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

¹Saarland University
²Max Plank Institute for Software Systems
³Microsoft Research, Cambridge
Eidesstattliche Erklärung

Ich erklärte hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 2010-08-31

(Thorsten Tarrach)
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 .................................................. 5
   1.1. Related work .......................................... 6
   1.2. Contributions ........................................ 6
   1.3. Overview ............................................. 7

2. DMinor .......................................................... 9
   2.1. DMinor language ........................................ 9
       2.1.1. Values ........................................... 9
       2.1.2. Types and expressions ........................... 10
   2.2. Semantics ............................................. 12
       2.2.1. Big-step semantics in Coq ....................... 12
       2.2.2. Purity ........................................... 13
       2.2.3. Environments and simultaneous substitution ... 13
       2.2.4. Relating operational and logical semantics ... 13
   2.3. Examples .............................................. 14
       2.3.1. Accumulate example .............................. 14
       2.3.2. Example of an execution error .................. 14
       2.3.3. Incompleteness of the type system ............. 15

3. Boogie .......................................................... 17
   3.1. Boogie language ....................................... 17
   3.2. Formalising a subset of Boogie ....................... 17
       3.2.1. Embedding of the logic .......................... 18
       3.2.2. Expressions .................................... 18
       3.2.3. The commands ................................... 20
       3.2.4. Calling convention for procedure calls ....... 21
       3.2.5. Operational semantics .......................... 22
       3.2.6. Call-depth-indexed operational semantics .... 24
   3.3. The Hoare rules ....................................... 25
       3.3.1. Definition of the semantic Hoare triples ...... 25
       3.3.2. Lemmas for deriving semantic Hoare triples .. 27
       3.3.3. Procedure calls .................................. 31
       3.3.4. Syntactic Hoare triples ........................ 32
   3.4. Inverting Hoare rules .................................. 34
       3.4.1. Weakest precondition ............................ 34
       3.4.2. Inversion functions ............................. 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.</td>
<td>Verification condition generation</td>
<td>35</td>
</tr>
<tr>
<td>4.</td>
<td>Translation</td>
<td>37</td>
</tr>
<tr>
<td>4.1.</td>
<td>Translation of type-tests</td>
<td>37</td>
</tr>
<tr>
<td>4.2.</td>
<td>Translation</td>
<td>37</td>
</tr>
<tr>
<td>4.3.</td>
<td>Examples</td>
<td>41</td>
</tr>
<tr>
<td>4.3.1.</td>
<td>Accumulate example</td>
<td>41</td>
</tr>
<tr>
<td>4.3.2.</td>
<td>Example of an execution error</td>
<td>42</td>
</tr>
<tr>
<td>4.3.3.</td>
<td>Incompleteness of the type system</td>
<td>43</td>
</tr>
<tr>
<td>4.4.</td>
<td>Soundness proof</td>
<td>43</td>
</tr>
<tr>
<td>4.4.1.</td>
<td>Intuition of the proof</td>
<td>44</td>
</tr>
<tr>
<td>4.4.2.</td>
<td>Closedness of the expression in the proof</td>
<td>47</td>
</tr>
<tr>
<td>4.4.3.</td>
<td>Relation with Hoare logics</td>
<td>47</td>
</tr>
<tr>
<td>4.4.4.</td>
<td>Relation with verification condition generation</td>
<td>48</td>
</tr>
<tr>
<td>4.4.5.</td>
<td>Relation to the type system</td>
<td>48</td>
</tr>
<tr>
<td>5.</td>
<td>Implementation</td>
<td>51</td>
</tr>
<tr>
<td>5.1.</td>
<td>DMinor type-checker</td>
<td>51</td>
</tr>
<tr>
<td>5.2.</td>
<td>Boogie tool</td>
<td>51</td>
</tr>
<tr>
<td>5.3.</td>
<td>Some implementation details</td>
<td>52</td>
</tr>
<tr>
<td>5.4.</td>
<td>Axiomatisation</td>
<td>53</td>
</tr>
<tr>
<td>5.4.1.</td>
<td>The General sort</td>
<td>53</td>
</tr>
<tr>
<td>5.4.2.</td>
<td>The Value sort</td>
<td>55</td>
</tr>
<tr>
<td>5.4.3.</td>
<td>Axiomatisation of DMinor operations</td>
<td>55</td>
</tr>
<tr>
<td>5.5.</td>
<td>Quantitative comparison of DMinor and DVerify</td>
<td>57</td>
</tr>
<tr>
<td>5.5.1.</td>
<td>Test suite and precision</td>
<td>58</td>
</tr>
<tr>
<td>5.5.2.</td>
<td>Verification times</td>
<td>58</td>
</tr>
<tr>
<td>5.5.3.</td>
<td>Example where DVerify is more precise</td>
<td>60</td>
</tr>
<tr>
<td>5.6.</td>
<td>Qualitative comparison of DMinor and DVerify</td>
<td>61</td>
</tr>
<tr>
<td>6.</td>
<td>Conclusion</td>
<td>65</td>
</tr>
<tr>
<td>6.1.</td>
<td>Summary</td>
<td>65</td>
</tr>
<tr>
<td>6.2.</td>
<td>Future work</td>
<td>65</td>
</tr>
<tr>
<td>6.2.1.</td>
<td>Implementation</td>
<td>65</td>
</tr>
<tr>
<td>6.2.2.</td>
<td>Theory</td>
<td>69</td>
</tr>
</tbody>
</table>

Bibliography | 71 |

A. DMinor big-step semantics | 75 |

B. Complete Coq definitions | 81 |
| B.1. | Imp.v | 82 |
| B.1.1. | Embedding of the logic | 83 |
| B.1.2. | Expressions | 83 |
| B.1.3. | The commands | 85 |
Contents

B.1.4. Calling convention for procedure calls ..................................... 86
B.1.5. Operational semantics ................................................................. 86
B.1.6. Call-depth-indexed operational semantics .................................. 88
B.2. Hoare.v ......................................................................................... 90
  B.2.1. Definition of the semantic Hoare triples .................................. 90
  B.2.2. Lemmas for deriving semantic Hoare triples ............................ 92
  B.2.3. Procedure calls ........................................................................... 95
  B.2.4. Syntactic Hoare triples ................................................................. 96
B.3. SubstState.v .................................................................................. 98
  B.3.1. Simultaneous substitution ............................................................ 99
  B.3.2. Relating operational and logical semantics. ................................. 101
B.4. Translate.v ..................................................................................... 101
  B.4.1. Lemmas for the soundness proof ................................................ 104
B.5. Soundness.v .................................................................................. 108
  B.5.1. Additional lemmas for the entity case ........................................ 108
  B.5.2. Additional lemmas for the accumulate case ............................... 110
  B.5.3. Closedness of the expression in the proof ................................. 112
  B.5.4. Relation with Hoare logics .......................................................... 112
  B.5.5. Relation with verification condition generation ........................ 113
  B.5.6. Relation to the type system ......................................................... 113

C. Complete axiomatisation .................................................................. 115
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 results¹ 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.

¹Since 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
  | 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 $\oplus$ e  
Applies the unary operator $\oplus$ to $e$. The only unary operator in DMinor is logical negation.

BinOp $\oplus$ e1 e2  
Applies the binary operator $\oplus$ 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  
e 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 \textbf{Error} or to \textbf{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 \textit{Eval} relation.

\[
\text{Inductive Result: Type :=}
\begin{align*}
\text{Error} : & \text{Result} \\
\text{Return} & : \text{Value \rightarrow Result}.
\end{align*}
\]

\[
\text{Inductive Eval : Exp \rightarrow Result \rightarrow Prop.}
\]

The \textit{Eval} relation takes an expression and returns a \textit{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. SEMANTICS

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 \( \text{Env} := \text{string} \rightarrow \text{Value} \).

Definition \( \text{env}_{\text{empty}} (x : \text{string}) := v_{\text{null}} \).

Definition \( \text{env}_{\text{update}} (\text{env} : \text{string} \rightarrow \text{Value}) (x : \text{string}) (v : \text{Value}) :=
\quad \text{fun } y \Rightarrow \text{if } \text{beq}_\text{str} x y \text{ then } v \text{ else } \text{env } y. \)

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 \( \text{subst}_{\text{state}} : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Exp} \) 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: \( \text{subst}_{\text{state} \_t} : \text{MType} \rightarrow \text{Env} \rightarrow \text{MType} \).

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 \( \text{eval}_F : \forall b \ T \ x \ s t,\)
\( \text{contains impure expressions} (\text{subst}_{\text{state} \_t} T \ s t) = \text{false} \rightarrow\)
\( \text{Eval} (\text{In} (\text{Value} x) (\text{subst}_{\text{state} \_t} T \ s t)) (\text{Return} (v_{\text{logical}} b)) \rightarrow\)
In order to relate \( W \) to the big-step semantics, we prove that, if the big-step semantics returns an \texttt{Error}, \( W \) will return true.

**Lemma eval\(_W\):** \( \forall T \times st, \)
contains\_impure\_expressions (subst\_state\_t \( T \) \( st \)) = false \( \rightarrow \)
\( \text{Eval (In (Value \( x \)) (subst\_state\_t \( T \) \( st \)) (\texttt{Error}))} \rightarrow \)
\( W \) \( T \times st = \text{true} \).

If the evaluation returns a value, then \( W \) returns false.

**Lemma eval\(_W\)\_false:** \( \forall T \times st \),
contains\_impure\_expressions (subst\_state\_t \( T \) \( st \)) = false \( \rightarrow \)
\( \text{Eval (In (Value \( x \)) (subst\_state\_t \( T \) \( st \)) (\texttt{Return} \( v \)))} \rightarrow \)
\( W \) \( T \times st = \text{false} \).

## 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 \( x \) from this collection is compared to the \texttt{null} value and if this comparison returns false, \( x \) is added to the resulting collection \( y \).

**Definition acc\_sample\_coll :=**
\( \text{(Acc} \( x \) (Value \( coll \)) \( y \) (Value \( \text{empty} \_\text{coll} \))}
\( \text{(If} (\text{BinOp} \text{OEq} \( \text{Var} \( x \)) (\text{Value} \( \text{v\_null} \)))}
\( \text{(Var} \( y \))
\( \text{(Add} \( \text{Var} \( x \)) \( \text{Var} \( y \))}
\( \text{)} \).

### 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 \texttt{Any} and is greater than 5. Since the greater operator only accepts operands of type \texttt{Integer}, \( v\_tt \) is not an acceptable operand. Therefore, in DMinor this causes an execution error.

**Definition in\_sample :=**
\( \text{In} (\text{Value} \( v\_tt \)) (\text{Refine} \( v \) \text{Any} (\text{BinOp} \text{OGt} \( \text{Var} \( v \)) (\text{Value} (\text{v\_int} 5))) \).

