A Survey on Embedding Programming Logics in a Theorem Prover

A. Azurat and **I.S.W.B. Prasetya**
Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands
{*ade,wishnu*}*@cs.uu.nl*

Abstract

Theorem provers were also called 'proof checkers' because that is what they were in the beginning. They have grown powerful, however, capable in many cases to automatically produce complicated proofs. In particular, higher order logic based theorem provers such as HOL and PVS became popular because the logic is well known and very expressive. They are generally considered to be potential platforms to embed a programming logic for the purpose of formal verification. In this paper we investigate a number of most commonly used methods of embedding programming logics in such theorem provers and expose problems we discover. We will also propose an alternative approach : *hybrid embedding*.

1 Introduction

Formal verification of a system starts from formal representations of (the implementation of) the system and its requirements. One should be able to make sure that the representations reflect the real systems and requirements and to check whether the implementation matches the requirements. Despite the general consensus that formal verification has great potential to help us produce dependable software/hardware products, the processes involved in doing so are difficult. The reasoning mechanism may not be obvious yet. The construction of proofs is often expensive and requires high expertise which is difficult to get. In the past the proofs themselves were also very error prone because they have to be constructed by hand.

The emerging of theorem provers and model checkers gives some expectation that formal verification can be done, to some extent, automatically. The two technologies have different approaches. A theorem prover is based on a (relatively small) logic system. A program and its requirements are formulas of the logic. Verifying a program amounts to using the inference rules of the logic to prove some formulas. On the other hand a model checker represents a program as an automaton. It simulates the automaton. As it generates all possible

states reachable by the program, it verifies if those states fulfil the program's requirements[6].

Model checkers are good in finding errors because they are highly automated, but they tend to have problems in representing infinite state spaces and infinite data structures (such as unbounded integers, strings, or lists). Theorem provers, in particular those based on a rich expression language, have no problem with infinite states and data structures. A theorem prover is also much more secure as it is based on a relatively small core logic. Theorem provers basically provide no automation. Their primary function is to check proofs supplied by the user rather than constructing the proofs themselves. Modern theorem provers, however, come with meta languages which can be used to program automatic proof generation. There are also some initiative to integrate theorem provers and model checkers.

1.1 What this Paper is About

To do formal verification of a program we need a programming logic. To do so with an existing theorem prover we usually need to *embed* the programming logic first in the theorem prover[1]. This is because the logic used by the theorem prover is typically not a programming logic. The purpose of this paper is to survey commonly used techniques for such embeddings. We will focus on theorem provers which are based on a higher order predicate logic. This class of theorem provers seems to be very popular because the logic is well known and its higher order feature makes it very expressive. Two embedding techniques are to be distinguished known: *shallow* and *deep* embedding. Both turn to be problematic. We will show the problems, by using concrete examples. In the examples we will assume the use of the HOL theorem prover, which is representative for the above mentioned class of theorem provers.

We will also explain an alternative approach, which we call *hybrid embedding*. In this approach only the semantics of a programming logic is embedded in a theorem prover. The syntactic machinery of the logic is implemented outside the theorem prover.

1.2 A Note on Notation

In order to show concrete examples of embedding we assume the HOL system is used as the host logic. We will however strip irrelevant HOL notational details from the code to give minimum distraction to non-HOL readers.

[1] This is not to be confused with the term *embedded system* which means a piece of hardware or software embedded as a component of an autonomous control system –such as the control system of a steam boiler. While the term *embedding* in this paper refers to a logic embedded inside another logic.

2 Embedding

In the formal verification community, *mechanizing* a logic means implementing the logic in a computer. The intent is to get the computer to check the well formedness of our formulas and the correctness of our proofs. This means, among other things, that the inference rules of the logic have to be implemented in the computer. Mechanization does not necessarily imply proof automation.

Embedding mechanizes a logic $\mathcal{L}$ by encoding $\mathcal{L}$ and its semantics inside another, already mechanized logic $\mathcal{L}'$, and in effect 'borrows' the mechanization of $\mathcal{L}'$. We will call the logic $\mathcal{L}$ the *guest* logic and $\mathcal{L}'$ the *host* logic.

Having the semantics represented in the host logic is what separates embedding from ordinary mechanization. It allows the verification of the soundness of the guest logic $\mathcal{L}$ in $\mathcal{L}'$. The inference rules of $\mathcal{L}$ are represented as formulas of $\mathcal{L}'$, so they can be verified just like any other $\mathcal{L}'$ formula.

Typically, we want a host logic to be expressive so that we can use it to represent (embed) a wide range of guest logics. We also want it to be small: it provides a minimum set of language constructs to express formulas and a minimum set of inference rules. This is for security reasons. Its primitiveness makes it much easier for us to convince ourselves of the soundness of the host logic. Obviously it is not safe to build an embedding on a host logic with questionable soundness. The HOL system [9] is an example of a theorem prover based on a small, yet very expressive logic (a higher order, typed predicate logic). Note that once proven, an embedded logic inherits the soundness of its host.

Unfortunately a minimalistic host logic also creates another problem. A programming logic is typically based on high level programming and specification languages. Encoding sentences of these languages as sentences of a more primitive language of the host logic can seriously reduce readability –an aspect that makes formal methods, in particular theorem provers, not very welcome in the industrial world[6].

