Saarland University

Faculty of Natural Sciences and Technology I

Department of Computer Science

Master's Programme in Computer Science

Master's Thesis

Automatically Verifying "M" Modelling Language Constraints

submitted by

Thorsten Tarrach
<thorstent@live.com>

on 2010-08-31

Supervisor

Prof. Dr. Michael Backes

Advisor

Cătălin Hrițcu [1]

Reviewers

Prof. Dr. Michael Backes [1] [2]

Andrew D. Gordon [3]

[1]Saarland University
[2]Max Plank Institute for Software Systems
[3]Microsoft Research, Cambridge

Abstract

This thesis investigates the relationship between a type system and a general-purpose verification tool. We aim to statically verify typing constraints for a first-order functional language featuring dynamic type-tests and refinement types by translating the code to a standard while language with assertions. Our translation generates assertions in such a way that they express the same constraints in the while program as the typing constraints in the original program. We use a generic verification condition generator together with an SMT solver to prove statically that these assertions succeed in all executions. We formalise our translation algorithm using an interactive theorem prover and provide a machine checkable proof of its soundness. Additionally, we provide a prototype implementation using Boogie (a generic verification condition generation back-end) and Z3 (an efficient SMT solver) that can already be used to verify a large number of test programs.

Acknowledgements

I would like to thank Michael Backes for again letting me write a thesis at his chair. After having successfully completed my Bachelor's thesis at his chair, I know how much effort Michael and his researchers put into supervising Bachelor/Master students.

I am also grateful to Andy Gordon for agreeing to review this thesis as a second reviewer. I would also like to thank Andy and Microsoft Research Cambridge for making the source code of DMinor available to us. Without this the implementation of my translation would have been much harder.

I am profoundly indebted to my advisor, Cătălin Hrițcu, for his continuous support during the last months. His help with Coq and the necessary proofs, explaining DMinor to me, his advice on the implementation and countless hours in his office proved invaluable for the success of this thesis. The fact that he took time during his holidays to correct this text saved me a lot of stress in the final week before the deadline.

Last but not least, I would like to thank my friends, family and especially Sanja for their understanding that I had little time for them in the past months.

"We all make choices, but in the end our choices make us."
— Andrew Ryan (Bioshock)

Contents

1. Introduction

In this thesis we investigate the relationship between a type system and a general-purpose verification tool. Both sophisticated type-checkers and general-purpose verification tools are used to check programming constrains statically; the former one using, for instance, dependent and refinement types, the latter one using assertions. We bring these two approaches together by defining a translation from a language with refinement types to a language with assertions and prove that they can indeed verify the same properties.

We start with DMinor, a first-order functional language featuring dynamic type-tests and refinement types, for which a type-checker is already available [BGHL10]. DMinor is a subset of a more general language called "M".

"M" is a programming language currently developed by Microsoft that allows the modelling of domains using text. Domains are defined as any collection of related concepts or objects. Modelling domains consists of selecting certain characteristics to include in the model and implicitly excluding others deemed irrelevant [Mic09]. Microsoft provides tools to export "M" models into SQL Server databases, where the types become tables and the functions stored procedures [Sel09]. "M" comes as part of Microsoft's SQL Server Modeling Services and is expected to ship with a future major release of SQL Server [Mic].

The DMinor language is expressive enough to encode the whole "M" language. It is developed by Microsoft Research Cambridge with the goal of statically type-checking "M" programs [BGHL10].

In order to emulate the DMinor type-checker using a general-purpose verification tool we translate the code to Boogie, a while language with assertions, which are statically checked by the Boogie verification condition generator back-end.

Both the intermediate language and the verification condition generator are called Boogie and they are developed by Microsoft Research as a back-end for verification tools, such as the Verified C Compiler (VCC) [DMS+09] and Spec# [BLS05].

Both Boogie and DMinor require a theorem prover to discharge the proof obligations they generate and both use by default Z3. Z3 is an automated theorem prover for Satisfiability Modulo Theories [RT06a]. It is sound but not complete; that means it may produce false negatives by rejecting formulae that actually hold [DMB08].

To formalise our translation and prove it sound we use Coq [BBC+09], an interactive theorem prover. It mechanically checks our proofs and thus provides a high degree of confidence in our theoretical results.

Lastly we use F# [Mar10] for our implementation. F# is a functional programming language that also encompasses imperative object-oriented programming paradigms.

1.1. Related work

Work on program verification was pioneered by Hoare and Floyd in the late 60s [Hoa69] [Flo67]. Winskel provides an excellent introduction to while languages, Hoare logics and verification condition generators in [Win93]. Pierce et al. formalised a simple while language in Coq in the Software Foundation course [PCG$^+$10] and give Hoare rules for the commands in their while language; we use these Coq files as a basis for our formalisation.

Nipkow formalises Hoare logic for a while language with local variables, non-determinism, exceptions and, most important, (mutually) recursive procedures [Nip02b] [Nip02a]. Although Nipkow's formalisation is done in Isabelle/HOL, we use most of the ideas from his paper as a starting point for the Hoare logic we define in this thesis.

Automatic program verification has been worked on for a long time and there are a variety of tools available. Leino describes in [Lei05] the weakest precondition generation which is the heart of Boogie. There is also ESC/Java [FLL$^+$02], one of the leading program checkers of its kind, which verifies Java bytecode, but is neither sound nor complete. In the Mobius project [Mob06] two verification condition generators for Java bytecode were created. One of them translates Java bytecode to Boogie and this translation was proven sound in Coq.

There is also a long history of building type-checkers for programs [Pie02]. Although type-checkers usually only check for simple programming errors, such as trying to add a string to a number, there have been type-checkers developed that can process complicated specifications expressed using refinement types. Backes et al. developed a type-checker for verifying security protocols modelled in a variant of the Spi calculus [AB05] which can deal with dependent and refinement types [BHM08]. Bengtson et al. developed a security type-checker called F7 that works on top of F# and is capable of statically verifying properties expressed with dependent and refinement types [BBF$^+$08] [BFG10].

Relating type systems and software model-checkers is a topic that has received attention recently from the research community [KO09] [NP08] [JMR10]. However, our approach is different in that we relate a type system to a more traditional verification approach using Hoare logic and a verification condition generator.

1.2. Contributions

This thesis focuses on employing a general-purpose verification tool to achieve the same goal as the DMinor type-checker: Statically verifying that DMinor programs do not cause type-errors when executed. Our approach is interesting because a general-purpose verification tool opens up the possibility to take advantage of the huge amount of proven techniques and ongoing research done on such verification tools. This will allow us for example to use techniques for loop invariant inference [BHMR07] or to implement extensions to the "M" language such as mutable state more easily. Most importantly however, we show that, for the first-order functional language we are considering, a generic verification tool can check the same properties a sophisticated type-checker can. With our translation algorithm we create a strong link between these two areas of ongoing research.

This thesis consists of two parts: A theory part and a practical part, whereas the focus is on the theory. On the theory side we formalised a subset of the Boogie language, which we need for our translation. We also formalised Hoare logic rules for this subset, proved them sound and formalised a verification condition generator, which we proved sound as well. Finally, we formalised our translation from the functional DMinor language to the imperative Boogie language and proved that our translation is sound. Soundness guarantees that, if a DMinor program raises a runtime error, then Boogie will reject its translation. If Boogie accepts the translation of a DMinor program, then the program does not cause runtime errors and, since the translation preserves operational behaviour, the transformed program returns the same results[1] as the original one.

On the practical side we wrote a tool in F# that takes DMinor source code and outputs a Boogie file. This translation is partial in the sense that there are valid programs that cannot be translated. It successfully works on a large set of DMinor programs, mostly taken from the DMinor test suite, and the vast majority of those translated files pass through Boogie with the same result as in DMinor. A more detailed analysis is given in Section 5.5.

1.3. Overview

Chapters 2 and 3 introduce DMinor and Boogie respectively. In both chapters we use the Coq notation to define the syntax and semantics of the languages. In Chapter 4 we introduce our translation algorithm and prove it sound. We also give a number of examples to illustrate the translation. Our implementation is introduced in Chapter 5, where we also compare our tool-chain to the DMinor type-checker. Finally, in Chapter 6 we conclude and give some ideas for future work.

The appendices list relevant code for reference purposes. Appendix A lists a part of the DMinor formalisation, Appendix B lists selected Coq files that are part of our formalisation and, last but not least, Appendix C lists an important part of our implementation.

[1]Since DMinor programs can be non-deterministic, there is a set of possible results a program can return.

2. DMinor

DMinor is a first-order functional programming language developed by Microsoft Research and based on the "M" programming language [Mic09]. It supports refinement types (the type of values satisfying a boolean expression) [BBF^{+}08] and type-tests (boolean expressions that determine at run-time whether a value has a specific type) [BGHL10]. Functions in DMinor can have pre- and postconditions in the form of refinement types.

DMinor supports semantic subtyping [FCB08], which means that pure expressions (i.e. expressions without side-effects) are interpreted as first-order terms and types as logical formulae. Subtying is then “semantically” defined as a valid implication between the interpretations of types. Z3 is used to determine the validity of the implication.

Syntactically DMinor is closely related to "M", but with the ambiguity of "M" removed. Part of the metatheory of DMinor was already formalised in Coq [BGHL10] and we build on top of this formalisation

2.1. DMinor language

We will explain the language elements in more detail now.

2.1.1. Values

Values in DMinor can be simple values (such as integers, strings, logicals or Null values), collections or entities.

General	• Integer • Text • Logical • Null
Collection	A collection is a finite multi-set of values.
Entity	An entity is a finite set of label-value pairs. In other programming languages this value is commonly called a record.

2.1.2. Types and expressions

The following Coq code excerpt is taken from the DMinor formalisation. It shows the definition of types and expressions, which are mutually recursive.

```
Inductive MType : Type :=
  | Any : MType
  | Integer : MType
  | Text : MType
  | Logical : MType
  | Coll : MType → MType
  | Entity : string → MType → MType
  | Refine : string → MType → Exp → MType
with Exp : Type :=
  | Var : string → Exp
  | Value : Value → Exp
  | UnOp : UnOp → Exp → Exp
  | BinOp : BinOp → Exp → Exp → Exp
  | If : Exp → Exp → Exp → Exp
  | Let : string → Exp → Exp → Exp
  | In : Exp → MType → Exp
  | Entity : list (string × Exp) → Exp
  | Dot : Exp → string → Exp
  | Add : Exp → Exp → Exp
  | Acc : string → Exp → string → Exp → Exp → Exp
  | App : string → Exp → Exp.
```

Types

DMinor has a number of build-in types. There are four primitive types:

Any	The Any type is a top type and all values are in this type.
Integer	Integers in DMinor are not restricted to a certain range as in other programming languages. All basic mathematical operations except division are supported on integers.
Logical	Logical is usually called boolean in other languages and supports all operations on booleans one would expect.
Text	Text is the type of strings, but unlike in other programming languages the only operation on texts is comparison.

There are three composite types that can be mutually recursive.

Coll T	A collection type is the type of collections containing only elements of type T.

Entity x T	An entity type is the type of an entity that has (at least) a field named x holding data of type T. Entity types have only one field because an entity with several fields can be represented using an intersection type. Intersection types can be encoded using refinements, type-test and boolean conjunction; the details are given in [BGHL10]
Refine x e T	A refinement type is a type with an additional boolean constraint e. A value v is an element of type Refine x e T if v has type T and additionally the expression e{v/x} evaluates to true.

Refinement types can be used to encode pre- and postconditions for functions. Furthermore refinement types can only use pure expressions, see Section 2.2.2.

Other types, like union, intersection and negation types can be represented using refinement types and dynamic type-tests [BGHL10].

Expressions

DMinor defines a number of expressions, most notably type-testing.

Var x	Returns the value of variable x.
Value v	Returns the constant v.
UnOp ⊕ e	Applies the unary operator ⊕ to e. The only unary operator in DMinor is logical negation.
BinOp ⊕ e1 e2	Applies the binary operator ⊕ to e1 and e2. A complete list of operators can be found in [BGHL10].
If e1 e2 e3	Evaluates e1 as a guard and evaluates e2 if the guard is true and e3 otherwise.
Let x e1 e2	Assigns e1 to x and evaluates e2.
In e T	Returns true if e is in type T, false otherwise.
Entity el	el is a list of label-value pairs and **Entity** returns a new entity. Although entities can also be created with the **Value** expression, this expression allows to create entities that are not constant and contain computed values. It is needed because entities cannot be updated once they are created.
Dot e l	Retrieves the value in field l of entity e.
Add e1 e2	Returns a new collection in which an element e1 is added to the collection e2.
Acc x e1 y e2 e3	Accumulate first evaluates e1 to a collection and e2 to an initial value. It then executes e3 for every element in the collection. x holds the current element from the collection and y begins with the initial value and then holds the value from the last iteration.
App f e	Calls function f with argument e. Multiple arguments can be encoded using entities.

2.2. Semantics

Above we gave an informal description of the semantics of DMinor. Bierman et al. give three equivalent formal semantics for DMinor [BGHL10].

First is a quite standard small-step operational semantics. Secondly, they give a logical semantics, which is only valid for pure expressions. The logical semantics interprets expressions as first-order terms and types as formulae and is used for semantic subtyping. The logical semantics is formalised using three mutually recursive functions: R returns a first-order logic term denoting the result of pure expressions. F tests if a value is in a specific type and W tests if a type-test goes wrong. The reason for the existence of the W function is that F is total and has to return a boolean value even if a type-test inside a refinement type raises an error. In the formal development we are only considering the error tracking logical semantics.

Third is a big-step semantics that is defined as a relation from an expression to either an **Error** or to **Return**(v), where v is the value that is returned. Errors are generated in a number of situations, for example in case one tries to access a field in an entity that does not have this field, or one tries to feed operands of a wrong type to an operator. A type-test can generate an error not only if the subexpression that is to be tested generates an error, but also if the type is refined with an expression that returns an error. All expressions bubble up errors if one subexpression returns an error and there is no way to recover from an error.

2.2.1. Big-step semantics in Coq

In our soundness proof in Section 4.4 we relate our translation to the big-step semantics, which is formalised in Coq using the Eval relation.

```
Inductive Result : Type :=
  | Error : Result
  | Return : Value → Result.
```

```
Inductive Eval : Exp → Result → Prop.
```

The Eval relation takes an expression and returns a Result. Unlike a function, a relation needs neither to be total nor deterministic.

A complete listing of the big-step semantics can be found in Appendix A.

2.2.2. Purity

The notion of purity is important for several parts in our translation. As stated above, the logical semantics is only defined on pure expressions and refinement types may only contain pure expressions. We quote Bierman et al. here [BGHL10]:

"The gist is that pure expressions must be terminating, have a unique result (which may be Error), and must only call functions whose bodies are pure."

2.2.3. Environments and simultaneous substitution

Environments

An environment in DMinor maps variable names to values.

```
Definition Env := string → Value.

Definition env_empty (x : string) := v_null.

Definition env_update (env : string → Value) (x : string) (v : Value) :=
  fun y ⇒ if beq_str x y then v else env y.
```

The environment is updated by returning a new function that returns the new value for the variable that is updated and the old value for any other variable.

Substituting free variables

We implement a simultaneous, capture-avoiding substitution subst_state : Exp → Env → Exp that eliminates all free variables from an expression and replaces them with the value from from an environment. The full code for the substitution algorithm is given in Appendix B.3.1. We also define a similar substitution function for types: subst_state_t : MType → Env → MType.

2.2.4. Relating operational and logical semantics.

For our translation we need a relation between the operational and logical semantics with respect to type-tests. While in the DMinor formalisation a relation between R and the big-step semantics is already proven, we prove a relation between F and the big-step semantics and between W and the big-step semantics. For F we prove that, if the big-step semantics returns b for testing if x is in T, then F also returns b, where b is either true or false.

```
Lemma eval_F : ∀ b T x st,
  contains_impure_expressions (subst_state_t T st) = false →
  Eval (In (Value x) (subst_state_t T st)) (Return (v_logical b)) →
  F T x st = b.
```

In order to relate W to the big-step semantics, we prove that, if the big-step semantics returns an **Error**, W will return true.

```
Lemma eval_W : ∀ T x st,
  contains_impure_expressions (subst_state_t T st) = false →
  Eval (In (Value x) (subst_state_t T st)) (Error) →
  W T x st = true.
```

If the evaluation returns a value, then W returns false.

```
Lemma eval_W_false : ∀ T x st v,
  contains_impure_expressions (subst_state_t T st) = false →
  Eval (In (Value x) (subst_state_t T st)) (Return v) →
  W T x st = false.
```

2.3. Examples

2.3.1. Accumulate example

This example shows a very simple accumulate that filters out all null values from a collection c. For simplicity the collection we accumulate over is a parameter of our definition. On each iteration one element x form this collection is compared to the null value and if this comparison returns false, x is added to the resulting collection y.

```
Definition acc_sample coll :=
  (Acc x (Value coll) y (Value empty_coll)
    (If (BinOp OEq (Var x) (Value v_null))
      (Var y)
      (Add (Var x) (Var y))
    )).
```

2.3.2. Example of an execution error

In this example we show how a refinement type can cause an execution error. We test if v_tt (the true value) is of type Any and is greater than 5. Since the greater operator only accepts operands of type Integer, v_tt is not an acceptable operand. Therefore, in DMinor this causes an execution error.

```
Definition in_sample :=
  In (Value v_tt) (Refine v Any (BinOp OGt (Var v) (Value (v_int 5)))).
```

We prove that this program evaluates indeed to **Error** using the big-step semantics.

Lemma in_sample_err : Eval in_sample **Error**.

2.3.3. Incompleteness of the type system

This example is similar to the one above, but this time we test if 5 is greater than 5. According to the operational semantics this is a valid type-test that will return v_ff (the false value). We show that this example is rejected by the DMinor type system due to the incompleteness of the type system with respect to the operational semantics.

Definition incom_sample :=
  In (**Value** (v_int 5)) (Refine x Any (**BinOp** OGt (**Var** x) (**Value** (v_int 5)))).

We prove that according to the big-step semantics this indeed evaluates to v_ff.

Lemma incom_sample_big : Eval incom_sample (**Return** v_ff).

Since the expression is pure, the logical semantics is also defined for this expression and it also evaluates to v_ff.

Lemma incom_sample_logical : R incom_sample env_empty = **Return** v_ff.

However, the expression fails to type-check in the type system. The reason is that our refinement type Refine x Any (...) specifies type Any for the variable x and the greater operator **BinOp** OGt (**Var** x) (**Value** (v_int 5)) does not type-check successfully with an argument of type Any. It does not matter that the actual type of the variable x during execution is indeed an Integer.

Lemma incom_sample_type : $\neg$ envT_empty $\vdash$ incom_sample : Logical.

The complete DMinor type system is given by Bierman et al. [BGHL10]. We will come back to this example when we relate our translation to the type system in Section 4.4.5.

3. Boogie

Boogie is an intermediate language and a static verification tool developed by Microsoft Research [Lei08] [BCD$^+$06] [DL05] [BL05]. Programs written in other languages, such as C and Spec# [BLS05], are translated to Boogie as an intermediate step and Boogie then verifies that the pre-, postconditions and asserts in the code hold. Boogie works by generating verification conditions and then checking them using an SMT solver.

3.1. Boogie language

In principle Boogie is a while language that supports certain additional commands like assume and assert for example. An assert takes a boolean expression and statically checks if that formula holds at a certain point in the program during any program run. Asserts are not necessarily computable, for example they can quantify over infinite structures. Boogie also supports manually adding assumptions and will only consider program runs where that assumption holds. If a false assumption is introduced to a branch, then that branch is always considered correct.

3.2. Formalising a subset of Boogie

As one of the main goals of this thesis we want to prove the translation from DMinor to Boogie sound. In order to do that, we need a formal representation of both Boogie and DMinor. We use the existing formalisation of DMinor as the source of our translation and we formalise Boogie as the target.

We do not formalise the full Boogie language, but only the subset which is relevant for our translation. Still, the subset of Boogie we formalise is considerably more expressive than a standard while language. Our formalisation supports collections, records, asserts, mutually recursive procedures, variable scoping and evaluation of logical formulae. Not all of these constructs are primitive in Boogie, but we encode the ones that are not primitive in our implementation described in Chapter 5.

3.2.1. Embedding of the logic

Some of the expressions and commands we define, such as type-tests and assertions, need to evaluate logical formulae, which may include quantifiers for instance. Generally speaking we could give a shallow or a deep embedding for these formulae. When using shallow embedding, logical formulae are written directly in the interactive theorem prover's logic as a function from Env to bool. Whereas for deep embedding one devises a datatype in the theorem prover that represents the syntax of these formulae and gives an evaluation function for this datatype. There is a more in-depth discussion of shallow vs. deep embedding in [WN04].

Deep embedding has the advantage that the power of the logic can be controlled. In our case we could have wanted to limit our logic to first-order. Nevertheless, we chose shallow embedding because the workload associated with defining and proving properties of syntactic operations on formulae would have been too high and there would be little benefit for this thesis. Nipkow makes the same decision for embedding logical formulae in Isabelle for his while language in [Nip02b].

```
Definition Assertion := Env → bool.
```

We enforce classical logic in our assertions by representing assertions as functions to bool, which satisfies the property of excluded middle. As Coq does not define quantifies for bool, we defined our own using a strong variant of the excluded middle axiom:

```
Definition forall_bool (A : Type) (P : A → bool) : bool :=
  match ClassicalEpsilon.excluded_middle_informative (∀ x, P x = true) with
  | left Ptrue ⇒ true
  | right Pfalsee ⇒ false
  end.
```

3.2.2. Expressions

The syntax of our while language is separated into two distinct classes: expressions, which are side-effect free, and commands, which have side-effects.

Our expressions allow basic operations on values, most of which directly correspond to the operations in DMinor. As for the Value type in the formalisation we assume that these expressions are native to Boogie; for the real Boogie tool we axiomatise them.

```
Inductive expr : Type :=
  | EValue : Value → expr
  | EVar : string → expr
  | ECollAdd : expr → expr → expr
  | ECollRem : expr → expr → expr
  | ECollEmpty : expr → expr
  | EEntUpd : string → expr → expr → expr
  | EDot : expr → string → expr
  | EIn : Assertion → expr
  | EOr : expr → expr → expr
  | ENot : expr → expr
  | EEq : expr → expr → expr
  | ELt : expr → expr → expr
  | EGt : expr → expr → expr
  | EPlus : expr → expr → expr
  | EMinus : expr → expr → expr
  | ETimes : expr → expr → expr
  | EAnd : expr → expr → expr.
```

Some of these constructs need further explanation, so we focus on the ones that do.

EValue v	Evaluates to a constant value v.
EVar x	Looks up the variable x in the current state (environment).
ECollAdd c e	Return a new collection that contains all the elements of c as well as the element e.
ECollRem c e	Returns a new collection that contains all the elements of c except for e. If e is not in c, then c is returned. If e is in the collection multiple times, only one instance is removed. This functionality is not present in DMinor, but can be encoded there. We need this expression to translate accumulate.
ECollEmpty c	Tests if the collection c is empty.
EEntUpd l v e	Returns a new entity that contains all the label-value pairs of e as well as the label l and the corresponding value v. This expression is not present in DMinor and cannot be encoded easily, but it could be added as a primitive easily. We need it to construct entities.
EDot e l	Retrieves the value with the label l from the entity e.
EIn a	EIn is a special construct that is used to translate the type-test (**in**) from DMinor. Given the assertion a it evaluates this assertion to either true or false as a value. A typical type-test to test if variable x is an integer is translated as EIn (fun env : Env ⇒ F Integer (env "x") env).

The remaining expressions work in the same way as their DMinor counterparts.

The semantics of expressions is defined using an evaluation function that takes a state and the expression to evaluate. It returns a Result, which can either be an **Error** or the value the evaluation yields (see Section 2.2.1). **Error** is used to signal runtime errors, and is returned for example when an integer is added to a string. We only show the most interesting cases here, the full code can be found in Appendix B.1.2.

```
Program Fixpoint eeval (st : Env) (e : expr) {struct e} : Result :=
  match e with
  | EValue v ⇒ Return v
  | EVar x ⇒ Return (st x)
  | ECollAdd c e ⇒ LBind (eeval st c) (fun c ⇒
          (LBind (eeval st e) (fun e ⇒
          if is_C c then Return (v_add e c) else Error)))
  | ECollEmpty c ⇒ LBind (eeval st c) (fun c ⇒
          if is_C c then Return (v_empty c) else Error)
  | ECollRem c e ⇒ LBind (eeval st c) (fun c ⇒
          (LBind (eeval st e) (fun e ⇒
          if is_C c then Return (v_remove e c) else Error)))
  | EEntUpd l v e ⇒ LBind (eeval st v) (fun val ⇒
          (LBind (eeval st e) (fun ent ⇒
          if is_E ent then Return (v_eupdate l val ent) else Error)))
  | EDot e l ⇒ LBind (eeval st e) (fun x ⇒
          if is_E x ∧ v_has_field l x then Return (v_dot l x) else Error)
  | EIn a ⇒ Return (v_logical (a st))
...
  end.
```

LBind is used to bubble up errors from subexpressions. Some constructs need guards, for instance evaluating EDot first makes sure e is actually an entity and has the field l. This is required because v_dot is a total function that will return a value for every input. The guards make sure an error is returned if the input is of a wrong type.

3.2.3. The commands

The first five commands are standard for a while language [Win93]. Evaluation of our commands works by relating two states (environments), a starting state and a result state (or an error state).

skip	This command does nothing.
c1; c2	A sequence first executes c1 and then c2.
l := a	Assigns the result of the expression a to the variable l.
if xb **then** e1 **else** e2	If xb is true then e1 is executed, otherwise e2. xb is a variable, not an expression. This ensures that the evaluation of the guard cannot yield an error, which simplifies the Hoare rule for conditionals.
while b **inv** a **do** c **end**	This while loop executes until the guard expression b evaluates to false. c is the body of the loop. The while command takes an additional assertion as an argument, which is operationally ignored, but used in verification condition generation.

In addition to these standard commands, we add the following ones:

assert a	If the logical formula a is valid this command does nothing, otherwise it returns a **CError**.

x := **pick** xc	This command non-deterministically chooses an element from a collection xc and writes this element into variable x. The reason this is a command and not an expression as the other operations on values is that Pick is non-deterministic, which cannot be modelled with the evaluation function for expressions, because said function has to return a single value. Pick is less general than the non-deterministic choice operator Nipkow defines [Nip02b], but is all we need for our translation.
call P	We have a global collection of procedures, which are called using the call command that takes only the name of the procedure to call. For procedure calls, operationally the callee works on the same variables as the caller. This makes variables global in a sense. There are no arguments or return values specified by the command, these are established below by a calling convention.
backup x **in** c	This command backs up the values of all variables, executes c and restores all values except for variable x. The intended use is a procedure call where all variables become local except for the return value x.

The formal definition of commands is a follows.

```
Inductive com : Type :=
  | CSkip : com
  | CAss : string → expr → com
  | CSeq : com → com → com
  | CIf : string → com → com → com
  | CWhile : expr → Assertion → com → com
  | CAssert : Assertion → com
  | CPick : string → string → com
  | CCall : proc_name → com
  | CBackup : string → com → com.
```

3.2.4. Calling convention for procedure calls

For procedure calls we use the following calling convention: Procedures take one argument, which is copied by the caller into the variable "arg". Multiple arguments can be encoded using an entity. The result is put into a variable called "ret" that is then copied by the caller into the designated return variable. A **backup** command makes sure no other variables but the return variable x are changed.

```
Definition call_arg := "arg".
Definition call_ret := "ret".
Definition fun_call x i e :=
  (backup x in (call_arg := e; (call i); x := EVar call_ret)).
```

The next command starts a module. This is a technicality that allows us to instantiate the module with a set of procedures. Otherwise the set of procedures would have to be passed to every function or be fixed at this point. We fix the procedures later in our translation. Since a module has to end when the Coq file ends we start a new module in every file. But all the modules have in common that they leave certain parameters for later instantiation. This module contains the operational semantics of commands.

```
Module Type ImpArgs.
  Parameter procs : proc_name → com.
End ImpArgs.
Module ImpM (Args : ImpArgs).
```

3.2.5. Operational semantics

We define a big-step semantics for our while language. As for the big-step semantics in DMinor (see Section 2.2.1) our evaluation can cause runtime errors. In the case that the evaluation is successful it results in a new state, otherwise it results in a **CError**.

```
Inductive CResult :=
  | CError : CResult
  | CReturn : Env → CResult.
```

Technically, our operational semantics is not defined with an evaluation function, but with a relation, just as the DMinor big-step semantics (see Section 2.2.1).