We prove that this program evaluates indeed to \texttt{Error} using the big-step semantics.
2.3. EXAMPLES

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 \( v_{\text{ff}} \) (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 :=
\[ \text{In} \ (\text{Value} \ (v_{\text{int}} \ 5)) \ (\text{Refine} \ x \ \text{Any} \ (\text{BinOp} \ \text{OGt} \ (\text{Var} \ x) \ (\text{Value} \ (v_{\text{int}} \ 5)))) \].

We prove that according to the big-step semantics this indeed evaluates to \( v_{\text{ff}} \).

Lemma incom_sample_big : Eval incom_sample (\text{Return} v_{\text{ff}}).

Since the expression is pure, the logical semantics is also defined for this expression and it also evaluates to \( v_{\text{ff}} \).

Lemma incom_sample_logical : \( R \) incom_sample env_empty = \text{Return} v_{\text{ff}}.

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

Lemma incom_sample_type : \( \neg \text{envT} \_\text{empty} \vdash \text{incom} \_\text{sample} : \text{Logical} \).

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 \( \text{Env} \) to \( \text{bool} \). 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 := \( \text{Env} \to \text{bool} \).

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 \( (A : \text{Type}) \ (P : A \to \text{bool}) : \text{bool} := \)

\[
\begin{align*}
&\text{match ClassicalEpsilon.excluded\_middle\_informative} (\forall x, P x = \text{true}) \text{ with} \\
&\mid \text{left } P\text{true} := \text{true} \\
&\mid \text{right } P\text{false} := \text{false} \\
&\text{end.}
\end{align*}
\]

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 :=

\[
\begin{align*}
&\text{EValue : Value \to expr} \\
&\text{EVar : string \to expr} \\
&\text{ECollAdd : expr \to expr \to expr} \\
&\text{ECollRem : expr \to expr \to expr} \\
&\text{ECollEmpty : expr \to expr} \\
&\text{EEntUpd : string \to expr \to expr \to expr}
\end{align*}
\]
3.2. FORMALISING A SUBSET OF BOOGIE

| 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 Returns 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 com-
mands 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 condi-
tionals.
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.
3.2. FORMALISING A SUBSET OF BOOGIE

\[ x := \text{pick} \ x_c \]

This command non-deterministically chooses an element from a collection \( x_c \) 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.

\textbf{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.

\textbf{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 as follows.

\begin{align*}
\text{Inductive com} \colon & \text{Type} := \\
& | \text{CSkip} : \text{com} \\
& | \text{CAss} : \text{string} \rightarrow \text{expr} \rightarrow \text{com} \\
& | \text{CSeq} : \text{com} \rightarrow \text{com} \rightarrow \text{com} \\
& | \text{CIf} : \text{string} \rightarrow \text{com} \rightarrow \text{com} \rightarrow \text{com} \\
& | \text{CWhile} : \text{expr} \rightarrow \text{Assertion} \rightarrow \text{com} \rightarrow \text{com} \\
& | \text{CAssert} : \text{Assertion} \rightarrow \text{com} \\
& | \text{CPick} : \text{string} \rightarrow \text{string} \rightarrow \text{com} \\
& | \text{CCall} : \text{proc\_name} \rightarrow \text{com} \\
& | \text{CBackup} : \text{string} \rightarrow \text{com} \rightarrow \text{com}.
\end{align*}

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 \textbf{backup} command makes sure no other variables but the return variable \( x \) are changed.

\begin{align*}
\text{Definition call\_arg} & := \text{"arg"}. \\
\text{Definition call\_ret} & := \text{"ret"}. \\
\text{Definition fun\_call \ x i e} & := \\
& (\text{backup} \ x \ in \ (\text{call\_arg} := e; (\text{call} \ i); x := \text{EVar\_call\_ret})).
\end{align*}
CHAPTER 3. BOOGIE

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:

\[
x := \text{EPlus}(\text{EValue}(v_{\text{int}} 42)) (\text{EValue}(v_{\text{ff}}))
\]

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)
3.2. FORMALISING A SUBSET OF BOOGIE

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 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,
\[
\begin{align*}
    c / st & \rightsquigarrow (CReturn st') \rightarrow \\
    (\text{backup } v \text{ in } c) / st & \rightsquigarrow (\text{CReturn } (\text{env_update } st \, v \, (st' \, v))) \\
| \text{CBackupErr} : \forall \, st \, v \, c, \\
    c / st & \rightsquigarrow \text{CError} \rightarrow \\
    (\text{backup } v \text{ in } c) / st & \rightsquigarrow \text{CError}
\end{align*}
\]

where "c1 / st \rightsquigarrow 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 \rightarrow com \rightarrow nat \rightarrow CResult \rightarrow Prop :=
\[
\begin{align*}
    & \ldots \\
    \mid \text{CEnCall : } \forall \, st \, st' \, pn \, n, \\
    & \text{Args.procs } pn \, / \, st \rightarrow n \rightsquigarrow st' \rightarrow \\
    & (\text{call } pn) / st \rightarrow S \, n \rightsquigarrow st' \\
\end{align*}
\]

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 \(i_1\) steps, it will also evaluate without diverging for every larger number of steps and the resulting state is the same.

Lemma ceval_step_more: \forall \, i_1 \, i_2 \, st \, st' \, c, \\
\[ i_1 \leq i_2 \rightarrow c / st \rightsquigarrow i_1 \rightsquigarrow st' \rightarrow \]
\[ c / st \rightsquigarrow i_2 \rightsquigarrow st'. \]

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: \forall \, c \, st \, st', \\
\[ c / st \rightsquigarrow st' \leftrightarrow \exists \, n, c / st \rightarrow n \rightsquigarrow st'. \]
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 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 : \forall Q z st',
    check_post Q z st' \leftrightarrow 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 :=
\forall st st',
    c / st \rightarrow st' 
\rightarrow (\forall z:Args.ZType, P z st = true 
\rightarrow 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 := \forall (z:Args.ZType) (st:Env), f z st = true.

Lemma hoare_triple_pre_valid_no_error : \forall c P Q,
    valid_formula P \rightarrow
    \{ P \} c \{ Q \} \rightarrow 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 |-{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 :=
\forall st st',
    c / st \rightarrow n \rightarrow st' 
\rightarrow (\forall z:Args.ZType, P z st = true 
\rightarrow 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**
\[
\text{context} := \text{list} (\text{Assertion} \times \text{com} \times \text{Assertion}).
\]

The function \(\text{sem\_context}\) checks if each of the Hoare triples in a context is correct.

\[
\text{Fixpoint}\ \text{sem\_context}\ ct \{\ \text{struct}\ ct\} := \\
\quad \text{match}\ ct\ \text{with}\ \\
\quad |\ \text{nil} \Rightarrow \text{True}
\quad |(P,c,Q)::cl \Rightarrow \{P\}c\{Q\} \land \text{sem\_context}\ cl
\quad \text{end.}
\]

\[
\text{Fixpoint}\ \text{sem\_context\_n}\ n\ n\ ct \{\ \text{struct}\ ct\} := \\
\quad \text{match}\ ct\ \text{with}\ \\
\quad |\ \text{nil} \Rightarrow \text{True}
\quad |(P,c,Q)::cl \Rightarrow \models n\{P\}c\{Q\} \land \text{sem\_context\_n}\ n\ cl
\quad \text{end.}
\]

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 \(\models n\{P\}c\{Q\}\) to describe such a Hoare triple where \(C\) is the context.

**Definition**
\[
\text{ext\_hoare\_triple}\ C P c Q := \\
\quad \text{sem\_context}\ C \rightarrow \{P\}c\{Q\}.
\]

**Definition**
\[
\text{ext\_hoare\_triple\_n}\ n\ C P c Q := \\
\quad \text{sem\_context\_n}\ n\ C \rightarrow \models n\{P\}c\{Q\}.
\]

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

We use the syntax \(\models n\ C1\ C2\) to describe a Hoare judgement where \(C1\) is the list of known Hoare triples and \(C2\) the Hoare triples we want to prove.

**Definition**
\[
\text{ext\_hoare\_judgement}\ C1\ C2 := \\
\quad \text{sem\_context}\ C1 \rightarrow \text{sem\_context}\ C2.
\]

**Definition**
\[
\text{ext\_hoare\_judgement\_n}\ (n:\text{nat})\ C1\ C2 := \\
\quad \text{sem\_context\_n}\ n\ C1 \rightarrow \text{sem\_context\_n}\ n\ C2.
\]

### 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.

\[
\begin{array}{c}
\{P\}c1\{Q\} \\
\{Q\}c2\{R\}
\end{array} 
\Rightarrow 
\{P\}c1;c2\{Q\}
\]

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 }. 
3.3. THE HOARE RULES

If command

To formalise the Hoare rule for conditionals we need to state in the precondition that a certain variable is true. We use \( \text{bassn} \) for that. The \( \text{bassn} \) 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** \( \text{bassn} \): Assertion :=
\[
\text{fun } z \text{ st } \Rightarrow \text{syn}_\text{beq}_\text{val}(\text{st } i) \text{ v } \text{tt}.
\]

We can now introduce the Hoare rule for conditionals. It states that \( P \) is a precondition and \( Q \) is a postcondition, if \( P \land b \) is a pre- and \( Q \) a postcondition for the then-part \( c_1 \) and \( P \land \neg b \) is a pre- and \( Q \) a postcondition for the else-part \( c_2 \). The variable \( b \) is the guard of the conditional.

\[
\frac{\{ P \land b \}\ c_1 \{ Q \}}{\{ P \}\ \text{if } b \text{ then } c_1 \text{ else } c_2 \{ Q \}}
\]

**Theorem** \( \text{hoare}_\text{if}_n \): \( \forall C \ P \ Q \ b \ c_1 \ c_2 \ n, \)
\[
C \models n \{ \text{fun } z \text{ Args.ZType st } \Rightarrow (P \ Z \ st) \land (\text{bassn } b \ Z \ st) \} \ c_1 \{ Q \} \rightarrow
C \models n \{ \text{fun } (z \text{ Args.ZType st } \Rightarrow (P \ Z \ st) \land (\neg(\text{bassn } b \ Z \ st))) \} \ c_2 \{ Q \} \rightarrow
C \models n \{ P \} \ \text{if } b \text{ then } c_1 \text{ else } c_2 \{ Q \}.
\]

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 \textbf{Error}, we define \( \text{eeval}_\text{bool} \) that returns false in case of an error and otherwise compares the resulting value to \( \text{v} \_\text{tt} \).