Embedding also inherits the problem of formal semantics. The semantics of a real programming language is very complicated, raising the issue of maintaining the correctness and the reliability of the semantics itself. We quote here from [23]:

> "Our EML experience suggests that, at least at the present time, tackling the problems of specification and formal development in a real programming language at a fully formal level is just too difficult. This seems to leave two ways forward: either remain fully formal but focus on a smaller and simpler language, perhaps building up from this gradually to something approaching a real language; or remain with a real language, but give up trying to achieve full formality."

In the theorem prover community, people often distinguish between so-called *shallow* and *deep* embedding [3]. Shallow embedding of a logic $\mathcal{L}$ embeds the

semantics of $\mathcal{L}$ but does only a minimal effort to represent the grammar[2] (syntax) $\mathcal{L}$ in $\mathcal{L}'$. Deep embedding embeds also the syntactic structures of the logic completely. The exact representation (e.g. whether we represent states with functions or records) may influence how much of the syntactic structures of the formulas of the guest logic can be maintained in the embedding –so people also speak of the *depth* of an embedding. Grammar is represented much better in deep embedding, as it uses functional data types to represent the grammar –it is known that there is strong correspondence between (context free) grammars and recursive data types [8]. Because of the explicit representation of the grammar, in deep embedding it is possible to encode and verify syntactic operations and analyses made on the formulas of the guest logic. This is not possible with shallow embedding. Unfortunately deep embedding has its own problems. For example, it takes lots of effort to set up. Changing the syntax of the guest logic may also incur lots of work.

In some theorem provers, such as COQ, there is no deep-shallow dichotomy [16]. COQ is based on a type system, which is even more primitive than the logic used by, for example, the HOL system. The disadvantage is that a guest logic must now be represented in an even more primitive language, which will further reduce the readability of the representations.

Lots of work has been invested in embedding in the past 10 years. We will mention some achievements –there are too many to mention all of them.

In [2, 21, 25], Andersen, Prasetya and Vos reported the embedding of the programming logic UNITY in HOL –UNITY is a simple logic [5] for reasoning about temporal properties of distributed systems. Andersen, Prasetya and Vos have used their embedding to mechanically verify non-trivial distributed algorithms. Furthermore, Prasetya also uses embedding to verify a rich self-stabilization theory, and Vos does the same for refinement theory. The embeddings are shallow, though Vos has a deeper embedding by having typed program variables more explicitly represented.

An example of deep embedding is the embedding of a simple imperative programming language called *Sunrise* in HOL by Homeier and Martin [11]. A *Verification Condition Generator* (VCG) is supplied, based on a method named *Diversion Verification Condition* to provide total correctness of mutually recursive functions. The VCG basically does a syntactic transformation on the Sunrise's programs and specification. It has been proven sound in HOL, which is only possible because of the deep embedding of Sunrise. However, the extendibility issue was not covered yet. So far the language only has numeric data type and *list* constructor, but even then the soundness proof of the VCG already involves 8 new types, 217 definitions, and 906 major theorems, using over 57,000 lines of proof, organized in 22 HOL theories.

An attempt to use embedding for a real programming language and logic was recently reported by Huisman [12] who embeds Java (without threads) in PVS and Isabelle. The embedding is part of a larger tool called LOOP. LOOP

[2]The grammar of a language is a set of rules specifying how the formulas of the language are constructed.

features a specification language for Java, called JML [13], and compilers to compile a Java program and its JML specifications to their embedded representations.

3 Shallow Embedding

Shallow embedding concentrates on how the semantics of the guest logic can be represented in a theorem prover. It is less concerned with various syntactic structures and constraints of the guest logic. Compared to deep embedding, shallow embedding takes much less work to set up. It is suitable when used in the initial phase of the development of a programming logic in which many features are still experimental. The languages of the logic and its inference rules are still unstable and we constantly need to verify the logic's consistency after each modification. It is still suitable for actual program verification work if the language used by the guest logic is still simple, such as the case in [21] where target programs are just flat structured UNITY programs.

To give a more concrete illustration of the approach, consider a very simple programming logic –abbreviated *VSPL*– described in Figure 1. The logic is too simple to be used in practice, but it will serve our purpose here. A shallow embedding of this logic will be given later. We will assume the HOL system to be used as the host logic.

A VSPL program is either a `skip`, an assignment, or a (possibly nested) `if-then-else` statement. There are only two kinds of values in VSPL: booleans and integers. VSPL expressions are very limited, with only $=$, $+$, $\Rightarrow$, $\wedge$, and $\neg$ as operators. Specifications are expressed in terms of Hoare triples with the usual interpretation. We also add one more kind of specification, which is a specification of the form $[P]$ where P is a boolean expression (predicate). Such a specification means that the predicate P is a tautology –so it holds on all possible states of a program. When used exhaustively, the inference rules of VSPL will reduce any Hoare triple specification into a specification of the form $[P]$. Although not explicitly mentioned in its description in Figure 1, VSPL assumes the availability of a mechanism to prove specifications of the latter form.

Remember that in embedding we need to have a semantics of the guest logic. This is because we basically embed a logic by writing its semantics as formulas of the host logic. The steps we typically have to perform are:

1. Identifying various semantic domains of the guest logic.
2. Describing the semantics in detail and encoding it in the host logic.
3. Representing and verifying the inference rules of the guest logic.