Runtime typing errors are handled by divergence. Divergence means that for a specific pair of command and state there does not exist a relation to continue evaluating from there. This can happen for instance when an expression in an assignment returns **Error**, or when we try to pick an element from an empty collection.

One such typing error would be the following command, which tries to add 42 to false:
x := EPlus (EValue (v_int 42)) (EValue (v_ff))

Since Boogie checks only for partial correctness, divergence is considered correct because the program never finishes. In our translation we avoid this by adding enough asserts so that every typing error becomes a **CError** in Boogie. This is consistent with how the Boogie tool works. Boogie has no way to type-check the custom operations we define on sort Value, so that we need to add enough asserts to check the parameters before executing a command.

```
Inductive ceval : Env → com → CResult → Prop :=
  | CESkip : ∀ st,
      skip / st ⇝ (CReturn st)
  | CEAss : ∀ st a1 (n:Value) l,
      eeval st a1 = (Return n) →
      (l := a1) / st ⇝ CReturn (env_update st l n)
```

```
| CESeq : ∀ c1 c2 st st′ st″,
      c1 / st ⇝ CReturn st′ →
      c2 / st′ ⇝ st″ →
      (c1; c2) / st ⇝ st″
| CESeqErr : ∀ c1 c2 st,
      c1 / st ⇝ CError →
      (c1; c2) / st ⇝ CError
| CEIfTrue : ∀ st st′ b1 c1 c2,
      syn_beq_val (st b1) v_tt = true →
      c1 / st ⇝ st′ →
      (if b1 then c1 else c2) / st ⇝ st′
| CEIfFalse : ∀ st st′ b1 c1 c2,
      syn_beq_val (st b1) v_tt = false →
      c2 / st ⇝ st′ →
      (if b1 then c1 else c2) / st ⇝ st′
| CEWhileEnd : ∀ b1 b a1 st c1,
      eeval st b = Return b1 →
      syn_beq_val b1 v_tt = false →
      (while b inv a1 do c1 end) / st ⇝ CReturn st
| CEWhileLoop : ∀ st st′ st″ b1 b a1 c1,
      eeval st b = Return b1 →
      syn_beq_val b1 v_tt = true →
      c1 / st ⇝ CReturn st′ →
      (while b inv a1 do c1 end) / st′ ⇝ st″ →
      (while b inv a1 do c1 end) / st ⇝ st″
| CEWhileLoopErr : ∀ st b1 b a1 c1,
      eeval st b = Return b1 →
      syn_beq_val b1 v_tt = true →
      c1 / st ⇝ CError →
      (while b inv a1 do c1 end) / st ⇝ CError
| CEAssert : ∀ (st:Env) (b:Assertion),
      b st = true →
      (assert b) / st ⇝ CReturn st
| CEAssertErr : ∀ st b,
      b st = false →
      (assert b) / st ⇝ CError
| CEPick : ∀ st x xc v,
      is_C (st xc) = true →
      v_mem v (st xc) = true →
      (x := pick xc) / st ⇝ CReturn (env_update st x v)
| CECall : ∀ st st′ pn,
      Args.procs pn / st ⇝ st′ →
      (call pn) / st ⇝ st′
| CEBackup : ∀ st st′ v c,
        c / st ⇝ (CReturn st′) →
        (backup v in c) / st ⇝ CReturn (env_update st v (st′ v))
  | CEBackupErr : ∀ st v c,
        c / st ⇝ CError →
        (backup v in c) / st ⇝ CError
  where "c1 / st ⇝ st'" := (ceval st c1 st′).
```

The **pick** command is non-deterministic if the collection contains more than one element. The only command that can *produce* a **CError** is **assert**. The remaining commands simply bubble up errors produced by a nested assert somewhere.

3.2.6. Call-depth-indexed operational semantics

As done by Nipkow [Nip02b], we define a second version of the above semantics that limits the maximum call depth to n and thereby guarantees that the program does no more than n nested procedure invocations. We need this trick to prove the soundness of the Hoare rules for the **call** statement. The **call** statement is the only statement that actually changes the index by decreasing it by 1 and it diverges if the index is not at least 1. Evaluating all other commands leaves the index unchanged.

```
Inductive ceval_indexed : Env → com → nat → CResult → Prop :=
...
  | CEnCall : ∀ st st′ pn n,
        Args.procs pn / st — n ⇝ st′ →
        (call pn) / st — S n ⇝ st′
  where "c1 '/' st '—' n '⇝' st'" := (ceval_indexed st c1 n st′).
```

One of the properties of this new semantics is that, if the command evaluates without diverging in i1 steps, it will also evaluate without diverging for every larger number of steps and the resulting state is the same.

```
Lemma ceval_step_more: ∀ i1 i2 st st′ c,
  i1 ≤ i2 → c / st — i1 ⇝ st′ →
  c / st — i2 ⇝ st′.
```

The following Lemma relates the two semantics by stating that there exists an n, for which the call-depth-indexed semantics gives the same result that the original one gives.

```
Lemma exec_iff_execn : ∀ c st st′,
  c / st ⇝ st′ ↔ ∃ n, c / st — n ⇝ st′.
```

3.3. The Hoare rules

The Hoare module is all about defining the Hoare logic of our commands. It is based on the Hoare chapter in [PCG$^+$10] and takes many ideas presented by Nipkow [Nip02b]. The Hoare rules are needed to prove the verification condition generator sound in Section 3.5.

The Hoare module needs in addition to the procedure list also a type parameter, which we call ZType. Its purpose becomes obvious later. We require that ZType is inhabited.

```
Module Type HoareArgs.
  Parameter procs : proc_name → com.
  Parameter ZType : Type.
  Parameter ZType_inhabited : ZType.
End HoareArgs.
Module HoareM (Args : HoareArgs).
Module ImpSpecificArgs := ImpM Args.
```

3.3.1. Definition of the semantic Hoare triples

As done in [PCG$^+$10], we start by working with semantic Hoare triples. We then define the corresponding syntactic triples and prove them sound using the semantic ones.

An assertion now depends on an auxiliary variable of type ZType. This definition shadows the original definition of assertions, which is still available under the name Imp.Assertion.

```
Definition Assertion := Args.ZType → Env → bool.
```

Since assertions are functions themselves evaluating them is done simply by passing them the state they should be evaluated against. However, because our evaluation relation for commands can produce errors, checking the postcondition is not as easy as the precondition. Our semantics for checking the postcondition is that, if the evaluation returns a **CError**, then the postcondition is constantly false, irrespective of the assertion. In case the evaluation succeeded and returned a new state the postcondition is evaluated against that.

```
Definition check_post Q (z:Args.ZType) st' :=
  match st' with
  | CError ⇒ False
  | CReturn st ⇒ Q z st = true
  end.
```

There is another way to define check_post, which is possibly more intuitive.

```
Definition check_post' Q (z:Args.ZType) st' :=
  ∃ st'', st' = CReturn st'' ∧ Q z st'' = true.
```

We give a short proof that these two definitions are indeed identical.

```
Lemma check_post_identical : ∀ Q z st′,
  check_post Q z st′ ↔ check_post′ Q z st′.
```

A semantic Hoare triple {P}c{Q} consists of a precondition, a command and a postcondition. It holds if whenever the precondition holds in the initial state, and the command is evaluated in this initial state, the resulting state is not an error and it has to satisfy the postcondition.

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st′,
        c / st ⇝ st′
    → (∀ z:Args.ZType, P z st = true
    → check_post Q z st′).
```

The z is an addition to the traditional Hoare triple and models auxiliary variables, whose treatment needs to be made explicit in the presence of recursive procedures. Our implementation of auxiliary variables resembles the one presented by Nipkow [Nip02b], who follows the solution of Morris [Mor] and Kleymann [Kle99].

A very important property of our semantic Hoare triples is that, if the precondition is universally valid and there is any postcondition Q that makes the triple hold, then the command c does not evaluate to a **CError** in any state. The property follows immediately from the definition of semantic Hoare triples.

```
Definition valid_formula f := ∀ (z:Args.ZType) (st:Env), f z st = true.
```

```
Lemma hoare_triple_pre_valid_no_error : ∀ c P Q,
  valid_formula P →
  { P } c { Q } → no_error c.
```

In order to avoid exposing the reader too much to the shallow embedding we define valid. This represents the place where usually the SMT-Solver would be called.

```
Definition valid (H:Prop) := H.
```

Corresponding to the Hoare triples above, there is a definition for the call-depth-indexed semantics. We use the suffix _n to indicate functions that use the call-depth-indexed semantics. Since the Hoare rule for **call** can only be proven using the call-depth-indexed semantics, we use that to prove our Hoare triples sound.

We use the syntax $\models_n$ {P}c{Q} to describe a Hoare triple where the maximal depth of the call stack is n.

```
Definition hoare_triple_n n (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st′,
        c / st — n ⇝ st′
    → (∀ z:Args.ZType, P z st = true
    → check_post Q z st′).
```

A context is a list of commands and pre- and postconditions for them. It is in a sense a list of Hoare triples.

```
Definition context := list (Assertion × com × Assertion).
```

The function sem_context checks if each of the Hoare triples in a context is correct.

```
Fixpoint sem_context ct { struct ct } :=
  match ct with
  | nil ⇒ True
  | (P,c,Q)::cl ⇒ { P } c { Q } ∧ sem_context cl
  end.
```

```
Fixpoint sem_context_n n ct { struct ct } :=
  match ct with
  | nil ⇒ True
  | (P,c,Q)::cl ⇒ ⊨ n { P } c { Q } ∧ sem_context_n n cl
  end.
```

An extended Hoare triple has, in addition to the Hoare triple from above, a context. The Hoare triples in that context are assumed to hold. These extended Hoare triples are used to prove the correctness of non-mutually recursive procedures.

We use the syntax $C \models_n \{P\}c\{Q\}$ to describe such a Hoare triple where C is the context.

```
Definition ext_hoare_triple C P c Q :=
  sem_context C → { P } c { Q }.
```

```
Definition ext_hoare_triple_n n C P c Q :=
  sem_context_n n C → ⊨ n { P } c { Q }.
```

Last but not least, the Hoare judgement states that a context C2 holds if a context C1 holds. The Hoare judgement is used to to prove the correctness of a number of mutually recursive procedures.

We use the syntax $C1 \models_n C2$ to describe a Hoare judgement where C1 is the list of known Hoare triples and C2 the Hoare triples we want to prove.

```
Definition ext_hoare_judgement C1 C2 :=
  sem_context C1 → sem_context C2.
```

```
Definition ext_hoare_judgement_n (n:nat) C1 C2 :=
  sem_context_n n C1 → sem_context_n n C2.
```

3.3.2. Lemmas for deriving semantic Hoare triples

In this chapter we prove lemmas that allow us to obtain semantic Hoare triples from other semantic Hoare triples for our commands.

Skip command

As the **skip** command does not change the state, any assertion that is true before **skip** also holds after the execution of the command.

```
Theorem hoare_skip_n : ∀ C P n,
      C ⊨ n { P } skip { P }.
```

Assignment command

The function assn_sub sets a variable in the state to the result of evaluating a certain expression. The case that this expression evaluates to an **Error** can be ignored because in that case the assignment command would diverge anyway.

Instead of a real substitution of the variable in the assertion P, assn_sub just replaces the variable in the state with the constant value before passing the state to P.

```
Definition assn_sub P x a : Assertion :=
  fun (z:Args.ZType) (st : Env) ⇒ P z (env_update st x (eeval_return st a)).
```

The Hoare rule for assignments intuitively corresponds to {P{x/a}}x := a{P}. The precondition is just the postcondition where the variable that is assigned to is replaced with the constant value of that expression.

```
Theorem hoare_asgn_n : ∀ C P x a n,
  C ⊨ n { assn_sub P x a } (x := a) { P }.
```

Sequence command

The sequence rule states that, if two individual commands c1 and c2, where the postcondition of c1 is the precondition of c2, are put into a sequence c1;c2, that sequence is again a valid Hoare triple.

$$\frac{\{P\}c1\{Q\} \qquad \{Q\}c2\{R\}}{\{P\}c1;c2\{Q\}}$$

```
Theorem hoare_seq_n : ∀ C P Q R c1 c2 n,
      C ⊨ n { Q } c2 { R }
  → C ⊨ n { P } c1 { Q }
  → C ⊨ n { P } c1;c2 { R }.
```

If command

To formalise the Hoare rule for conditionals we need to state in the precondition that a certain variable is true. We use bassn for that. The bassn function creates an assertion that checks if a certain variable in the state is true. We use the fact here that the guard of a conditional is always a variable.

```
Definition bassn i : Assertion :=
  fun z st ⇒ syn_beq_val (st i) v_tt.
```

We can now introduce the Hoare rule for conditionals. It states that P is a precondition and Q is a postcondition, if P ∧ b is a pre- and Q a postcondition for the then-part c1 and P ∧ ¬ b is a pre- and Q a postcondition for the else-part c2. The variable b is the guard of the conditional.

$$\frac{\{P \land b\}c1\{Q\} \qquad \{P \land \neg b\}c2\{Q\}}{\{P\}\textbf{if}\ b\ \textbf{then}\ c1\ \textbf{else}\ c2\{Q\}}$$

```
Theorem hoare_if_n : ∀ C P Q b c1 c2 n,
  C ⊨ n { fun (z:Args.ZType) st ⇒ (P z st) ∧ (bassn b z st) } c1 { Q } →
  C ⊨ n { fun (z:Args.ZType) st ⇒ (P z st) ∧ (¬ (bassn b z st)) } c2 { Q } →
  C ⊨ n { P } if b then c1 else c2 { Q }.
```

While command

A while loop evaluates the guard before every loop iteration. So we could not simply make the guard a variable as for the conditional. Because an evaluation can return an **Error**, we define eeval_bool that returns false in case of an error and otherwise compares the resulting value to v_tt.

```
Definition eeval_bool st e :=
  match eeval st e with
  | Return v ⇒ syn_beq_val v v_tt
  | Error ⇒ false
  end.
```

The Hoare rule for the while loop requires that the loop body has P as an invariant, that the precondition for the loop body is P and that the guard holds. The postcondition for the while rule states that b does not hold.

$$\frac{\{P \land b\}c\{Q\}}{\{P\}\textbf{while}\ b\ \textbf{inv}\ P\ \textbf{do}\ c\ \textbf{end}\{Q \land \neg b\}}$$

```
Theorem hoare_while_n : ∀ C P b c z n,
  C ⊨ n { fun z st ⇒ P z st ∧ eeval_bool st b } c { P } →
  C ⊨ n { P }
    (while b inv (fun st ⇒ P z st) do c end)
```

{ fun z st ⇒ P z st ∧ ¬ (eeval_bool st b) }.

Assert Command

The Hoare rule for assertions requires as a precondition that the assertion holds.

$$\{Q \wedge b\}\textbf{assert}\ b\{Q\}$$

```
Theorem hoare_assert_n : ∀ C (Q:Assertion) (b:Imp.Assertion) n,
  C ⊨ n { fun z st ⇒ b st ∧ Q z st } assert b { Q }.
```

Pick command

The Hoare rule for **pick** works similar to the assignment rule, just that it requires that the postcondition holds when x is assigned any element of the collection.

$$\{\forall v \in xc, P\{v/x\}\}x := \textbf{pick}\ xc\{P\}$$

```
Theorem hoare_pick_n : ∀ C P xc x n,
  C ⊨ n { fun z st ⇒
  forall_bool (fun v ⇒ implb (v_mem v (st xc)) (assn_sub P x (EValue v) z st)) }
  x := pick xc
  { P }.
```

Backup command

The **backup** x **in** c command evaluates the Hoare triple for c using the same state for the pre- and postcondition, except for variable x which is updated. The tricky thing is to “transfer” the state from the pre- to the postcondition. We do that by quantifying over a new state st′ that we require to be equal to the state in the precondition.

This is similar to the auxiliary variable z we define for semantic Hoare triples in general in Section 3.3.1, but we cannot use z because the type of z is opaque.

$$\frac{\forall st', \{\text{fun } st \Rightarrow P\ st \wedge st' = st\}c\{\text{fun } st \Rightarrow Q\{st\ x/x\}\ st'\}}{\{P\}\textbf{backup}\ x\ \textbf{in}\ c\{Q\}}$$

```
Theorem hoare_backup_n : ∀ C P Q x c n,
      (∀ st′, C ⊨ n { fun z st ⇒ P z st ∧ beq_env st′ st }
         c
      { fun z st ⇒ Q z (env_update st′ x (st x)) } )
  → C ⊨ n { P } backup x in c { Q }.
```

The consequence rule

Before we go on with the Hoare rule for **call** we introduce the consequence rule. The consequence rule describes, in what way the pre- and postcondition can be changed for a given Hoare triple. This will become clearer with the derived rules below.

$$\frac{\{P'\}c\{Q'\} \qquad \forall st\ st', (\forall z, P'\ z\ st = true \rightarrow Q'\ z\ st') \rightarrow (\forall z, P\ z\ st = true \rightarrow Q\ z\ st')}{\{P\}c\{Q\}}$$

```
Theorem hoare_consequence_n : ∀ C (P P′ Q Q′ : Assertion) c n,
  C ⊨ n { P′ } c { Q′ } →
  (∀ st st′,
  (∀ (z:Args.ZType), P′ z st = true → check_post Q′ z st′) →
  (∀ (z:Args.ZType), P z st = true → check_post Q z st′)) →
  C ⊨ n { P } c { Q }.
```

The hoare_consequence_pre corollary proves that a precondition P' can be replaced by a stronger precondition P. Stronger means that precondition P implies P' for all states.

$$\frac{\{P'\}c\{Q\} \qquad \forall st\ z, (P\ z\ st \rightarrow P'\ z\ st)}{\{P\}c\{Q\}}$$

```
Corollary hoare_consequence_pre_n : ∀ C (P P′ Q : Assertion) c n,
  C ⊨ n { P′ } c { Q } →
  (P ⟶ P′) →
  C ⊨ n { P } c { Q }.
```

Conversely, the postcondition can be replaced by a weaker postcondition.

$$\frac{\{P\}c\{Q'\} \qquad \forall st\ z, (Q'\ z\ st \rightarrow Q\ z\ st)}{\{P\}c\{Q\}}$$

```
Corollary hoare_consequence_post_n : ∀ C (P Q Q′ : Assertion) c n,
  C ⊨ n { P } c { Q′ } →
  (Q′ ⟶ Q) →
  C ⊨ n { P } c { Q }.
```

3.3.3. Procedure calls

We first give a Hoare rule for simple, possibly recursive procedures as long as they are not mutually recursive. The context C contains the other procedures that can be called. The context rule proves that all Hoare triples in the context are correct, which is by definition of the extended Hoare triple.

$$\frac{(P, c, Q) \in C}{\forall n, C \models_n \{P\}c\{Q\}}$$

Theorem hoare_context_n : $\forall$ C P c Q n,
  In (P,c,Q) C $\rightarrow$
  C $\models_{n}$ { P } c { Q }.

To justify a call to the procedure x we first assume that the recursive procedure call, if there is any, satisfies the Hoare triple $\{P\}$**call** x$\{Q\}$. We then require that the body of x satisfies that Hoare triple under the assumption that the recursive call satisfies the Hoare triple.

$$\frac{\forall n', (P, \textbf{call}\ x, Q) :: C \models_{n'} \{P\}\text{procs } x\{Q\}}{\forall n, C \models_{n} \{P\}\textbf{call}\ x\{Q\}}$$

To prove the Hoare rule for procedures, we need a number of lemmas. A complete list can be found in Appendix B.2.3.

Theorem hoare_call_simple_n : $\forall$ C P x Q,
  ($\forall$ n′, (P, **call** x, Q) :: C $\models_{n'}$ { P } Args.procs x { Q }) $\rightarrow$
  ($\forall$ n, C $\models_{n}$ { P } **call** x { Q }).

Mutually recursive procedure calls

Several mutually recursive procedures are justified using a Hoare judgement. The judgement reasons over two contexts, C1 and C2. The Hoare triples in C1 are assumed to hold, whereas we want to prove the Hoare triples in C2.

To justify that a judgement C2 holds under C1, two preconditions are required: Firstly, C2 only contains **call** x statements. Secondly, each Hoare triple for a procedure x in C2 is justified under the combined context of C1 and C2. This means that, like for the normal recursive procedures, we assume for recursive calls that the Hoare triple actually holds.

$$\frac{\forall n'\ P\ Q\ x, \{P\}\textbf{call}\ x\{Q\} \in C2 \rightarrow C1 \cup C2 \models_{n'} \{P\}\text{procs } x\{Q\}}{\forall n, C1 \models_{n} C2}$$

Theorem hoare_call_n : $\forall$ C1 C2,
  ($\forall$ P c Q, In (P,c,Q) C2 $\rightarrow$ $\exists$ x, c = **call** x) $\rightarrow$
  ($\forall$ n′ (P Q : Assertion) x, In (P,CALL x,Q) C2 $\rightarrow$ C1 ++ C2 $\models_{n'}$ { P } Args.procs x { Q }) $\rightarrow$
  ($\forall$ n, C1 $\models_{n}$ C2).

3.3.4. Syntactic Hoare triples

As a final step we define syntactic Hoare triples and syntactic Hoare judgements. The syntactic rules are pieces of inductively defined syntax, which we prove to be sound with respect to the semantic triples. Because of the inductive definition, the rules are the only way to prove syntactic Hoare triples. Whereas semantic ones can be proven both, using the lemmas and directly with respect to the operational semantics. This distinction would play a very important role if one wanted to prove completeness [Nip02b]. The syntactic

Hoare triples are also the link between the call-depth-indexed semantic variant of the Hoare triples and the standard semantic Hoare triples. The rules are the same as for the semantic triples. We denote syntactic Hoare triples with C ⊢ {P}c{Q}.

```
Inductive syn_ext_triple : context → Assertion → com → Assertion → Prop :=
  | SSkip : ∀ C P, C ⊢ { P } CSkip { P }
  | SAsgn : ∀ C P V a, C ⊢ { assn_sub P V a } V := a { P }
...

  where "C ⊢ { P } c { Q }" := (syn_ext_triple C P c Q)

with syn_judgement : context → context → Prop :=
...
  where "C1 ⊢ C2" := (syn_judgement C1 C2).
```

We prove that, if a syntactic triple is valid, then the call-depth-indexed semantic triple is also valid.

```
Lemma hoare_soundness_n : ∀ C P c Q,
  C ⊢ { P } c { Q } →
  ∀ n, C ⊨ n { P } c { Q }.
```

This theorem states that the validity of the syntactic triples implies the validity of the Hoare rules that do not take the call-depth into account.

```
Theorem hoare_soundness : ∀ C P c Q,
  C ⊢ { P } c { Q } →
  C ⊨ { P } c { Q } .
```

Syntactic judgements are related to semantic judgements in the same way as the Hoare triples above.

```
Lemma hoare_jsoundness_n : ∀ C1 C2,
  C1 ⊢ C2 →
  ∀ n, C1 ⊨ n C2.
```

```
Theorem hoare_jsoundness : ∀ C1 C2,
  C1 ⊢ C2 → C1 ⊨ C2.
```

We have only proven the soundness of the Hoare rules above, however, we expect them to also be complete. Such a proof could probably follow the general proof structure of Nipkow's proof [Nip02b], but it was not necessary for the current work where we only prove the soundness of the transformation to Boogie.

3.4. Inverting Hoare rules

The Hoare rules above show how a valid Hoare triple for an expression can be derived from valid Hoare triples of the subexpressions. We now want to invert the Hoare rules and derive valid Hoare triples for the subexpressions from a valid Hoare triple of an expression.

3.4.1. Weakest precondition

To do that we first define a weakest precondition function and prove certain facts about it. The weakest precondition function WeakestPre takes a command and a postcondition and generates the weakest possible precondition. It is based on the definition by Nipkow [Nip02b]. This precondition is not in first-order logic because it uses a quantification over states, which are themselves functions. Essentially the weakest precondition states that for every state the postcondition has to hold after executing the command c.

```
Definition WeakestPre (c : com) (Q : Assertion) : Assertion :=
  fun z st ⇒ forall_bool_prop (fun st′ ⇒ c / st ⇝ st′ → check_post Q z st′).
```

We prove that this weakest precondition is indeed a precondition as we would expect.

```
Lemma WeakestPre_pre : ∀ c Q, { WeakestPre c Q } c { Q }.
```

This lemma proves that the weakest precondition is indeed the *weakest* precondition.

```
Lemma WeakestPre_weakest : ∀ c P Q,
  { P } c { Q } →
  ∀ z st, P z st = true →
  WeakestPre c Q z st = true.
```

The counterpart of the weakest precondition is the strongest postcondition. Even though it would be useful to have such a function, it is not possible to define one in our setting. The reason is that commands can produce errors. In the case of an error, the check_post function always evaluates to false, regardless of the postcondition in the Hoare triple. So even with a constant-true postcondition it will be impossible to satisfy such a Hoare triple. For example, the command **assert** (fun st ⇒ false) will always evaluate to **CError** and no strongest postcondition can be generated.

3.4.2. Inversion functions

We can use the weakest precondition to invert Hoare rules. We show this in two cases.

The inversion of a sequence states that, if a Hoare triple involving a sequence is valid, it can be split into two individual Hoare triples that are both valid. The post-/precondition in the middle is generated by the weakest precondition function.

```
Lemma invert_seq : ∀ c1 c2 (P Q : Assertion),
  { P } c1;c2 { Q } → { P } c1 { WeakestPre c2 Q } ∧ { WeakestPre c2 Q } c2 { Q }.
```

The inversion of a conditional gives us a Hoare triple for each branch. This is very simple and does not require the weakest precondition function. The postcondition of each branch is identical to the postcondition of the conditional. The precondition for the then-branch includes additionally the fact that b is true, whereas for the else-branch ¬ b has to be true.

```
Lemma invert_if : ∀ b c1 c2 (P Q : Assertion),
  { P } if b then c1 else c2 { Q } →
  { fun z st ⇒ P z st ∧ bassn b z st } c1 { Q }
  ∧ { fun z st ⇒ P z st ∧ ¬ (bassn b z st) } c2 { Q }.
```

3.5. Verification condition generation

The verification condition generator closely resembles the way Boogie works. Boogie starts at the end of a function and calculates a precondition for each command. At the beginning of the function Boogie tries to prove that the generated precondition is weaker than the precondition annotated for the function.

```
Fixpoint VCgen (C:context) (c : com) (Q : Assertion) {struct c} : Assertion :=
  match c with
    | skip ⇒ Q
    | l := a1 ⇒
        assn_sub Q l a1
    | c1; c2 ⇒
        let P′ := VCgen C c2 Q in
        VCgen C c1 P′
    | if b then c1 else c2 ⇒
        let Pif := VCgen C c1 Q in
        let Pelse := VCgen C c2 Q in
        fun z st ⇒ (bassn b z st ∧ Pif z st) || (¬ (bassn b z st) ∧ Pelse z st)
    | while b inv P do c1 end ⇒
        fun _ st ⇒ P st ∧ forall_bool (fun z ⇒
            forall_bool (fun st′ ⇒ implb (P st′ ∧ ¬ (eeval_bool st′ b)) (Q z st′)
                ∧ (implb (P st′ ∧ eeval_bool st′ b) (VCgen C c1 (fun z ⇒ P) z st′))))
    | assert b ⇒
        fun z st ⇒ b st ∧ Q z st
    | i := pick e ⇒ fun z st ⇒
        forall_bool (fun v ⇒ implb (v_mem v (st e)) (assn_sub Q i (EValue v) z st))
    | call n ⇒ fun z st ⇒
        exists_bool_prop (fun P ⇒ In (P,CALL n,Q) C ∧ P z st=true)
    | backup x in c ⇒
        fun z st ⇒ VCgen C c (fun z′ st′ ⇒ Q z′ (env_update st x (st′ x))) z st
  end.
```

Unlike the weakest precondition function, the verification condition generator is is not guaranteed to return the *weakest* precondition. A special case is the while loop, which requires an additional loop invariant P to be specified. Therefore the verification condition generation is not complete, because a wrong loop invariant may be provided. The same is true for procedures calls: the user needs to annotate proper triples for all procedures.

We prove that the precondition VCgen generates is indeed a precondition. In this theorem we reason over the call-depth-indexed semantic Hoare triples, because our lemmas for the Hoare rules are only specified using the call-depth-indexed semantics.

```
Theorem VCgen_sound_n : ∀ C c Q n,
  C ⊨ₙ { VCgen C c Q } c { Q }.
```

To prove this theorem the Hoare lemmas from Section 3.3.1 were needed for each case. As a corollary we prove that VCgen is also sound with respect to the normal semantic Hoare triples, which follows directly from the above theorem.