**Definition** \( \text{eeval}_\text{bool} \): st e :=
\[
\text{match } \text{eeval st e} \text{ with}
| \text{Return } v \Rightarrow \text{syn}_\text{beq}_\text{val} v \text{ v } \text{tt}
| \text{Error} \Rightarrow \text{false}
\text{end}.
\]

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.

\[
\frac{\{ P \land b \}\ c\{ Q \}}{\{ P \}\ \text{while } b \ \text{inv} \ P \ \text{do } c \ \text{end}\{ Q \land \neg b \}}
\]

**Theorem** \( \text{hoare}_\text{while}_n \): \( \forall C \ P \ b \ c \ z \ n, \)
\[
C \models n \{ \text{fun } z \text{ st } \Rightarrow P \ Z \ st \land \text{eeval}_\text{bool} \ st \ b \} \ c \{ P \} \rightarrow
C \models n \{ P \}
\]
\[
(\text{while } b \ \text{inv} \ (\text{fun } st \Rightarrow P \ Z \ st) \ \text{do } c \ \text{end})
\]
\{ \text{fun } z \text{ st } \Rightarrow P \text{ st } \land \neg (\text{eval_bool st b}) \}. \\

**Assert Command**

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

\{ Q \land b \} \text{assert } b \{ Q \}

Theorem hoare_assert_n : \forall C \text{ (Q:Assertion) (b:Imp.Assertion) n}, 
\quad C \models_n \{ \text{fun st st } \Rightarrow b \text{ st } \land Q \text{ st st } \} \text{ assert } b \{ Q \}.

**Pick command**

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

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

Theorem hoare_pick_n : \forall C \text{ xc x n}, 
\quad C \models_n \{ \text{fun z st } \Rightarrow \text{forall Bool (fun } v \Rightarrow \text{implb (v mem v (st xc)) (assn_sub P x (EValue v) z st))} \} \text{ x := pick } xc \{ P \}.

**Backup command**

The \text{backup x in c} command evaluates the Hoare triple for \text{c} using the same state for the pre- and postcondition, except for variable \text{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 \text{st}' that we require to be equal to the state in the precondition.

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

\forall st', \{ \text{fun st } \Rightarrow P \text{ st } \land st' = st \} c \{ \text{fun st } \Rightarrow Q\{st x/x\} st' \} \quad \{ P \} \text{backup x in c} \{ Q \}

Theorem hoare_backup_n : \forall C \text{ P Q x n}, 
\quad \{ \forall st', C \models_n \{ \text{fun z st } \Rightarrow P \text{ z st } \land \text{beq_env st' st} \} \} \quad c \{ \text{fun z st } \Rightarrow Q \text{ z (env_update st' x (st x))} \} \} \quad \rightarrow C \models_n \{ P \} \text{ backup x in c} \{ Q \}.
3.3. THE HOARE RULES

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.

\[
\frac{\{P\} c \{Q\} \quad \forall s t s', (\forall z, P' z s' = true \rightarrow Q' z s')}{\{P\} c \{Q\}}
\]

Theorem hoare_consequence_n: \(\forall C \{P P' Q Q' : \text{Assertion}\} c n,\)
\[C \models_n \{P'\} c \{Q'\} \rightarrow\]
\[(\forall z: \text{Args.ZType}, P' z s = true \rightarrow \text{check_post} Q' z s') \rightarrow\]
\[(\forall z: \text{Args.ZType}, P z s = true \rightarrow \text{check_post} Q z s') \rightarrow\]
\[C \models_n \{P\} c \{Q\}.\]

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.

\[
\frac{\{P\} c \{Q\} \quad \forall s t z, (P z s \rightarrow P' z s)}{\{P\} c \{Q\}}
\]

Corollary hoare_consequence_pre_n: \(\forall C \{P P' Q : \text{Assertion}\} c n,\)
\[C \models_n \{P'\} c \{Q\} \rightarrow\]
\[(P \rightarrow P') \rightarrow\]
\[C \models_n \{P\} c \{Q\}.\]

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

\[
\frac{\{P\} c \{Q'\} \quad \forall s t z, (Q' z s \rightarrow Q z s)}{\{P\} c \{Q\}}
\]

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

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: \( \forall C P c Q n, \)
\[ \text{In} (P,c,Q)\) \( \Rightarrow \)
\[ C \models \{ P \} c \{ Q \}. \]

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

\[ \forall n', (P, \text{call} x, Q) :: C \models \{ P \} \text{procs} x \{ Q \} \]
\[ \forall n, C \models \{ P \} \text{call} x \{ Q \} \]

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: \( \forall C P x Q, \)
\[ (\forall n', (P, \text{call} x, Q) :: C \models \{ P \} \text{procs} x \{ Q \} ) \rightarrow \]
\[ (\forall n, C \models \{ P \} \text{call} x \{ Q \} ). \]

**Mutually recursive procedure calls**

Several mutually recursive procedures are justified using a Hoare judgement. The judgement reasons over two contexts, \( C_1 \) and \( C_2 \). The Hoare triples in \( C_1 \) are assumed to hold, whereas we want to prove the Hoare triples in \( C_2 \).

To justify that a judgement \( C_2 \) holds under \( C_1 \), two preconditions are required: Firstly, \( C_2 \) only contains \( \text{call} x \) statements. Secondly, each Hoare triple for a procedure \( x \) in \( C_2 \) is justified under the combined context of \( C_1 \) and \( C_2 \). This means that, like for the normal recursive procedures, we assume for recursive calls that the Hoare triple actually holds.

\[ \forall n' P Q x, \{ P \} \text{call} x \{ Q \} \in C_2 \rightarrow C_1 \cup C_2 \models \{ P \} \text{procs} x \{ Q \} \]
\[ \forall n, C_1 \models \{ P \} \text{call} x \{ Q \} \]

**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 $C \vdash \{ P \} c \{ Q \}$.

**Inductive syn_ext_triple : context → Assertion → com → Assertion → Prop :=**

- $SSkip : \forall C P, C \vdash \{ P \} CSkip \{ P \}$
- $SAsgn : \forall C P V a, C \vdash \{ \text{assn_sub P V a} \} V := a \{ P \}$

... where "$C \vdash \{ P \} c \{ Q \}" := (syn_ext_triple C P c Q)

with **syn_judgement : context → context → Prop :=**

... where "$C1 \vdash C2$" := (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 : \forall C P c Q,**

$C \vdash \{ P \} c \{ Q \} \rightarrow$

$\forall n, C \models_n \{ P \} c \{ Q \}.$

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 : \forall C P c Q,**

$C \vdash \{ P \} c \{ Q \} \rightarrow$

$C \models \{ P \} c \{ Q \}.$

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

**Lemma hoare_jsoundness_n : \forall C1 C2,**

$C1 \vdash C2 \rightarrow$

$\forall n, C1 \models_n C2.$

**Theorem hoare_jsoundness : \forall C1 C2,**

$C1 \vdash C2 \rightarrow C1 \models C2.$

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),
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 \( \neg b \) has to be true.

Lemma invert_if: \( \forall b \, c_1 \, c_2 \, (P \, Q: \text{Assertion}), \)
\[
\{ P \} \text{if} \ b \ \text{then} \ c_1 \ \text{else} \ c_2 \{ Q \} \rightarrow \\
\{ \text{fun} \ z \ st \Rightarrow P \ z \ st \land \text{bassn} \ b \ z \ st \} \ c_1 \{ Q \} \\
\land \{ \text{fun} \ z \ st \Rightarrow P \ z \ st \land \neg(\text{bassn} \ b \ z \ st) \} \ c_2 \{ Q \}.
\]

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 \land Pif z st) || (\neg(bassn b z st) \land Pelse z st)
| while b inv P do c1 end ⇒
  fun z st ⇒ P z st \land forall_bool (fun z ⇒
    forall_bool (fun st' ⇒ implb (P st' \land \neg(eeval_bool st' b)) (Q z st'))
    \land (implb (P st' \land eeval_bool st' b) (VCgen C c1 (fun z ⇒ P z st'))))
| assert b ⇒
  fun z st ⇒ b st \land 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 \land 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 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 \( \text{VCgen} \) 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.

\[ \forall C \ c \ Q \ n, \quad C \models_n \{ \text{VCgen} \ C \ c \ Q \} \ c \{ \ Q \}. \]

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

\[ \forall C \ c \ Q, \quad C \models \{ \text{VCgen} \ C \ c \ Q \} \ c \{ \ Q \}. \]

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.

\[ \forall c \ Q \ C, \quad \text{sem\_context} \ C \rightarrow \text{valid\_formula} (\text{VCgen} \ C \ c \ Q) \rightarrow \text{no\_error} \ c. \]

Furthermore, we define a verification condition generation for procedures, which does two things. For once it ensures that the context contains only \text{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.

\[ \text{Definition} \ \text{VCgen\_procs} \ (C:\text{context}) : \text{Prop} := \]
\[ (\forall P \ c \ Q, \text{In} \ (P, c, Q) \ C \rightarrow \exists x, \ c = \text{call} \ x) \land \]
\[ \forall n, \forall P, \forall Q, \text{In} \ (P, \text{call} \ n, Q) \ C \rightarrow \]
\[ P \rightarrow (\text{VCgen} \ C \ (\text{Args\_procs} \ n) \ Q). \]

Soundness of the \text{VCgen\_procs} function is defined by stating that, if \text{VCgen\_procs} generates a valid formula for the context \( C \), then \( C \) is indeed a valid context.

\[ \text{Lemma} \text{VCgen\_procs\_sound} : \forall C, \]
\[ \text{valid} (\text{VCgen\_procs} \ C) \rightarrow \text{sem\_context} \ C. \]

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 \( \text{translateT} \) function takes a type and the variable that should be tested, and it returns an assertion (a function from \( \text{Env} \) to \( \text{bool} \)).

\[
\text{Definition} \quad \text{translateT} \quad (t: \text{MType}) \quad (x: \text{string}) \quad : \quad \text{Imp} : \quad \text{Assertion} := \\
\quad \text{fun} \quad \text{env} \Rightarrow F \quad (t) \quad (\text{env} \ x) \quad \text{env}.
\]

\( F \) 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 typing 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).

\[
\text{Definition} \quad \text{translateT\_err} \quad (t: \text{MType}) \quad (x: \text{string}) \quad : \quad \text{Imp} : \quad \text{Assertion} := \\
\quad \text{fun} \quad \text{env} \Rightarrow \neg \quad (W \quad (t) \quad (\text{env} \ x) \quad \text{env}).
\]

The \( \text{translateT\_err} \) function has the same signature as \( \text{translateT} \). Our translation adds an assertion before every type-test, verifying that \( \text{translateT\_err} \) holds.

4.2. Translation

Below we introduce the actual translation function \( \text{translate} \). 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 \( \text{translate} \) 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
  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′))
        | OOr ⇒ CAss outvar (EOOr (EVar e1′) (EVar e2′))
        | OAnd ⇒ CAss outvar (EAnd (EVar e1′) (EVar e2′))
        | OMinit ⇒ CAss outvar (EMinit (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 e1 ⇒
4.2. TRANSLATION

let ent' := proj1_sig(fresh (avoid)) in
let temp' := proj1_sig (ent'::avoid)) in
ent' := EValue_vEmpty;
(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 OGe (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 OGe (Var v) (Value (v_int 5)))) "fresh");
"outx" := EIn
(translateT
 (Refine v Any (BinOp OGe (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.