The amount of detail we want to put in the semantics may depend on the application of the guest logic. For example, the execution models of most imperative programming language typically include a program counter to keep track

1. Grammar:

$$
\begin{array}{lll}
\textit{Stmt} & :: & \texttt{skip} \\
 & | & \textit{Identifier} := \textit{Expr} \\
 & | & \texttt{if } \textit{Expr} \texttt{ then } \textit{Stmt} \texttt{ else } \textit{Stmt} \\
 & & \\
\textit{Expr} & :: & \textit{Expr} = \textit{Expr} \\
 & | & \textit{Expr} \wedge \textit{Expr} \\
 & | & \textit{Expr} \Rightarrow \textit{Expr} \\
 & | & \textit{Expr} + \textit{Expr} \\
 & | & \neg\ \textit{Expr} \\
 & | & \textit{Bool} \\
 & | & \textit{Integer} \\
 & & \\
\textit{Spec} & :: & \{\textit{Expr}\}\ \textit{Stmt}\ \{\textit{Expr}\} \\
 & | & \texttt{[} \textit{Expr} \texttt{]}
\end{array}
$$

2. Inference rules:

Skip Rule
$$\frac{[P \Rightarrow Q]}{\{P\}\ \texttt{skip}\ \{Q\}}$$

Assignment Rule
$$\frac{[P \Rightarrow Q[E/x]]}{\{P\}\ x\ \texttt{:=}\ E\ \{Q\}}$$

If-then-else Rule
$$\frac{\{P \wedge g\}\ A\ \{Q\} \quad , \quad \{P \wedge \neg g\}\ B\ \{Q\}}{\{P\}\ \texttt{if}\ g\ \texttt{then}\ A\ \texttt{else}\ B\ \{Q\}}$$

Figure 1: VSPL

of the control location of a program during its execution. In some setup, such as when concurrency is an issue, we may have to explicitly include program counters in the semantics. However if we only deal with sequential programs specified by Hoare triples, such as in VSPL, we can hide the program counter from the semantics.

There are many ways one can define the semantics of the same programming logic. In some representations, programs variables are represented by first class values in the host logic. Depending on the application this can be an important aspect. We need first class representation of variables names in order to express syntactic constraints on the set of variables of a program. They are quite useful and occur quite often as conditions in program transformations, composition, and refinements. On the other hand, approaches in which variables are not represented as first class values typically produce cleaner representations of programs and specifications. We will show examples of both approaches. There are also approaches which are somewhat in between, e.g. Huisman uses the combination of records and functions [12]. We will not cover this kind of approach.

4 Shallow Embedding I

This section will show a shallow embedding of VSPL in HOL using an approach where VSPL variables are represented by first class HOL values.

4.1 Step 1: Identifying Semantic Domains

A state of a program at a given moment describes the values of the program's variables at that moment. We can represent a state by a *function* from variables to values. We can use `string` to represent variables (actually, variables' names)[3]. This representation is quite simple and is used by many others, e.g. [2, 17, 21, 25]. Alternatively, one can also use lists to represent states.

Since VSPL only has two kind of values, booleans and integers, we can represent VSPL values in HOL with the following HOL data type:

Code 4.1 :

```
datatype Value_IB = fromInt int | fromBool bool
```

□

However, if we use this specific data-type the resulting embedding of VSPL inference rules will only work on that specific representation of data values: they cannot be reused if one decides to extend VSPL with new types of data values. To allow reuse, we can represent VSPL data values with a HOL type variable

[3] Although we will not do this here, for concrete applications it is typically useful to maintain separate lists to keep track which variables a given program owns and what their access modes are.

which can later be instantiated by some concrete HOL type. Alternatively, we can represent VSPL data values by a specific HOL type whose property is now *left unspecified* and later we provide injections to, for example, integers and booleans. Both approaches are almost equivalent. The second approach is slightly more flexible[4], so we will focus on it.

So, assume an unspecified HOL type `Value` representing all possible data values of VSPL programs. An expression is evaluated on a state and returns the value of the expression in that state. So, a VSPL expression will be represented by a function from state to `value`.

VSPL statements are deterministic. They can be represented by *functions* that take an initial state and return a new state (in other settings where statements are non-deterministic, or may block, *relations* can be used). Finally, a VSPL specification will semantically be represented by a boolean value, since a specification is either valid (*true*) or invalid (*false*).

To summarize, these are our semantic domains:

Definition 4.2 : Semantic domains of VSPL

```
type State = string -> Value
type Expr  = State  -> Value
type Pred  = State  -> bool
type Stmt  = State  -> State
```

□

4.2 Step 2: Specifying the Semantics

A semantics assigns meaning to each syntactic category of a language. For example, the following gives the semantics of VSPL statements in terms of the semantic domains given in Definition 4.2:

VSPL statements	semantics
`skip`	`(\s. s)`
`if g then A else B`	`(\s. if (g s) then A s else B s)`
`x=E`	`(\s. (\v. if v=x then E s else s v))`

where `s` is of type `State`.

This semantics can be directly translated to HOL. For each syntactic construct of *Stmt* we introduce a new HOL constant to represent it. The semantics is then assigned to each constant as its definition. So this is how VSPL statements are represented in HOL:

[4]In both approaches the inference rules are reusable. However, in the first approach – in which the space of VSPL data values is represented by a type variable, say, V– programs verified based on a concrete instance T_1 of V cannot be composed with programs verified based on a different instance T_2 of V. One will have to upgrade both programs using a common and larger base T_3. However, verification of both programs will have to be re-executed. Alternatively, one can also provide mappings between T_1, T_2 and T_3.