```
Corollary VCgen_sound : ∀ C c Q,
  C ⊨ { VCgen C c Q } c { Q }.
```

Lastly, we prove that, if the generated verification condition is valid, then the result of the evaluation of c will be not be an error.

```
Corollary VCgen_no_error : ∀ c Q C,
  sem_context C →
  valid_formula (VCgen C c Q) →
  no_error c.
```

Furthermore, we define a verification condition generation for procedures, which does two things. For once it ensures that the context contains only **call** commands and it ensures that all triples in the context have a stronger precondition than the one the verification condition generator generates. This definition relies heavily on shallow embedding.

```
Definition VCgen_procs (C:context) : Prop :=
  (∀ P c Q, In (P,c,Q) C → ∃ x, c = call x) ∧
  ∀ n, ∀ P, ∀ Q, In (P, call n, Q) C →
  P ⟶ (VCgen C (Args.procs n) Q).
```

Soundness of the VCgen_procs function is defined by stating that, if VCgen_procs generates a valid formula for the context C, then C is indeed a valid context.

```
Lemma VCgen_procs_sound : ∀ C,
  valid (VCgen_procs C) →
  sem_context C.
```

Having modelled the verification condition generator we have a complete model of Boogie. We will use this model to show that our translation is indeed sound with respect to our model of Boogie.

4. Translation

In this chapter we will outline the translation of code from DMinor to Boogie, give a number of examples to illustrate it and prove the translation sound.

4.1. Translation of type-tests

Before introducing the actual translation we introduce the translation of type-tests. For checking if a variable x is in a specific type we use the F function that is defined by Bierman et al. [BGHL10] and formalised in Coq. The translateT function takes a type and the variable that should be tested, and it returns an assertion (a function from Env to bool).

```
Definition translateT (t:MType) (x:string) : Imp.Assertion :=
  fun env ⇒ F t (env x) env.
```

F has the property that it always returns a boolean value, which is a problem because we expect our translated code to fail if there is a typing error in a type. The only type that can produce a tying error during evaluation is a refinement type containing a type-test. We use the W function to check in advance whether the expressions in a type cause errors or not. W returns false if the type test does not cause an error (see Section 2.2.4 for a formal statement of this).

```
Definition translateT_err (t:MType) (x:string) : Imp.Assertion :=
  fun env ⇒ ¬ (W t (env x) env).
```

The translateT_err function has the same signature as translateT. Our translation adds an assertion before every type-test, verifying that translateT_err holds.

4.2. Translation

Below we introduce the actual translation function translate. The function takes, apart from the expression we want to translate, an avoid list and the output variable. The avoid list contains variable names that are not to be chosen when we generate fresh temporary variables. Names of binders are added to the avoid list that is passed to recursive calls of the translate function. The output variable specifies the variable name where the result should be written eventually. The translation generates code in such a way that after executing the generated code, the specified variable indeed contains the result value.

```
Fixpoint translate avoid e outvar {struct e} :=
  let avoid_o := outvar :: avoid in
  let avoid := outvar :: fv_exp e ++ avoid in
  backup outvar in
  match e with
     | Var x ⇒ CAss outvar (EVar x)
     | Value v ⇒ CAss outvar (EValue v)
     | UnOp o e ⇒
          let e′ := proj1_sig (fresh (avoid)) in
          translate (e′::avoid) e e′;
          (assert (translateT (fst (op_type_un o)) e′));
          match o with
          | ONot ⇒ CAss outvar (ENot (EVar e′))
          end
     | BinOp o e1 e2 ⇒
          let e1′ := proj1_sig (fresh (avoid)) in
          let e2′ := proj1_sig (fresh (e1′::avoid)) in
          translate (e1′::e2'::avoid) e1 e1′;
          translate (e1′::e2'::avoid) e2 e2′;
          (assert (translateT (fst3 (op_type_bi o)) e1′));
          (assert (translateT (snd3 (op_type_bi o)) e2′));
          match o with
          | OEq ⇒ CAss outvar (EEq (EVar e1′) (EVar e2′))
          | OLt ⇒ CAss outvar (ELt (EVar e1′) (EVar e2′))
          | OGt ⇒ CAss outvar (EGt (EVar e1′) (EVar e2′))
          | OAnd ⇒ CAss outvar (EAnd (EVar e1′) (EVar e2′))
          | OOr ⇒ CAss outvar (EOr (EVar e1′) (EVar e2′))
          | OPlus ⇒ CAss outvar (EPlus (EVar e1′) (EVar e2′))
          | OMinus ⇒ CAss outvar (EMinus (EVar e1′) (EVar e2′))
          | OTimes ⇒ CAss outvar (ETimes (EVar e1′) (EVar e2′))
          end
     | If e1 e2 e3 ⇒
          let e1′ := proj1_sig (fresh (avoid)) in
          translate (e1′::avoid) e1 e1′;
          CAssert (translateT Logical e1′);
          if e1′ then translate (e1′::avoid) e2 outvar
          else translate (e1′::avoid) e3 outvar
     | Let x e1 e2 ⇒ translate avoid e1 x; translate avoid e2 outvar
     | In e t ⇒
          let e′ := proj1_sig (fresh (avoid)) in
          translate (e′::avoid) e e′;
          (assert (translateT_err t e′));
          outvar := EIn (translateT t e′)
     | Entity el ⇒
```

```
          let ent′ := proj1_sig(fresh (avoid)) in
          let temp′ := proj1_sig (fresh (ent′::avoid)) in
          ent′ := EValue v_eempty;
          (fix les_to_entity el :=
          match el with
          | nil ⇒ skip
          | (l, e) :: el′ ⇒
             (translate (ent′::temp'::fv_exp e ++ avoid_o) e temp′;
             ent′ := EEntUpd l (EVar temp′) (EVar ent′);
             les_to_entity el′)
          end) el;
          outvar := EVar ent′
      | Dot e l ⇒
           let e′ := proj1_sig (fresh (avoid)) in
           (translate (e′::avoid) e e′; assert (translateT (Entity l Any)) e′);
           CAss outvar (EDot (EVar e′) l)
      | Add e1 e2 ⇒
           let e1′ := proj1_sig (fresh (avoid)) in
           let e2′ := proj1_sig (fresh (e1′::avoid)) in
           translate (e1′::e2'::avoid) e1 e1′; translate (e1′::e2'::avoid) e2 e2′;
           (assert (translateT (Coll Any)) e2′);
           CAss outvar (ECollAdd (EVar e2′) (EVar e1′))
      | Acc x e1 y e2 e3 ⇒
           let e1′ := proj1_sig (fresh (x::y::avoid)) in
           translate (x::y::e1'::avoid) e1 e1′;
           (assert (translateT (Coll Any)) e1′);
           translate (x::y::e1'::avoid) e2 y;
           while (ENot (ECollEmpty (EVar e1′))) do
              CPick x e1′;
              e1′ := ECollRem (EVar e1′) (EVar x);
              translate (x::y::e1'::avoid) e3 y
           end;
           outvar := (EVar y)
      | App f e ⇒
          translate (call_arg::avoid) e call_arg;
          (call f);
          outvar := EVar call_ret
    end.
```

A number of points should be noted about the translate function. Firstly, there is a **backup** at the beginning of every translation, which effectively results in a **backup** at every step of the translated program. While this is not always necessary, it gives us the property that after execution of a translated expression only the outvar variable changed.

All subexpressions of the translated expression are first translated and the values assigned to temporary variables. These variables are used in the translation of the actual expression. The temporary variables are chosen to be fresh, so they are neither free variables of the expression we intend to translate nor the output variable where the result is written to.

The cases for variables and values are obvious. For the unary and binary operators we first translate the arguments. Then we use an **assert** to verify that the arguments have the required type. Similarly, for the conditional the guard e1 is required to be of type Logical. Since all these type-assertions test for primitive types, such as Logical or Integer, we do not need to use translateT_err.

We translate **In** by asserting that a type-test, if expression e is in type t, does not produce a typing error. That assertion is generated using translateT_err. The function translateT is then used in the EIn construct to check the type during runtime and return a Logical.

Entities are created using a loop in Coq, meaning that after the translation the entity creation is completely unrolled. We start by creating an empty entity and then adding fields one by one.

Acc is translated using a while loop in Boogie. First the collection to accumulate over, e1, and the initial value e2 of the accumulator are translated and assigned to the variables x and y respectively. The while loop continues as long as there are elements in the collection. In each iteration one element is picked non-deterministically from the collection and then the body of the accumulate, e3, is applied to that element.

For a procedure call the argument is translated and put into the call_arg variable as specified by the calling convention we defined in Section 3.2.4. The result is then found in call_ret after procedure execution. We do not need an additional **backup** because the translation already adds a **backup** for every translated expression. The translation of the procedure definitions requires that we initialise the procedure list we abstracted as a module argument in Section 3.2.4.

```
Module Type TranslateArgs.
  Parameter ZType : Type.
  Parameter ZType_inhabited : ZType.
End TranslateArgs.
Module TranslateM (Args : TranslateArgs).
```

When instantiating the VCgen module we need to pass a set of procedures. This set is passed all the way to the Imp module. We instantiate this by making every procedure a translation of the DMinor function with the same name. Functions in DMinor are accessible using the functions Coq-function and have an argument name specified while our calling convention mandates the argument to be called call_arg. We therefore copy the argument to the specified variable name and then translate the function body while ensuring that the result is put into call_ret.

```
Module HoareArgs.
  Definition procs s :=
    let (arg,ex) := functions s in
    arg := EVar call_arg; translate nil ex call_ret.
  Definition ZType := Args.ZType.
  Definition ZType_inhabited := Args.ZType_inhabited.
End HoareArgs.
Module VCgenSpecificArgs := VCgenM HoareArgs.
End TranslateM.
```

4.3. Examples

To illustrate this translation process we use the same examples as in Section 2.3 and give their translation. To increase readability we leave out the **backup** command at the beginning of each translation step. We also manually give names to fresh variables, which would otherwise be defined in terms of long avoid lists.

4.3.1. Accumulate example

```
Definition acc_sample coll :=
  (Acc x (Value coll) y (Value empty_coll)
    (If (BinOp OEq (Var x) (Value v_null))
      (Var y)
      (Add (Var x) (Var y))
    )).
```

This is the translation to Boogie of the above program.

```
Definition acc_translated coll :=
  "fresh1" := EValue coll;
  (assert translateT (Coll Any) "fresh1");
  y := EValue empty_coll;
  (while ENot (ECollEmpty (EVar "fresh1")) do
    (x := pick "fresh1");
    "fresh1" := ECollRem (EVar "fresh1") (EVar x);
    ("fresh2" := EVar x;
    "fresh3" := EValue v_null;
    (assert translateT Any "fresh2");
    (assert translateT Any "fresh3");
    "fresh4" := EEq (EVar "fresh2") (EVar "fresh3"));
    (assert translateT Logical "fresh4");
    if "fresh4" then y := EVar y
      else "fresh5" := EVar x;
      "fresh6" := EVar y;
      (assert translateT (Coll Any) "fresh6");
      y := ECollAdd (EVar "fresh5") (EVar "fresh6")
  end);
  "outx" := EVar y.
```

Typical for our translations is that every subexpression is first assigned to a fresh variable before it is used, even if that means just copying a variable in the case that the subexpression is **Var** x.

This example also shows all the **assert** commands our translation adds, which make sure operands are of the right type. Before iterating over it, the **assert** checks if "fresh1" is indeed a collection. Sometimes the translation also adds useless assertions, such as for "fresh2" and "fresh3". They are useless because the equality operator requires no specific types for its arguments. These type-tests will always succeed as every value is in Any.

4.3.2. Example of an execution error

```
Definition in_sample :=
  In (Value v_tt) (Refine v Any (BinOp OGt (Var v) (Value (v_int 5)))).
```

This is the translation to Boogie of the above program.

```
Definition in_translated :=
  "fresh" := EValue v_tt;
  (assert translateT_err
    (Refine v Any
      (BinOp OGt (Var v) (Value (v_int 5)))) "fresh");
  "outx" := EIn
    (translateT
      (Refine v Any (BinOp OGt (Var v) (Value (v_int 5))))
         "fresh").
```

Our translation adds an **assert** before using EIn to ensure the expression in the refinement type is well-typed. This is required for our translation to be sound because a type-test using translateT will always return a value (see Section 4.1). The added assertion fails because translateT_err will return false for this input, precisely for the same reason as the DMinor program above fails: Because the greater operator does not accept arguments of type Logical.

We prove that in_translated indeed evaluates to an error.

Lemma in_translated_err : in_translated / env_empty $\rightsquigarrow$ **CError**.

4.3.3. Incompleteness of the type system

We recall the incompleteness example and will demonstrate here that its translation behaves according to the operational semantics.

```
Definition incom_sample :=
  In (Value (v_int 5)) (Refine x Any (BinOp OGt (Var x) (Value (v_int 5)))).
Definition incom_translated :=
  "fresh" := EValue (v_int 5);
  (assert translateT_err
    (Refine x Any
      (BinOp OGt (Var x) (Value (v_int 5)))) "fresh");
  "outx" := EIn
    (translateT
      (Refine x Any (BinOp OGt (Var x) (Value (v_int 5))))
        "fresh").
```

The translation is identical to the example above except for the fact that "fresh" is now 5. We prove that the translated program satisfies a Hoare triple.

```
Lemma incom_translated_no_error :
  { fun z st ⇒ true } incom_translated { fun z st ⇒ translateT Logical "outx" st }.
```

4.4. Soundness proof

The soundness proof is the heart of this thesis; we relate the DMinor big-step semantics to the Boogie big-step semantics.

We actually prove two things: Firstly, if the evaluation of the DMinor program raises an error, our translated program will also evaluate to an error in Boogie. This is done by adding enough assertions during the translation so that Boogie will fail for every incorrect program. Secondly, we prove that, if the evaluation of the DMinor program produces a value v, then the translation of this program will store v in the output variable when executed. These two things have to be proved together by mutual induction.

We have three additional preconditions: The expression we want to translate must not contain impure refinements, none of the functions contains impure refinements and the only free variable a function may have is the argument. An impure refinement is where an impure expression (see Section 2.2.2) is used in a refinement type.

We use subst_state to make the original DMinor expression closed before evaluating it. This is explained in more detail in Section 4.4.2.

```
Theorem translation_sound : ∀ ex r outx avoid st,
  (∀ s arg exp, (arg,exp) = functions s →
        contains_impure_refinements exp = false) →
  (∀ s arg exp, (arg,exp) = functions s →
        fv_exp exp = nil ∨ fv_exp exp = (arg::nil)) →
  Eval (subst_state ex st) r →
  contains_impure_refinements (subst_state ex st) = false →
  (r = Error → (translate avoid ex outx) / st ⇝ CError)
    ∧
  (∀ v, r = (Return v) →
     ∃ st′, (translate avoid ex outx) / st ⇝ CReturn st′ ∧ st′ outx = v).
```

4.4.1. Intuition of the proof

We prove this theorem by induction on the big-step semantics, which gives us 42 cases. In each case we have to prove that the translated code evaluates to the same result as the big-step semantics in DMinor (see Appendix A). On the Boogie side we use the big-step semantics we defined in Section 3.2.5.

The first step in the proof is always to remove the **backup** command that is added by every translation. For this we use two lemmas, one for the error case and one for the case a resulting state is returned. In the case of a resulting state only the output variable outx changed after executing the **backup** command.

```
Lemma backup_err : ∀ c st outx,
   (backup outx in c) / st ⇝ CError ↔
   c / st ⇝ CError.
```

```
Lemma backup_ret : ∀ c st st′ outx v,
  c / st ⇝ CReturn st′ →
  st′ outx = v →
  (backup outx in c) / st ⇝ CReturn (env_update st outx v).
```

Then, depending on the case, we have to either prove that the evaluation succeeds with a **CReturn** (v), where v must be the same value as DMinor returns, or that a **CError** is returned.

The value case

In this case, a constant is returned. In the DMinor semantics it is formalised as follows:

```
| eval_value : ∀ v,
  Eval (Value v) (Return v)
```

The generated proof obligation is:

```
∃ st′ : Env,
   (backup outx in outx := EValue v) / st ⇝ CReturn st′ ∧ st′ outx = v
```

Without diving into technical details, it becomes clear that st′ outx will indeed contain v.

Failing operand

The above example showed a case where the evaluation succeeds and returns a value. Before we move on to more complicated cases, we show a case that evaluates to **Error**.

```
eval_un_op_1 : ∀ e op,
   Eval e Error →
   Eval (UnOp op e) Error
```

This case evaluates to **Error** because the operand e evaluates to **Error**, so it is a simple bubbling up of errors. This is the proof obligation Coq gives us, where the **backup** is already removed:

```
(translate avoid′ e′ a;
   (assert translateT (fst (op_type_un u)) a);
   match u with
   | ONot ⇒ outx := ENot (EVar a)
   end) / st ⇝ CError
```

This is again easy to prove, because the induction hypothesis gives us the fact that the translation of e′ evaluates to a **CError**: translate avoid′ e′ a / st ⇝ **CError**

A number of cases can be proven simply by the induction hypothesis. However, some cases were very complicated to prove. The two most difficult cases were entity creation and accumulation.

Entity creation

Entity creation consists of two cases. In the first case, one subexpression from the expression-list evaluates to an **Error**, which causes the whole expression to evaluate to an **Error**.

```
| eval_entity_1 : ∀ les1 les2 lj ej,
   (∀ l e, In (l,e) les1 → ∃ v, Eval e (Return v)) →
   Eval ej Error →
   Eval (Entity (les1 ++ (lj,ej) :: les2)) Error
```

The intuition is that les1 ++ (lj,ej) :: les2 is the list of expressions, where the expressions in les1 evaluate successfully, the expression ej evaluates to an error and the expressions in les2 could do anything, even diverge.

Our translation translates this case using a loop in Coq that adds elements to the entity one-by-one. This case is proven by induction on les1, the expressions we know to succeed. The fact that the elements in les1 evaluate to some value is important for us to prove that the evaluation of the translated expressions form les1 does not diverge.

The challenge was that we needed to strengthen the induction hypothesis before inducting on the list les1. The strengthened statement has two additional hypotheses: The first is that the entity, which is being constructed in the temporary variable a is of type Entity and secondly, that except for the two temporary variables a and b the state stays the same for each loop iteration. These are in a sense loop invariants for the Coq loop.

```
assert (∀ st″, is_E (st″ a) = true →
  (∀ x : string, a ≠ x → b ≠ x → st x = st″ x) →
  (fix les_to_entity (les : list (string × Exp)) : com :=
  match les with
  | nil ⇒ skip
  | pair l e :: les′ ⇒
  translate (a :: b :: fv_exp e ++ outx :: avoid) e b;
  a := EEntUpd l (EVar b) (EVar a); les_to_entity les′
  end) (les1′ ++ (li, ei) :: les2′) / st″ ⇝ CError).
```

The case when the entity creation succeeds has the added difficulty that we need to prove that the resulting value of both the original and the translated expression is the same after evaluation. We use backwards induction on the list of expressions to prove this case. We defined and proved our own backwards induction principle for this purpose.

```
Lemma list_ind_rev : ∀ (A : Type) (P : list A → Prop),
  P nil →
  (∀ (a : A) (l : list A), P l → P (l ++ a :: nil)) →
  ∀ l : list A, P l.
```

A number of additional lemmas are needed for this case, e.g., lemmas that reason over removing elements from an entity. They are listed in Appendix B.5.1.

Accumulation

The accumulation case faces similar difficulties as the entity case above. Here is the DMinor evaluation rule:

```
| eval_accum_lets : ∀ x e1 y e2 e3 r v1 vs,
  Eval e1 (Return v1) →
  is_C v1 = true →
  (Permutation vs (vb_to_vs (out_C v1))) →
  Eval (Let y e2 (let_seq y (List.map (fun v ⇒ subst_exp e3 v x) vs) (Var y))) r →
  Eval (Acc x e1 y e2 e3) r
```

Similar to the entity case we also do an induction on the list of elements in the collection. Even though collections are encoded as lists in Coq, a collection has no inherent order, which is expressed by the Permutation hypothesis above. We translate an accumulate into a while loop that repeatedly picks elements from the collection. The main difficulty was to relate the **Let** sequence to our **while** loop, which we did using two lemmas, one for the **while** loop (Lemma unroll_loop) and one for the **Let** sequence generated by let_seq (Lemma translate_let_seq_inv), which take an element from the collection and unroll the sequence or the loop by one. These lemmas and other additional lemmas can be found in Appendix B.5.2.

4.4.2. Closedness of the expression in the proof

In our proof we used subst_state because Eval is only defined for closed expressions and subst_state makes every expression closed. We can prove as a corollary that the above theorem also holds if we require ex to be closed instead of using subst_state. In the theorem above subst_state was required in order to strengthen the induction hypothesis, because in the induction case the expression could not always be guaranteed to be closed.

```
Corollary translation_closed_sound : ∀ ex r outx avoid st,
  fv_exp ex = nil →
  (∀ s arg exp, (arg,exp) = functions s →
       contains_impure_refinements exp = false) →
  (∀ s arg exp, (arg,exp) = functions s →
       fv_exp exp = nil ∨ fv_exp exp = (arg::nil)) →
  Eval ex r →
  contains_impure_refinements ex = false →
  (r = Error → (translate avoid ex outx) / st ⇝ CError)
    ∧
  (∀ v, r = (Return v) →
  ∃ st′, (translate avoid ex outx) / st ⇝ CReturn st′ ∧ st′ outx = v).
```

4.4.3. Relation with Hoare logics

As a second corollary we prove that, if a Hoare triple with a valid precondition can be established for the translated program, then the original DMinor program will not raise an error when it is evaluated.

```
Corollary soundness_hoare_plus_translation : ∀ avoid ex outx P Q,
  fv_exp ex = nil →
  (∀ s arg exp, (arg,exp) = functions s →
       contains_impure_refinements exp = false) →
  (∀ s arg exp, (arg,exp) = functions s →
       fv_exp exp = nil ∨ fv_exp exp = (arg::nil)) →
  contains_impure_refinements (ex) = false →
  nil ⊢ { P } translate avoid ex outx { Q } →
  valid_formula P →
  ¬ Eval ex Error.
```

This corollary has a very simple proof, as it follows directly from translation_closed_sound and hoare_triple_pre_valid_no_error from Section 3.3.1.

4.4.4. Relation with verification condition generation

We prove that, if the verification condition VCgen generates for the translated code is valid, then the original program will not evaluate to **Error**. This proof connects Boogie to DMinor, because VCgen models the behaviour of Boogie.

```
Corollary soundness_vcgen_plus_translation : ∀ avoid ex outx Q C,
  fv_exp ex = nil →
  (∀ s arg exp, (arg,exp) = functions s →
       contains_impure_refinements exp = false) →
  (∀ s arg exp, (arg,exp) = functions s →
       fv_exp exp = nil ∨ fv_exp exp = (arg::nil)) →
  contains_impure_refinements (ex) = false →
  valid (VCgen_procs C) →
  valid_formula (VCgen C (translate avoid ex outx) Q) →
  ¬ Eval ex Error.
```

The proof is very simple by using VCgen_no_error from Section 3.5 together with translation_closed_sound.

4.4.5. Relation to the type system

As we have shown in Section 4.3.3, the DMinor type system is incomplete with respect to the operational semantics. From that fact follows that, if a translated expression is valid in a Hoare triple with true as a precondition, it does not hold in general that the expression is well-typed.

```
Lemma counterexample : ¬ (∀ e env outx T avoid,
  fv_exp e = nil →
  env ⊢ T →
  ¬ In outx (dom env) →
  nil ⊨ { fun _ _ ⇒ true } translate (app avoid (dom env)) e outx
           { fun _ st ⇒ translateT T outx st } →
  env ⊢ e : T).
```

The proof works by using incom_sample to construct a counterexample. We have proven lemma incom_translated_no_error (see Section 4.3.3) that states that a valid Hoare triple with true as the precondition and "translateT Logical "outx" st" as a postcondition exists. We have also proven lemma incom_sample_type (see Section 2.3.3) that shows that incom_sample is not of type Logical.

If we wanted our translation to be sound with respect to the type system, we would have to change the translation to add more assertions to match the incompleteness of the DMinor type system, which would actually decrease precision and in practise also performance.

5. Implementation

Matching the theory we wrote an implementation, called DVerify, that transforms a DMinor program into a Boogie program. DVerify is written in F# 2.0 [Mar10] and consists of more than 1200 lines of code as well as a 700 line axiomatisation that defines the DMinor types and functions in Boogie. The main purpose of it is to serve as a proof-of-concept and to show that Boogie is indeed able to correctly verify the translation of a considerable number of samples.

But before introducing our axiomatisation and implementation we introduce the tools that match the theory given in Chapters 2 and 3.

5.1. DMinor type-checker

A prototype implementation for DMinor is written in F# and works using a bidirectional type-checking algorithm [PT98]. Bidirectional means that it uses type-synthesis to generate a type for any expression and type-checking to check if an expression has a certain type. These two algorithms are mutually recursive. Type-synthesis is similar to a strongest[1] postcondition algorithm in a verification setting.

As outlined in Chapter 2, DMinor uses semantic subtyping and the type-checking function calls Z3 to determine if the formula encoding a subtyping test is valid. Since Z3 knows nothing about the types and functions DMinor uses, Bierman et al. created an axiomatisation of those types and functions that is fed to Z3 along with the formula to be proven.

5.2. Boogie tool

The Boogie tool takes a Boogie program as input and outputs either an error message that describes points in the program where certain postconditions or assertions may not hold [LMS05] or otherwise prints a message indicating that the program has been verified successfully.

[1]It is not guaranteed that the type returned by the type-synthesis is indeed the strongest. In fact, some of the type-synthesis rules [BGHL10] seem to trade some “strongness” (i.e. theoretical completeness) for compositionality, efficiency and practical completeness.

Boogie supports uninterpreted function symbols, which we did not model in our formalisation. For an uninterpreted function symbol only the signature is given and the function's input/output behaviour is specified using axioms. For that reason Boogie code is not executable, however, it is not necessary to execute Boogie code because said code was usually generated from executable code in the first place. In our formalisation we abstracted function symbols away by adding the functions we require as primitives to the language, see Section 3.2.2.

5.3. Some implementation details

We will not go into great detail about DVerify here, but we will give a high-level overview. We use the DMinor implementation as a library so that we do not have to reimplement existing functionality. This is mainly the parser for DMinor files, the purity checking and a weak form of type-synthesis.

The heart of our translation consists of a recursive function that goes over a DMinor expression and translates it into Boogie code. This function is called once per DMinor function and produces a Boogie procedure. Types in DMinor are translated into Boogie function symbols returning a `bool`, using another recursive function in our implementation. For each translation of **In** we add an additional assert, just as in the theory (see Section 4.2).

Every time we translate a refinement type we use the purity checking function from DMinor and if we encounter an impure expression in a refinement we raise an exception during the translation. This means that in such cases, it is not Boogie that fails on the translated code, as usual, but it is the actual translation routine that fails. That is necessary because there is no way to check purity using Boogie.

The while loops produced by the translation of **Acc** are annotated with a type for the accumulator, so we use that type annotation as an invariant for our `while` loop. In the future we intend to infer such loop invariants automatically using the Boogie infrastructure for this task. The DMinor language as implemented by the type-checker allows for one more construct to define a loop, called Bind. In theory Bind can be encoded using **Acc**, but in the DMinor implementation it is considered a primitive in the interest of efficiency and to reduce the type annotation burden. Since Bind does not carry a type annotation, we have to find one during translation. For that we use type-synthesis from DMinor. However, we modified the original type-synthesis algorithm as used by DMinor so that it no longer calls the type-checking algorithm, and therefore never fails to synthesize a type for an expression. This also means that our solution never calls Z3 during type-synthesis, because Z3 is only needed to check subtyping.

5.4. Axiomatisation

A major part of the implementation is the axiomatisation of DMinor values and functions in Boogie. This becomes necessary because Boogie as such understands only two sorts, `bool` and `int`, whereas DMinor has a number of primitive and composite values as listed in Section 2.1.1. In the Coq formalisation we circumvented this problem by directly using values, thereby assuming DMinor values are native to Boogie as outlined in Section 3.2.2. Our axiomatisation is similar to the axiomatisation the DMinor type-checker feeds to Z3. The complete axiomatisation is given in Appendix C.

As an example we describe in detail the axiomatisation of the `General` sort.

5.4.1. The `General` sort

General is the sort of primitive values (i.e. values having a primitive DMinor type), listed in Section 2.1.1.

In the axiomatisation `General` is a sort. Since Boogie does not allow algebraic datatypes to be defined directly, like for instance in F#, we have to use function symbols that take a specific sort and output `General`. These function symbols do not have implementations because Boogie is not a language that is run, but rather these function symbols are passed directly to Z3.