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 :=**

\[ \text{In} (\text{Value}(v_{\text{int}} 5)) (\text{Refine} x\text{ Any}(\text{BinOp OGt} \text{ Var} x \text{ (Value} (v_{\text{int}} 5)))\).\]

**Definition incom_translated :=**

"fresh" :\text{ EValue} (v_{\text{int}} 5);
\[\text{assert translateT} \_\text{err} (\text{Refine} x\text{ Any} (\text{BinOp OGt} \text{ Var} x \text{ (Value} (v_{\text{int}} 5)))) "fresh";\]
"outx" :\text{ Eln} (\text{translateT} (\text{Refine} x\text{ Any}(\text{BinOp OGt} \text{ Var} x \text{ (Value} (v_{\text{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 \Rightarrow true \} incom_translated \{ fun z st \Rightarrow \text{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 :** \( \forall \text{ ex r outx avoid st}, \)
\[ (\forall s \text{ arg exp}, (\text{arg,exp}) = \text{functions} s \rightarrow \)
contains_impure_refinements \exp = \text{false}) \rightarrow
(\forall \ arg, \ (\arg, \exp) = \text{functions} \ s \rightarrow
\fv\exp = \text{nil} \lor \fv\exp = (\arg : \text{nil})) \rightarrow
\text{Eval} (\text{subst}\_\text{state} \ ex \ \st) \ r \rightarrow
\text{contains_impure_refinements} (\text{subst}\_\text{state} \ ex \ \st) = \text{false} \rightarrow
(r = \text{Error} \rightarrow (\text{translate avoid ex outx}) / \st \rightsquigarrow \text{CError})
\land
(\forall v, r = (\text{Return} v) \rightarrow
\exists \st', \ (\text{translate avoid ex outx}) / \st \rightsquigarrow \text{CReturn} \ \st' \land \st' \ \text{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:** \( \forall c \ st \ outx, (\text{backup outx in } c) / \st \rightsquigarrow \text{CError} \leftrightarrow c / \st \rightsquigarrow \text{CError}. \)

**Lemma backup_ret:** \( \forall c \ st \ st' \ outx \ v, c / \st \rightsquigarrow \text{CReturn} \ st' \rightarrow
st' \ \text{outx} = v \rightarrow
(\text{backup outx in } c) / \st \rightsquigarrow \text{CReturn} (\text{env_update} \ st \ outx \ v). \)

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:

\[
\text{eval_value} : \forall v,
\text{Eval (Value v) (Return v)}
\]

The generated proof obligation is:

\( \exists \st' : \text{Env}, \)
(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: ∀ 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) (les′ ++ (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_accumLets : ∀ 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
4.4. SOUNDNESS PROOF

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 : \forall ex r outx avoid st,
fv_exp ex = nil →
(\forall s arg exp, (arg,exp) = functions s → contains_impure_refinements exp = false) →
(\forall 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)
∧
(\forall v, r = (Return v) →
\exists 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 : \forall avoid ex outx P Q,
fv_exp ex = nil →
(\forall s arg exp, (arg,exp) = functions s → contains_impure_refinements exp = false) →
(\forall 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 strongest¹ 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.

¹It 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. Axiomatization

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, \texttt{bool} and \texttt{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 \texttt{General} sort.

5.4.1. The \texttt{General} sort

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

In the axiomatisation \texttt{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 \texttt{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.

\begin{verbatim}
type String;

type General;

// Constructors
function G_Integer(int) returns (General);
function G_Text(String) returns (General);
function G_Logical(bool) returns (General);
const G_Null : General;
\end{verbatim}

The sort \texttt{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 \texttt{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:

\begin{verbatim}
const unique str_Hallo : String; // the string "Hallo"
const unique str_foo : String; // the string "foo"
\end{verbatim}

As a next step we define a number of tags, which are used to identify what content a variable of sort \texttt{General} has. There is a function symbol \texttt{get_GTag} that returns a tag for a specific \texttt{General} variable. A tag is a unique constant of sort \texttt{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.

```plaintext
// 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).