Definition 4.3 : Semantics of *Stmt*

```
SKIP                =  (\s. s)
IF g THEN A ELSE B  =  (\s. if (g s) then A s else B s)
x ASG E             =  (\s. (\v. if v=x then E s else s v))
```

□

VSPL specifications can be represented as follows:

Definition 4.4 : Semantics of *Spec*

```
VALID P    =  (!s. P s)
HOA A P Q  =  (!s. P s ==> Q (A s))
```

□

where `HOA A P Q` represents a specification of the form $\{P\}\ A\ \{Q\}$ and `VALID P` represents a specification of the form $[P]$.

Expressions, in particular boolean valued expressions (predicates) can be represented as follows:

Definition 4.5 : Semantics of *Expr*

```
p AND q  =  (\s.  p s /\  q s)
p IMP q  =  (\s.  p s ==> q s)
NOT p    =  (\s. ~p s)
```

□

where `AND`, `IMP`, and `NOT` represent VSPL's $\wedge$, $\Rightarrow$, and $\neg$. The bounded variable `s` is of type `State`.

In shallow embedding we may not want to represent all syntactic structures of the guest logic. Notice that the inference rules of VSPL only assumes the existence of $\wedge$, $\Rightarrow$, and $\neg$ operators with their respective arities. The inference rules do not care about other syntactic structures that *Expr* may have. So, for the purpose of verifying the soundness of VSPL we do not need to represent those other syntactic structures of *Expr* (such as the operators $=$ and $+$) and to provide their semantics.

4.3 Step 3: Representing and Verifying Inference Rules

VSPL's inference rules can be represented by HOL formulas. Proving these formulas essentially justifies the soundness of the inference rules they represent. The formula representation may seem inadequate, since rules are computation entities (they accept VSPL specifications and produce new specifications) and formulas are plain data. However, once proven, the resulting theorems can be 'executed' by applying them to some VSPL specifications. Applying a HOL theorem is usually done by calling some combination of HOL's own inference rules for Modus Ponens and Substitution.

1. **Skip-rule**

```
|- (P ==> Q)
    ==>
    HOA SKIP P Q
```

2. **Cond-rule.**

```
|- HOA A (P AND g) Q       /\
   HOA B (P AND NOT g) Q
   ==>
   HOA (IF g THEN A ELSE B) P Q
```

3. **Assign-rule**

```
|- VALID (p IMP (q o (x ASG E)))
   ==>
   HOA (x ASG E) p q
```

Figure 2: VSPL's inference rules in HOL

Figure 2 shows the resulting HOL theorems, representing VSPL's inference rules.

Proving those inference rules in HOL is very easy. Essentially we just rewrite them using their own definitions (using their own semantics) and then call HOL's simplifier and first order prover. As example, here is the proof code of the **Cond-rule**:

```
prove
     --‘
    HOA A (P AND g) Q  /\ HOA B (P AND NOT g) Q
    ==>
    HOA (IF g THEN A ELSE B) P Q
   ‘--,
   RW_TAC std_ss [HOA_def,IFTHENELSE_def,AND_def,NOT_def]
   THEN PROVE_TAC[]
   );
```

4.4 Representing Programs and Specifications

The shallow embedding of VSPL is now set. We can use it to represent and verify concrete VSPL programs. Consider the following VSPL specification:

Example 4.6 :

$\{b \wedge \neg(x = y)\}$

```
if x=y then x=x+1 else y=y+1
```

$\{b \wedge \neg(x+1=y)\}$

□

One may expect to represent, for example, the post condition $b \wedge \neg(x+1=y)$ like this:

```
(\s. s "b") AND NOT(\s. s "x" + 1 = s "y")
```

However this is not type correct. It requires states to be functions returning boolean values (as in `s "b"`) as well as integers (as in `s "x" + 1`). This is also not consistent with the typing of states: we have decided that they are functions from `State` to `Value`. Fortunately we have left the property of `Value` unspecified in the semantics. We can now say that it is at least large enough to contain booleans and integers, and furthermore we also have constructors and destructors to construct a `Value` from a boolean or integer vice-versa. We can impose on the existence of the following functions:

Definition 4.7 : Constructors and Destructors of `Value`

```
fromInt  : int->Value
fromBool : bool->Value
toInt    : Value->Int
toBool   : Value->Bool
```

with the expected property that each `from-` function forms an injection and their respective `from-` counterparts form the inverses.
□

Now we can correctly represent the specification in the Example 4.6 as follows:

Code 4.8 :

```
HOA

(IF (\s. (toInt o s) "x" = (toInt o s) "y")
    THEN ("x" ASG (fromInt o (\s. (toInt o s) "x" + 1)))
    ELSE ("y" ASG (fromInt o (\s. (toInt o s) "y" + 1))))

((\s. (toBool o s) "b") AND
 NOT(\s. (toInt o s) "x" = (toInt o s) "y"))

((\s. (toInt o s) "b")  AND
 NOT(\s. (toInt o s) "x" + 1 = (toInt o s) "y"))
```

□

The reader may notice that expressions at the right hand side of an assignment is translated differently from the guard of an `if-then-else`, pre-condition, and post-condition. The latter are always boolean typed expressions (predicates). On the other hand, an expression at the right hand side of an assignment can be of any type, which in this embedding is represented by a function that returns a `Value`. Because, for example, `(\s. (toInt o s) "x" + 1)` returns an integer, we still need the function `fromInt` to cast the result to `Value`.