```
type String;

type General;

// Constructos
function G_Integer(int) returns (General);
function G_Text(String) returns (General);
function G_Logical(bool) returns (General);
const G_Null : General;
```

The sort `String` needs some explaining: Boogie does not have any primitive representation for strings, so we chose the simplest possible representation for strings. Every string is a unique constant of sort `String`, two identical strings are translated to the same constant, simply by making the string the name of the constant. This is possible because DMinor has no operation on strings other than comparison. Here is an example of those string constants:

```
const unique str_Hallo : String; // the string "Hallo"
const unique str_foo : String;   // the string "foo"
```

As a next step we define a number of tags, which are used to identify what content a variable of sort `General` has. There is a function symbol `get_GTag` that returns a tag for a specific `General` variable. A tag is a unique constant of sort `GTag`, similar to an enumeration in imperative programming languages. There is no implementation for this function symbol, but a number of axioms that define the function symbol's behaviour: Depending on the function symbol used to create a certain variable of type `General` the corresponding tag is returned. The tester function symbols return true or false, depending on whether the variable was created with the corresponding function symbol.

```
// Tags
type GTag;
const unique GTag_Integer : GTag;
const unique GTag_Text : GTag;
const unique GTag_Logical : GTag;
const unique GTag_Null : GTag;

function get_GTag(General) returns (GTag);
axiom (forall i : int :: { get_GTag(G_Integer(i)) }
  get_GTag(G_Integer(i)) == GTag_Integer);
axiom (forall s : String :: { get_GTag(G_Text(s))}
  get_GTag(G_Text(s)) == GTag_Text);
axiom (forall b : bool :: { get_GTag(G_Logical(b)) }
  get_GTag(G_Logical(b)) == GTag_Logical);
axiom (get_GTag(G_Null) == GTag_Null);

// Testers
function is_Integer(g : General) returns (bool) { get_GTag(g) ==
    GTag_Integer}
function is_Text(g : General) returns (bool) { get_GTag(g) ==
    GTag_Text}
function is_Logical(g : General) returns (bool) { get_GTag(g) ==
    GTag_Logical}
function is_Null(g : General) returns (bool) { get_GTag(g) ==
    GTag_Null}
```

The last set of function symbols concerning `General` are the out-function symbols. They return the primitive that a `General` variable contains (`int`, `bool`, or `String`).

```
// Accessors
function of_G_Integer(General) returns (int);
axiom (forall i : int :: of_G_Integer(G_Integer(i)) == i);
function of_G_Text(General) returns (String);
axiom (forall s : String :: of_G_Text(G_Text(s)) == s);
function of_G_Logical(General) returns(bool);
axiom (forall b : bool :: of_G_Logical(G_Logical(b)) == b);
```

The return value of each of these function symbols is only defined if the `General` passed to it contains a value of that type. Since function symbols in first-order logic are total, in other cases anything can be returned. So if we call for example `of_G_Logical(G_Integer(5))` either `true` or `false` is returned, but there is no assumption which one. This is a major difference to Coq where a default value needs to be explicitly given in that case.

5.4.2. The Value sort

The Value sort is built in the same way as General. To keep this section concise we will only display the function symbols that construct a Value.

```
type Value;
type VList;
type VOption;

// Constructors (values)
function G(General) returns (Value);
function E([String]VOption) returns (Value);
function C([Value]int) returns (Value);
function L(VList) returns (Value);

// Lists
const Nil : VList;
function Cons(Value,VList) returns (VList);

// Options used for enties
const NoValue : VOption;
function SomeValue(Value) returns (VOption);
```

General requires no more explanation. Entities are maps from String to VOption. In Boogie a map means a finite support function that in this case gives a VOption for every possible String. The empty Entity is a map where all strings are mapped to NoValue. During entity construction some strings are then updated so that they map to SomeValue. Collections (multi-sets) are also represented using maps, this time from Value to int. So for every possible Value the map stores the number of occurrences in the multi-set. Initially this is 0 for all values and is then updated to any positive number.

Lists are not primitive in the theory (see Section 2.1.1), but are present in the implementation of DMinor. Even though they can be encoded [BGHL10], they are implemented as a primitive for performance reasons. In Boogie we represent them as using Nil and Cons constructors as usual for lists.

5.4.3. Axiomatisation of DMinor operations

DMinor offers a number of expressions that perform operations on values as outlined in Section 2.1.2. These built-in expressions simply become function symbols in Boogie. Since Boogie distinguishes between function symbols and procedures we define both a function symbol and a procedure for those DMinor expressions. The reason for this dual axiomatisation is that procedures are used when these expressions appear inside DMinor functions, whereas the function symbols are used when expressions are used inside refinement types.

We translate DMinor types as function symbols, and those function symbols cannot call procedures. For that reason we need to have an axiomatisation of DMinor expressions as both, function symbols and procedures.

Procedures

In Boogie procedures consist of a body and a set of pre- and postconditions. Boogie checks once if the procedure body conforms to these pre- and postconditions and then the procedure becomes opaque. That means that when the procedure is called somewhere, the body of the procedure is ignored. Boogie merely checks if the procedure's preconditions are satisfied and then assumes the postconditions. So it is essential that the postconditions are strong enough to reason about the procedure's functionality.

Function symbols

Function symbols are interpreted (in first-order logic models) by total mathematical functions and are specified with number of axioms. A function symbol can be underspecified; in that case it is unknown what value a function will return for certain inputs. The other way to define a function symbol's behaviour in Boogie is by giving a body to the function, but that is only syntactic sugar for an axiom. Functions can also be called in asserts, pre- and postconditions, which is not true for procedures.

Example: addition

As a very simple demonstration of the relation of functions and procedures, we show here the addition operator that takes two integers and adds them.

```
function O_Sum(v1:Value,v2:Value) returns (v:Value);
axiom (forall v1 : Value, v2 : Value :: { O_Sum(v1,v2) }
  Integer(v1) && Integer(v2) ==>
    O_Sum(v1,v2) ==
      v_Int(of_G_Integer(of_V_General(v1))
          + of_G_Integer(of_V_General(v2))));

procedure pO_Sum(v1:Value,v2:Value) returns (v:Value)
  requires Integer(v1) && Integer(v2);
  ensures Integer(v) && v == O_Sum(v1,v2);
{
  v := v_Int(of_G_Integer(of_V_General(v1))
           + of_G_Integer(of_V_General(v2)));
}
```

The body of the function and that of the procedure are very similar and are mainly unboxing the integers, adding them and boxing the result. The procedure makes sure through the preconditions that the arguments are indeed integers. In the postcondition the procedure relates to the function symbol to enable a caller to reason about the procedure's result. Because of the implication, the function symbol is defined only for the case that the operands are integers. For the case that they are not, no assumption can be made what value is returned, the value must not even be an integer at all. We can express that fact with this assertion, which indeed fails:

```
assert (Integer(O_Sum(v_true, v_false)));
```

This is different from our theory, where we model operators using error-tracking so that, in case the arguments are not integers an error is returned. So in theory `O_Sum` should be defined to return a Result instead of a Value.

```
| EPlus a b ⇒ LBind (eeval st a) (fun x1 ⇒
        (LBind (eeval st b) (fun x2 ⇒
        if In_Integer x1 ∧ In_Integer x2 then Return (O_Sum x1 x2) else Error)))
```

When we began developing early prototypes of DVerify we axiomatised operators and other functions as fully specified function symbols; in case of addition a function symbol that always returns an integer. This would render our translation unsound, because if an integer was added to a logical in a type-test, said type-test would be verified successfully, even though according to the operational semantics it would return an error when executed.

It would have been possible to stay faithful to the theory in our axiomatisation and explicitly track errors, but it would have required widespread changes and we took the pragmatic decision to go with the easier solution of underspecifying functions.

The intuition is that by underspecifying them, these function symbols work in a similar way to their theoretical counterparts that bubble up errors: If one of the operands cannot be proven to have a specific type, then the result will also not have any specific type. We know that function symbols are only used in type-tests and refinement types; in all other cases procedures are used. In those two cases the result of the expression must be either true or false, which we check in our translated code. If deep inside the refinement type there is a logical fed to an addition operator, then for the whole expression it cannot be proven that the result is true or false and therefore the program is not verified by Boogie.

While we have some intuition on this solution, there is no formal proof that would guarantee this is sound in all cases. However, all examples we tried so far worked fine with this solution.

5.5. Quantitative comparison of DMinor and DVerify

We will give an overview here how our implementation compares to the DMinor typechecker in terms of precision, efficiency and the predictability of verification time. We test against DMinor 0.1.1 from September 2010.

5.5.1. Test suite and precision

Microsoft Research gave us access to their DMinor test suite that contains 109 sample DMinor programs, plus some additional tests for the parser and interpreter. We do not consider parser and interpreter tests because we use the parser from DMinor and the interpreter is not necessary for static program verification. Out of these 109 tests 33 should fail type-checking because they are ill-typed DMinor programs and 76 should successfully type-check. Out of these 76 DMinor cannot verify 10 tests that are all examples of incompleteness of the type-checker.

	Type-checking success	Type-checking failure	Total
Original test suite	76	33	109
DMinor passes	66	33	99
DVerify passes	62	31	93

We wrote a test program in the form of a Visual Studio Test Project that automatically runs the test suite through DMinor and then through DVerify.

From 66 cases on which DMinor succeeds, DVerify manages to verify 62 as correct. Out of the 33 that fail in DMinor, DVerify fails on 31. The other two are correct operationally, but have typing errors. That means overall that DVerify succeeds on 94% of the cases DMinor succeeds on and is able to verify two correct programs DMinor cannot verify.

For the 4 cases Boogie cannot verify the most common problem is that a too complicated loop invariant is created by the type-synthesis. This invariant is not necessarily wrong, but together with our axiomatisation Z3 fails to prove it.

5.5.2. Verification times

We modified the test program to also take time measurements. For each component two measurements were taken: Firstly, we measured the overall wall-clock time that is needed by the two tools, which includes the time the operating system requires to start the process. Secondly, we measured the time excluding initialisation and parsing, which we call “internal time”. Since we are dealing with a large number of small files and both tools are managed (.NET) assemblies, initialisation is a major factor. To get the internal time we added code to each component that measures and outputs that time. This was possible because we had access to the source code of all components.

We measure timings for those 66 examples that are successfully verified by both tools and present the statistics below. The measurements were taken on a 2.1 GHz laptop with 4GB of RAM running the Windows 7 x64 operating system.

	Overall time				Internal time			
	Min	Avg[1]	Max	SD[2]	Min	Avg[1]	Max	SD[2]
DMinor	0,69s	1,13s	4,62s	0,50s	0,09s	0,50s	3,97s	0,49s
DVerify	1,56s	1,81s	2,97s	0,25s	0,30s	0,49s	1,35s	0,17s
Translation only	0,80s	0,97s	1,75s	0,13s	0,12s	0,26s	0,42s	0,08s
Boogie only	0,76s	0,85s	1,71s	0,16s	0,16s	0,23s	0,94s	0,11s

[1] Arithmetic mean
[2] Standard deviation

DVerify takes 60% longer on average than DMinor. However, the times without parsing and initialisation are the same on average. DVerify takes 60% longer, mostly because we have twice the initialisation overhead, once for the translation and once for Boogie. Initialisation and parsing take particularly long for Boogie. We also list the times DVerify spends translating separately from the time taken by Boogie. The standard deviation of our translation and Boogie is much lower, which indicates a better predictability, meaning that irrespective of the example the running time is almost the same.

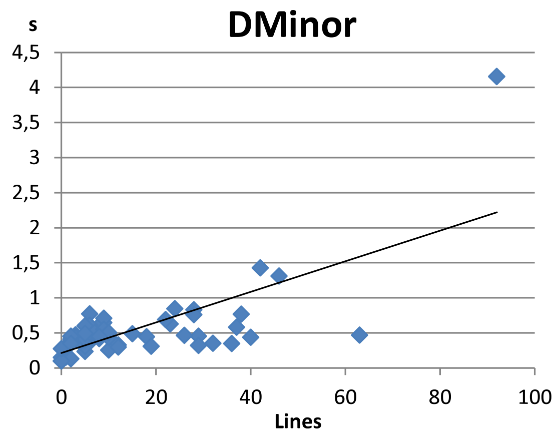
Included in above results is a simple optimisation we did that sped Boogie up by 40%: In the original setup the whole axiomatisation was passed along with the translated file to Boogie. That meant that all the procedures in the axiomatisation had to be checked as well, every time a translated file was processed by Boogie. We removed this additional workload by removing all procedure bodies from the axiomatisation once boogie verified them, leaving only the signatures and the pre- and postconditions. Consistent usage of quantifier patterns [LM09] improved performance by another 10%. We did not just add patterns to the quantifiers in the library, but also to the quantifiers we generate during our translation, e.g. for loop invariants.

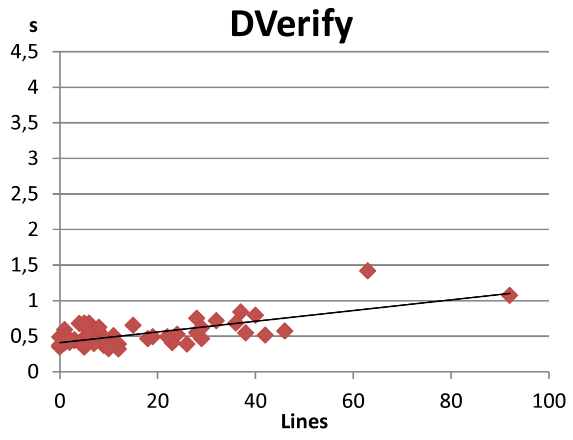
Another way to speed both tools up would be by using the Native Image Generator (NGen) that is part of the .NET Framework [Wil05]. .NET assemblies are compiled by default to the Common Intermediate Language (CIL) [ECM06] and then compiled to native code during runtime. This just-in-time compilation does not just have an impact on the startup time, but also on the internal time because a procedure is only compiled once it is called for the first time. While for long-running applications the additional time of the just-in-time compilation may be negligible, in our case it has a very big impact because a new instance of each program is started for every file we test. NGen performs an ahead-of-time compilation and thereby reduces start-up and running time significantly. We only tested this for DMinor and it resulted in a speed-up of 33%. This is not included in the above figures, which are measured using the normal .NET assemblies.

Another way to speed up things slightly would be to start Boogie after the translation in the same process. That would require one less process initialisation, which takes time because the Common Language Runtime has to be loaded into a every managed process. We did not experimentally try this solution.

Scalability

We also tried to test scalability by plotting the verification time over the program size. It seems that Boogie has a better scalability than DMinor, meaning that verification times of Boogie are nearly constant and do not change much when the complexity of the program increases. However, it is hard to draw definitive conclusions out of this experiment, since all our test programs are very small, the largest program being 92 lines.





The charts both show the time (in seconds) the verification takes per line of code and a linear trendline. For DMinor the time per line grows faster than for Boogie.

5.5.3. Example where DVerify is more precise

We showed in Section 4.3.3 that the type system is incomplete with respect to the operational semantics. This carries over from the theory to the implementations and we demonstrate here that for this example DVerify is more precise than DMinor. This is the example in DMinor syntax:

```
a() : Logical
{
  5 in Any where value > 5
}
```

DMinor rejects this program, because in the refinement type we use type Any and that cannot be applied to the greater operator. DVerify, on the other hand, checks if this refinement is correct with respect to the operational semantics, which it is, because the value in question (5) which is indeed an Integer. So Boogie accepts the translated program below.

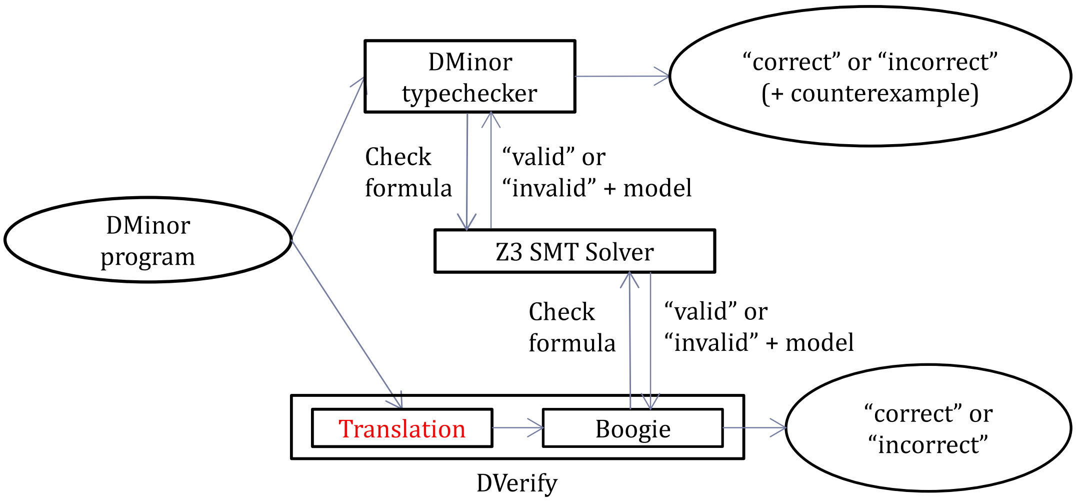
```
procedure a() returns (out: Value)
  ensures Logical(out);
{
  assert (O_GT(v_Int(5), v_Int(5)) == v_true ||
          O_GT(v_Int(5), v_Int(5)) == v_false) &&
          Any(v_Int(5));
  out := v_Logical(O_GT(v_Int(5), v_Int(5)) == v_true &&
         Any(v_Int(5)));
}
```

If we change the above example slightly so that we pass a Logical to the greater operator, both DMinor and Boogie will fail as expected.

```
b() : Logical
{
  true in Any where value > 5
}
```

5.6. Qualitative comparison of DMinor and DVerify

During our work we realised that even though type-checking and verification are usually considered different, in our setting, not only can we use both to achieve the same thing, but they also, from a highly abstract point of view, have similarities in how they work.



On the DMinor side the source code is directly fed to the DMinor type-checker, which then invokes Z3 for every subtyping check (see Section 5.1). On the DVerify side the code is first translated by our tool and then fed to Boogie, which generates one proof obligation for every postcondition or assert inside a procedure. Leino et al. propose a technique called verification condition splitting that would lead to a larger number of smaller proof obligations being discharged [LMSL08]. We tried enabling this feature in Boogie and it indeed decreased the maximum verification time, but had no impact on the average time. For this reason our implementation does not currently use verification condition splitting.

Area	**Our verification approach (DVerify)**	**Type-checking approach (DMinor)**
Verification cond. generation	Weakest precondition	Bidirectional type-checking (type synthesis ≈ strongest postcondition)
Formulae discharged	One per postcondition/ assertion (larger, but less obligations)	One per subtyping test (smaller, but more obligations)
Backend	Boogie + SMT-Solver (Z3)	SMT-Solver (Z3)
Loop invariants	In principle Boogie could infer some (even for accumulates)	For some constructs they are inferred (but not for accumulates)
Error reporting	Abstract trace	Counterexample
Performance (overall)	slightly worse (more initialization)	slightly better
Performance (internal)	similar	similar
Scalability	possibly better	possibly worse
Precision (practise)	similar	similar
Completeness (theory)	possibly better	possibly worse (type system)

Both DMinor and Boogie require an axiomatisation of the Value type and the build-in functions. In DMinor this axiomatisation is written in SMT syntax [RT06b] and directly fed to Z3 with the proof obligation. Our axiomatisation is in Boogie language and Boogie translates it to Simplify syntax [DNS05] and feeds it to Z3 along with the verification conditions it generates.

Regarding error reporting DMinor gives us counterexamples which are more useful in practise than the abstract traces Boogie gives us. We will discuss error reporting and possibilities to improve that in DVerify in Section 6.2.1.

Both use Z3 as an SMT solver, but DMinor calls Z3 repeatedly, whereas Boogie calls Z3 only once, but with a proof obligation for each procedure. Overall performance is better in DMinor by roughly 60%. However, when we excluded initialisation and parsing times both took on average the same time. The smaller standard deviation indicates that DVerify is more predictable. Regrading scalability results show that DVerify could be better, however, we only only tested this on a number of very small examples.

Regarding precision our results are mixed. Better precision means that less correct programs are rejected. In theory, DVerify could be better because it is close to the operational semantics, whereas type systems are inherently incomplete. However, we never proved completeness for our Hoare logic and VCgen is inherently incomplete for loops, which require an invariant annotation. Comparing the implementations, the result depends on the example, sometimes DMinor is more precise and sometimes DVerify. The fact that we tested

DVerify on a test suite that was written for DMinor biases the result as one could find more examples that are verified by DVerify, but not by DMinor. Finally, one should keep in mind that both are just meant to be prototypes.

6. Conclusion

6.1. Summary

This thesis investigated the relationship between a general-purpose verification tool and a type-checker. We started by introducing a subset of Boogie and the DMinor language in a theoretical context. We used Coq to formalise the subset of Boogie, a formalisation of DMinor was already available. Next, we introduced our translation and proved it sound with respect to the big-step semantics of the two languages. Lastly, we presented the prototype implementation of the translation and how our tool chain works compared to DMinor.

On the theory side we showed that our translation is able to identify runtime errors in DMinor programs statically. On the practical side we were not yet able to match the precision of DMinor for all examples, but we presented one example where DVerify is more precise than DMinor. Regarding efficiency we were able to match DMinor when not taking initialisation into account.

Our formal development consists of 5000 lines of Coq and our proofs are done in full detail. This is made on top of the DMinor formalisation consisting of 4000 lines [BGHL10], which makes the total size of the formalisation approach 9000 lines of Coq. The soundness proof alone consists of 1300 lines of Coq code and the Coq proof checker takes more than 2½ minutes to check the proof. Three custom Coq tactics were defined to simplify the proof. A list of how these 5000 lines are split between the nine files of our formalisation can be found in Appendix B.

The Coq formalisation and the DVerify source code are available under permissive licenses [Tar].

6.2. Future work

6.2.1. Implementation

Due to our focus on the formalisation of the theory in Coq, the current implementation should be considered only a proof-of-concept prototype. However, during the inception of this thesis we had a number of ideas that could make DVerify a much more powerful tool for verifying DMinor programs.

Extensions to DMinor

The fact that we use a general verification tool for checking DMinor programs could allow us to increase the expressivity of the DMinor language more easily.

The "M" language is still in development and a sensible addition would be to support state. State means that there are mutable global variables, which can be read and written from inside functions. Adding support for state would be easy in Boogie, because, as Boogie is used mainly for imperative programming languages, it has built-in support for global variables [CMTS09]. An interesting consequence of using Boogie is that it should be easy to support strong updates (i.e. updates that change the type of variables), which is hard to achieve with a type-checker.

Other extensions could target features of "M" that are not implemented in DMinor. "M" supports for example modules, fully qualified names and breaking modules into several files. This is only syntactic sugar, but "M" also supports default values for entity types. That means that whenever no explicit value is assigned to a field the default value is returned. One can dynamically cast an entity to an entity type with default values and thereby add fields to the entity [Mic09].

One of the most ambitious additions would be concurrency. We do not know whether "M" will ever explicitly support concurrency, but it would be a sensible addition as "M" is intended to model databases. Currently "M" does not have a notion of state, so there is no need to introduce locking mechanisms to coordinate access to shared memory. The VCC uses Boogie to verify concurrent C programs [CMST09] [DMS$^+$09], so Boogie is well suited to deal with concurrent programs. Having both state and concurrency available we could properly model "M" extents, which represent a read-only volatile state.

DMinor already allows quantification over collections [BGHL10], but not over the Value type. Such a quantification would be easy in Boogie. It is however questionable how useful that would be.

Performance optimisations

From our tool chain the theorem prover Z3 takes the most time and would therefore be a suitable target to optimise performance. Even though we cannot change Z3 as such, its performance is impacted by the input, largely our axiomatisation, which consists of a lot of quantifiers. Z3 comes with a tool called axiom profiler that shows which axioms are instantiated the most. The number of instantiations can be reduced by using so-called quantifier patterns [LM09], which tell Z3 when to instantiate a quantifier. A lot of our axioms already have patterns, but we did not use profiling techniques to further optimise axioms and patterns. However, too aggressive application of quantifier patterns will reduce the precision of DVerify, because Z3 will reject more correct assertions if it cannot instantiate the necessary quantifiers.

Better scalability testing

We only tested scalability on very small samples. To get a more meaningful result one should test larger samples, which could be obtained by porting to DMinor some of the examples the "M" product group released.

Inferring loop invariants

DMinor requires that each loop is annotated with a loop invariant for the accumulator, because type synthesis cannot in general synthesise such an invariant. Boogie has build-in support for abstract interpretations [BCD+06]. Currently this is very limited, but it seems possible to extend the abstract interpretation in Boogie to include support for our DMinor types.

There are many other techniques for loop invariant inference, but to us most promising seem to be [GMR09] and [BHMR07].

Generating better invariants for Bind

Currently DVerify rejects some of the programs that DMinor can successfully verify. The most common reason is the Bind construct. Unlike **Acc**, Bind does not carry a type annotation and we have to infer one to produce a proper loop invariant.

For Bind, the DMinor type-synthesis can be used to infer loop invariants and it indeed produces correct loop invariants, but these are sometimes too complicated for Boogie to prove them. We tried to simplify them, which works great for most samples, but for some samples the invariant is then too weak. In the current setup of DVerify, a program is translated once and then fed to Boogie. In the future we could change DVerify such that it tries several loop invariants. When Boogie rejects one program, that program is translated again with a different loop invariant and fed again to Boogie. If one of the translations is accepted by Boogie, it means that the original program does not produce a runtime error, because our translation is always sound.

More precise purity checking

Another way to improve precision, and therefore reject less programs, is to improve purity checking. Currently there is a big gap between the theoretical definition of purity by Bierman et al. [BGHL10], see Section 2.2.2, and the practical implementation in DMinor. The problematic part in the purity definition is termination. The DMinor implementation currently uses a very simple algorithm to determine if an expression is terminating: Any recursive function is considered to be not terminating, so the expression in question must not call any recursive function. Termination is undecidable in general, but there are better algorithms to prove termination. One is to check if all recursive calls decrease one of the arguments, such as an entity or a natural number. This is what Coq checks if a recursive function is defined there.

Error reporting

When an assertion fails, Boogie reports that fact, including an abstract execution trace that outlines which branches were taken to reach the failing assertion and where that assertion is located in the code. We have three ideas on how this can be improved.

Firstly, the assertion that fails could be used to reason what kind of error the original DMinor program has. If for example an assertions that checks an operand to the greater operator fails, we know that said operand is not of type Integer as required. To implement that, we would need additional information that maps the lines of the translated program to the lines or expressions in the original program. In programming languages this information is called debug symbols, which have to be generated during the translation process.

The next step would be to map the trace information Boogie gives us back to DMinor and thereby give a trace in the original program. DMinor has only two expressions that can branch: conditional and accumulate. There is a one-to-one correspondence between these and their translation, so mapping the Boogie trace to the corresponding DMinor expressions should be relatively easy provided a mapping of line numbers exists as outlined above. DMinor cannot currently output such a trace.

The last and most complicated step would be counterexamples. A counterexample is one state under which the assertion fails, meaning it gives values for relevant variables. For example, a function that takes values of type Any and then performs the type-test we described in our example in Section 2.3.3. While this type-test works for values of type Integer, it would cause a runtime error for values of other types. So a valid counterexample would be the value true for instance. Counterexamples can be generated by the output Z3 provides. Z3 produces a partial model that indicates, which constructors were used to create the values of certain variables. That information can be used to reconstruct a DMinor value, which can be displayed to the user. The fact that the models are partial means in particular that they can be wrong, so that executing the program with these values does not yield an error. This is especially true because Z3 produces false-positives when it fails to prove an assertion that is actually correct. DMinor has the ability to generate counterexamples and solves the false-positive problem by evaluating type-tests to check if it is indeed a counterexample. If not it still reports the error, but without a counterexample. To make this even more precise one could add the wrong counterexample as an axiom to the proof obligation and run Z3 again, thereby forcing Z3 to come up with another counterexample.

6.2.2. Theory

In this thesis we focused on the theory and formalisation in Coq. Therefore the theory leaves less open ends than the implementation.

Proving completeness of the transformation

A theoretical goal would be to prove the completeness of our translation, rather than just soundness. Whereas soundness gives us that, if the DMinor program raises a runtime error, then Boogie will reject its translation, completeness gives us the additional property that, if Boogie rejects the program, then the original program evaluates to an error. This would guarantee, in theory, that there are no false positives (programs that are rejected by Boogie, but are actually correct with respect to the DMinor big-step semantics). A crucial step in this direction would be to show the translation complete.