```plaintext
// 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.

```plaintext
// 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 entries
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:

\[
\text{assert } (\text{Integer}(\text{O}_\text{Sum}(v_{true}, v_{false})));\]

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 \(\text{O}_\text{Sum}\) should be defined to return a Result instead of a Value.

\[
\text{EPlus } a \ b \Rightarrow \text{LBind } (\text{eeval } st \ a) (\text{fun } x_1 \Rightarrow
\text{LBind } (\text{eeval } st \ b) (\text{fun } x_2 \Rightarrow
\text{if } \text{In}_{\text{Integer}} x_1 \land \text{In}\_\text{Integer} x_2 \text{ then Return } (\text{O}_\text{Sum} x_1 x_2) \text{ else Error}))\]

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 type-checker 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.

<table>
<thead>
<tr>
<th>Type-checking success</th>
<th>Type-checking failure</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original test suite</td>
<td>76</td>
<td>33</td>
</tr>
<tr>
<td>DMinor passes</td>
<td>66</td>
<td>33</td>
</tr>
<tr>
<td>DVerify passes</td>
<td>62</td>
<td>31</td>
</tr>
</tbody>
</table>

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.
### 5.5. Quantitative Comparison of DMinor and DVerify

<table>
<thead>
<tr>
<th></th>
<th>Overall time</th>
<th>Internal time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Avg(^1)</td>
</tr>
<tr>
<td>DMinor</td>
<td>0.69s</td>
<td>1.13s</td>
</tr>
<tr>
<td>DVerify</td>
<td>1.56s</td>
<td>1.81s</td>
</tr>
<tr>
<td>Translation only</td>
<td>0.80s</td>
<td>0.97s</td>
</tr>
<tr>
<td>Boogie only</td>
<td>0.76s</td>
<td>0.85s</td>
</tr>
</tbody>
</table>

\(^1\) 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 ||
```

```
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.
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 \cite{RT06b} and directly fed to Z3 with the proof obligation. Our axiomatisation is in Boogie language and Boogie translates it to Simplify syntax \cite{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 practice 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. Regarding 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

<table>
<thead>
<tr>
<th>Area</th>
<th>Our verification approach (DVerify)</th>
<th>Type-checking approach (DMinor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification cond.</td>
<td>Weakest precondition</td>
<td>Bidirectional type-checking (type synthesis ≈ strongest postcondition)</td>
</tr>
<tr>
<td>generation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Formulae discharged</td>
<td>One per postcondition/</td>
<td>One per subtyping test (smaller, but more obligations)</td>
</tr>
<tr>
<td></td>
<td>assertion (larger, but less obligations)</td>
<td></td>
</tr>
<tr>
<td>Backend</td>
<td>Boogie + SMT-Solver (Z3)</td>
<td>SMT-Solver (Z3)</td>
</tr>
<tr>
<td>Loop invariants</td>
<td>In principle Boogie could infer some</td>
<td>For some constructs they are inferred (but not for accumulates)</td>
</tr>
<tr>
<td></td>
<td>(even for accumulates)</td>
<td></td>
</tr>
<tr>
<td>Error reporting</td>
<td>Abstract trace</td>
<td>Counterexample</td>
</tr>
<tr>
<td>Performance (overall)</td>
<td>slightly worse (more initialization)</td>
<td>slightly better</td>
</tr>
<tr>
<td>Performance (internal)</td>
<td>similar</td>
<td>similar</td>
</tr>
<tr>
<td>Scalability</td>
<td>possibly better</td>
<td>possibly worse</td>
</tr>
<tr>
<td>Precision (practise)</td>
<td>similar</td>
<td>similar</td>
</tr>
<tr>
<td>Completeness (theory)</td>
<td>possibly better</td>
<td>possibly worse (type system)</td>
</tr>
</tbody>
</table>
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 \[\text{BCD}^+\text{06}\]. 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 \[\text{GMR09}\] and \[\text{BHMR07}\].

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. \[\text{BGHL10}\], 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 assertion 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.

\[ \text{Theorem translation_complete : } \forall \ ex r \ outx \ avoid \ st, \]
\[ (\forall s \ arg \ ex \ v, (arg,ex) = \text{functions } s \rightarrow \]
\[ \text{contains_impure_refinements (subst_exp ex v arg) = false} \rightarrow \]
\[ (\forall s \ arg \ ex, (arg,ex) = \text{functions } s \rightarrow \]
\[ \text{fv_exp ex = nil} \lor \text{fv_exp ex = (arg:nil)} \rightarrow \]
\[ \text{contains_impure_refinements (subst_state ex st) = false} \rightarrow \]
\[ (((\text{translate avoid ex outx}) / st \rightsquigarrow \text{CError}) \land \]
\[ \forall st', \]
\[ (\text{translate avoid ex outx}) / st \rightsquigarrow \text{CReturn} \ st' \rightarrow \]
\[ \text{Eval (subst_state ex st) Error} \land \]
\[ \forall st', \]
\[ (\text{translate avoid ex outx}) / st' \rightsquigarrow \text{CReturn} \ st' \rightarrow \]
\[ \text{Eval (subst_state ex st) (st' outx)}. \]

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 \cite{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 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 \cite{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 \cite{RGP08}.
Bibliography


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].

\[
\text{Inductive Eval: Exp} \rightarrow \text{Result} \rightarrow \text{Prop} := \\
| \text{eval}_{\text{value}}: \forall v, \\
\quad \text{Eval (Value} v \text{) (Return} v \text{)} \\
| \text{eval}_{\text{un}_{\text{op}}_1}: \forall e \text{ op}, \\
\quad \text{Eval e Error} \rightarrow \\
\quad \text{Eval (UnOp op e) Error} \\
| \text{eval}_{\text{un}_{\text{op}}_2}: \forall e \text{ op v}, \\
\quad \text{Eval e (Return} v \text{) \quad (} \exists v', \text{un_op_eval op v v'} \text{)} \rightarrow \\
\quad \text{Eval (UnOp op e) Error} \\
| \text{eval}_{\text{un}_{\text{op}}_3}: \forall e \text{ op v v'}, \\
\quad \text{Eval e (Return} v \text{) \quad un_op_eval op v v'} \rightarrow \\
\quad \text{Eval (UnOp op e) (Return} v' \text{)} \\
| \text{eval}_{\text{bin}_{\text{op}}_1_1}: \forall e_1 e_2 \text{ op}, \\
\quad \text{Eval e_1 Error} \rightarrow \\
\quad \text{Eval (BinOp op e_1 e_2) Error} \\
| \text{eval}_{\text{bin}_{\text{op}}_1_2}: \forall e_1 e_2 \text{ op v}, \\
\quad \text{Eval e_1 (Return} v \text{) \quad Eval e_2 Error} \rightarrow \\
\quad \text{Eval (BinOp op e_1 e_2) Error} \\
| \text{eval}_{\text{bin}_{\text{op}}_2}: \forall e_1 e_2 \text{ op v1 v2}, \\
\quad \text{Eval e_1 (Return} v_1 \text{) \quad Eval e_2 (Return} v_2 \text{) \quad (} \exists v', \text{bin_op_eval op v1 v2 v'} \text{)} \rightarrow \\
\quad \text{Eval (BinOp op e_1 e_2) Error} \\
| \text{eval}_{\text{bin}_{\text{op}}_3}: \forall e_1 e_2 \text{ op v1 v2 v'}, \\
\quad \text{Eval e_1 (Return} v_1 \text{) \quad Eval e_2 (Return} v_2 \text{) \quad bin_op_eval op v1 v2 v'} \rightarrow \\
\quad \text{Eval (BinOp op e_1 e_2) (Return} v' \text{)}
\]
| 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 : \forall e_1 e_2 v_1, 
  Eval e_1 (Return v_1) → 
  Eval e_2 Error → 
  Eval (Add e_1 e_2) Error

 eval_add_2_2 : \forall e_1 e_2 v_1 v_2, 
  Eval e_1 (Return v_1) → 
  Eval e_2 (Return v_2) → 
  is_C v_2 = false → 
  Eval (Add e_1 e_2) Error

 eval_add_3 : \forall e_1 e_2 v_1 v_2, 
  Eval e_1 (Return v_1) → 
  Eval e_2 (Return v_2) → 
  is_C v_2 = true → 
  Eval (Add e_1 e_2) (Return (v_add v_1 v_2))

 eval_appl_1 : \forall f e, 
  Eval e Error → 
  Eval (App f e) Error

 eval_appl_2 : \forall 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 : \forall x y e_1 e_2 e_3, 
  Eval e_1 Error → 
  Eval (Acc x e_1 y e_2 e_3) Error

 eval_accum_1_2 : \forall x y e_1 e_2 e_3 v_1, 
  Eval e_1 (Return v_1) → 
  is_C v_1 = false → 
  Eval (Acc x e_1 y e_2 e_3) Error

 eval_accum_lets : \forall x e_1 y e_2 e_3 r v_1 vs, 
  Eval e_1 (Return v_1) → 
  is_C v_1 = true → 
  (Permutation vs (vb_to_vs (out_C v_1))) → 
  Eval (Let y e_2 (let_seq y (List.map (fun v ⇒ subst_exp e_3 v x) vs) (Var y))) r → 
  Eval (Acc x e_1 y e_2 e_3) r

 eval_test_wrong : \forall e T, 
  Eval e Error → 
  Eval (In e T) Error

 eval_test_any : \forall e v, 
  Eval e (Return v) → 
  Eval (In e Any) (Return v_tt)

 eval_test_integer : \forall e v, 
  Eval e (Return v) → 
  Eval (In e Integer) (Return (v_logical (In_Integer v))))
APPENDIX A. DMINOR BIG-STEP SEMANTICS

| 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 (Entity l T)) (Return v_ff)
| eval_test_coll_1 : ∀ e T v,
| Eval e (Return v) →
| is_C v = false →
| Eval (In (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 (\textbf{Return} v1) \rightarrow
eval \text{test\_refine\_2} : \forall e1 x T e2 v1 v' v2,
Eval e1 (\textbf{Return} v1) \rightarrow
Eval (In (Value v1) \top) (\textbf{Return} v') \rightarrow
Eval (subst\_exp e2 v1 x) \textbf{Error} \rightarrow
Eval (In e1 (\textbf{Refine} x T e2)) \textbf{Error}

\mid eval \text{test\_refine\_3} : \forall e1 x T e2 v1 v' v2,
Eval e1 (\textbf{Return} v1) \rightarrow
Eval (In (Value v1) \top) (\textbf{Return} v') \rightarrow
Eval (subst\_exp e2 v1 x) (\textbf{Return} v2) \rightarrow
In\_Logical v' \land In\_Logical v2 = \text{false} \rightarrow
Eval (In e1 (\textbf{Refine} x T e2)) \textbf{Error}

Eval e1 (\textbf{Return} v1) \rightarrow
Eval (In (Value v1) \top) (\textbf{Return} v') \rightarrow
Eval (subst\_exp e2 v1 x) (\textbf{Return} v2) \rightarrow
In\_Logical v' \land In\_Logical v2 = \text{true} \rightarrow
Eval (In e1 (\textbf{Refine} x T e2)) (\textbf{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.

<table>
<thead>
<tr>
<th>File</th>
<th>Lines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lib.v</td>
<td>374</td>
<td>This file contains basic lemmas, which are useful, but not present in the Coq library.</td>
</tr>
<tr>
<td>Imp.v</td>
<td>658</td>
<td>In Imp.v we give our while language and the operational semantics. A detailed explanation is given in Section 3.2.</td>
</tr>
<tr>
<td>Hoare.v</td>
<td>1014</td>
<td>This file contains our Hoare logics, which is described in Section 3.3.</td>
</tr>
<tr>
<td>WP.v</td>