4.5 Verifying Programs

Specifications such as the one in Example 4.6 are verified by successively applying the guest logic's inference rules in a certain order. In the case of VSPL this can be easily automated in HOL. We can write something like this:

```
(REPEAT o FIRST  o map MATCH_MP_TAC)
         [SKIP_thm, ASG_thm, IFTHENELSE_thm]
```

where, for example, `SKIP_thm` is the HOL theorem representing the inference rule for the `skip` statement. The above HOL code will repeatedly apply VSPL inference rules until no further application is possible. How much automation we can get depends largely on the guest logic itself. In any case, HOL comes with a powerful meta language.

In the case of VSPL, exhaustively applying the rules do not prove the specification, yet. Rather, this step produces a specification of the form `VALID P`. After unfolding the definition of `VALID` we will get a verification 'goal' of the form `P s` which has to be proven for an arbitrary `s`. For example, for the specification in Example 4.6 the following goal will be produced:

Code 4.9 :

```
(   (toBool o s) "b"                          /\
   ~((toInt o s) "x" = (toInt o s) "y")       /\
    ((toInt o s) "x" = (toInt o s) "y")
    ==>
    (toBool o s) "b"                          /\
   ~((toInt o s) "x" + 1 + 1 = (toInt o s) "y"))

/\

(   (toBool o s) "b"                          /\
   ~((toInt o s) "x" = (toInt o s) "y")       /\
   ~((toInt o s) "x" = (toInt o s) "y")
    ==>
    (toBool o s) "b"                          /\
   ~((toInt o s) "x" + 1 = (toInt o s) "y" + 1))
```

which has to be proven to hold for an arbitrary `s`.
□

The formula represents:

$$(b \wedge \neg(x = y) \wedge (x = y) \;\Rightarrow\; b \wedge \neg(x + 1 + 1 = y))$$
$$\wedge$$
$$(b \wedge \neg(x = y) \wedge \neg(x = y) \;\Rightarrow\; b \wedge \neg(x + 1 = y + 1))$$

The above can be easily proven in HOL. Note that from this point on the proof no longer depends on the structure of the program. Instead, it depends on the properties of the data values –in this case the properties of booleans and integers.

There are also a number of subtle points about the `to-` and the `from-` functions. Consider the specification:

Example 4.10 :

```
{ x=0 }   x = x + 1   { x=1 }
```

which is represented by this HOL formula in the embedding:

```
HOA   ("x" ASG (fromInt o (\s. (toInt o s) "x" + 1)))

      (\s. (toInt o s) "x" = 0)

      (\s. (toInt o s) "x" = 1)
```

□

After applying the `ASG` Rule to the specification above, and after some trivial simplifications, we will get this as a goal:

Code 4.11 :

```
VALID
  ((\s. (toInt o s) "x" = 0)
   IMP
   (\s. toInt (fromInt ((toInt o s) "x" + 1)) = 1))
```

□

The goal above cannot be proven if we cannot reduce the expression at the right hand side of the implication to `(\s. (toInt o s) "x" + 1 = 1)`. The following three properties are necessary to enable this reduction

1. The `to-` and the `from-` functions have to be distributive. For example, we have to be able to reduce:

   ```
   toInt (fromInt ((toInt o s) "x" + 1))
   ```

   to:

   ```
   toInt (fromInt (toInt o s) "x")  +  toInt (fromInt 1)
   ```

2. The fact that the `to-` functions are injections with their respective `from-` counterparts as inverses is crucial, otherwise we cannot, for example, reduce `toInt (fromInt 1)` to 1.

3. We need to add an explicit assumption that `s "x"` is a `Value` within the domain of `toInt`. This can also be seen as a requirement for the author of the program to confirm to HOL that `x` is indeed an integer valued variable. Without this assumption we cannot reduce:

   ```
   toInt (fromInt (toInt o s) "x")
   ```

   to `(toInt o s) "x"`

4.5.1 Syntax Extension

What if we want to extend the syntax of the guest logic? For example, let us extend VSPL by adding the sequential composition operator '`;`' for statements. So the syntax of the *Stmt* is now extended as shown below. We also need a new inference rule to deal with the new operator.

Example 4.12 :

$$
\begin{array}{lll}
\mathit{Stmt} & :: & \texttt{skip} \\
 & & \mid \mathit{Identifier} := \mathit{Expr} \\
 & & \mid \texttt{if } \mathit{Expr} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \\
 & & \mid \mathit{Stmt} \texttt{ ; } \mathit{Stmt}
\end{array}
$$

Seq Rule: $\dfrac{\{P\}\ A\ \{Q\} \quad , \quad \{Q\}\ B\ \{R\}}{\{P\}\ A\ \texttt{;}\ B\ \{R\}}$

□

To represent '`;`' we need to add a new HOL constant and assign some meaning to it:

Definition 4.13 : Semantics of `;`

```
SEQ A B = B o A
```

□

The new inference rule can be represented in the same way existing VSPL rules are represented in HOL, namely by the formula:

```
HOA A P Q  /\ HOA B Q R
==>
HOA (SEQ A B) P R
```

which can be proven with as much ease as we prove the other rules.

Note that incrementally extending the syntax of the guest logic does not require existing shallow embedding to be rebuilt. As we will see later, this property does not hold for deep embedding.

4.5.2 Value Space Extension

VSPL only supports two kinds of data values: booleans and integers. What if we want to add new types? As an example, let us add the type of lists of integers. We also extend the syntax of *Expr* with a few list operators as shown below. No new inference rule is introduced.