```
Theorem translation_complete : ∀ ex r outx avoid st,
  (∀ s arg ex v, (arg,ex) = functions s →
       contains_impure_refinements (subst_exp ex v arg) = false) →
  (∀ s arg ex, (arg,ex) = functions s →
       fv_exp ex = nil ∨ fv_exp ex = (arg::nil)) →
  contains_impure_refinements (subst_state ex st) = false →
  ((translate avoid ex outx) / st ⇝ CError →
    Eval (subst_state ex st) Error
  ∧
  ∀ st′,
    (translate avoid ex outx) / st ⇝ CReturn st′ →
    Eval (subst_state ex st) (st′ outx)).
```

As for soundness, the completeness of the translation can probably be combined with the completeness of the Hoare logic. We expect our Hoare logic to be complete because Nipkow proved completeness for a similar set of Hoare rules [Nip02b]. The verification condition generator is, however, inherently incomplete, because of the user-provided annotations for loop invariants and procedure pre- and postconditions. However, for loop-and-procedure-free programs a completeness proof should be possible even for the verification condition generator. Even more, one should be able to prove the *expressive completeness* of the verification condition generator: for every operationally correct program without annotations, there exists a set of annotations that makes the verification condition generator output a valid formula.

Certified implementation

We have proven in theory that our translation is sound and we implemented this translation in DVerify and tested it to be sound on a number of samples. However, there is no proof that our implementation in F# is sound or indeed implements our proven translation. Coq has the ability to generate OCaml code from Coq source files, which is called extraction [BBC$^+$09]. This feature could be used to create a certified implementation, by exporting our Coq translation function to OCaml code that can then be used as part of our F# project.

To make this extracted code produce proper Boogie programs in practise, we would have to deal with a very big number of implementation details we ignored so far. For example, we would would need to deal with the shallow embedding of the logic in Coq and relate our formalisation of Boogie to the real Boogie. Our formalisation also has many primitives, which would have to be mapped to our axiomatisation.

Translating higher-order functional languages

In this thesis we have related a type system with refinement types for a first-order functional language to standard verification techniques. The open question still remains: Can type systems for higher-order languages, such as F7 [BBF$^+$08], be explained in terms of standard verification techniques? The Hoare logic for call-by-value functional programs by Régis-Gianas et al. could be a good starting point in this direction [RGP08].

Bibliography

[AB05] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM (JACM)*, 52(1):102–146, 2005.

[BBC$^+$09] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual. *INRIA, version*, 8(2), 2009.

[BBF$^+$08] J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. *21st IEEE Computer Security Foundations Symposium (CSF 2008)*, 2008. `http://research.microsoft.com/F7/`.

[BCD$^+$06] M. Barnett, B.Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, page 364–387. Springer, 2006.

[BFG10] K. Bhargavan, C. Fournet, and A.D. Gordon. Modular verification of security protocol code by typing. *ACM SIGPLAN Notices*, 45(1):445–456, 2010.

[BGHL10] G. Bierman, A. Gordon, C. Hriţcu, and D. Langworthy. Semantic Subtyping with an SMT Solver. *International Conference on Functional Programming*, 2010.

[BHM08] M. Backes, C. Hriţcu, and M. Maffei. Type-checking Zero-knowledge. *21st IEEE Computer Security Foundations Symposium (CSF 2008)*, 2008. Implementation available at `http://www.infsec.cs.uni-sb.de/projects/zk-typechecker`.

[BHMR07] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Verification, Model Checking, and Abstract Interpretation*, page 378–394. Springer, 2007.

[BL05] M. Barnett and K.R.M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, page 82–87. Citeseer, 2005.

[BLS05] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. *Construction and analysis of safe, secure, and interoperable smart devices*, page 49–69, 2005.

[CMST09] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. *Microsoft Research*, 2009.

[CMTS09] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electronic Notes in Theoretical Computer Science*, 254:85–103, 2009.

[DL05] R. DeLine and K.R.M. Leino. *BoogiePL: A typed procedural language for checking object-oriented programs*, 2005.

[DMB08] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, page 337–340, 2008.

[DMS$^+$09] 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*. Citeseer, 2009.

[DNS05] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):473, 2005.

[ECM06] ECMA. *Standard ECMA-335: Common Language Infrastructure (CLI)*, 4th edition, June 2006.

[FCB08] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.

[FLL$^+$02] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, page 234–245. ACM, 2002.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[GMR09] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. *Tools and Algorithms for the Construction and Analysis of Systems*, page 262–276, 2009.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[JMR10] R. Jhala, R. Majumdar, and A. Rybalchenko. Refinement type inference via abstract interpretation. *Arxiv preprint arXiv:1004.2884*, 2010.

[Kle99] T. Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

[KO09] N. Kobayashi and C.H.L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of LICS*, volume 2009. Citeseer, 2009.

[Lei05] K.R.M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[Lei08] K.R.M. Leino. This is Boogie 2. *Manuscript KRML*, 178, 2008.

[LM09] K.R.M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of the 2009 ACM symposium on Applied Computing*, page 615–622. ACM, 2009.

[LMS05] K.R.M. Leino, T. Millstein, and J.B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, 2005.

[LMSL08] K.R.M. Leino, M. Moskal, W. Schulte, and R. Leino. Verification condition splitting. *Submitted manuscript, September*, 2008.

[Mar10] C. Marinos. An Introduction to Functional Programming for .NET Developers. *MSDN Magazine*, April 2010.

[Mic] Microsoft. *Questions and Answers – SQL Server Modeling Services*. `http://msdn.microsoft.com/en-us/library/dd578299.aspx`.

[Mic09] Microsoft. *The Microsoft code name "M" Modeling Language Specification*, October 2009. `http://msdn.microsoft.com/en-us/library/dd548667.aspx`.

[Mob06] Mobius Project. *Byte Code Level Specification Language and Program Logic*, 2006.

[Mor] J.H. Morris. Comments on “procedures and parameters”. Undated and unpublished.

[Nip02a] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *Computer Science Logic*, page 155–182. Springer, 2002.

[Nip02b] T Nipkow. Hoare Logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341--367. Kluwer, 2002.

[NP08] M. Naik and J. Palsberg. A type system equivalent to a model checker. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):29, 2008.

[PCG$^+$10] B. Pierce, C. Casinghino, M. Greenberg, V. Sjöberg, and B. Yorgey. *Software Foundations*. `http://www.cis.upenn.edu/~bcpierce/sf/`, 2010.

[Pie02] B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[PT98] B. C. Pierce and D. N. Turner. Local type inference. In *ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 252--265. ACM, 1998.

[RGP08] Y. Régis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In *Mathematics of Program Construction*, page 305–335. Springer, 2008.

[RT06a] S. Ranise and C. Tinelli. Satisfiability modulo theories. *Trends and Controversies-IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006.

[RT06b] S. Ranise and C. Tinelli. The satisfiability modulo theories library (SMT-LIB). www. *SMT-LIB. org*, 2006.

[Sel09] C. Sells. Build Metadata-Based Applications With The “Oslo” Platform. *MSDN Magazine*, February 2009.

[Tar] T. Tarrach. DVerify. `http://www.infsec.cs.uni-saarland.de/projects/dverify`.

[Wil05] R. Wilkes. NGen Revs Up Your Performance with Powerful New Features. *MSDN Magazine*, April 2005.

[Win93] G. Winskel. *The formal semantics of programming languages: an introduction*. The MIT Press, 1993.

[WN04] M. Wildmoser and T. Nipkow. Certifying Machine Code Safety: Shallow versus Deep Embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305--320. Springer, 2004.

A. DMinor big-step semantics

In this appendix we reproduce the complete DMinor big-step semantics as it is formalised in Coq. This work was not done by us, but taken from the DMinor formalisation [BGHL10].

```
Inductive Eval : Exp → Result → Prop :=
  | eval_value : ∀ v,
      Eval (Value v) (Return v)
  | eval_un_op_1 : ∀ e op,
      Eval e Error →
      Eval (UnOp op e) Error
  | eval_un_op_2 : ∀ e op v,
      Eval e (Return v) →
      (~exists v′, un_op_eval op v v′) →
      Eval (UnOp op e) Error
  | eval_un_op_3 : ∀ e op v v′,
      Eval e (Return v) →
      un_op_eval op v v′ →
      Eval (UnOp op e) (Return v′)
  | eval_bin_op_1_1 : ∀ e1 e2 op,
      Eval e1 Error →
      Eval (BinOp op e1 e2) Error
  | eval_bin_op_1_2 : ∀ e1 e2 op v,
      Eval e1 (Return v) →
      Eval e2 Error →
      Eval (BinOp op e1 e2) Error
  | eval_bin_op_2 : ∀ e1 e2 op v1 v2,
      Eval e1 (Return v1) →
      Eval e2 (Return v2) →
      (~exists v′, bin_op_eval op v1 v2 v′) →
      Eval (BinOp op e1 e2) Error
  | eval_bin_op_3 : ∀ e1 e2 op v1 v2 v′,
      Eval e1 (Return v1) →
      Eval e2 (Return v2) →
      bin_op_eval op v1 v2 v′ →
      Eval (BinOp op e1 e2) (Return v′)
```

```
| eval_cond_1 : ∀ e1 e2 e3 r,
    Eval e1 r →
    r ≠ Return (v_tt) →
    r ≠ Return (v_ff) →
    Eval (If e1 e2 e3) Error
| eval_cond_2_1 : ∀ e1 e2 e3 r,
    Eval e1 (Return v_tt) →
    Eval e2 r →
    Eval (If e1 e2 e3) r
| eval_cond_2_2 : ∀ e1 e2 e3 r,
    Eval e1 (Return v_ff) →
    Eval e3 r →
    Eval (If e1 e2 e3) r
| eval_let_1 : ∀ x e1 e2,
    Eval e1 Error →
    Eval (Let x e1 e2) Error
| eval_let_2 : ∀ x e1 e2 v r,
    Eval e1 (Return v) →
    Eval (subst_exp e2 v x) r →
    Eval (Let x e1 e2) r
| eval_entity_1 : ∀ les1 les2 lj ej,
    (∀ l e, In (l,e) les1 → ∃ v, Eval e (Return v)) →
    Eval ej Error →
    Eval (Entity (les1 ++ (lj,ej) :: les2)) Error
| eval_entity_2 : ∀ les v,
    is_E v = true →
    (∀ li ei, In (li,ei) les → ∃ vi,
      Eval ei (Return vi) ∧ v_has_field li v = true ∧ v_dot li v = vi) →
    (∀ li, v_has_field li v = true → ∃ ei, In (li, ei) les) →
    Eval (Entity les) (Return v)
| eval_dot_1_1 : ∀ e l,
    Eval e Error →
    Eval (Dot e l) Error
| eval_dot_1_2 : ∀ e l v,
    Eval e (Return v) →
    is_E v ∧ v_has_field l v = false →
    Eval (Dot e l) Error
| eval_dot_2 : ∀ e l v,
    Eval e (Return v) →
    is_E v ∧ v_has_field l v = true →
    Eval (Dot e l) (Return (v_dot l v))
| eval_add_1 : ∀ e1 e2,
    Eval e1 Error →
    Eval (Add e1 e2) Error
```

```
| eval_add_2_1 : ∀ e1 e2 v1,
    Eval e1 (Return v1) →
    Eval e2 Error →
    Eval (Add e1 e2) Error
| eval_add_2_2 : ∀ e1 e2 v1 v2,
    Eval e1 (Return v1) →
    Eval e2 (Return v2) →
    is_C v2 = false →
    Eval (Add e1 e2) Error
| eval_add_3 : ∀ e1 e2 v1 v2,
    Eval e1 (Return v1) →
    Eval e2 (Return v2) →
    is_C v2 = true →
    Eval (Add e1 e2) (Return (v_add v1 v2))
| eval_appl_1 : ∀ f e,
    Eval e Error →
    Eval (App f e) Error
| eval_appl_2 : ∀ f arg ex r v e,
    Eval e (Return v) →
    (arg,ex) = functions f →
    Eval (subst_exp ex v arg) r →
    Eval (App f e) r
| eval_accum_1_1 : ∀ x y e1 e2 e3,
    Eval e1 Error →
    Eval (Acc x e1 y e2 e3) Error
| eval_accum_1_2 : ∀ x y e1 e2 e3 v1,
    Eval e1 (Return v1) →
    is_C v1 = false →
    Eval (Acc x e1 y e2 e3) Error
| eval_accum_lets : ∀ x e1 y e2 e3 r v1 vs,
    Eval e1 (Return v1) →
    is_C v1 = true →
    (Permutation vs (vb_to_vs (out_C v1))) →
    Eval (Let y e2 (let_seq y (List.map (fun v ⇒ subst_exp e3 v x) vs) (Var y))) r →
    Eval (Acc x e1 y e2 e3) r
| eval_test_wrong : ∀ e T,
    Eval e Error →
    Eval (In e T) Error
| eval_test_any : ∀ e v,
    Eval e (Return v) →
    Eval (In e Any) (Return v_tt)
| eval_test_integer : ∀ e v,
    Eval e (Return v) →
    Eval (In e Integer) (Return (v_logical (In_Integer v)))
```

```
| eval_test_text : ∀ e v,
    Eval e (Return v) →
    Eval (In e Text) (Return (v_logical (In_Text v)))
| eval_test_logical : ∀ e v,
    Eval e (Return v) →
    Eval (In e Logical) (Return (v_logical (In_Logical v)))
| eval_test_entity_1 : ∀ e l T v r,
    Eval e (Return v) →
    is_E v ∧ v_has_field l v = true →
    Eval (In (Value (v_dot l v)) T) r →
    Eval (In e (Entity l T)) r
| eval_test_entity_2 : ∀ e l T v,
    Eval e (Return v) →
    is_E v ∧ v_has_field l v = false →
    Eval (In e (Entity l T)) (Return v_ff)
| eval_test_coll_1 : ∀ e T v,
    Eval e (Return v) →
    is_C v = false →
    Eval (In e (Coll T)) (Return v_ff)
| eval_test_coll_2 : ∀ e T v v′,
    Eval e (Return v) →
    is_C v = true →
    v_mem v′ v = true →
    Eval (In (Value v′) T) Error →
    Eval (In e (Coll T)) Error
| eval_test_coll_3_1 : ∀ e T v,
    Eval e (Return v) →
    is_C v = true →
    (∀ v′, v_mem v′ v = true → Eval (In (Value v′) T) (Return v_tt)) →
    Eval (In e (Coll T)) (Return v_tt)
| eval_test_coll_3_2 : ∀ e T v v‴,
    Eval e (Return v) →
    is_C v = true →
    (∀ v′, v_mem v′ v = true → ∃ v″,
        (Eval (In (Value v′) T) (Return v″) ∧ In_Logical v″ = true)
    ) →
    v_mem v‴ v = true →
    Eval (In (Value v‴) T) (Return v_ff) →
    Eval (In e (Coll T)) (Return v_ff)
| eval_test_refine_1 : ∀ e1 x T e2 v,
    Eval e1 (Return v) →
    Eval (In (Value v) T) Error →
    Eval (In e1 (Refine x T e2)) Error
| eval_test_refine_2_1 : ∀ e1 x T e2 v1 v′,
    Eval e1 (Return v1) →
    Eval (In (Value v1) T) (Return v′) →
    Eval (subst_exp e2 v1 x) Error →
    Eval (In e1 (Refine x T e2)) Error
| eval_test_refine_2_2 : ∀ e1 x T e2 v1 v′ v2,
    Eval e1 (Return v1) →
    Eval (In (Value v1) T) (Return v′) →
    Eval (subst_exp e2 v1 x) (Return v2) →
    In_Logical v′ ∧ In_Logical v2 = false →
    Eval (In e1 (Refine x T e2)) Error
| eval_test_refine_3 : ∀ e1 x T e2 v1 v′ v2,
    Eval e1 (Return v1) →
    Eval (In (Value v1) T) (Return v′) →
    Eval (subst_exp e2 v1 x) (Return v2) →
    In_Logical v′ ∧ In_Logical v2 = true →
    Eval (In e1 (Refine x T e2)) (Return (O_And v′ v2)).
```

B. Complete Coq definitions

This appendix shows a list of relevant Coq files that contain the most important definitions, theorems and lemmas. The proofs are not printed here as they are very hard to read without the interactive proof assistant showing the proof state. The reader is encouraged to take a look at the digital Coq files for details on the proofs.

File	Lines	Description
Lib.v	374	This file contains basic lemmas, which are useful, but not present in the Coq library.
Imp.v	658	In Imp.v we give our while language and the operational semantics. A detailed explanation is given in Section 3.2.
Hoare.v	1014	This file contains our Hoare logics, which is described in Section 3.3.
WP.v	275	The weakest precondition is explained in Section 3.4.
VCgen.v	245	The file VCgen.v contains the verification condition generator, which is described in detail in Section 3.5
Sets.v	97	This file contains just a number of wrappers for the Coq Uniset implementation. These wrappers make the usage of sets simpler, because we know that we only want sets containing strings.
SubstState.v	749	This file contains both, the the simultaneous substitution we briefly outlined in Section 2.2.3 and the relation of operational and logical semantics from Section 2.2.4.
Translate.v	448	Apart from the translation described in Section 4.2, the file Translate.v also contains a number of helping lemmas needed by the soundness proof.
Soundness.v	1581	Last but not least, the file Soundness.v contains the soundness proof described in Section 4.4 and the corollaries relating the soundness of the translation to the Hoare logics and the verification condition generator.
Sum	**5441**	

The files Lib.v, WP.v, VCgen.v and Sets.v are not listed below, either because all important lemmas from them are explained in the thesis already or because they merely contain helper lemmas.

B.1. Imp.v

```
Require Export Lib.

Require Export Model.
Require Export LogicalSemantics.
Require Import Coq.Logic.FunctionalExtensionality.
Require Import BigStepSemantics.

Theorem env_update_eq : ∀ n V st,
  (env_update st V n) V = n.

Theorem env_update_neq : ∀ V2 V1 n st,
  (beq_str V2 V1 = false) →
  (env_update st V2 n) V1 = (st V1).

Definition beq_env (env1:Env) env2 :=
  forall_bool (fun x ⇒ syn_beq_val (env1 x) (env2 x)).

Lemma beq_env_true_x : ∀ env1 env2 x,
  beq_env env1 env2 = true →
  env1 x = env2 x.

Lemma beq_env_true : ∀ env1 env2,
  beq_env env1 env2 = true →
  env1 = env2.

Theorem env_update_shadow : ∀ x1 x2 k1 k2 (f : Env),
    (env_update (env_update f k2 x1) k2 x2) k1 = (env_update f k2 x2) k1.

Theorem env_update_shadow_fun : ∀ x1 x2 k2 (f : Env),
    (env_update (env_update f k2 x1) k2 x2) = (env_update f k2 x2).

Theorem env_update_same : ∀ x1 k1 k2 (f : Env),
  f k1 = x1 →
  (env_update f k1 x1) k2 = f k2.

Theorem env_update_same_fun : ∀ x1 k1 (f : Env),
  f k1 = x1 →
  (env_update f k1 x1) = f.

Theorem env_update_permute : ∀ x1 x2 k1 k2 k3 f,
  (beq_str k2 k1 = false) →
  (env_update (env_update f k2 x1) k1 x2) k3 = (env_update (env_update f k1 x2) k2 x1) k3.

Theorem env_update_permute_fun : ∀ x1 x2 k1 k2 f,
  (beq_str k2 k1 = false) →
  (env_update (env_update f k2 x1) k1 x2) = (env_update (env_update f k1 x2) k2 x1).

Theorem env_update_lookup : ∀ x1 k1 (f:Env),
  env_update f k1 x1 k1 = x1.

Theorem env_update_lookup' : ∀ x1 k1 (f:Env) k,
  beq_str k1 k = true →
  env_update f k1 x1 k = x1.
Theorem env_update_ignore : ∀ x1 k1 k2 (f:Env),
  beq_str k1 k2 = false →
  env_update f k1 x1 k2 = f k2.
Theorem env_update_VAR : ∀ st a n0 x n1 y,
  beq_str a x = false →
  (env_update (env_update (env_update st a n0) x n1) a (st a)) y
  = (env_update st x n1) y.
Theorem env_update_VAR_fun : ∀ st a n0 x n1,
  beq_str a x = false →
  (env_update (env_update (env_update st a n0) x n1) a (st a))
  = (env_update st x n1).
Definition X : string := "X".
Definition Y : string := "Y".
```

B.1.1. Embedding of the logic

```
Definition Assertion := Env → bool.
```

B.1.2. Expressions

```
Inductive expr : Type :=
  | EValue : Value → expr
  | EVar : string → expr
  | ECollAdd : expr → expr → expr
  | ECollRem : expr → expr → expr
  | ECollEmpty : expr → expr
  | EEntUpd : string → expr → expr → expr
  | EDot : expr → string → expr
  | EIn : Assertion → expr
  | EOr : expr → expr → expr
  | ENot : expr → expr
  | EEq : expr → expr → expr
  | ELt : expr → expr → expr
  | EGt : expr → expr → expr
  | EPlus : expr → expr → expr
  | EMinus : expr → expr → expr
  | ETimes : expr → expr → expr
  | EAnd : expr → expr → expr.
```

```
Program Fixpoint eeval (st : Env) (e : expr) {struct e} : Result :=
  match e with
  | EValue v ⇒ Return v
  | EVar x ⇒ Return (st x)
  | ECollAdd c e ⇒ LBind (eeval st c) (fun c ⇒
        (LBind (eeval st e) (fun e ⇒
        if is_C c then Return (v_add e c) else Error)))
  | ECollEmpty c ⇒ LBind (eeval st c) (fun c ⇒
        if is_C c then Return (v_empty c) else Error)
  | ECollRem c e ⇒ LBind (eeval st c) (fun c ⇒
        (LBind (eeval st e) (fun e ⇒
        if is_C c then Return (v_remove e c) else Error)))
  | EEntUpd l v e ⇒ LBind (eeval st v) (fun val ⇒
        (LBind (eeval st e) (fun ent ⇒
        if is_E ent then Return (v_eupdate l val ent) else Error)))
  | EDot e l ⇒ LBind (eeval st e) (fun x ⇒
        if is_E x ∧ v_has_field l x then Return (v_dot l x) else Error)
  | EIn a ⇒ Return (v_logical (a st))

  | ENot e ⇒ LBind (eeval st e) (fun x ⇒
        if In_Logical x then Return (O_Not x) else Error)
  | EEq a b ⇒ LBind (eeval st a) (fun a ⇒
        (LBind (eeval st b) (fun b ⇒ Return (O_EQ a b))))
  | ELt a b ⇒ LBind (eeval st a) (fun x1 ⇒
        (LBind (eeval st b) (fun x2 ⇒
        if In_Integer x1 ∧ In_Integer x2 then Return (O_LT x1 x2) else Error)))
  | EGt a b ⇒ LBind (eeval st a) (fun x1 ⇒
        (LBind (eeval st b) (fun x2 ⇒
        if In_Integer x1 ∧ In_Integer x2 then Return (O_GT x1 x2) else Error)))
  | EPlus a b ⇒ LBind (eeval st a) (fun x1 ⇒
        (LBind (eeval st b) (fun x2 ⇒
        if In_Integer x1 ∧ In_Integer x2 then Return (O_Sum x1 x2) else Error)))
  | EMinus a b ⇒ LBind (eeval st a) (fun x1 ⇒
        (LBind (eeval st b) (fun x2 ⇒
        if In_Integer x1 ∧ In_Integer x2 then Return (O_Minus x1 x2) else Error)))
  | ETimes a b ⇒ LBind (eeval st a) (fun x1 ⇒
        (LBind (eeval st b) (fun x2 ⇒
        if In_Integer x1 ∧ In_Integer x2 then Return (O_Mult x1 x2) else Error)))
  | EAnd a b ⇒ LBind (eeval st a) (fun x1 ⇒
        (LBind (eeval st b) (fun x2 ⇒
        if In_Logical x1 ∧ In_Logical x2 then Return (O_And x1 x2) else Error)))
  | EOr a b ⇒ LBind (eeval st a) (fun x1 ⇒
        (LBind (eeval st b) (fun x2 ⇒
        if In_Logical x1 ∧ In_Logical x2 then Return (O_Or x1 x2) else Error)))
  end.
Definition proc_name := string.
```

B.1.3. The commands

```
Inductive com : Type :=
  | CSkip : com
  | CAss : string → expr → com
  | CSeq : com → com → com
  | CIf : string → com → com → com
  | CWhile : expr → Assertion → com → com
  | CAssert : Assertion → com
  | CPick : string → string → com
  | CCall : proc_name → com
  | CBackup : string → com → com.
Tactic Notation "com_cases" tactic(first) tactic(c) :=
  first;
  [ c "CSkip" | c "CAss" | c "CSeq" | c "CIf" | c "CWhile"
    | c "CAssert" | c "CPick" | c "CCall" | c "CBackup" ].
Notation "'SKIP'" :=
  CSkip.
Notation "c1 ; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "l ':=' a" :=
  (CAss l a) (at level 60).
Notation "'WHILE' b 'DO' c 'LOOP'" :=
  (CWhile b (fun st ⇒ true) c) (at level 80, right associativity).
Notation "'WHILE' b 'INV' a 'DO' c 'LOOP'" :=
  (CWhile b a c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'ASSERT' b" :=
  (CAssert b) (at level 80, right associativity).
Notation "i ':= pick' e" :=
  (CPick i e) (at level 80, right associativity).
Notation "'CALL' i" :=
  (CCall i) (at level 80, right associativity).
Notation "'BACKUP' x 'IN' c" :=
  (CBackup x c) (at level 80, right associativity).
```

B.1.4. Calling convention for procedure calls

```
Definition call_arg := "arg".
Definition call_ret := "ret".
Definition fun_call x i e :=
  (backup x in (call_arg := e; (call i); x := EVar call_ret)).
Module Type ImpArgs.
  Parameter procs : proc_name → com.
End ImpArgs.
Module ImpM (Args : ImpArgs).
```

B.1.5. Operational semantics

```
Inductive CResult :=
  | CError : CResult
  | CReturn : Env → CResult.
Reserved Notation "c1 '/' st '⇝' st'" (at level 40).
Inductive ceval : Env → com → CResult → Prop :=
  | CESkip : ∀ st,
      skip / st ⇝ (CReturn st)
  | CEAss : ∀ st a1 (n:Value) l,
      eeval st a1 = (Return n) →
      (l := a1) / st ⇝ CReturn (env_update st l n)
  | CESeq : ∀ c1 c2 st st′ st″,
      c1 / st ⇝ CReturn st′ →
      c2 / st′ ⇝ st″ →
      (c1; c2) / st ⇝ st″
  | CESeqErr : ∀ c1 c2 st,
      c1 / st ⇝ CError →
      (c1; c2) / st ⇝ CError
  | CEIfTrue : ∀ st st′ b1 c1 c2,
      syn_beq_val (st b1) v_tt = true →
      c1 / st ⇝ st′ →
      (if b1 then c1 else c2) / st ⇝ st′
  | CEIfFalse : ∀ st st′ b1 c1 c2,
      syn_beq_val (st b1) v_tt = false →
      c2 / st ⇝ st′ →
      (if b1 then c1 else c2) / st ⇝ st′
  | CEWhileEnd : ∀ b1 b a1 st c1,
      eeval st b = Return b1 →
      syn_beq_val b1 v_tt = false →
```

```
        (while b inv a1 do c1 end) / st ⇝ CReturn st
  | CEWhileLoop : ∀ st st′ st″ b1 b a1 c1,
        eeval st b = Return b1 →
        syn_beq_val b1 v_tt = true →
        c1 / st ⇝ CReturn st′ →
        (while b inv a1 do c1 end) / st′ ⇝ st″ →
        (while b inv a1 do c1 end) / st ⇝ st″
  | CEWhileLoopErr : ∀ st b1 b a1 c1,
        eeval st b = Return b1 →
        syn_beq_val b1 v_tt = true →
        c1 / st ⇝ CError →
        (while b inv a1 do c1 end) / st ⇝ CError
  | CEAssert : ∀ (st:Env) (b:Assertion),
        b st = true →
        (assert b) / st ⇝ CReturn st
  | CEAssertErr : ∀ st b,
        b st = false →
        (assert b) / st ⇝ CError
  | CEPick : ∀ st x xc v,
        is_C (st xc) = true →
        v_mem v (st xc) = true →
        (x := pick xc) / st ⇝ CReturn (env_update st x v)
  | CECall : ∀ st st′ pn,
        Args.procs pn / st ⇝ st′ →
        (call pn) / st ⇝ st′
  | CEBackup : ∀ st st′ v c,
        c / st ⇝ (CReturn st′) →
        (backup v in c) / st ⇝ CReturn (env_update st v (st′ v))
  | CEBackupErr : ∀ st v c,
        c / st ⇝ CError →
        (backup v in c) / st ⇝ CError
  where "c1 / st ⇝ st′" := (ceval st c1 st′).