<td>275</td>
<td>The weakest precondition is explained in Section 3.4.</td>
</tr>
<tr>
<td>VCgen.v</td>
<td>245</td>
<td>The file VCgen.v contains the verification condition generator, which is described in detail in Section 3.5.</td>
</tr>
<tr>
<td>Sets.v</td>
<td>97</td>
<td>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.</td>
</tr>
<tr>
<td>SubstState.v</td>
<td>749</td>
<td>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.</td>
</tr>
<tr>
<td>Translate.v</td>
<td>448</td>
<td>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.</td>
</tr>
<tr>
<td>Soundness.v</td>
<td>1581</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 BigStepSemantics.

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

Theorem env_update_neq : \forall V2 V1 n st,
\text{beq} \_\text{str} V2 V1 = \text{false} \rightarrow
(env_update st V2 n) V1 = (st V1).

Definition beq_env (env:Env) env2 :=
forall bool (fun x \Rightarrow \text{syn_beq_val}(env x) (env2 x)).

Lemma beq_env_true_x : \forall env1 env2 x,
\text{beq}_\text{env} env1 env2 = \text{true} \rightarrow
env1 x = env2 x.

Lemma beq_env_true : \forall env1 env2,
\text{beq}_\text{env} env1 env2 = \text{true} \rightarrow
env1 = env2.

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

Theorem env_update_same : \forall x1 k1 k2 (f: Env),
f k1 = x1 \rightarrow
(env_update f k1 x1) k2 = f k2.

Theorem env_update_same_fun : \forall x1 k1 (f: Env),
f k1 = x1 \rightarrow
(env_update f k1 x1) = f.

Theorem env_update_permute : \forall x1 x2 k1 k2 k3 f,
\text{beq_str} k2 k1 = \text{false} \rightarrow
(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 : \forall x1 x2 k1 k2 f,
\text{beq_str} k2 k1 = \text{false} \rightarrow
(env_update (env_update f k2 x1) k1 x2) = (env_update (env_update f k1 x2) k2 x1).

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

Theorem env_update_lookup' : \forall x1 k1 (f:Env) k,
beq_str k1 k = true \rightarrow 
env_update f k1 x1 k = x1.

Theorem env_update_ignore : \( \forall x1 k1 k2 (f:Env), \)
beq_str k1 k2 = false \rightarrow 
env_update f k1 x1 k2 = f k2.

Theorem env_update_VAR : \( \forall st a n0 x n1 y, \)
beq_str a x = false \rightarrow 
(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 : \( \forall st a n0 x n1, \)
beq_str a x = false \rightarrow 
(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 \rightarrow bool.

B.1.2. Expressions

Inductive expr : Type :=
| EValue : Value \rightarrow expr
| EVar : string \rightarrow expr
| ECollAdd : expr \rightarrow expr \rightarrow expr
| ECollRem : expr \rightarrow expr \rightarrow expr
| ECollEmpty : expr \rightarrow expr
| EEntUpd : string \rightarrow expr \rightarrow expr \rightarrow expr
| EDot : expr \rightarrow string \rightarrow expr
| EIn : Assertion \rightarrow expr
| EOOr : expr \rightarrow expr \rightarrow expr
| ENot : expr \rightarrow expr
| EEq : expr \rightarrow expr \rightarrow expr
| ELt : expr \rightarrow expr \rightarrow expr
| EGt : expr \rightarrow expr \rightarrow expr
| EPlus : expr \rightarrow expr \rightarrow expr
| EMinus : expr \rightarrow expr \rightarrow expr
| ETimes : expr \rightarrow expr \rightarrow expr
| EAnd : expr \rightarrow expr \rightarrow 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 ln_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 ln_Integer x1 ∧ ln_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 ln_Integer x1 ∧ ln_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 ln_Integer x1 ∧ ln_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 ln_Integer x1 ∧ ln_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 ln_Integer x1 ∧ ln_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 ln_Logical x1 ∧ ln_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 ln_Logical x1 ∧ ln_Logical x2 then Return (O_Or x1 x2) else Error)))))
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 \rightarrow \text{CReturn } st

| CEWhileLoop : \forall \text{ st st' st'' b1 b a1 c1},
eeval st b = \text{Return } b1 \rightarrow
\text{syn\_beq\_val b1 v.tt = true } \rightarrow
c1 / st \rightarrow \text{CReturn } st' \rightarrow
(while b inv a1 do c1 end) / st' \rightarrow st'' →
(while b inv a1 do c1 end) / st \rightarrow st'' →

| CEWhileLoopErr : \forall \text{ st b1 b a1 c1},
eeval st b = \text{Return } b1 \rightarrow
\text{syn\_beq\_val b1 v.tt = true } \rightarrow
c1 / st \rightarrow \text{CError } \rightarrow
(while b inv a1 do c1 end) / st \rightarrow \text{CError}

| CEAssert : \forall \text{ (st:Env) (b:Assertion)},
b st = true \rightarrow
(assert b) / st \rightarrow \text{CReturn } st

| CEAssertErr : \forall \text{ st b},
b st = false \rightarrow
(assert b) / st \rightarrow \text{CError}

| CEPick : \forall \text{ st x xc v},
is\_C (st xc) = true \rightarrow
v\_mem v (st xc) = true \rightarrow
(x := \text{pick xc}) / st \rightarrow \text{CReturn } (\text{env\_update st x v)}

| CECall : \forall \text{ st st' pn},
\text{Args.procs pn / st \rightarrow st' } \rightarrow
(call pn) / st \rightarrow \text{st'}

| CEBackup : \forall \text{ st st' v c},
c / st \rightarrow (\text{CReturn } st') \rightarrow
(backup v in c) / st \rightarrow \text{CReturn } (\text{env\_update st v (st' v)})

| CEBackupErr : \forall \text{ st v c},
c / st \rightarrow \text{CError } \rightarrow
(backup v in c) / st \rightarrow \text{CError}

where "c1 / st \rightarrow st'"::= (\text{ceval st c1 st')}.

\text{Tactic Notation "ceval\_cases" tactic(first) tactic(c) := first; [}
c "\text{CESkip}\] | c "\text{CEAss}\] | c "\text{CSeq}\] | c "\text{CSeqErr}\] |
c "\text{CEIfTrue}\] | c "\text{CEIfFalse}\] | c "\text{CEWhileEnd}\] | c "\text{CEWhileLoop}\] |
c "\text{CEWhileLoopErr}\] | c "\text{CEAss}\] | c "\text{CEAssErr}\] | c "\text{CEPick}\] |
c "\text{CECall}\] | c "\text{CEBackup}\] | c "\text{CEBackupErr}\].

\text{Lemma seq\_ass : \forall \text{ c1 c2 c3 st st'},}
\text{(c1; c2; c3) / st \rightarrow st' } \leftrightarrow
\text{((c1; c2; c3) / st \rightarrow 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) ! nidx,
    syn_beq_val (st a1) v tt = true →
    (CAss ! a1) / st — nidx <— CResult (env_update st ! 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 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 := \( \forall (z: \text{Args.ZType}) (st:\text{Env}), f \, z \, st = \text{true} \).

Lemma hoare triple pre valid no error : \( \forall c \, P \, Q, \)
valid formula P \( \rightarrow \)
\( \{ P \} \, c \{ Q \} \rightarrow \text{no_error} \, c. \)

Definition valid (H:Prop) := H.

Definition hoare triple n n (P:Assertion) (c:com) (Q:Assertion) : Prop :=
\( \forall \, st \, st', \)
\( c / st \rightarrow n \rightarrow st' \)
\( \rightarrow (\forall z: \text{Args.ZType}, P \, z \, st = \text{true} \)
\( \rightarrow \text{check_post} \, Q \, z \, st'). \)

Notation "\( n \) "\( \{ P \} \, c \{ Q \} " := (hoare triple n n P c Q) (at level 89).

Definition context := list (Assertion \times \text{com} \times \text{Assertion}).

Fixpoint sem context ct { struct ct } :=
match ct with
| nil \Rightarrow True
| (P,c,Q)::cl \Rightarrow \{ P \} \, c \{ Q \} \land \text{sem context} \, cl
end.

Fixpoint sem context n n ct { struct ct } :=
match ct with
| nil \Rightarrow True
| (P,c,Q)::cl \Rightarrow \{ n \} \, c \{ Q \} \land \text{sem context} \, n \, n \, cl
end.

Lemma sem context n in : \( \forall \, n \, ct \, P \, c \, Q, \)
\( \text{ln} \, (P,c,Q) \, ct \rightarrow \)
\( \text{sem context} \, n \, n \, ct \rightarrow \)
\( \{ n \} \, c \{ Q \}. \)

Definition ext hoare triple C P c Q :=
\text{sem context} \, C \rightarrow \{ 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 :=
\text{sem context} \, n \, n \, C \rightarrow \{ n \} \, 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 :=
\text{sem context} \, C1 \rightarrow \text{sem context} \, C2.

Notation "C1 \( \{ P \} \, c \{ Q \} " := (ext hoare judgement C1 C2) (at level 85).

Definition ext hoare judgement n n (n:nat) C1 C2 :=
\text{sem context} \, n \, n \, C1 \rightarrow \text{sem context} \, n \, n \, C2.

Notation "C1 \( \{ P \} \, c \{ Q \} " := (ext hoare judgement n n C1 C2) (at level 85).

Lemma hoare iff n : \( \forall P \, Q, \{ P \} \, c \{ Q \} \leftrightarrow \forall n, \{ n \} \, c \{ Q \}. \)
Lemma $\text{sem}_\iff n : \forall C, \text{sem\_context} C \leftrightarrow \forall n, \text{sem\_context}_n n C$.

Lemma $\text{ext\_hoare\_if}_n : \forall C \; P \; c \; Q, \forall n, (C \vdash_n P \{Q\}) \rightarrow (C \vdash_n \{P\} \{Q\})$.

Lemma $\text{no\_context\_nil}_n : \forall P \; R \; c \; n, \text{nil} \vdash_n \{P\} \{R\} \leftrightarrow \vdash_n \{P\} \{R\}$.

Definition $\text{eeval\_return} env e :=$

| Error | $\Rightarrow$ env$\null$ |
| Return | $x \Rightarrow x$ |

\section*{B.2.2. Lemmas for deriving semantic Hoare triples}

\textbf{Skip command}

Theorem $\text{hoare\_skip}_n : \forall C \; P \; n, C \vdash_n \{P\} \{\text{skip}\} P$.

\textbf{Assignment command}

Definition $\text{assn\_sub} P x a : \text{Assertion} :=$

| fun ($z : \text{Args.ZType}$) (st : $\text{Env}$) $\Rightarrow$ P $z$ (st$\_update$ st x (eeval$\_return$ st a)). |

Theorem $\text{hoare\_asgn}_n : \forall C \; P \; x \; a \; n, C \vdash_n \{\text{assn\_sub} P x a\} \{x := a\} \{P\}$.

Theorem $\text{hoare\_asgn\_eq} : \forall C \; P \; P' \; V \; a \; n, P' = \text{assn\_sub} P V a$

$\rightarrow C \vdash_n \{P'\} \{V := a\} \{P\}$.

\textbf{Sequence command}

Theorem $\text{hoare\_seq}_n : \forall C \; P \; Q \; R \; c_1 \; c_2 \; n,$

$C \vdash_n \{Q\} \{c_2\} \{R\}$$

$\rightarrow C \vdash_n \{P\} \{c_1\} \{Q\}$

$\rightarrow C \vdash_n \{P\} \{c_1; c_2\} \{R\}$.

Corollary $\text{hoare\_seq} : \forall C \; P \; Q \; R \; c_1 \; c_2,$
Lemma hoare_seq_ass: \forall c_1 c_2 c_3 P Q, 
\{ P \} (c_1; c_2; c_3) \{ Q \} \leftrightarrow \{ P \} ((c_1; c_2); c_3) \{ Q \}.

If command

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

Lemma bassn_z_independent: \forall b (z z' : Args.ZType) st, 
bassn b z st = bassn b z' st.

Lemma bexp_eval_true : \forall b st (z:Args.ZType), 