Example 4.14 :

$$
\begin{array}{lll}
\mathit{Expr} & :: & \mathit{Expr} = \mathit{Expr} \\
 & | & \mathit{Expr} \wedge \mathit{Expr} \\
 & | & \mathit{Expr} \rightarrow \mathit{Expr} \\
 & | & \mathit{Expr} + \mathit{Expr} \\
 & | & \neg\ \mathit{Expr} \\
 & | & \mathit{Expr}\ \texttt{::}\ \mathit{Expr} \quad \texttt{/* add a new element to a list */} \\
 & | & \texttt{[]} \qquad\qquad\qquad\ \texttt{/* make an empty list */} \\
 & | & \mathit{Bool} \\
 & | & \mathit{Integer}
\end{array}
$$

□

Note that since we have no inference rules that require evaluation of a list expression, there is no need to explicitly add the list operators in the embedding (like we did to, for example, the $\wedge$ and the $\neg$ operators).

To represent programs and specifications that contain list expressions we need to extend our assumptions on `Value`. Recall that this is an unspecified HOL type representing all possible data values of VSPL programs. Since we now also have lists of integers we need to add an assumption that `Value` is also large enough to include all lists of integers. We can do this by imposing the existence of the following two functions:

Definition 4.15 : `Value` TO LIST INJECTION

```
fromIntList  : Int list->Value
toIntList    : Value->Int list
```

with the property that `toIntList` is an injection and `fromIntList` is its inverse. □

Here is an example of a program that does something with lists and its representation:

Example 4.16 :
A VSPL program:

```
if z=[] then skip else z = 1 :: z
```

Its representation:

```
IF (\s. (toIntList o s) "z" = [])
   THEN SKIP
   ELSE ("x" ASG (\s. 1 :: (toIntList o s) "z"))
```

□

It would be nice to have other kind of lists, e.g. lists of booleans or nested lists. The elegant way to get this is by adding polymorphic types to VSPL. Unfortunately embedding polymorphism in a theorem prover such as HOL is problematic. For example, suppose now we want to extend VSPL's lists so that their elements can have an arbitrary type (so we can have lists of booleans too, and multi level lists). We will have to add the following constructor and destructor to the type `Value`:

Code 4.17 :

```
fromList  : 'a list->Value
toList    : Value->'a list
```

□

where `'a` denotes a type variable.

Since `'a` is variable, it can be `Value` itself. No injective function `fromList` can be provided without introducing the Russel paradox[5].

This is very unfortunate. Polymorphism is very useful to specify general properties of programs, or to specify programs that build sophicticated data structures, e.g. distributed programs (building spanning trees) and database programs (building tables). Note that the used programming language itself does not need to support sophisticated expressions. Specifying programs on the other hand, may require a powerful expression language.

We can propose a solution to this problem. The type `Value` has to be introduced as a 'second order' HOL type with an axiom stating that it is large enough to provide an injection to any 'normal' HOL type. To prevent the Russel paradox, we put the restriction that type variables in HOL can only range over normal HOL types. So they cannot be instantiated with `Value`.

The solution can be easily implemented if types are first class HOL objects so we can freely manipulate them. Unfortunately, they are not. So, implementing it will require some hacking into HOL implementation –not the kind of things ordinary users can and want to do. People may also try other theorem provers, such as the COQ system, where types are first class objects.

As a side note, one may point out this problem of embedding polymorphic state space may be circumvented had we chosen to represent VSPL data values with a polymorphic type V. Unfortunately this is not the case. For example, for the program in Example 4.10 one may suggest to use the following instantiation of V:

```
datatype 'a Value_IBL =
      fromInt int
    | fromBool bool
    | fromList ('a list)
```

[5]Without restriction, `Value` can be extremely large that it contains everything. Such a set leads to a certain paradox called Russel paradox. See [14]

Though the type system of HOL (and most type systems) will make sure we do not run into the Russel paradox, the above type is actually not what we want. Consider the (destructor) function `toList` which inverts `fromList`:

```
toList : 'a Value_IBL -> 'a list
```

Consider an expression `a::b` where `a` is of type `int list` and `b` is of type `int list list`. The expression will be represented like this in the embedding:

```
(\s. (toList o s) a :: (toList o s) b)
```

The (intended) types of the first and second `toList` are respectively:

```
toList :: int Value_IBL -> int list
toList :: int list Value_IBL -> int list list
```

Consequently, the second `s` has the (intended) type `int Value_IBL` whereas the third `s` has type `int list Value_IBL`. This is not type correct since both `s`'es are bound to the same `s`!

4.6 Guest Logic Type Checking

Representing states as functions from variables to values raises a problem in keeping the type of embedded expressions consistent. For example, the following is not a type correct VSPL program:

Example 4.18 :

```
if x then x=false else x=x+1
```

□

Yet its representation in HOL:

```
IF (\s. (toBool o s) "x")
   THEN ("x" ASG (fromBool o (\s. F)))
   ELSE ("x" ASG (fromInt  o (\s. (toInt o s) "x" + 1)))
```

is a well-typed HOL expression. An even simpler example is the following:

Example 4.19 :

```
"x" ASG (fromInt o (\s. (toInt o fromBool) T))
```

□

which is also a well-type HOL expression, but it has no VSPL counterpart.