Tactic Notation "ceval_cases" tactic(first) tactic(c) := first; [
    c "CESkip" | c "CEAss" | c "CESeq" | c "CESeqErr" |
    c "CEIfTrue" | c "CEIfFalse" | c "CEWhileEnd" | c "CEWhileLoop" |
    c "CEWhileLoopErr" | c "CEAssert" | c "CEAssertErr" | c "CEPick" |
    c "CECall" | c "CEBackup" | c "CEBackupErr" ].

Lemma seq_ass : ∀ c1 c2 c3 st st′,
  (c1; c2; c3) / st ⇝ st′ ↔
  ((c1; c2); c3) / st ⇝ st′.
```

B.1.6. Call-depth-indexed operational semantics

```
Definition CProper res :=
  match res with
  | CError ⇒ False
  | CReturn st ⇒ True
  end.

Definition out_CResult res :=
  match res with
  | CError ⇒ env_empty
  | CReturn x ⇒ x
  end.

Definition bind_CResult (xo : CResult) (f : Env → CResult)
                                : CResult :=
  match xo with
    | CError ⇒ CError
    | CReturn x ⇒ f x
  end.

Reserved Notation "c1 '/' st '— ' n '⇝' st'" (at level 40).

Inductive ceval_indexed : Env → com → nat → CResult → Prop :=

  | CEnSkip : ∀ st n,
      ceval_indexed st CSkip n (CReturn st)
  | CEnAss : ∀ st a1 (n:Value) l nidx,
      eeval st a1 = Return n →
      (CAss l a1) / st — nidx ⇝ CReturn (env_update st l n)
  | CEnSeq : ∀ c1 c2 st st′ st″ n,
      c1 / st — n ⇝ CReturn st′ →
      c2 / st′ — n ⇝ st″ →
      (CSeq c1 c2) / st — n ⇝ st″
  | CEnSeqErr : ∀ c1 c2 st n,
      c1 / st — n ⇝ CError →
      (CSeq c1 c2) / st — n ⇝ CError
  | CEnIfTrue : ∀ st st′ b1 c1 c2 n,
      syn_beq_val (st b1) v_tt = true →
      c1 / st — n ⇝ st′ →
      (CIf b1 c1 c2) / st — n ⇝ st′
  | CEnIfFalse : ∀ st st′ b1 c1 c2 n,
      syn_beq_val (st b1) v_tt = false →
      c2 / st — n ⇝ st′ →
      (CIf b1 c1 c2) / st — n ⇝ st′
  | CEnWhileEnd : ∀ b1 b a1 st c1 n,
      eeval st b = Return b1 →
      syn_beq_val b1 v_tt = false →
      (CWhile b a1 c1) / st — n ⇝ CReturn st
  | CEnWhileLoop : ∀ st st′ st″ b1 b a1 c1 n,
      eeval st b = Return b1 →
      syn_beq_val b1 v_tt = true →
      c1 / st — n ⇝ CReturn st′ →
      (CWhile b a1 c1) / st′ — n ⇝ st″ →
      (CWhile b a1 c1) / st — n ⇝ st″
  | CEnWhileLoopErr : ∀ st b1 b a1 c1 n,
      eeval st b = Return b1 →
      syn_beq_val b1 v_tt = true →
      c1 / st — n ⇝ CError →
      (CWhile b a1 c1) / st — n ⇝ CError
  | CEnAssert : ∀ (st:Env) (b:Assertion) n,
      b st = true →
      (CAssert b) / st — n ⇝ CReturn st
  | CEnAssertErr : ∀ st b n,
      b st = false →
      (CAssert b) / st — n ⇝ CError
  | CEnPick : ∀ st x xc v n,
      is_C (st xc) = true →
      v_mem v (st xc) = true →
      (CPick x xc) / st — n ⇝ CReturn (env_update st x v)

  | CEnCall : ∀ st st′ pn n,
      Args.procs pn / st — n ⇝ st′ →
      (call pn) / st — S n ⇝ st′

  | CEnBackup : ∀ st st′ v c n,
      c / st — n ⇝ (CReturn st′) →
      (CBackup v c) / st — n ⇝ CReturn (env_update st v (st′ v))
  | CEnBackupErr : ∀ st v c n,
      c / st — n ⇝ CError →
      (CBackup v c) / st — n ⇝ CError

  where "c1 '/' st '— ' n '⇝' st'" := (ceval_indexed st c1 n st′).

Lemma ceval_step_more: ∀ i1 i2 st st′ c,
  i1 ≤ i2 → c / st — i1 ⇝ st′ →
  c / st — i2 ⇝ st′.

Lemma exec_iff_execn : ∀ c st st′,
  c / st ⇝ st′ ↔ ∃ n, c / st — n ⇝ st′.

End ImpM.
```

B.2. Hoare.v

```
Require Export Imp.

Module Type HoareArgs.
  Parameter procs : proc_name → com.
  Parameter ZType : Type.
  Parameter ZType_inhabited : ZType.
End HoareArgs.

Module HoareM (Args : HoareArgs).
Module ImpSpecificArgs := ImpM Args.
Export ImpSpecificArgs.
```

B.2.1. Definition of the semantic Hoare triples

```
Definition Assertion := Args.ZType → Env → bool.

Definition check_post Q (z:Args.ZType) st' :=
  match st' with
  | CError ⇒ False
  | CReturn st ⇒ Q z st = true
  end.

Definition check_post' Q (z:Args.ZType) st' :=
  ∃ st'', st' = CReturn st'' ∧ Q z st'' = true.

Lemma check_post_identical : ∀ Q z st',
  check_post Q z st' ↔ check_post' Q z st'.

Lemma check_post_true : ∀ Q (z:Args.ZType) st,
  check_post Q z (CReturn st) →
  Q z st = true.

Lemma check_post_holds : ∀ Q (z:Args.ZType) st,
  Q z st = true →
  check_post Q z (CReturn st).

Definition no_error c :=
  ∀ st st', c/st ⇝ st' →
  CProper st'.

Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st',
        c / st ⇝ st'
    → (∀ z:Args.ZType, P z st = true
    → check_post Q z st').

Notation "{ P } c { Q }" := (hoare_triple P c Q) (at level 90).
```

```
Definition valid_formula f := ∀ (z:Args.ZType) (st:Env), f z st = true.
Lemma hoare_triple_pre_valid_no_error : ∀ c P Q,
  valid_formula P →
  { P } c { Q } → no_error c.
Definition valid (H:Prop) := H.
Definition hoare_triple_n n (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st',
        c / st — n ⇝ st'
     → (∀ z:Args.ZType, P z st = true
     → check_post Q z st').
Notation "⊨ '_n' { P } c { Q }" := (hoare_triple_n n P c Q) (at level 89).
Definition context := list (Assertion × com × Assertion).
Fixpoint sem_context ct { struct ct } :=
  match ct with
  | nil ⇒ True
  | (P,c,Q)::cl ⇒ { P } c { Q } ∧ sem_context cl
  end.
Fixpoint sem_context_n n ct { struct ct } :=
  match ct with
  | nil ⇒ True
  | (P,c,Q)::cl ⇒ ⊨ n { P } c { Q } ∧ sem_context_n n cl
  end.
Lemma sem_context_n_in : ∀ n ct P c Q,
  In (P,c,Q) ct →
  sem_context_n n ct →
  ⊨ n { P } c { Q }.
Definition ext_hoare_triple C P c Q :=
  sem_context C → { P } c { Q }.
Notation "C ⊨ { P } c { Q }" := (ext_hoare_triple C P c Q) (at level 85).
Definition ext_hoare_triple_n n C P c Q :=
  sem_context_n n C → ⊨ n { P } c { Q }.
Notation "C ⊨ n { P } c { Q }" := (ext_hoare_triple_n n C P c Q) (at level 85).
Definition ext_hoare_judgement C1 C2 :=
  sem_context C1 → sem_context C2.
Notation "C1 ⊨ C2" := (ext_hoare_judgement C1 C2) (at level 85).
Definition ext_hoare_judgement_n (n:nat) C1 C2 :=
  sem_context_n n C1 → sem_context_n n C2.
Notation "C1 '⊨' '_n' C2" := (ext_hoare_judgement_n n C1 C2) (at level 85).
Lemma hoare_iff_n : ∀ P c Q, { P } c { Q } ↔ ∀ n, ⊨ n { P } c { Q }.
Lemma sem_iff_n : ∀ C, sem_context C ↔ ∀ n, sem_context_n n C.
```

```
Lemma ext_hoare_if_n : ∀ C1 P c Q,
  (∀ n, C1 ⊨ n { P } c { Q } ) → C1 ⊨ { P } c { Q }.
```

```
Lemma no_context_nil_n : ∀ P R c n,
  nil ⊨ n { P } c { R } ↔ ⊨ n { P } c { R }.
```

```
Lemma no_context_nil : ∀ P R c,
  nil ⊨ { P } c { R } ↔ { P } c { R }.
```

```
Definition eeval_return env e :=
  match eeval env e with
  | Error ⇒ v_null
  | Return x ⇒ x
  end.
```

B.2.2. Lemmas for deriving semantic Hoare triples

Skip command

```
Theorem hoare_skip_n : ∀ C P n,
     C ⊨ n { P } skip { P }.
```

Assignment command

```
Definition assn_sub P x a : Assertion :=
  fun (z:Args.ZType) (st : Env) ⇒ P z (env_update st x (eeval_return st a)).
```

```
Theorem hoare_asgn_n : ∀ C P x a n,
  C ⊨ n { assn_sub P x a } (x := a) { P }.
```

```
Theorem hoare_asgn_eq : ∀ C P P′ V a n,
     P′ = assn_sub P V a
  → C ⊨ n { P′ } (V := a) { P }.
```

Sequence command

```
Theorem hoare_seq_n : ∀ C P Q R c1 c2 n,
     C ⊨ n { Q } c2 { R }
  → C ⊨ n { P } c1 { Q }
  → C ⊨ n { P } c1;c2 { R }.
```

```
Corollary hoare_seq : ∀ C P Q R c1 c2,
    C ⊨ { Q } c2 { R }
  → C ⊨ { P } c1 { Q }
  → C ⊨ { P } c1;c2 { R }.
Lemma hoare_seq_ass : ∀ c1 c2 c3 P Q,
  { P } (c1; c2; c3) { Q } ↔
  { P } ((c1; c2); c3) { Q }.
```

If command

```
Definition bassn i : Assertion :=
  fun z st ⇒ syn_beq_val (st i) v_tt.
Lemma bassn_z_independent : ∀ b (z z' : Args.ZType) st,
  bassn b z st = bassn b z' st.
Lemma bexp_eval_true : ∀ b st (z:Args.ZType),
  syn_beq_val (st b) v_tt = true → (bassn b) z st = true.
Lemma bexp_eval_true_inv : ∀ b st (z:Args.ZType),
  (bassn b) z st = true → syn_beq_val (st b) v_tt = true.
Lemma bexp_eval_false : ∀ b st (z:Args.ZType),
  syn_beq_val (st b) v_tt = false → (bassn b) z st = false.
Lemma bexp_eval_false_inv : ∀ b st (z:Args.ZType),
  (bassn b) z st = false → syn_beq_val (st b) v_tt = false.
Theorem hoare_if_n : ∀ C P Q b c1 c2 n,
  C ⊨ n { fun (z:Args.ZType) st ⇒ (P z st) ∧ (bassn b z st) } c1 { Q } →
  C ⊨ n { fun (z:Args.ZType) st ⇒ (P z st) ∧ (¬ (bassn b z st)) } c2 { Q } →
  C ⊨ n { P } if b then c1 else c2 { Q }.
```

While command

```
Definition eeval_bool st e :=
  match eeval st e with
  | Return v ⇒ syn_beq_val v v_tt
  | Error ⇒ false
  end.
Theorem hoare_while_n : ∀ C P b c z n,
  C ⊨ n { fun z st ⇒ P z st ∧ eeval_bool st b } c { P } →
  C ⊨ n { P }
    (while b inv (fun st ⇒ P z st) do c end)
    { fun z st ⇒ P z st ∧ ¬ (eeval_bool st b) }.
```

Assert Command

```
Theorem hoare_assert_n : ∀ C (Q:Assertion) (b:Imp.Assertion) n,
  C ⊨n { fun z st ⇒ b st ∧ Q z st } assert b { Q }.
```

Pick command

```
Theorem hoare_pick_n : ∀ C P xc x n,
  C ⊨n { fun z st ⇒
  forall_bool (fun v ⇒ implb (v_mem v (st xc)) (assn_sub P x (EValue v) z st)) }
  x := pick xc
  { P }.
```

Backup command

```
Theorem hoare_backup_n : ∀ C P Q x c n,
      (∀ st', C ⊨n { fun z st ⇒ P z st ∧ beq_env st' st }
        c
      { fun z st ⇒ Q z (env_update st' x (st x)) } )
  → C ⊨n { P } backup x in c { Q }.
```

The consequence rule

```
Theorem hoare_consequence_n : ∀ C (P P' Q Q' : Assertion) c n,
  C ⊨n { P' } c { Q' } →
  (∀ st st',
  (∀ (z:Args.ZType), P' z st = true → check_post Q' z st') →
  (∀ (z:Args.ZType), P z st = true → check_post Q z st')) →
  C ⊨n { P } c { Q }.
Definition assimp (P Q:Assertion) :=
  ∀ (z:Args.ZType) st, P z st = true → Q z st = true.
Notation "P ⟶ Q" := (assimp P Q) (at level 89).
Corollary hoare_consequence_pre_n : ∀ C (P P' Q : Assertion) c n,
  C ⊨n { P' } c { Q } →
  (P ⟶ P') →
  C ⊨n { P } c { Q }.
Definition postimp (P Q:Assertion) :=
  ∀ (z:Args.ZType) st, check_post P z st → check_post Q z st.
Notation "P ⟶ Q" := (postimp P Q) (at level 89).
```

Corollary hoare_consequence_post_n : $\forall$ C (P Q Q$'$: Assertion) c n,
  C $\models_n$ { P } c { Q$'$ } $\rightarrow$
  (Q$'$ $\longrightarrow$ Q) $\rightarrow$
  C $\models_n$ { P } c { Q }.

Theorem hoare_consequence_post : $\forall$ (P Q Q$'$: Assertion) c,
  { P } c { Q$'$ } $\rightarrow$
  (Q$'$ $\longrightarrow$ Q) $\rightarrow$
  { P } c { Q }.

B.2.3. Procedure calls

Theorem hoare_context_n : $\forall$ C P c Q n,
  In (P,c,Q) C $\rightarrow$
  C $\models_n$ { P } c { Q }.

Lemma ext_hoare_hoare : $\forall$ C1,
  ($\forall$ (P Q : Assertion) c, In (P,c,Q) C1 $\rightarrow$ C1 $\models$ { P } c { Q }).

Lemma split_judgement : $\forall$ C1 C2 P c Q,
  ext_hoare_judgement C1 ((P, c, Q) :: C2) $\leftrightarrow$
  C1 $\models$ { P } c { Q } $\wedge$ (C1 $\models$ C2).

Lemma split_judgement_n : $\forall$ C1 C2 P c Q n,
  C1 $\models_n$ ((P, c, Q) :: C2) $\leftrightarrow$
  C1 $\models_n$ { P } c { Q } $\wedge$ (C1 $\models_n$ C2).

Lemma hoare_call_lookup : $\forall$ C P Q x,
  C $\models$ { P } **call** x { Q } $\leftrightarrow$ C $\models$ { P } Args.procs x { Q }.

Lemma hoare_context_more : $\forall$ P c Q i1 i2,
  i1 $\leq$ i2 $\rightarrow$
  $\models_{i2}$ { P } c { Q } $\rightarrow$ $\models_{i1}$ { P } c { Q }.

Lemma sem_context_more : $\forall$ C i1 i2,
  i1 $\leq$ i2 $\rightarrow$
  sem_context_n i2 C $\rightarrow$ sem_context_n i1 C.

Lemma call_lookup_n : $\forall$ x st st$'$ n,
  (**call** x) / st — S n $\leadsto$ st$'$ $\leftrightarrow$ Args.procs x / st — n $\leadsto$ st$'$.

Lemma hoare_call_lookup_n : $\forall$ n P Q x,
  $\models_{S(n)}$ { P } **call** x { Q } $\leftrightarrow$ $\models_n$ { P } Args.procs x { Q }.

Theorem hoare_call_simple_n : $\forall$ C P x Q,
  ($\forall$ n$'$, (P, **call** x, Q) :: C $\models_{n'}$ { P } Args.procs x { Q }) $\rightarrow$
  ($\forall$ n, C $\models_n$ { P } **call** x { Q }).

Mutually recursive procedure calls

```
Lemma ext_judgement_if_n : ∀ C1 C2,
  (∀ n, C1 ⊨_n C2) → C1 ⊨ C2.

Lemma sem_context_n_app : ∀ n C1 C2,
  sem_context_n n (C1 ++ C2) ↔ sem_context_n n C1 ∧ sem_context_n n C2.

Lemma sem_context_0_call : ∀ C,
  (∀ P c Q, In (P,c,Q) C → ∃ x, c = call x) → sem_context_n 0 C.

Theorem hoare_call_n : ∀ C1 C2,
  (∀ P c Q, In (P,c,Q) C2 → ∃ x, c = call x) →
  (∀ n′ (P Q : Assertion) x, In (P,CALL x,Q) C2 → C1 ++ C2 ⊨_n′ { P } Args.procs x { Q } ) →
  (∀ n, C1 ⊨_n C2).
```

B.2.4. Syntactic Hoare triples

```
Reserved Notation "C ⊢ { P } c { Q }" (at level 89).
Reserved Notation "C1 ⊢ C2" (at level 89).

Inductive syn_ext_triple : context → Assertion → com → Assertion → Prop :=
  | SSkip : ∀ C P, C ⊢ { P } CSkip { P }
  | SAsgn : ∀ C P V a, C ⊢ { assn_sub P V a } V := a { P }

  | SSeq : ∀ C c1 c2 P Q R, C ⊢ { Q } c2 { R }
                 → C ⊢ { P } c1 { Q }
                 → C ⊢ { P } c1;c2 { R }
  | SIf : ∀ C c1 c2 P Q b, C ⊢ { fun (z:Args.ZType) st ⇒ andb (P z st) (bassn b z st) } c1 { Q }
                 → C ⊢ { fun (z:Args.ZType) st ⇒ andb (P z st) (¬ (bassn b z st)) } c2 { Q }
                 → C ⊢ { P } CIf b c1 c2 { Q }
  | SWhile : ∀ C c P b z, C ⊢ { fun z st ⇒ P z st ∧ eeval_bool st b } c { P }
                 → C ⊢ { P } while b inv (fun st ⇒ P z st) do c end
                          { fun z st ⇒ P z st ∧ ¬ (eeval_bool st b) }
  | SConsq : ∀ C P P′ c Q Q′, C ⊢ { P′ } c { Q′ }
                 → (∀ st st′,
                      (∀ (z:Args.ZType), P′ z st = true → check_post Q′ z st′)
                 → (∀ (z:Args.ZType), P z st = true → check_post Q z st′))
                 → C ⊢ { P } c { Q }
  | SConsqPre : ∀ C P P′ c Q, C ⊢ { P′ } c { Q }
                 → (P ⟶ P′)
                 → C ⊢ { P } c { Q }
  | SConsqPost : ∀ C P c Q Q′, C ⊢ { P } c { Q′ }
                 → (Q′ ⟶ Q)
                 → C ⊢ { P } c { Q }
  | SCallSimp : ∀ P x Q C, ((P, call x, Q) :: C) ⊢ { P } Args.procs x { Q }
                  → C ⊢ { P } call x { Q }
  | SBackup : ∀ C P c x Q,
                     (∀ st′, C ⊢ { fun z st ⇒ P z st ∧ beq_env st′ st }
                        c
                     { fun z st ⇒ Q z (env_update st′ x (st x)) } )
                  → C ⊢ { P } backup x in c { Q }
  | SPick : ∀ C P x xc,
                       C ⊢ { fun z st ⇒
                       forall_bool (fun v ⇒ implb (v_mem v (st xc)) (assn_sub P x (EValue v) z st))
}
                          x := pick xc
                       { P }
  | SCtx : ∀ C P c Q, In (P,c,Q) C
                  → C ⊢ { P } c { Q }


  where "C ⊢ { P } c { Q }" := (syn_ext_triple C P c Q)

with syn_judgement : context → context → Prop :=

  | SCall : ∀ C1 C2,
         (∀ P c Q, In (P,c,Q) C2 → ∃ x, c = call x)
         → (∀ (P Q : Assertion) x, In (P,CALL x,Q) C2 → C1 ++ C2 ⊢ { P } Args.procs x { Q } )
         → C1 ⊢ C2
  | STriple : ∀ C1 C2,
         (∀ P c Q, In (P,c,Q) C2 → C1 ⊢ { P } c { Q } )
         → C1 ⊢ C2

  where "C1 ⊢ C2" := (syn_judgement C1 C2).

Lemma hoare_soundness_n : ∀ C P c Q,
  C ⊢ { P } c { Q } →
  ∀ n, C ⊨ n { P } c { Q }.

Theorem hoare_soundness : ∀ C P c Q,
  C ⊢ { P } c { Q } →
  C ⊨ { P } c { Q } .

Lemma hoare_jsoundness_n : ∀ C1 C2,
  C1 ⊢ C2 →
  ∀ n, C1 ⊨ n C2.

Theorem hoare_jsoundness : ∀ C1 C2,
  C1 ⊢ C2 → C1 ⊨ C2.

End HoareM.
```

B.3. SubstState.v

```
Require Import Imp.
Require Import BigStepSemantics.
Require Import Purity.
Require Export Sets.
Require Export Calculus.

Lemma R_subst_exp : ∀ e3 v x s a a0 s0 a1 a2,
  syn_beq_res
    (LBind (R e3 (env_update (env_update a s a1) s0 a2))
        (fun z : Value ⇒
          R e3 (env_update (env_update a s a0) s0 z)))
    (LBind (R e3 (env_update (env_update a s a0) s0 a2))
        (fun z : Value ⇒
          R e3 (env_update (env_update a s a1) s0 z))) = true
  →
  syn_beq_res
    (LBind (R (subst_exp e3 v x) (env_update (env_update a s a1) s0 a2))
        (fun z : Value ⇒
          R (subst_exp e3 v x) (env_update (env_update a s a0) s0 z)))
    (LBind (R (subst_exp e3 v x) (env_update (env_update a s a0) s0 a2))
        (fun z : Value ⇒
          R (subst_exp e3 v x) (env_update (env_update a s a1) s0 z))) = true.

Lemma pure_subst_exp : ∀ s v e,
  is_pure e = is_pure (subst_exp e s v).

Theorem beq_str_false : ∀ (s s' : string),
  beq_str s s' = false → s ≠ s'.

Lemma list_remove_ignore : ∀ l b s,
  In b l →
  b ≠ s →
  In b (List.remove eq_str_dec s (l)).

Lemma rem_intro : ∀ x a s,
  x ≠ s →
  In x (List.remove eq_str_dec s (a)) →
  In x a.

Lemma rem_remove : ∀ x a s,
  In x a →
  s ≠ x →
  In x (List.remove eq_str_dec s a).
```

B.3.1. Simultaneous substitution

```
Fixpoint subst_state_avoid e (st:Env) avoid :=
  match e with
  | Var x0 ⇒ if set_mem x0 avoid then Var x0 else Value (st x0)
  | Value g ⇒ Value g
  | UnOp o e1 ⇒ UnOp o (subst_state_avoid e1 st avoid)
  | BinOp o e1 e2 ⇒ BinOp o (subst_state_avoid e1 st avoid) (subst_state_avoid e2 st avoid)
  | If e1 e2 e3 ⇒ If (subst_state_avoid e1 st avoid) (subst_state_avoid e2 st avoid)
                                  (subst_state_avoid e3 st avoid)
  | Let y e1 e2 ⇒
        Let y (subst_state_avoid e1 st avoid) (subst_state_avoid e2 st (set_add y avoid))
  | In e1 T ⇒ In (subst_state_avoid e1 st avoid) (subst_state_t_avoid T st avoid)
  | Entity les ⇒
      Entity (List.map (fun le ⇒
                              match le with
                              | (li,ei) ⇒ (li, subst_state_avoid ei st avoid)
                              end) les)
  | Dot e1 l ⇒ Dot (subst_state_avoid e1 st avoid) l
  | Add e1 e2 ⇒ Add (subst_state_avoid e1 st avoid) (subst_state_avoid e2 st avoid)
  | Acc y1 e1 y2 e2 e3 ⇒
        Acc y1 (subst_state_avoid e1 st avoid) y2 (subst_state_avoid e2 st avoid)
                     (subst_state_avoid e3 st (set_add y1 (set_add y2 avoid)))
  | App f e ⇒
      App f (subst_state_avoid e st avoid)
  end
with
  subst_state_t_avoid (U : MType) (st:Env) avoid : MType :=
  match U with
  | Coll T ⇒ Coll (subst_state_t_avoid T st avoid)
  | Entity l T ⇒ Entity l (subst_state_t_avoid T st avoid)
  | Refine y T e ⇒
        Refine y (subst_state_t_avoid T st avoid) (subst_state_avoid e st (set_add y avoid))
  | _ ⇒ U
  end.
Definition subst_state e st := subst_state_avoid e st set_empty.
Definition subst_state_t e st := subst_state_t_avoid e st set_empty.

Lemma fv_subst : ∀ e′ st avoid, incl (fv_exp (subst_state_avoid e′ st avoid)) avoid
with fv_subst_t : ∀ t′ st avoid, incl (fv_type (subst_state_t_avoid t′ st avoid)) avoid.
Lemma impure_subst: ∀ e v x,
  contains_impure_refinements (e) = false →
  contains_impure_refinements (subst_exp e v x) = false
```

```
with impure_subst_t : ∀ t v x,
   contains_impure_expressions t = false →
   contains_impure_expressions (subst_type t v x) = false
with impure_subst_p : ∀ e v x,
   is_pure e = true →
   is_pure (subst_exp e v x) = true.
Lemma fv_subst' : ∀ (e:Exp) e' st,
   e = subst_state_avoid e' st set_empty →
   fv_exp e = nil.
Lemma subst_state_ignore : ∀ e' e a v st av,
   e = subst_state_avoid e' st av →
   ¬ In a (fv_exp e') ∨ set_mem a av = true →
   e = subst_state_avoid e' (env_update st a v) av.
Lemma subst_state_trans_subst' : ∀ e' v s' st av,
   set_mem s' av = false →
   subst_exp (subst_state_avoid e' st (set_add s' av)) v s'
        = subst_state_avoid e' (env_update st s' v) av.
Lemma subst_state_trans_subst : ∀ e' e v s st av,
   set_mem s av = false →
   e = subst_state_avoid e' st (set_add s av) →
   subst_exp e v s = subst_state_avoid e' (env_update st s v) av.
Lemma app_eq_single A : ∀ l1 l2 (s:A),
   l1 ++ l2 = s :: nil →
   (l1 = nil ∨ l1 = s::nil) ∧ (l2 = nil ∨ l2 = s::nil).
Lemma subst_state_eq_subst : ∀ e v s st av,
   set_mem s av = false →
   fv_exp e = nil ∨ fv_exp e = (s::nil) →
   subst_exp e v s = subst_state_avoid e (env_update st s v) av.
Lemma Pure_subst_state : ∀ e st av,
   Pure e ↔
   Pure (subst_state_avoid e st av).
Lemma pure_subst_state : ∀ e st av,
   is_pure e = is_pure (subst_state_avoid e st av).
Lemma subst_swap : ∀ e v1 s st av,
   subst_state_avoid (subst_exp e v1 s) st av =
   subst_exp (subst_state_avoid e st (set_add s av)) v1 s.
Lemma subst_state_closed : ∀ e st av,
   fv_exp e = nil →
   subst_state_avoid e st av = e.