syn_beq_val (st b) v_tt = true ⇒ \(\text{bassn b} z st = true\).

Lemma bexp_eval_true_inv : \forall b st (z:Args.ZType), 
(bassn b) z st = true ⇒ syn_beq_val (st b) v_tt = true.

Lemma bexp_eval_false : \forall b st (z:Args.ZType),
syn_beq_val (st b) v_tt = false ⇒ \(\text{bassn b} z st = false\).

Lemma bexp_eval_false_inv : \forall b st (z:Args.ZType),
(bassn b) z st = false ⇒ syn_beq_val (st b) v_tt = false.

Theorem hoare_if_n : \forall C P Q b c_1 c_2 n,
C \models \{ \text{fun} (z:Args.ZType) st ⇒ (P z st) \land (\text{bassn b} z st) \} c_1 \{ Q \} →
C \models \{ \text{fun} (z:Args.ZType) st ⇒ (P z st) \land (\neg (\text{bassn b} z st)) \} c_2 \{ Q \} →
C \models \text{if b then} c_1 \text{ else } c_2 \{ Q \}.

While command

Definition eeval_bool st e :=
  match eeval st e with
  \text{Return} v ⇒ syn_beq_val v v_tt
  \text{Error} ⇒ false
end.

Theorem hoare_while_n : \forall C P b c z n,
C \models \{ \text{fun} z st ⇒ P z st \land \text{eeval_bool st} b \} \{ P \} →
C \models \{ P \} (\text{while} b \text{ inv} (\text{fun} st ⇒ P z st) \text{ do } c \text{ end})
\{ \text{fun} z st ⇒ P z st \land (\neg (\text{eeval_bool st} b)) \}. 
**Assert Command**

Theorem hoare_assert_n : ∀ C : 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 x c 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 : Assertion P P' Q Q' : Assertion) c n,
C ⊨ n { P'} c { Q'} →
(∀ 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: ∀ C (P Q Q' : Assertion) c n,
C | n {P} c {Q'} → (Q' → Q) → C | n {P} c {Q}.

Theorem hoare_consequence_post: ∀ (P Q Q' : Assertion) c,
{P} c {Q'} → (Q' → Q) → {P} c {Q}.

B.2.3. Procedure calls

Theorem hoare_context_n: ∀ C P c Q n,
In (P, c, Q) C → C | n {P} c {Q}.

Lemma ext_hoare_hoare: ∀ C1,
(∀ (P Q : Assertion) c, In (P, c, Q) C1 → C1 | {P} c {Q})

Lemma split_judgement: ∀ C1 C2 P c Q,
ext_hoare_judgement C1 ((P, c, Q) :: C2) ↔ C1 | {P} c {Q} ∧ (C1 | C2).

Lemma split_judgement_n: ∀ C1 C2 P c Q n,
C1 | n ((P, c, Q) :: C2) ↔ C1 | n {P} c {Q} ∧ (C1 | n C2).

Lemma hoare_call_lookup: ∀ C P Q x,
C | {P} call x {Q} ↔ C | {P} Args.procs x {Q}.

Lemma hoare_context_more: ∀ P c Q i1 i2,
i1 ≤ i2 → || = i2 {P} c {Q} → || = i1 {P} c {Q}.

Lemma sem_context_more: ∀ C i1 i2,
i1 ≤ i2 → sem_context_n i2 C → sem_context_n i1 C.

Lemma call_lookup_n: ∀ x st st' n,
(call x) / st → S n → st' ↔ Args.procs x / st → n → st'.

Lemma hoare_call_lookup_n: ∀ n P Q x,
|| = S(n) {P} call x {Q} ↔ || = n {P} Args.procs x {Q}.

Theorem hoare_call_simple_n: ∀ C P Q x,
(∀ n', (P, call x, Q) :: C | n' {P} Args.procs x {Q}) →
(∀ n, C | = n {P} call x {Q}).
Mutually recursive procedure calls

Lemma ext_judgement_if_n : \forall C1 C2,
(\forall n. C1 \equiv n C2) \rightarrow C1 \equiv C2.

Lemma sem_context_n_app : \forall n C1 C2,
sem_context n n (C1 ++ C2) \leftrightarrow sem_context n n C1 \land sem_context n n C2.

Lemma sem_context_0_call : \forall C,
(\forall P Q. \in (P,c,Q) C \rightarrow \exists x. c = call x) \rightarrow sem_context n 0 C.

Theorem hoare_call_n : \forall C1 C2,
(\forall P 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 \equiv n' \{ P \} Args.procs x \{ Q \}) \rightarrow
(\forall n. C1 \equiv n C2).

B.2.4. Syntactic Hoare triples

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

Inductive syn_ext_triple : context \rightarrow Assertion \rightarrow com \rightarrow Assertion \rightarrow Prop :=
| SSkip : \forall C P. C \vdash \{ P \} CSkip \{ P \}
| SASgn : \forall C P V a. C \vdash \{ assn_sub P V a \} V := a \{ P \}
| SSeq : \forall C c1 c2 P Q R. C \vdash \{ Q \} c2 \{ R \}
  \rightarrow C \vdash \{ P \} c1 \{ Q \}
  \rightarrow C \vdash \{ P \} c1;c2 \{ R \}
| SIf : \forall C c1 c2 P Q b. C \vdash \{ \text{fun (z:Args.ZType) st \Rightarrow andb (P z st) (bassn b z st)} \} c1 \{ Q \}
  \rightarrow C \vdash \{ \text{fun (z:Args.ZType) st \Rightarrow andb (P z st) (~ (bassn b z st))} \} c2 \{ Q \}
  \rightarrow C \vdash \{ P \} CIf b c1 c2 \{ Q \}
| SWhile : \forall C c P b z. C \vdash \{ \text{fun z st \Rightarrow P z st} \land (\text{eeval_bool st b}) \} c \{ P \}
  \rightarrow C \vdash \{ P \} while b inv (\text{fun st \Rightarrow P z st}) do \text{c end}
  \{ \text{fun z st \Rightarrow P z st} \land (~ (\text{eeval_bool st b}) \}
| SConsq : \forall C P P' c Q Q'. C \vdash \{ P' \} c \{ Q' \}
  \rightarrow (\forall st st',
    (\forall (z:Args.ZType), P' z st = true \rightarrow \text{check_post Q' z st'})
  \rightarrow (\forall (z:Args.ZType), P z st = true \rightarrow \text{check_post Q z st'})
  \rightarrow C \vdash \{ P \} c \{ Q \}
| SConsqPre : \forall C P P' c Q. C \vdash \{ P' \} c \{ Q \}
  \rightarrow (P \rightarrow P')
  \rightarrow C \vdash \{ P \} c \{ Q \}
| SConsqPost : \forall C P c Q Q'. C \vdash \{ P \} c \{ Q' \}
  \rightarrow (Q' \rightarrow Q)
  \rightarrow C \vdash \{ P \} c \{ Q \}
B.2. HOARE.V

| 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, ln (P, c, Q) C2 → ∃ x, c = call x) → (∀ (P : Assertion) x, ln (P, CALL x, Q) C2 → C1 ++ C2 ⊣ (P) Args.procs x {Q}) → C1 ⊣ C2
| STriple : ∀ C1 C2,
(∀ P c Q, ln (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 : \( \forall e3 v x s a0 s0 a1 a2, \)
\( \text{syn beq res} (\text{LBind} (R e3 (\text{env update} (\text{env update} a s a1) s0 a2)) \)
\( (\text{fun z : Value} \Rightarrow R e3 (\text{env update} (\text{env update} a s a0) s0 z))) \)
\( (\text{LBind} (R e3 (\text{env update} (\text{env update} a s a0) s0 a2)) \)
\( (\text{fun z : Value} \Rightarrow R e3 (\text{env update} (\text{env update} a s a1) s0 z))) = \text{true} \)
\( \rightarrow \text{syn beq res} (\text{LBind} (R (\text{subst exp} e3 v x) (\text{env update} (\text{env update} a s a1) s0 a2)) \)
\( (\text{fun z : Value} \Rightarrow R (\text{subst exp} e3 v x) (\text{env update} (\text{env update} a s a0) s0 z))) \)
\( (\text{LBind} (R (\text{subst exp} e3 v x) (\text{env update} (\text{env update} a s a0) s0 a2)) \)
\( (\text{fun z : Value} \Rightarrow R (\text{subst exp} e3 v x) (\text{env update} (\text{env update} a s a1) s0 z))) = \text{true} \).

Lemma pure subst exp : \( \forall s v e, \)
\( \text{is pure e} = \text{is pure (subst exp e s v)}. \)

Theorem beq str false : \( \forall (s s' : \text{string}), \)
\( \text{beq str s} s' = \text{false} \rightarrow s \neq s'. \)

Lemma list remove ignore : \( \forall l b s, \)
\( \text{In b l} \rightarrow \)
\( b \neq s \rightarrow \)
\( \text{In b (List.remove eq str dec s l)}. \)

Lemma rem intro : \( \forall x a s, \)
\( x \neq s \rightarrow \)
\( \text{In x (List.remove eq str dec s a)} \rightarrow \)
\( \text{In x a}. \)

Lemma rem remove : \( \forall x a s, \)
\( \text{In x a} \rightarrow \)
\( s \neq x \rightarrow \)
\( \text{In x (List.remove eq str dec s a)}. \)
### 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 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,
B.4. TRANSLATE.V

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_implexure_expressions (subst_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_implexure_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_implexure_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 Purity.
Require Export SubstState.

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′;
(\textbf{assert} (translateT (fst3 (op\_type\_bi o)) e1′));
(\textbf{assert} (translateT (snd3 (op\_type\_bi o)) e2′));
match o with
| \textbf{OEq} ⇒ CAss outvar (EEq (EVar e1′) (EVar e2′))
| \textbf{Olt} ⇒ CAss outvar (Elt (EVar e1′) (EVar e2′))
| \textbf{OAnd} ⇒ CAss outvar (EAnd (EVar e1′) (EVar e2′))
| \textbf{OOr} ⇒ CAss outvar (EOr (EVar e1′) (EVar e2′))
| \textbf{OPlus} ⇒ CAss outvar (EPlus (EVar e1′) (EVar e2′))
| \textbf{OMinus} ⇒ CAss outvar (EMinus (EVar e1′) (EVar e2′))
| \textbf{OTimes} ⇒ CAss outvar (ETimes (EVar e1′) (EVar e2′))
end

| \textbf{If} e1 e2 e3 ⇒
| let e1′ := proj1_sig(fresh(avoid)) in
| translate (e1′::avoid) e1 e1;
| CAssert (translateT Logical e1′);
| \textbf{if} e1′ \textbf{then} translate (e1′::avoid) e2 outvar
| \textbf{else} translate (e1′::avoid) e3 outvar

| \textbf{Let} x e1 e2 ⇒ translate avoid e1 x; translate avoid e2 outvar

| \textbf{In} e t ⇒
| let e′ := proj1_sig(fresh(avoid)) in
| translate (e′::avoid) e e′;
| (\textbf{assert} (translateT err t e′));
| outvar := EIn (translateT t e′)

| \textbf{Entity} el ⇒
| let ent′ := proj1_sig(fresh(avoid)) in
| let temp′ := proj1_sig(fresh(ent′::avoid)) in
| ent′ := EValue v_empty;
| (fix les\_to\_entity el :=
| match el with
| | \textbf{nil} ⇒ \textbf{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′

| \textbf{Dot} e l ⇒
| let e′ := proj1_sig(fresh(avoid)) in
| (translate (e′::avoid) e e′; \textbf{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(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′,
Lemma \textbf{un\_op\_ONot} : \forall v v',
un\_op\_eval O\textit{Not} v v' \rightarrow v' = O\_Not v.

Lemma \textbf{un\_op\_not\_logical} : \forall v,
\neg (\exists v' : \text{Value},\ un\_op\_eval \textit{ONot} v v') \rightarrow \text{In\_Logical} v = \text{false}.

Lemma \textbf{fresh\_is\_fresh} : \forall (a:\text{string}) b avoid,
\neg \text{In} b (a :: avoid) \rightarrow
b \neq a.

Lemma \textbf{fresh\_is\_fresh\_beq} : \forall (a:b avoid),
\neg \text{In} b (a :: avoid) \rightarrow
\text{beq\_str} b a = \text{false}.

Lemma \textbf{backup\_ret} : \forall \text{rest st st'} outx v,
\text{rest} / \text{st} \rightsquigarrow \text{CReturn st'} \rightarrow
\text{st'} outx = v \rightarrow
(\textbf{backup} outx in rest) / \text{st} \rightsquigarrow \text{CReturn} (\text{env\_update} st outx v).

Lemma \textbf{backup\_err} : \forall \text{rest st outx},
(\textbf{backup} outx in rest) / \text{st} \rightsquigarrow \text{CError} \leftrightarrow
\text{rest} / \text{st} \rightsquigarrow \text{CError}.

Lemma \textbf{lookup\_equiv} : \forall st st' outx s,
(outx := E\textit{Value} (s st)) / \text{st} \rightsquigarrow \text{CReturn st'} \leftrightarrow
(outx := E\textit{Var} s) / \text{st} \rightsquigarrow \text{CReturn st'}.

Lemma \textbf{forall\_holds} : \forall v2,