The injection of VSPL's data types into the HOL type `Value` ambiguates the original VSPL typing information. So it is not a surprise that HOL's type system is having problems in type checking embedded VSPL programs.

As a rule of thumb, syntax analysis algorithms[6] cannot be incorporated inside a shallow embedding as they typically require the syntactic structure of an embedded sentence to be explicitly represented. Some limited form of syntactic analysis may still be possible. For example the embedding approach described in this section allows the testing of constraints on the variables of a program –see Subsection 4.7. Another approach uses records to represent states –see Section 5– and with this approach it is possible to use HOL's own type checker to type check embedded VSPL sentences.

It is also a moot point whether one should build a syntax analysis algorithm in HOL. It seems somewhat wasteful, since this kind of algorithms can often be reliably implemented using high level programming languages, e.g. functional languages such as ML or Haskell. In the case of VSPL type checking, an external type checker can be easily writen. We can use it type check a VSPL program before translating it in HOL, and hence making sure that HOL only recieves type correct VSPL programs.

Of course having a foreign type checker embedded in HOL opens a way to verify properties about the type checker. It is also useful to, for example, guard the type consistency of program transformations. A simple example of a transformation is guard strengthening. The following law, for example, says that we can strengthen the guard of a conditional statement:

Example 4.20 :

$$\frac{\{P\}\ \texttt{if}\ g \wedge h\ \texttt{then}\ A\ \texttt{else}\ B\ \{Q\}}{\{P \wedge g\}\ A\ \{Q\}}{\{P\}\ \texttt{if}\ g\ \texttt{then}\ A\ \texttt{else}\ B\ \{Q\}}$$

□

Since program transformations change a program, we may want to make sure that the resulting programs are type correct again. To verify this in HOL, we will have to define in HOL what 'type correctness' means, which essentially amounts to encoding the type checking algorithm in HOL.

4.7 Program Transformations with Variable Constraints

The best way to model program transformations with variable constraints in HOL is with deep embedding because checking a variable constraint is a syntactic analysis operation. To verify the consistency of such a transformation we still need a semantics. Because we have represented VSPL variables with (first class) HOL values we can easily define the semantics of variable constraints.

Variable constraints are a useful kind of constraints occuring in many program transformations. Think for example of the parallel composition of two programs which are constrained to share no write variables. Another example is adding assignments into a program where the new assignments should be assignments to fresh variables.

[6] In addition to type checking, think also of recognizing nested loops, recognizing recursion, recognizing unread variables, and so on.

As a concrete example, consider the following transformation which reverses the transformation in Example 4.20. It states that we can also safely weaken the guard $g \wedge h$ of an `if-then-else` statement by dropping the h provided the `else` branch can realize the post-condition if h does not hold:

Example 4.21 :

$$\frac{\{P\}\ \texttt{if}\ g\ \texttt{then}\ A\ \texttt{else}\ B\ \{Q\} \qquad \{P \wedge \neg h\}\ B\ \{Q\}}{\{P\}\ \texttt{if}\ g \wedge h\ \texttt{then}\ A\ \texttt{else}\ B\ \{Q\}}$$

□

However we can also assert something stronger. The following rule states that the constraint in Example 4.21 is implied if the `else` branch does not modify any (free) variables of the post-condition Q if h does not hold:

Example 4.22 :

$$\frac{Q\ \textit{is confined by}\ V \qquad B\ \textit{preserves}\ V\ \textit{when}\ \neg h \qquad [P \wedge \neg h \Rightarrow Q]}{\{P \wedge \neg h\}\ B\ \{Q\}}$$

□

Given a predicate Q and set of variables V, Q *is confined by* V means that V is a subset of the free variables of V. This can be easily checked if the syntactic structure of Q is also embedded in HOL. Unfortunately in shallow embedding we do not have that. It is, however, possible to define confinement 'semantically'.

Given two functions f and g, they are partially equal over V if both return the same value on all $x \in V$. This can be defined as follows in HOL:

Definition 4.23 : Partial Equality

```
s EQV t ON V =  (!x. x IN V ==> (s x = t x))
```

□

Q is confined by V if for any state s on which Q is satisfied, then Q is also sastisfied on all other states t which are partially equivalent to s on V. In HOL:

Definition 4.24 : Confinement

```
Q IS CONFINED BY V  =  (!s t. s EQV t ON V ==> (Q s = Q t))
```

□

Finally, a statement B *preserves* a set of variables V *when* h if when h holds the statement preserves any predicate confined by V. In HOL:

Definition 4.25 : Preservation

```
Q PRESERVES V WHEN h
=
(!P. P IS CONFINED BY V ==> HOA B (h AND P) P)
```

□

The inference rules in Examples 4.21 and 4.22 can now be expressed by the following theorems in HOL (they can be easily proven).

Theorem 4.26 :

```
|- HOA (IF g THEN A ELSE B) P Q    /\
   HOA B (NOT h AND P) Q
   ==>
   HOA (IF (g AND h) THEN A ELSE B) P Q
```

□

Theorem 4.27 :

```
|- Q CONFINED BY V                 /\
   Q PRESERVES V WHEN (NOT h)      /\
   VALID (P AND NOT h  IMP  Q)
   ==>
   HOA B (NOT h AND P) Q
```

□

One may point out that using them on concrete programs may be problematic because of the universal quantifications on potentially infinite domains in the definition of partial equality and preservation. Fortunately, this is not the case [22]. For confinement, consider as an example the predicate $x = 0$. This predicate is confined by the set $\{x, y\}$. This is represented by the following in HOL:

```
(\s. (toInt o s) "x" = 0) IS CONFINED BY {"x","y"}
```

After rewriting with the definition of confinement and partial equality we get the following 'goal':

```
(!s t. (!x. x IN {"x","y"} ==> (s x = t x))
       ==>
      (((toInt o s) "x" = 0)  =  ((toInt o t) "x" = 0))
)
```

which can be easily proven.