Lemma L30' : ∀ e r st,
   Pure (subst_state_avoid e st set_empty) →
   Eval (subst_state_avoid e st set_empty) r →
   R e st = r.
Lemma EIn_logical : ∀ T e v,
   Eval (In e T) (Return v) →
   In_Logical v = true.
Lemma F_env_ignore : ∀ T v env a vx,
   ¬ In a (fv_type T) →
   F T v (env_update env a vx) = F T v env.
Lemma R_env_ignore : ∀ e env a vx,
   ¬ In a (fv_exp e) →
   R e (env_update env a vx) = R e env.
Lemma alg_purity_sound : ∀ e,
   is_pure e = true → Pure e.
```

B.3.2. Relating operational and logical semantics.

```
Lemma eval_F : ∀ b T x st,
  contains_impure_expressions (subst_state_t T st) = false →
  Eval (In (Value x) (subst_state_t T st)) (Return (v_logical b)) →
  F T x st = b.
Lemma eval_W : ∀ T x st,
  contains_impure_expressions (subst_state_t T st) = false →
  Eval (In (Value x) (subst_state_t T st)) (Error) →
  W T x st = true.
Lemma eval_W_false : ∀ T x st v,
  contains_impure_expressions (subst_state_t T st) = false →
  Eval (In (Value x) (subst_state_t T st)) (Return v) →
  W T x st = false.
```

B.4. Translate.v

```
Require Export Hoare.
Require Export DeclarativeTypeSystem.
Require Export VCgen.
Require Export BigStepSemantics.
Require Export Coq.Program.Wf.
Require Export Purity.
Require Export SubstState.
Require Export Coq.Logic.FunctionalExtensionality.

Fixpoint binders (e:Exp) {struct e} :=
  match e with
    | UnOp o e ⇒ binders e
    | BinOp o e1 e2 ⇒ binders e1 ++ (binders e2)
    | If c t e ⇒ binders c ++ (binders t) ++ (binders e)
    | Let v e e2 ⇒ v :: binders e ++ binders e2
    | In e m ⇒ binders e
    | Entity les ⇒
       (fix bi_entity les :=
       match les with
       | nil ⇒ nil
       | (_, e) :: les′ ⇒ binders e ++ (bi_entity les′)
       end) les
    | Dot e s ⇒ binders e
    | Add e1 e2 ⇒ binders e1 ++ (binders e2)
    | Acc v e s e2 e3 ⇒ v :: s :: binders e ++ binders e2 ++ binders e2
    | App s e ⇒ binders e
    | _ ⇒ nil
  end.

Program Definition unique (e:Exp) : {e:Exp | NoDup (binders e ++ fv_exp e)} := e.

Definition translateT (t:MType) (x:string) : Imp.Assertion :=
  fun env ⇒ F t (env x) env.

Definition translateT_err (t:MType) (x:string) : Imp.Assertion :=
  fun env ⇒ ¬ (W t (env x) env).

Fixpoint translate avoid e outvar {struct e} :=
  let avoid_o := outvar :: avoid in
  let avoid := outvar :: fv_exp e ++ avoid in
  backup outvar in
  match e with
    | Var x ⇒ CAss outvar (EVar x)
    | Value v ⇒ CAss outvar (EValue v)
    | UnOp o e ⇒
         let e′ := proj1_sig (fresh (avoid)) in
         translate (e′::avoid) e e′;
         (assert (translateT (fst (op_type_un o)) e′));
         match o with
         | ONot ⇒ CAss outvar (ENot (EVar e′))
         end
    | BinOp o e1 e2 ⇒
```

```
        let e1′ := proj1_sig (fresh (avoid)) in
        let e2′ := proj1_sig (fresh (e1′::avoid)) in
        translate (e1′::e2'::avoid) e1 e1′;
        translate (e1′::e2'::avoid) e2 e2′;
        (assert (translateT (fst3 (op_type_bi o)) e1′));
        (assert (translateT (snd3 (op_type_bi o)) e2′));
        match o with
        | OEq ⇒ CAss outvar (EEq (EVar e1′) (EVar e2′))
        | OLt ⇒ CAss outvar (ELt (EVar e1′) (EVar e2′))
        | OGt ⇒ CAss outvar (EGt (EVar e1′) (EVar e2′))
        | OAnd ⇒ CAss outvar (EAnd (EVar e1′) (EVar e2′))
        | OOr ⇒ CAss outvar (EOr (EVar e1′) (EVar e2′))
        | OPlus ⇒ CAss outvar (EPlus (EVar e1′) (EVar e2′))
        | OMinus ⇒ CAss outvar (EMinus (EVar e1′) (EVar e2′))
        | OTimes ⇒ CAss outvar (ETimes (EVar e1′) (EVar e2′))
        end
  | If e1 e2 e3 ⇒
        let e1′ := proj1_sig (fresh (avoid)) in
        translate (e1′::avoid) e1 e1′;
        CAssert (translateT Logical e1′);
        if e1′ then translate (e1′::avoid) e2 outvar
        else translate (e1′::avoid) e3 outvar
  | Let x e1 e2 ⇒ translate avoid e1 x; translate avoid e2 outvar
  | In e t ⇒
        let e′ := proj1_sig (fresh (avoid)) in
        translate (e′::avoid) e e′;
        (assert (translateT_err t e′));
        outvar := EIn (translateT t e′)
  | Entity el ⇒
      let ent′ := proj1_sig(fresh (avoid)) in
      let temp′ := proj1_sig (fresh (ent′::avoid)) in
      ent′ := EValue v_eempty;
      (fix les_to_entity el :=
      match el with
      | nil ⇒ skip
      | (l, e) :: el′ ⇒
        (translate (ent′::temp'::fv_exp e ++ avoid_o) e temp′;
        ent′ := EEntUpd l (EVar temp′) (EVar ent′);
        les_to_entity el′)
      end) el;
      outvar := EVar ent′
  | Dot e l ⇒
        let e′ := proj1_sig (fresh (avoid)) in
        (translate (e′::avoid) e e′; assert (translateT (Entity l Any)) e′);
```

```
            CAss outvar (EDot (EVar e′) l)
      | Add e1 e2 ⇒
            let e1′ := proj1_sig (fresh (avoid)) in
            let e2′ := proj1_sig (fresh (e1′::avoid)) in
            translate (e1′::e2'::avoid) e1 e1′; translate (e1′::e2'::avoid) e2 e2′;
            (assert (translateT (Coll Any)) e2′);
            CAss outvar (ECollAdd (EVar e2′) (EVar e1′))
      | Acc x e1 y e2 e3 ⇒
            let e1′ := proj1_sig (fresh (x::y::avoid)) in
            translate (x::y::e1'::avoid) e1 e1′;
            (assert (translateT (Coll Any)) e1′);
            translate (x::y::e1'::avoid) e2 y;
            while (ENot (ECollEmpty (EVar e1′))) do
               CPick x e1′;
               e1′ := ECollRem (EVar e1′) (EVar x);
               translate (x::y::e1'::avoid) e3 y
            end;
            outvar := (EVar y)
      | App f e ⇒
           translate (call_arg::avoid) e call_arg;
           (call f);
           outvar := EVar call_ret
   end.

Module Type TranslateArgs.
   Parameter ZType : Type.
   Parameter ZType_inhabited : ZType.
End TranslateArgs.

Module TranslateM (Args : TranslateArgs).

Module HoareArgs.
   Definition procs s :=
      let (arg,ex) := functions s in
      arg := EVar call_arg; translate nil ex call_ret.
   Definition ZType := Args.ZType.
   Definition ZType_inhabited := Args.ZType_inhabited.
End HoareArgs.

Module VCgenSpecificArgs := VCgenM HoareArgs.
Export VCgenSpecificArgs.
```

B.4.1. Lemmas for the soundness proof

```
Lemma un_op_logical : ∀ v v′,
  un_op_eval ONot v v′ → In_Logical v = true.
Lemma un_op_ONot : ∀ v v′,
  un_op_eval ONot v v′ → v′ = O_Not v.
Lemma un_op_not_logical : ∀ v,
  ¬ (∃ v′ : Value, un_op_eval ONot v v′) → In_Logical v = false.
Lemma fresh_is_fresh : ∀ (a:string) b avoid,
  ¬ In b (a :: avoid) →
  b ≠ a.
Lemma fresh_is_fresh_beq : ∀ a b avoid,
  ¬ In b (a :: avoid) →
  beq_str b a = false.
Lemma fresh_is_fresh2 : ∀ (a:string) b c avoid,
  ¬ In b (c::a :: avoid) →
  b ≠ a.
Lemma fresh_is_fresh2_beq : ∀ a b c avoid,
  ¬ In b (c::a :: avoid) →
  beq_str b a = false.
Lemma backup_ret : ∀ rest st st′ outx v,
  rest / st ⇝ CReturn st′ →
  st′ outx = v →
  (backup outx in rest) / st ⇝ CReturn (env_update st outx v).
Lemma backup_err : ∀ rest st outx,
   (backup outx in rest) / st ⇝ CError ↔
   rest / st ⇝ CError.
Lemma lookup_equiv : ∀ st st′ outx s,
(outx := EValue (st s)) / st ⇝ CReturn st′ ↔
(outx := EVar s) / st ⇝ CReturn st′.
Lemma forall_holds : ∀ v2,
  forall_bool (fun v : Value ⇒ implb (v_mem v v2) true) = true.
Lemma coll_not_empty : ∀ v v1,
  v_mem v v1 = true →
  O_Not (v_empty v1) = v_tt.
Lemma coll_has_member : ∀ v1,
  is_C v1 = true →
  v_empty v1 = v_ff →
  ∃ v, v_mem v v1 = true.
Lemma not_in : ∀ a (x:string) avoid,
  ¬ In a (x :: avoid) → beq_str a x = false.
Lemma fresh_not_avoid : ∀ x avoid,
  In x avoid →
  (proj1_sig (fresh avoid)) ≠ x.
Lemma vars_not_changed : ∀ e x avoid outx st st′,
  translate (avoid) e outx / st ⇝ CReturn st′ →
  outx ≠ x →
  st x = st′ x.
Lemma state_unchanged : ∀ e avoid outx st st′,
  translate (avoid) e outx / st ⇝ CReturn st′ →
  st′ = env_update st outx (st′ outx).
Lemma uneq_update : ∀ st st′ a,
  (∀ x, a<>x → st x = st′ x) →
  st = env_update st′ a (st a).
Lemma subst_state_trans′ : ∀ e2 a st st′,
  (∀ x, a<>x → st x = st′ x) →
  ¬ In a (fv_exp e2) →
  subst_state_avoid e2 st set_empty = subst_state_avoid e2 st′ set_empty.
Lemma subst_state_trans′2 : ∀ e2′ a b st st′,
  (∀ x, a<>x → b<>x → st x = st′ x) →
  ¬ In a (fv_exp e2′) →
  ¬ In b (fv_exp e2′) →
  subst_state_avoid e2′ st set_empty = subst_state_avoid e2′ st′ set_empty.
Lemma subst_state_trans : ∀ e2 e2′ e a avoid st st′,
  translate avoid e a / st ⇝ CReturn st′ →
  ¬ In a (fv_exp e2′) →
  (e2 = subst_state_avoid e2′ st set_empty ↔ e2 = subst_state_avoid e2′ st′ set_empty).
Lemma subst_state_trans_t : ∀ t t′ e a avoid st st′,
  translate avoid e a / st ⇝ CReturn st′ →
  ¬ In a (fv_type t′) →
  (t = subst_state_t_avoid t′ st set_empty ↔ t = subst_state_t_avoid t′ st′ set_empty).
Tactic Notation "prepare" hyp(Himp) hyp(IHEval) constr(outx1) constr(st1) constr(avoid1)
constr(e) "as" ident(out) :=
  let f1 := fresh "IH" out "1" in
  let f1r := fresh "IH" out "1r" in
  let f2 := fresh "IH" out "2" in
  let f2r := fresh "IH" out "2r" in
  let f2′ := fresh "IH" out "2'" in
  let st′ := fresh "st'" out in
  ((destruct IHEval with (outx:=outx1) (st:=st1) (avoid:=avoid1) (ex0:=e) as [f1 f2r]; try
(apply Himp));
  try assumption;
  try (specialize (f1 (refl_equal _)));
  try (edestruct f2r as [st′ [f2 f2′] ]; [ eassumption | ]); try (edestruct f2r as [st′ [f2 f2′] ]; [
reflexivity | ])).
Tactic Notation "prepare_err" hyp(Himp) hyp(IHEval) constr(outx1) constr(st1) constr(avoid1)
constr(e) "as" ident(out) :=
  rewrite backup_err;
  prepare Himp IHEval outx1 st1 avoid1 e as out.
Tactic Notation "prep_ret" :=
  eexists; split; try (eapply backup_ret).
Tactic Notation "prepare_ret" hyp(Himp) hyp(IHEval) constr(outx1) constr(st1) constr(avoid1)
constr(e) "as" ident(out) :=
  prepare Himp IHEval outx1 st1 avoid1 e as out;
  prep_ret.
Ltac remembertaca x a :=
  let tx := fresh "t" x in
  let px := fresh "P" x in
  let x := fresh x in
  let H := fresh "Heq" x in
  (set (tx:=proj1_sig(a)) in ×; assert (H: tx=proj1_sig(a)) by reflexivity;
  clearbody tx; simpl in H; rewrite ← H in ×;
  destruct (a) as [x px]; simpl in H; rewrite H in ×; clear H; clear tx).
Tactic Notation "remember_adv" constr(c) "as" ident(x) := remembertaca x c.
Tactic Notation "resolve_subst" hyp(n) :=
  rewrite ← subst_state_trans; [ eassumption | eassumption | In_contra n].
Ltac W_res1 Himp Hsubst Hsubstt Pa Hrew :=
  unfold translateT_err; rewrite negb_rome; first [ apply eval_W | eapply eval_W_false ];
unfold subst_state_t;
  [ simpl; rewrite subst_state_trans_t in Hsubst;
    [ rewrite ← Hsubst; try apply Himp | try eassumption | unfold fv_type in Pa; In_contra
Pa ] |
      rewrite subst_state_trans_t in Hsubstt; [ rewrite ← Hsubstt; rewrite Hrew | try
eassumption | In_contra Pa ]
      ].
Tactic Notation "W_resolve" hyp(Himp) hyp(Hsubst) hyp(Hsubstt) hyp(Pa) hyp(Hrew)
:=
  first [ apply CESeqErr; apply CEAssertErr; W_res1 Himp Hsubst Hsubstt Pa Hrew
  | eapply CESeq; [apply CEAssert; W_res1 Himp Hsubst Hsubstt Pa Hrew | ] ].
Lemma lookup : ∀ outx v st,
  env_update st outx v outx = v.
End TranslateM.
```

B.5. Soundness.v

```
Require Import Translate.
Module SoundnessM (Args : TranslateArgs).
Module TranslateSpecificArgs := TranslateM Args.
Export TranslateSpecificArgs.
```

B.5.1. Additional lemmas for the entity case

```
Definition v_eremove (l:string) (e:Value) : Value.
Admitted.
Lemma rem_add : ∀ l v x,
  v_dot l x = v →
  v_eupdate l v (v_eremove l x) = x.
Lemma v_has_field_rem : ∀ li l1′ vx,
  v_has_field li (v_eremove l1′ vx) = true →
  v_has_field li vx = true.
Lemma v_dot_rem : ∀ li l1′ vx v,
  v_dot li (v_eremove l1′ vx) = v →
  v_dot li vx = v.
Lemma v_has_field_ign : ∀ li l1′ vx,
  li ≠ l1′ →
  v_has_field li vx = true →
  v_has_field li (v_eremove l1′ vx) = true.
Lemma v_dot_ign : ∀ li l1′ vx v,
  li ≠ l1′ →
  v_dot li vx = v →
  v_dot li (v_eremove l1′ vx) = v.
Lemma v_has_field_false : ∀ li vx,
  v_has_field li (v_eremove li vx) = false.
Lemma v_eremove_E : ∀ l1′ vx,
  is_E (v_eremove l1′ vx) = true.
Lemma les_entity_app : ∀ l l1′ e1′ a b outx avoid st″ v,
  ((fix les_to_entity (les : list (string × Exp)) : com :=
   match les with
   | nil ⇒ skip
   | pair l0 e :: les′ ⇒
       translate (a :: b :: fv_exp e ++ outx :: avoid) e b;
       a := EEntUpd l0 (EVar b) (EVar a); les_to_entity les′
   end) l;
   translate (a :: b :: fv_exp e1′ ++ outx :: avoid) e1′ b;
   a := EEntUpd l1′ (EVar b) (EVar a))
   / st″ ⇝ CReturn v
→
   (fix les_to_entity (les : list (string × Exp)) : com :=
   match les with
   | nil ⇒ skip
   | pair l0 e :: les′ ⇒
       translate (a :: b :: fv_exp e ++ outx :: avoid) e b;
       a := EEntUpd l0 (EVar b) (EVar a); les_to_entity les′
   end) (l ++ (l1′, e1′) :: nil) / st″ ⇝ CReturn v.
Lemma fv_entity_app : ∀ l l1′ e1′,
  (fix fv_entity (les : list (string × Exp)) : list string :=
             match les with
             | nil ⇒ nil
             | pair _ e :: les′ ⇒ fv_exp e ++ fv_entity les′
             end) (l ++ (l1′, e1′) :: nil) =
  (fix fv_entity (les : list (string × Exp)) : list string :=
             match les with
             | nil ⇒ nil
             | pair _ e :: les′ ⇒ fv_exp e ++ fv_entity les′
             end) l ++ fv_exp e1′.
Lemma cir_entity_app : ∀ l l1′ e1′ st,
  (fix cir_entity (les : list (string × Exp)) : bool :=
          match les with
          | nil ⇒ false
          | pair _ e :: les′ ⇒
              contains_impure_refinements e || cir_entity les′
          end)
         (map
            (fun le : string × Exp ⇒
             let (li, ei) := le in (li, subst_state_avoid ei st set_empty))
            (l ++ (l1′, e1′) :: nil)) = false
→
  (fix cir_entity (les : list (string × Exp)) : bool :=
          match les with
          | nil ⇒ false
          | pair _ e :: les′ ⇒
              contains_impure_refinements e || cir_entity les′
          end)
         (map
            (fun le : string × Exp ⇒
             let (li, ei) := le in (li, subst_state_avoid ei st set_empty))
            l) = false ∧ contains_impure_refinements (subst_state_avoid e1′ st set_empty) =
false.
Lemma In_swap A : ∀ l1 l2 (x:A),
  In x (l1 ++ l2) ↔ In x (l2 ++ l1).
Definition nil_list_str : list string := nil.
```

B.5.2. Additional lemmas for the accumulate case

```
Lemma let_seq_imp : ∀ e3,
  contains_impure_refinements e3 = false →
  ∀ vs y x, contains_impure_refinements
  (let_seq y (map (fun v : Value ⇒ subst_exp e3 v x) vs) (Var y)) = false.
Lemma translate_avoid : ∀ av1 av2 st r out e,
  translate av1 e out / st ⇝ r →
  List.incl av1 av2 →
  translate av2 e out / st ⇝ r.
Lemma translate_update : ∀ av e out st st′ x v,
  translate av e out / st ⇝ CReturn st′ →
  x ≠ out →
  ¬ In x (fv_exp e) →
  translate av e out / env_update st x v ⇝ CReturn (env_update st′ x v).
Lemma translate_update_rem : ∀ av e out st st′ x v,
  translate av e out / st ⇝ CReturn st′ →
  ¬ In x (fv_exp e) →
  translate av e out / env_update st x v ⇝ CReturn st′.
Lemma translate_update_add : ∀ av e out st st′ x v,
  translate av e out / env_update st x v ⇝ CReturn st′ →
  ¬ In x (fv_exp e) →
  translate av e out / st ⇝ CReturn st′.
Lemma translate_update_err : ∀ av e out st x v,
  translate av e out / st ⇝ CError →
  x ≠ out →
  ¬ In x (fv_exp e) →
  translate av e out / env_update st x v ⇝ CError.
Lemma translate_upd_subst : ∀ av e out st st′ x v,
  translate av (subst_exp e v x) out / st ⇝ st′ ↔
  translate av e out / (env_update st x v) ⇝ st′.
Lemma let_seq_subst : ∀ y e′2 e′3 x vs st,
Let y (subst_state_avoid e′2 st set_empty)
  (let_seq y
     (map
        (fun v : Value ⇒
                subst_exp (subst_state_avoid e′3 st (set_add x (set_add y set_empty))) v x)
              vs) (Var y)) =
subst_state_avoid
   (Let y e′2
        (let_seq y (map (fun v : Value ⇒ subst_exp e′3 v x) vs) (Var y)))
   st set_empty.
```

```
Lemma v_mem_remove : ∀ x y c,
   v_mem x (v_remove y c) = true →
   v_mem x c = true.
```

```
Lemma unroll_loop : ∀ st st′ b v1 a x y e avoid,
   x ≠ b →
   y ≠ b →
   ¬ In b (fv_exp e) →
   is_C v1 = true →
   v_mem a v1 = true →
   st b = v1 →
   (b := ECollRem (EVar b) (EValue a);
   (while ENot (ECollEmpty (EVar b)) inv fun _ : Env ⇒ true
     do (x := pick b);
       b := ECollRem (EVar b) (EVar x);
       translate (avoid) e y end);
   x := EValue a;
   translate (avoid) e y) / st ⇝
   CReturn st′
   →
   (while ENot (ECollEmpty (EVar b)) inv fun _ : Env ⇒ true
   do (x := pick b);
       b := ECollRem (EVar b) (EVar x);
       translate (avoid) e y end) / st ⇝
   CReturn st′.
```

```
Lemma translate_let_seq_inv : ∀ vs avoid y e′3 x st st′ a,
      List.incl (x :: y :: List.remove eq_str_dec x (List.remove eq_str_dec y (fv_exp e′3))) avoid
→
      translate
              avoid
              (fold_right (fun e e′ : Exp ⇒ Let y e e′)
                    (Let y (subst_exp e′3 a x) (Var y))
                    (map (fun v : Value ⇒ subst_exp e′3 v x) vs)) y / st ⇝
          CReturn st′ →
      (translate
              avoid
              (fold_right (fun e e′ : Exp ⇒ Let y e e′)
                    (Var y)
          (map (fun v : Value ⇒ subst_exp e′3 v x) vs)) y;
    translate avoid (subst_exp e′3 a x) y) / st ⇝
      CReturn st′.
```

```
Lemma Permutation_accum A : ∀ vs (a:A) vl,
  Permutation (vs ++ a :: nil) vl →
  Permutation (a :: vs) vl.
```

```
Theorem translation_sound : ∀ ex r outx avoid st,
  (∀ s arg exp, (arg,exp) = functions s →
      contains_impure_refinements exp = false) →
  (∀ s arg exp, (arg,exp) = functions s →
      fv_exp exp = nil ∨ fv_exp exp = (arg::nil)) →
  Eval (subst_state ex st) r →
  contains_impure_refinements (subst_state ex st) = false →
  (r = Error → (translate avoid ex outx) / st ⇝ CError)
   ∧
  (∀ v, r = (Return v) →
    ∃ st′, (translate avoid ex outx) / st ⇝ CReturn st′ ∧ st′ outx = v).
```

B.5.3. Closedness of the expression in the proof

```
Corollary translation_closed_sound : ∀ ex r outx avoid st,
  fv_exp ex = nil →
  (∀ s arg exp, (arg,exp) = functions s →
      contains_impure_refinements exp = false) →
  (∀ s arg exp, (arg,exp) = functions s →
      fv_exp exp = nil ∨ fv_exp exp = (arg::nil)) →
  Eval ex r →
  contains_impure_refinements ex = false →
  (r = Error → (translate avoid ex outx) / st ⇝ CError)
   ∧
  (∀ v, r = (Return v) →
  ∃ st′, (translate avoid ex outx) / st ⇝ CReturn st′ ∧ st′ outx = v).
```

B.5.4. Relation with Hoare logics

```
Lemma eval_subst_state : ∀ ex st av,
  Eval ex Error →
  Eval (subst_state_avoid ex st av) Error.
```

```
Corollary soundness_hoare_plus_translation : ∀ avoid ex outx P Q,
  fv_exp ex = nil →
  (∀ s arg exp, (arg,exp) = functions s →
        contains_impure_refinements exp = false) →
  (∀ s arg exp, (arg,exp) = functions s →
        fv_exp exp = nil ∨ fv_exp exp = (arg::nil)) →
  contains_impure_refinements (ex) = false →
  nil ⊢ { P } translate avoid ex outx { Q } →
  valid_formula P →
  ¬ Eval ex Error.
```

B.5.5. Relation with verification condition generation

```
Corollary soundness_vcgen_plus_translation : ∀ avoid ex outx Q C,
  fv_exp ex = nil →
  (∀ s arg exp, (arg,exp) = functions s →
        contains_impure_refinements exp = false) →
  (∀ s arg exp, (arg,exp) = functions s →
        fv_exp exp = nil ∨ fv_exp exp = (arg::nil)) →
  contains_impure_refinements (ex) = false →
  valid (VCgen_procs C) →
  valid_formula (VCgen C (translate avoid ex outx) Q) →
  ¬ Eval ex Error.
```

B.5.6. Relation to the type system

```
Definition x := "x".
Definition incom_sample :=
  In (Value (v_int 5)) (Refine x Any (BinOp OGt (Var x) (Value (v_int 5)))).

Lemma incom_sample_type : ¬ envT_empty ⊢ incom_sample : Logical.

Lemma incom_translated_no_error :
  { fun (_ : HoareArgs.ZType) (_ : Env) ⇒ true }
  translate (nil ++ dom envT_empty) incom_sample "outx"
  { fun (_ : HoareArgs.ZType) (st : Env) ⇒ translateT Logical "outx" st }.

Lemma counterexample : ¬ (∀ e env outx T avoid,
  fv_exp e = nil →
  env ⊢ T →
  ¬ In outx (dom env) →
  nil ⊨ { fun _ _ ⇒ true } translate (app avoid (dom env)) e outx
            { fun _ st ⇒ translateT T outx st } →
  env ⊢ e : T).

End SoundnessM.
```

C. Complete axiomatisation

This appendix shows the complete axiomatisation we wrote for our implementation (see Section 5.4).