\text{forall}\_\textit{bool}(\text{fun} v : \text{Value} \Rightarrow \text{implb}(v\_\text{mem} v v2) \text{true}) = \text{true}.

Lemma \textbf{coll\_not\_empty} : \forall v v1,
v\_\text{mem} v v1 = \text{true} \rightarrow
O\_\textit{Not} (v\_\text{empty} v1) = v\_tt.

Lemma \textbf{coll\_has\_member} : \forall v1,
is\_C v1 = \text{true} \rightarrow
v\_\text{empty} v1 = v\_ff \rightarrow
\exists v, v\_\text{mem} v v1 = \text{true}.

Lemma \textbf{not\_in} : \forall (a: \text{string}) avoid,
\neg \text{In} a (x :: avoid) \rightarrow \text{beq\_str} a x = \text{false}.

Lemma \textbf{fresh\_not\_avoid} : \forall x avoid,
\neg \text{In} x (\text{avoid}) \rightarrow \text{beq\_str} x x = \text{false}.
Lemma vars : \forall x \ avoid \ outx \ st \ st',
translate (avoid) e outx / st \leadsto \texttt{CReturn} \ st' \rightarrow
outx \neq x \rightarrow
st \ x = st' \ x.

Lemma state_unchanged : \forall e \ avoid \ outx \ st \ st',
translate (avoid) e outx / st \leadsto \texttt{CReturn} \ st' \rightarrow
st' = env_update st outx (st' outx).

Lemma subst_state_trans' : \forall e2 \ a \ st \ st',
(\forall x, a<>x \rightarrow st \ x = st' \ x) \rightarrow
\ln a (fv_exp e2) \rightarrow
\lnot \ subst_state\_avoid\ e2 \ st \ set\_empty = subst_state\_avoid\ e2 \ st' \ set\_empty.

Lemma subst_state_trans'2 : \forall e2' \ b \ st \ st',
(\forall x, a<>x \rightarrow b<>x \rightarrow st \ x = st' \ x) \rightarrow
\ln a (fv_exp e2') \rightarrow
\lnot \ ln b (fv_exp e2') \rightarrow
subst_state\_avoid\ e2' \ st \ set\_empty = subst_state\_avoid\ e2' \ st' \ set\_empty.

Lemma subst_state_trans : \forall e2 \ e2' \ a \ avoid \ st \ st',
translate avoid e a / st \leadsto \texttt{CReturn} \ st' \rightarrow
\lnot \ ln a (fv_exp e2') \rightarrow
(e2 = subst_state\_avoid\ e2' \ st \ set\_empty \leftrightarrow e2 = subst_state\_avoid\ e2' \ st' \ set\_empty).

Lemma subst_state_trans_t : \forall t \ t' \ e \ avoid \ st \ st',
translate avoid e a / st \leadsto \texttt{CReturn} \ st' \rightarrow
\ln a (fv_type t') \rightarrow
(t = subst_state\_t\_avoid\ t' \ st \ set\_empty \leftrightarrow t = subst_state\_t\_avoid\ t' \ st' \ set\_empty).

Tactic Notation "prepare" hyp(Himp) hyp(HEval) 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 HEval with (outx:=outx1) (st:=st1) (avoid:=avoid1) (ex0:=e) as [f1 f2r]; try
apply Himp));
try assumption;
try (specialize (f1 (refl_equal _)));
try (deestruct f2r as [st' [f2 f2'] ]; [ eassumption | ]); try (deestruct 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; simp in H; rewrite ← H in ×;
    destruct (a) as [x px]; simp 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 | ln_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; ln_contra Pa]
     [ rewrite ← Hsubstt; rewrite Hrew | try eassumption | ln_contra Pa]
   ].

Tactic Notation "W_resolve" hyp(Himp) hyp(Hsubst) hyp(Hsubstt) hyp(Pa) hyp(Hrew) :=
   first [ apply CESeqErr; apply CEAssErr; W_res1 Himp Hsubst Hsubstt Pa Hrew
   | eapply CESeq; [apply CEAss; 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_ерemove (l: string) (e: Value): Value.
  Admitted.

Lemma rem_add : ∀ l v x,
  v dot l x = v →
  v update l v (v_ерemove l x) = x.

Lemma v_has_field_rem : ∀ l li l1' vx,
  v_has_field li (v_ерemove l1' vx) = true →
  v_has_field li vx = true.

Lemma v_dot_rem : ∀ l li l1' vx v,
  v dot li (v_ерemove l1' vx) = v →
  v dot li vx = v.

Lemma v_has_field_ign : ∀ l li l1' vx,
  li ≠ l1' →
  v_has_field li vx = true →
  v_has_field li (v_ерemove l1' vx) = true.

Lemma v_dot_ign : ∀ l li l1' vx v,
  li ≠ l1' →
  v dot li vx = v →
  v dot li (v_ерemove l1' vx) = v.

Lemma v_has_field_false : ∀ li vx,
  v_has_field li (v_ерemove li vx) = false.

Lemma v_ерemove_E : ∀ l l1' vx,
  is_E (v_ерemove 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" \rightarrow \text{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" \rightarrow \text{CReturn } v.

Lemma \text{fv_entity_app} : \forall l l1′ e1′,
(fix \text{fv_entity} (les : list (string × Exp)) : list string :=
match les with
| nil ⇒ nil
| pair _ e :: les′ ⇒ \text{fv_exp} e ++ \text{fv_entity} les′
end) (l ++ (l1′, e1′) :: nil) =

Lemma \text{cir_entity_app} : \forall l l1′ e1′ \text{ st},
(fix \text{cir_entity} (les : list (string × Exp)) : bool :=
match les with
| nil ⇒ false
| pair _ e :: les′ ⇒
  contains_impure_refinements e || \text{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 \text{cir_entity} (les : list (string × Exp)) : bool :=
match les with
| nil ⇒ false
| pair _ e :: les′ ⇒
  contains_impure_refinements e || \text{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 ⇒


\[
\text{subst\_exp } (\text{subst\_state\_avoid } e'3 \text{ st } (\text{set\_add } x \text{ (set\_add } y \text{ set\_empty)))) \text{ v } x) \\
\text{vs} (\text{Var y}) = \\
\text{subst\_state\_avoid} \\
\text{(Let y e'2) \\
\text{(let\_seq y } (\text{map } (\text{fun } v : \text{Value } \Rightarrow \text{subst\_exp e'3 v } x) \text{ vs} ) (\text{Var y}))}
\]

Lemma \text{v\_mem\_remove} : \forall x y c, \\
v\_mem x (v\_remove y c) = true \rightarrow \\
v\_mem x c = true.

Lemma \text{unroll\_loop} : \forall st st' b v1 a x y e avoid, \\
x \neq b \rightarrow \\
y \neq b \rightarrow \\
\neg \text{ln b } (\text{fv\_exp e}) \rightarrow \\
is\_C v1 = true \rightarrow \\
v\_mem a v1 = true \rightarrow \\
\text{st b } = v1 \rightarrow \\
(b := \text{ECollRem } (\text{EVar b}) (\text{EValue a}); \\
(\text{while } \text{ENot } (\text{ECollEmpty } (\text{EVar b})) \text{ inv fun : Env } \Rightarrow true \\
\text{do } (x ::= \text{pick b}); \\
b := \text{ECollRem } (\text{EVar b}) (\text{EVar x}); \\
\text{translate (avoid) e y end}); \\
x := \text{EValue a}; \\
\text{translate (avoid) e y) / st } \rightsquigarrow \\
\text{CReturn st'} \\
\rightarrow \\
(\text{while } \text{ENot } (\text{ECollEmpty } (\text{EVar b})) \text{ inv fun : Env } \Rightarrow true \\
\text{do } (x ::= \text{pick b}); \\
b := \text{ECollRem } (\text{EVar b}) (\text{EVar x}); \\
\text{translate (avoid) e y end) / st } \rightsquigarrow \\
\text{CReturn st'}.

Lemma \text{translate\_let\_seq\_inv} : \forall vs avoid y e'3 x st st' a, \\
List.incl (x :: y :: \text{List\_remove eq\_str\_dec x } (\text{List\_remove eq\_str\_dec y } (\text{fv\_exp e'3}))) \text{ avoid} \\
\rightarrow \\
\text{translate} \\
\text{avoid} \\
(\text{fold\_right } (\text{fun } e e' : \text{Exp } \Rightarrow \text{Let y e e'}) \\
(\text{Let y } (\text{subst\_exp e'3 a } x) (\text{Var y})) \\
(\text{map } (\text{fun } v : \text{Value } \Rightarrow \text{subst\_exp e'3 v } x) \text{ vs} ) y / st \rightsquigarrow \\
\text{CReturn st'} \\
(\text{translate} \\
\text{avoid} \\
(\text{fold\_right } (\text{fun } e e' : \text{Exp } \Rightarrow \text{Let y e e'}) \\
(\text{Var y}))}
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 →
B.5. SOUNDNESS

\[ (\forall s \text{ arg exp}, (\text{arg},\text{exp}) = \text{functions s} \rightarrow \ \text{contains_impure_refinements exp} = \text{false} \rightarrow \]}

\[ (\forall s \text{ arg exp}, (\text{arg},\text{exp}) = \text{functions s} \rightarrow \ \text{fv_exp exp} = \text{nil} \lor \text{fv_exp exp} = (\text{arg}:\text{nil}) \rightarrow \]}

\[ \text{contains_impure_refinements (ex)} = \text{false} \rightarrow \]}

\[ \text{nil} \vdash \{P\} \text{translate avoid ex outx} \{Q\} \rightarrow \]}

\[ \text{valid_formula P} \rightarrow \]}

\[ \neg \text{Eval ex Error}. \]

B.5.5. Relation with verification condition generation

**Corollary soundness_vcgencode_translation:** \( \forall \text{avoid ex outx Q C}, \)

\[ \text{fv_exp exp} = \text{nil} \rightarrow \]}

\[ (\forall s \text{ arg exp}, (\text{arg},\text{exp}) = \text{functions s} \rightarrow \ \text{contains_impure_refinements exp} = \text{false} \rightarrow \]}

\[ (\forall s \text{ arg exp}, (\text{arg},\text{exp}) = \text{functions s} \rightarrow \ \text{fv_exp exp} = \text{nil} \lor \text{fv_exp exp} = (\text{arg}:\text{nil}) \rightarrow \]}

\[ \text{contains_impure_refinements (ex)} = \text{false} \rightarrow \]}

\[ \text{valid (VCgen_procs C)} \rightarrow \]}

\[ \text{valid_formula (VCgen C (translate avoid ex outx) Q)} \rightarrow \]}

\[ \neg \text{Eval ex Error}. \]

B.5.6. Relation to the type system

**Definition x := "x".**

**Definition incom_sample :=**

\[ \text{In (Value (v_int 5)) (Refine x Any (BinOp OGt (Var x) (Value (v_int 5)))}. \]

**Lemma incom_sample_type : \( \neg \text{envT_empty} \vdash \text{incom_sample} : \text{Logical}. \)**

**Lemma incom_translated_no_error :**

\[ \{ \text{Fun}_\_ : \text{HoareArgs.ZType} \}_\_ : \text{Env} \Rightarrow \text{true} \}

\[ \text{translate (nil ++ dom envT_empty)} \text{incom_sample "outx"} \]

\[ \{ \text{Fun}_\_ : \text{HoareArgs.ZType} \}_\_ : \text{st : Env} \Rightarrow \text{translateT Logical "outx" st } \} . \]

**Lemma counterexample : \( \neg (\forall e \text{ env outx T avoid}, \)

\[ \text{fv_exp e} = \text{nil} \rightarrow \]}

\[ \text{env} \vdash \text{T} \rightarrow \]}

\[ \neg \text{In outx (dom env)} \rightarrow \]}

\[ \text{nil} \models \{ \text{fun}_\_ \Rightarrow \text{true} \} \text{translate (app avoid (dom env)) e outx} \]

\[ \{ \text{fun}_\_ \text{ st} \Rightarrow \text{translateT T outx st } \} \rightarrow \]}

\[ \text{env} \vdash e : T \} . \]

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;

// Constructors

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 entries
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); {
119

```plaintext
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));

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(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)));
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}
f
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);
en
ensures out == Coll_Distinct(coll);
out := Coll_Distinct(coll);
}

// Lists

type Closure = [Value, Value] Value;

custom 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_Llist(list);
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)));
  }