```
////////////////////////////////////////
// General
////////////////////////////////////////

type String;

type General;

// Constructos
function G_Integer(int) returns (General);
function G_Text(String) returns (General);
function G_Logical(bool) returns (General);
const G_Null : General;

// Tags
type GTag;
const unique GTag_Integer : GTag;
const unique GTag_Text : GTag;
const unique GTag_Logical : GTag;
const unique GTag_Null : GTag;

function get_GTag(General) returns (GTag);
axiom (forall i : int :: { get_GTag(G_Integer(i)) }
  get_GTag(G_Integer(i)) == GTag_Integer);
axiom (forall s : String :: { get_GTag(G_Text(s))}
  get_GTag(G_Text(s)) == GTag_Text);
axiom (forall b : bool :: { get_GTag(G_Logical(b)) }
  get_GTag(G_Logical(b)) == GTag_Logical);
axiom (get_GTag(G_Null) == GTag_Null);

// Testers
function is_Integer(g : General) returns (bool) { get_GTag(g) ==
    GTag_Integer}
function is_Text(g : General) returns (bool) { get_GTag(g) ==
    GTag_Text}
function is_Logical(g : General) returns (bool) { get_GTag(g) ==
    GTag_Logical}
```

```
function is_Null(g : General) returns (bool) { get_GTag(g) ==
   GTag_Null}

// Accessors
function of_G_Integer(General) returns (int);
axiom (forall i : int :: of_G_Integer(G_Integer(i)) == i);
function of_G_Text(General) returns (String);
axiom (forall s : String :: of_G_Text(G_Text(s)) == s);
function of_G_Logical(General) returns(bool);
axiom (forall b : bool :: of_G_Logical(G_Logical(b)) == b);

///////////////////////////////////////
// Value
///////////////////////////////////////

type Value;
type VList;
type VOption;

// Constructors (values)
function G(General) returns (Value);
function E([String]VOption) returns (Value);
function C([Value]int) returns (Value);
function L(VList) returns (Value);

// Lists
const Nil : VList;
function Cons(Value,VList) returns (VList);

// Options used for enties
const NoValue : VOption;
function SomeValue(Value) returns (VOption);

// Tags
type ValueTag; // should be finite
const unique VTag_General : ValueTag;
const unique VTag_Entity : ValueTag;
const unique VTag_Coll : ValueTag;
const unique VTag_List : ValueTag;

type VListTag;
const unique VLTag_Nil : VListTag;
const unique VLTag_Cons : VListTag;
axiom ( forall x:VListTag :: x==VLTag_Nil || x==VLTag_Cons );

type VOptionTag;
const unique VOTag_NoValue : VOptionTag;
const unique VOTag_SomeValue : VOptionTag;
```

```
function get_VTag(Value) returns (ValueTag);
axiom (forall g : General :: { get_VTag(G(g)) }
  get_VTag(G(g)) == VTag_General);
axiom (forall e : [String]VOption :: { get_VTag(E(e))}
  get_VTag(E(e)) == VTag_Entity);
axiom (forall c : [Value]int :: { get_VTag(C(c)) }
  get_VTag(C(c)) == VTag_Coll);
axiom (forall l : VList :: { get_VTag(L(l)) }
  get_VTag(L(l)) == VTag_List);

// new axioms
axiom (forall l:VList :: (is_Cons(l) ==> (exists v:Value, l1:VList ::
    l==Cons(v,l1))));
axiom (forall l:VList :: (is_Nil(l) ==> l == Nil));


function get_VLTag(VList) returns (VListTag);
axiom (forall v : Value, c : VList :: { get_VLTag(Cons(v, c)) }
  get_VLTag(Cons(v, c)) == VLTag_Cons);
axiom (get_VLTag(Nil) == VLTag_Nil);

function get_VOTag(VOption) returns (VOptionTag);
axiom (forall v : Value :: { get_VOTag(SomeValue(v)) }
  get_VOTag(SomeValue(v)) == VOTag_SomeValue);
axiom (get_VOTag(NoValue) == VOTag_NoValue);


// Testers
function is_General(v : Value) returns (bool) { get_VTag(v) ==
   VTag_General}
function is_Entity(v : Value) returns (bool) { get_VTag(v) ==
   VTag_Entity}
function is_Coll(v : Value) returns (bool) { get_VTag(v) == VTag_Coll
    && Positive(of_V_Coll(v))}
function is_List(v : Value) returns (bool) { get_VTag(v) == VTag_List
   }

function is_Cons(v : VList) returns (bool) { get_VLTag(v) ==
   VLTag_Cons}
function is_Nil(v : VList) returns (bool) { get_VLTag(v) == VLTag_Nil
   }

function is_NoValue(v : VOption) returns (bool) { get_VOTag(v) ==
   VOTag_NoValue }
function is_SomeValue(v : VOption) returns (bool) { get_VOTag(v) ==
   VOTag_SomeValue }

// Positiveness of collections (alternative - define a type nat)
// type nat; const Z : nat; function S(x:nat) returns (nat);
```

```
function Positive(a : [Value]int) returns (bool) {(forall v : Value
    :: a[v] >= 0)}

// Accessors
function of_V_General(Value) returns (General);
axiom (forall g : General :: of_V_General(G(g)) == g);
function of_V_Entity(Value) returns ([String]VOption);
axiom (forall e : [String]VOption :: {of_V_Entity(E(e))} of_V_Entity(
    E(e)) == e);
function of_V_Coll(Value) returns([Value]int);
axiom (forall c : [Value]int :: {of_V_Coll(C(c))} of_V_Coll(C(c)) ==
    c);
function of_V_List(Value) returns(VList);
axiom (forall l : VList :: {of_V_List(L(l))} of_V_List(L(l)) == l);

function of_VL_Hd(VList) returns(Value);
axiom (forall l : VList, v : Value :: { of_VL_Hd(Cons(v,l)) }
    of_VL_Hd(Cons(v,l)) == v);
function of_VL_Tl(VList) returns(VList);
axiom (forall l : VList, v : Value :: { of_VL_Tl(Cons(v,l)) }
    of_VL_Tl(Cons(v,l)) == l);

function of_SomeValue(VOption) returns (Value);
axiom (forall v : Value :: {of_SomeValue(SomeValue(v))} of_SomeValue(
    SomeValue(v)) == v);


///////////////////////////////////////
// Entities
///////////////////////////////////////

const Entity_Empty : [String]VOption;
axiom (forall s : String :: {Entity_Empty[s]} Entity_Empty[s] ==
    NoValue);

function has_field(v:Value, s:String) returns (bool) { is_SomeValue(
    of_V_Entity(v)[s]) }

function dot(v:Value, s:String) returns (Value);
axiom (forall v:Value, s:String :: { dot(v,s) }
  is_Entity(v) && has_field(v,s) ==> dot(v,s) ==
     of_SomeValue(of_V_Entity(v)[s]));

procedure pdot(v:Value, s:String) returns (o:Value)
requires is_Entity(v);
requires has_field(v,s);
ensures o==dot(v,s);
{
  o := dot(v,s);
}

///////////////////////////////////////
// Collections
///////////////////////////////////////

// This is extensionality
// axiom (forall a, b : [Value]int :: (forall v : Value :: a[v] == b[
   v]) ==>  a == b);
// Would be nice to have, but is expensive
// Ext helps us to prove:  a[v1:=n1][v2:=n2] == a[v2:=n2][v1:=n1]
// and (forall a : [int]int,  x : int :: a[x:=a[x]] == a)

function Coll_IsEmpty(v:Value) returns (bool);
axiom (forall a:Value :: { Coll_IsEmpty(a) } is_Coll(a) ==>
   Coll_IsEmpty(a) == (forall s:Value :: of_V_Coll(a)[s] == 0));

procedure xyz()
{
assert (forall c:Value:: Coll_IsEmpty(c) ==> Coll_Count(Coll_Empty)
   == v_Int(0));
}

const Coll_Empty : Value;
axiom ( is_Coll(Coll_Empty) == true );
axiom ( Coll_IsEmpty(Coll_Empty) == true );

function Coll_Pick(v:Value,element:Value) returns (out:Value) { C(
   of_V_Coll(v)[element:=of_V_Coll(v)[element]-1]) }

procedure pColl_Pick(v:Value) returns (out:Value, element:Value)
requires is_Coll(v);
requires !Coll_IsEmpty(v);
ensures is_Coll(out);
ensures of_V_Coll(v)[element] > 0;
ensures out == Coll_Pick(v,element);
{
  var list:[Value]int;
  list := of_V_Coll(v);
  havoc element;
  assume (list[element] > 0);
  list[element] := list[element] - 1;
  out := C(list);
}


function Coll_Add(c1:Value, c2:Value) returns (out:Value);
```

```
axiom (forall c1,c2:Value :: { Coll_Add(c1,c2) } is_Coll(c1) &&
   is_Coll(c2) ==> (forall s:Value :: of_V_Coll(Coll_Add(c1,c2))[s]
   == of_V_Coll(c1)[s]+of_V_Coll(c2)[s]) );
axiom (forall c1,c2:Value :: { Coll_Add(c1,c2) } is_Coll(c1) &&
   is_Coll(c2) ==> (is_Coll(Coll_Add(c1,c2))));

procedure pColl_Add(c1:Value, c2:Value) returns (out:Value)
requires is_Coll(c1) && is_Coll(c2);
ensures is_Coll(out);
ensures out == Coll_Add(c1,c2);
ensures Coll_IsEmpty(c1) && Coll_IsEmpty(c2) <==> Coll_IsEmpty(out);
ensures (forall s:Value :: of_V_Coll(out)[s] == of_V_Coll(c1)[s]+
   of_V_Coll(c2)[s]);
{
  out:=Coll_Add(c1,c2);
}



function Coll_Add_num(coll:Value, element:Value, number:int) returns
   (out:Value);
axiom (forall coll:Value, element:Value, number:int :: {Coll_Add_num(
   coll,element,number)}
  is_Coll(coll) ==> Coll_Add_num(coll,element,number) == C(of_V_Coll(
     coll)[element:=of_V_Coll(coll)[element]+number]));


procedure pColl_Add_num(coll:Value, element:Value, number:int)
   returns (out:Value)
requires is_Coll(coll);
requires number >= 0;
ensures is_Coll(out);
ensures Coll_Add_num(coll, element, number) == out;
{
  var list:[Value]int;
  list := of_V_Coll(coll);
  list[element] := list[element] + number;
  out := C(list);
}

function fColl_Count(coll:Value) returns (out:int);
axiom ( forall coll:Value :: { fColl_Count(coll) } Coll_IsEmpty(coll)
    ==> fColl_Count(coll) == 0 );
axiom ( forall coll:Value :: { fColl_Count(coll) } (forall ele:Value,
    num:int, coll1:Value :: Coll_Add_num(coll1,ele,num)==coll ==>
   fColl_Count(coll) == fColl_Count(coll1)+num) );
axiom ( forall coll:Value :: { fColl_Count(coll) } (forall ele:Value,
    coll1:Value :: Coll_Pick(coll1,ele)==coll ==> fColl_Count(coll)
   == fColl_Count(coll1)-1) );
```

```
axiom ( forall coll:Value :: { fColl_Count(coll) } (forall coll1,
    coll2 :Value :: Coll_Add(coll1,coll2)==coll ==> fColl_Count(coll)
    == fColl_Count(coll1)+fColl_Count(coll2)) );

function Coll_Count(coll:Value) returns (out:Value);
axiom (forall coll:Value :: {Coll_Count(coll)}
  is_Coll(coll) ==> Coll_Count(coll) == v_Int(fColl_Count(coll)));

axiom (forall c:Value :: O_Or(O_EQ(Coll_Count(c), v_Int(0)), O_GT(
    Coll_Count(c), v_Int(0))) == v_true);
axiom (forall c :Value :: (O_EQ(Coll_Count(c), v_Int(0)) == v_true
    ==> Coll_IsEmpty(c) == true));

procedure pColl_Count(coll:Value) returns (out:Value)
requires is_Coll(coll);
ensures Integer(out);
ensures out==Coll_Count(coll);
{
  var c : int;
  var co : Value;
  var x : Value;

  c := 0;
  co := coll;
  while (!Coll_IsEmpty(co))
  invariant is_Coll(co);
  invariant c == fColl_Count(coll) - fColl_Count(co);
  {
    call co,x := pColl_Pick(co);
    c := c + 1;
  }
 out := v_Int(c);
}

function Coll_Contains_bool(coll:Value,ele:Value) returns (bool) {
    of_V_Coll(coll)[ele]>=1}
function Coll_Contains(coll:Value,ele:Value) returns (Value);
axiom (forall coll:Value,ele:Value :: {Coll_Contains(coll,ele)}
  is_Coll(coll) ==> Coll_Contains(coll,ele) == v_Logical(
      Coll_Contains_bool(coll,ele)));

axiom (forall c:Value :: {Coll_IsEmpty(c)} !Coll_IsEmpty(c) ==> (
    exists v:Value :: Coll_Contains_bool(c, v)));

procedure pColl_Contains(coll:Value,ele:Value) returns (out:Value)
requires is_Coll(coll);
ensures out == Coll_Contains(coll,ele);
{
  out := Coll_Contains(coll,ele);
}

function Coll_Distinct(coll:Value) returns (out:Value);
axiom (forall c,c' : Value :: {Coll_Distinct(c),is_Coll(c')} c' ==
    Coll_Distinct(c) <==> is_Coll(c') && (forall ele:Value :: ite(
    of_V_Coll(c)[ele]>=1,1,0)==of_V_Coll(c')[ele]));

procedure pColl_Distinct(coll:Value) returns (out:Value)
requires is_Coll(coll);
ensures is_Coll(out);
ensures out == Coll_Distinct(coll);
{
  out := Coll_Distinct(coll);
}

///////////////////////////////////////
// Lists
///////////////////////////////////////

type Closure = [Value, Value] Value;

const List_Empty : Value;
axiom (List_Empty == L(Nil));

function List_Empty(list:Value) returns (bool) { is_Nil(of_V_List(
    list)) }

axiom (forall x:Value :: x!=List_Empty ==> !List_Empty(x));

function List_Contained_bool(element:Value,list:Value) returns (out:
    bool)
{ ((List_Empty(list)) ==> false) && ((!List_Empty(list)) ==> (
    of_VL_Hd(of_V_List(list))==element || List_Contained_bool(element,
    L(of_VL_Tl(of_V_List(list))))) ) }

function List_Contained(element:Value,list:Value) returns (out:Value)
{ v_Logical(List_Contained_bool(element,list)) }

function VList_Length(list:VList) returns (out:int);
axiom ( VList_Length(Nil) == 0);
axiom ( forall l:VList, v:Value :: { VList_Length(Cons(v,l)) }
    VList_Length(Cons(v,l)) == VList_Length(l)+1);

function List_Length(list:Value) returns (out:Value);
axiom (forall list:Value :: {List_Length(list)}
  is_List(list) ==> List_Length(list) == v_Int(VList_Length(of_V_List
      (list))));

procedure pList_Length(list:Value) returns (out:Value)
```

```
requires is_List(list);
ensures Integer(out);
ensures out==List_Length(list);
{
  var l : VList;
  var l_old : VList;
  var c : int;
  l := of_V_List(list);
  l_old := l;
  c := 0;
  while (!is_Nil(l))
  invariant is_Nil(l) || is_Cons(l);
  invariant c == VList_Length(l_old)-VList_Length(l);
  {
    l := of_VL_Tl(l);
    c := c+1;
  }

  out := v_Int(c);

}

function VList_Contains(ele:Value,list:VList) returns (out:bool);
axiom (forall v:Value :: {VList_Contains(v,Nil)} !(VList_Contains(v,
    Nil)));
axiom (forall v:Value, vs:VList, u:Value :: {VList_Contains(u,Cons(v,
    vs))} VList_Contains(u, Cons(v,vs)) <==> (u == v) &&
    VList_Contains(u, vs));

function List_Contains(list:Value,ele:Value) returns (out:Value);
axiom (forall ele:Value,list:Value :: {List_Contains(list,ele)}
  is_List(list) ==> List_Contains(list,ele) == v_Logical (
     VList_Contains(ele,of_V_List(list))));

procedure pList_Contains(list:Value,ele:Value) returns (out:Value)
requires is_List(list);
ensures Logical(out);
ensures out == List_Contains(list, ele);
{
  out := List_Contains(list, ele);
}

function List_Cons(element:Value,list:Value) returns (out:Value);
axiom (forall element:Value,list:Value :: {List_Cons(element,list)}
  is_List(list) ==> List_Cons(element,list) ==
  L(Cons(element,of_V_List(list))));

procedure pList_Cons(element:Value,list:Value) returns (out:Value)
requires is_List(list);
```

```
ensures is_List(out);
ensures out == List_Cons(element,list);
ensures VList_Length(of_V_List(list)) + 1 == VList_Length(of_V_List(
    out));
ensures (forall v:Value :: List_Contained_bool(v,list) ==>
    List_Contained_bool(v,out));
ensures (List_Contained_bool(element,out));
ensures (forall v:Value :: List_Contained_bool(v,out) ==>
    List_Contained_bool(v,list) || v==element);
{
  var l : VList;
  l := of_V_List(list);
  out := L(Cons(element,l));
}

function List_Head(list:Value) returns (out:Value);
axiom (forall list:Value :: {List_Head(list)}
  is_List(list) && !List_Empty(list) ==> List_Head(list) ==
  of_VL_Hd(of_V_List(list)));

procedure pList_Head(list:Value) returns (out:Value)
requires is_List(list);
requires !List_Empty(list);
ensures out == List_Head(list);
ensures (List_Contained_bool(out,list));
{
  out:=List_Head(list);
}

function List_Tail(list:Value) returns (out:Value);
axiom (forall list:Value :: {List_Tail(list)}
  is_List(list) && !List_Empty(list) ==> List_Tail(list) ==
  L(of_VL_Tl(of_V_List(list))));


axiom (forall l:Value :: { List_Tail(l) } VList_Length(of_V_List(l))
    -1 ==  VList_Length(of_V_List(List_Tail(l))));

procedure pList_Tail(list:Value) returns (out:Value)
requires is_List(list);
requires !List_Empty(list);
ensures out == List_Tail(list);
ensures VList_Length(of_V_List(list)) - 1 == VList_Length(of_V_List(
    out));
ensures (forall v:Value :: List_Contained_bool(v,out) ==>
    List_Contained_bool(v,list));
{
  out:=List_Tail(list);
}
```

```
function VList_Nth(list:VList,num:int) returns (out:Value);
axiom ( forall l:VList, v:Value :: { VList_Nth(Cons(v,l),0) }
    VList_Nth(Cons(v,l),0) == v);
axiom ( forall l:VList, v:Value, n:int :: { VList_Nth(Cons(v,l),n) }
    n>0 ==>  VList_Nth(Cons(v,l),n)==VList_Nth(l,n-1));

function List_Nth(list:Value,num:Value) returns (out:Value);
axiom (forall list:Value,num:Value :: {List_Nth(list,num)}
  is_List(list) && Integer(num) ==>
  List_Nth(list,num) == VList_Nth(of_V_List(list),of_G_Integer(
     of_V_General(num))));

axiom ( forall l:Value, n:Value :: { List_Nth(l,n) }
    List_Contained_bool(List_Nth(l,n),l));

procedure pList_Nth(list:Value,num:Value) returns (out:Value)
requires is_List(list);
requires Integer(num);
requires (of_G_Integer(of_V_General(num))>=0);
requires O_GT(List_Length(list),num)==v_true;
ensures out == List_Nth(list,num);
ensures List_Contained_bool(out,list);
{
  out := List_Nth(list,num);
}

function VList_Fold(list:VList, state:Value, cls:Closure) returns (
    out:Value);
axiom (forall state:Value,cls:Closure :: { VList_Fold(Nil,state,cls)
    } VList_Fold(Nil,state,cls) == state);
axiom (forall lh:Value, lt:VList, state:Value, cls:Closure :: {
    VList_Fold(Cons(lh,lt),state,cls)}
  VList_Fold(Cons(lh,lt),state,cls) == VList_Fold(lt,cls[state,lh],
     cls));

function List_Fold(list:Value, state:Value, cls:Closure) returns (out
    :Value)
  { VList_Fold(of_V_List(list), state, cls) }

procedure pList_Fold(list:Value, state:Value, cls:Closure) returns (
    out:Value)
requires is_List(list);
ensures out == List_Fold(list, state, cls);
{
  out := List_Fold(list, state, cls);
}

function VList_Append(list1:VList,list2:VList) returns (out:VList);
```

```
axiom (forall list2:VList :: { VList_Append(Nil,list2) } VList_Append
    (Nil,list2) == list2);
axiom (forall list1,list2:VList :: { VList_Append(list1,list2) }
    VList_Append(list1, list2) == VList_Append(of_VL_Tl(list1),Cons(
    of_VL_Hd(list1),list2)));

function List_Append(list1:Value,list2:Value) returns (out:Value);
axiom (forall list1,list2:Value :: {List_Append(list1,list2)}
  is_List(list1) && is_List(list2) ==> List_Append(list1,list2) ==
    L(VList_Append(of_V_List(list1),of_V_List(list2))));

procedure pList_Append(list1:Value, list2:Value) returns (out:Value)
requires is_List(list1);
requires is_List(list2);
ensures is_List(out);
ensures VList_Length(of_V_List(out)) == VList_Length(of_V_List(list1)
    ) + VList_Length(of_V_List(list2));
ensures VList_Append(of_V_List(list1),of_V_List(list2)) == of_V_List(
    out);
ensures (forall v:Value :: List_Contained_bool(v,list1) ==>
    List_Contained_bool(v,out));
ensures (forall v:Value :: List_Contained_bool(v,list2) ==>
    List_Contained_bool(v,out));
ensures (forall v:Value :: List_Contained_bool(v,out) ==>
    List_Contained_bool(v,list1) || List_Contained_bool(v,list2));
{
  var l1 : Value;
  var l2 : Value;
  var ele : Value;
  l2 := list2;
  l1 := list1;
  while (!List_Empty(l1))
    invariant is_List(l1);
    invariant is_List(l2);
    invariant VList_Length(of_V_List(l1)) + VList_Length(of_V_List(l2
       )) == VList_Length(of_V_List(list1)) + VList_Length(of_V_List(
       list2));
   invariant VList_Append(of_V_List(l1),of_V_List(l2)) ==
       VList_Append(of_V_List(list1),of_V_List(list2));
    invariant (forall v:Value :: List_Contained_bool(v,list1) ==>
       List_Contained_bool(v,l1) || List_Contained_bool(v,l2));
    invariant (forall v:Value :: List_Contained_bool(v,list2) ==>
       List_Contained_bool(v,l2));
    invariant (forall v:Value :: List_Contained_bool(v,l1) ==>
       List_Contained_bool(v,list1));
    invariant (forall v:Value :: List_Contained_bool(v,l2) ==>
       List_Contained_bool(v,list1) || List_Contained_bool(v,list2));
  {
    call ele := pList_Head(l1);
    assert  (List_Contained_bool(ele,list1));
    call l1 := pList_Tail(l1);
    call l2 := pList_Cons(ele,l2);
  }
  out := l2;
}

///////////////////////////////////////
// If then else
///////////////////////////////////////
function ite<X>(c : bool, t1 : X, t2 : X) returns (X);

axiom(forall<X> c:bool, t1:X, t2:X :: {ite(c,t1,t2)} c ==> ite(c,t1,
   t2) == t1);
axiom(forall<X> c:bool, t1:X, t2:X :: {ite(c,t1,t2)} !c ==> ite(c,t1,
   t2) == t2);

function ite_value<X>(c : Value, t1 : X, t2 : X) returns (X) { ite(
   of_G_Logical(of_V_General(c)),t1,t2) }


///////////////////////////////////////
// Constants
///////////////////////////////////////

const v_true : Value;
axiom (v_true == G(G_Logical(true)));

const v_false : Value;
axiom (v_false == G(G_Logical(false)));

///////////////////////////////////////
// Generation functions
///////////////////////////////////////

function v_Int(n:int) returns (Value) { G(G_Integer(n)) }

function v_Logical(n:bool) returns (Value) { G(G_Logical(n)) }

function v_Text(s:String) returns (Value) { G(G_Text(s)) }

///////////////////////////////////////
// Pre- / Postconditions
///////////////////////////////////////

function Integer(v:Value) returns (bool) {
  is_General(v) && is_Integer(of_V_General(v))
}
```

```
function Logical(v:Value) returns (bool) {
  is_General(v) && is_Logical(of_V_General(v))
}

function Text(v:Value) returns (bool) {
  is_General(v) && is_Text(of_V_General(v))
}

function Any(v:Value) returns (bool) {
  is_valid(v)
}

function is_V_Null(v:Value) returns (bool) {
  is_Null(of_V_General(v))
}

///////////////////////////////////////
// Operators
///////////////////////////////////////

function O_GT(v1 : Value, v2 : Value) returns (Value);

axiom (forall v1 : Value, v2 : Value :: { O_GT(v1,v2) }
  Integer(v1) && Integer(v2) ==> ((of_G_Integer(of_V_General(v1)) >
      of_G_Integer(of_V_General(v2)) ==> O_GT(v1,v2) == v_true ) &&
  (of_G_Integer(of_V_General(v1)) <= of_G_Integer(of_V_General(v2))
      ==> O_GT(v1,v2) == v_false )));

procedure pO_GT(v1:Value,v2:Value) returns (v:Value)
  requires Integer(v1) && Integer(v2);
  ensures Logical(v) && v == O_GT(v1,v2);
{
  v := O_GT(v1,v2);
}

function O_LT(v1 : Value, v2 : Value) returns (Value);

procedure pO_LT(v1:Value,v2:Value) returns (v:Value)
  requires Integer(v1) && Integer(v2);
  ensures Logical(v) && v == O_LT(v1,v2);
{
  v := O_LT(v1,v2);
}

axiom (forall v1 : Value, v2 : Value :: { O_LT(v1,v2) }
  Integer(v1) && Integer(v2) ==> (of_G_Integer(of_V_General(v1)) <
      of_G_Integer(of_V_General(v2)) ==> O_LT(v1,v2) == v_true ) &&
  (of_G_Integer(of_V_General(v1)) >= of_G_Integer(of_V_General(v2))
      ==> O_LT(v1,v2) == v_false ));
```

```
function O_EQ(v1:Value, v2:Value) returns (Value);

axiom (forall v1 : Value, v2 : Value :: { O_EQ(v1,v2) }
  is_valid(v1) && is_valid(v2) ==> ((v1 == v2 ==> O_EQ(v1,v2) ==
     v_true ) &&
  (v1 != v2 ==> O_EQ(v1,v2) == v_false )));
axiom (forall v1 : Value, v2 : Value :: { O_EQ(v1,v2) }
  ((v1 == v2 <== O_EQ(v1,v2) == v_true ) &&
  (v1 != v2 <== O_EQ(v1,v2) == v_false )));

function is_valid(v:Value) returns (bool) {is_General(v) || is_Entity
   (v) || is_Coll(v) || is_List(v)}


function O_NE(v1:Value, v2:Value) returns (Value);

axiom (forall v1 : Value, v2 : Value :: { O_NE(v1,v2) }
  is_valid(v1) && is_valid(v2) ==> ((v1 != v2 ==> O_NE(v1,v2) ==
     v_true ) &&
  (v1 == v2 ==> O_NE(v1,v2) == v_false )));
axiom (forall v1 : Value, v2 : Value :: { O_NE(v1,v2) }
  ((v1 != v2 <== O_NE(v1,v2) == v_true ) &&
  (v1 == v2 <== O_NE(v1,v2) == v_false )));


function O_Sum(v1:Value,v2:Value) returns (v:Value);
axiom (forall v1 : Value, v2 : Value :: { O_Sum(v1,v2) }
  Integer(v1) && Integer(v2) ==>
    O_Sum(v1,v2) ==
      v_Int(of_G_Integer(of_V_General(v1))
          + of_G_Integer(of_V_General(v2))));

procedure pO_Sum(v1:Value,v2:Value) returns (v:Value)
  requires Integer(v1) && Integer(v2);
  ensures Integer(v) && v == O_Sum(v1,v2);
{
  v := v_Int(of_G_Integer(of_V_General(v1))
           + of_G_Integer(of_V_General(v2)));
}

function O_Minus(v1:Value,v2:Value) returns (v:Value);
axiom (forall v1 : Value, v2 : Value :: { O_Minus(v1,v2) }
  Integer(v1) && Integer(v2) ==> O_Minus(v1,v2) ==
    v_Int(of_G_Integer(of_V_General(v1)) - of_G_Integer(of_V_General(
       v2))));

procedure pO_Minus(v1:Value,v2:Value) returns (v:Value)
  requires Integer(v1) && Integer(v2);
  ensures Integer(v) && v == O_Minus(v1,v2);
{
  v := v_Int(of_G_Integer(of_V_General(v1)) - of_G_Integer(
      of_V_General(v2)));
}

function O_Mult(v1:Value,v2:Value) returns (v:Value);
axiom (forall v1 : Value, v2 : Value :: { O_Mult(v1,v2) }
  Integer(v1) && Integer(v2) ==> O_Mult(v1,v2) ==
    v_Int(of_G_Integer(of_V_General(v1)) * of_G_Integer(of_V_General(
       v2))));

procedure pO_Mult(v1:Value,v2:Value) returns (v:Value)
  requires Integer(v1) && Integer(v2);
  ensures Integer(v) && v == O_Mult(v1,v2);
{
  v := v_Int(of_G_Integer(of_V_General(v1)) * of_G_Integer(
      of_V_General(v2)));
}

function O_And(v1:Value,v2:Value) returns (v:Value);
axiom (forall v1 : Value, v2 : Value :: { O_And(v1,v2) }
  Logical(v1) && Logical(v2) ==> O_And(v1,v2) ==
    v_Logical(of_G_Logical(of_V_General(v1)) && of_G_Logical(
       of_V_General(v2))));

procedure pO_And(v1:Value,v2:Value) returns (v:Value)
  requires Logical(v1) && Logical(v2);
  ensures Logical(v) && v == O_And(v1,v2);
{
  v := v_Logical(of_G_Logical(of_V_General(v1)) && of_G_Logical(
      of_V_General(v2)));
}

function O_Or(v1:Value,v2:Value) returns (v:Value);
axiom (forall v1 : Value, v2 : Value :: { O_Or(v1,v2) }
  Logical(v1) && Logical(v2) ==> O_Or(v1,v2) ==
    v_Logical(of_G_Logical(of_V_General(v1)) || of_G_Logical(
       of_V_General(v2))));

procedure pO_Or(v1:Value,v2:Value) returns (v:Value)
  requires Logical(v1) && Logical(v2);
  ensures Logical(v) && v == O_Or(v1,v2);
{
  v := v_Logical(of_G_Logical(of_V_General(v1)) || of_G_Logical(
      of_V_General(v2)));
}
```

```
function O_Not(v1:Value) returns (v:Value);
axiom (forall v1 : Value :: { O_Not(v1) }
  Logical(v1) ==> O_Not(v1) ==
    v_Logical(!of_G_Logical(of_V_General(v1))));

procedure pO_Not(v1:Value) returns (v:Value)
  requires Logical(v1);
  ensures Logical(v) && v == O_Not(v1);
{
  v := v_Logical(!of_G_Logical(of_V_General(v1)));
}
```