Proving preservation properties is also not problematic. For VSPL, the following theorems can be easily proven in HOL. They specify conditions on which a preservation property of a VSPL statement can be proven without having to quantify over all confined predicates.

Theorem 4.28 :

```
|- ~("x" IN V) ==> ("x" ASG E) PRESERVES V WHEN h
```

□

Theorem 4.29 :

```
|- A PRESERVES V WHEN h
   ==>
   (IF g THEN A ELSE B) PRESERVES V WHEN (g AND h)

|- B PRESERVES V WHEN h
   ==>
   (IF g THEN A ELSE B) PRESERVES V WHEN (NOT g AND h)
```

□

5 Shallow Embedding II

In the previous section we have shown an embedding of VSPL in which VSPL program states are represented by functions from variables to values. This section will briefly show another shallow embedding approach where (concrete) states are represented by records (alternatively, although less sophisticated, one can use tuples). For example, consider a program P with two variables, namely b and x. A state of P in which the value of b is true and the value of x is 0 is represented by the following record: `<| b=T; x=0 |>`.

The approach produces cleaner representations of programs and specifications. The approach is widely used, for example as in [24, 1, 19]. Unfortunately the field names of a record are not first class values in HOL. This has certain disadvantages.

5.1 Semantics

In this embedding we will use the following semantic domains. As in Section 4, the type `Pred` represents predicates, and the type `Stmt` represents statements. However, they are now parameterized by a type variable `'s` which represents the type of states of an arbitrary VSPL programs.

Definition 5.1 : Semantic domains

```
type 's Pred  = 's -> bool
type 's Stmt  = 's -> 's
```

□

With the exception of assignment, VSPL statements and VSPL boolean operators can be defined in the same way as in Section 4, though they should now be defined in terms of the above semantic domains.

An assignment, for example, $x = x + 1$ can be represented by a state transition function: `(\s. s with x := s.x + 1)`. So we (re-) define the HOL constant `ASG` as follows:

Definition 5.2 : Semantics of Assignment

```
ASG f = f
```

□

And the inference rule for assignment now looks like this in HOL:

Definition 5.3 : Assign Rule

```
|- VALID (p IMP (q o f))
   ==>
   HOA (ASG f) p q
```

□

5.2 Representing Program and Specification

Consider again the specification in Example 4.6. The program in this specification has three variables: a boolean variable `b`, and two integer variables `x` and `y`. The states of the program will now be represented by records of the following type:

Code 5.4 :

```
datatype State_A = <| b:bool; x:int; y:int |>
```

□

The specification in Example 4.6 is now represented as follows:

Code 5.5 :

```
HOA

(IF (\s. s.x = s.y)
    THEN (ASG (\s. s with x := (s.x + 1)))
    ELSE (ASG (\s. s with y := (s.y + 1))))

((\s. s.b) AND NOT(\s. s.x = s.y))

((\s. s.b) AND NOT(\s. s.x + 1 = s.y))
```

□

Compared to the representation of the previous section, this one is apparently cleaner.

5.3 Value Space Extension

Unlike in the functional model of states, using records we can easily represent programs with different value spaces (different set of variables). For example, the program in Example 4.10, unlike the one in Example 4.6, has only one variable `z` of type `int list`. To represent this in HOL we simply construct a new record type to represent its states:

Code 5.6 :

```
datatype State_B = <| z : int list |>
```

□

The program can now be represented as follows:

Code 5.7 :

```
IF (\s. s.z = [])
   THEN SKIP
   ELSE ASG (\s. s with z := 1 :: s.z)
```

□

Unfortunately, combining programs with different value spaces is problematic. Consider the programs in Examples 4.6 and 4.10. Let us call the first `A` and the second `B`. Notice that in HOL their types are:

```
A : State_A Stmt
B : State_B Stmt
```

where `State_A` and `State_B` are defined in Codes 5.4 and 5.6. So both programs have different(HOL) types! Composing them, for example like this:

Example 5.8 :

```
IF (\s. s.x=s.y) THEN A ELSE B
```

□

cannot be done because `IF-THEN-ELSE` expects two programs of the same type.

A possible solution is to upgrade `A` and `B` so that they are based on a record type which is big enough to represent the union of their variables, for example:

```
datatype State_C
=
<| b : bool; x : int; y : int; z : int list |>
```

The drawback is that all previous HOL proofs about `A` and `B` have to be re-built.

Another option is to use injections from `State_C` to the old value space of `A` and `B` into `State_C`. For example, if `toA` and `toB` are two such functions, and `fromA` and `fromB` are their inverses:

```
toA   : State_C -> State_A
toB   : State_C -> State_B
fromA : State_A -> State_C
fromB : State_B -> State_C
```

The program in Example 5.8 can now be represented as follows:

```
IF (\s. s.x=s.y)
   THEN (fromA o A o toA)
   ELSE (fromB o B o toB)
```

Old proofs do not have to be rebuilt, but the representation becomes more cluttered[7].

5.4 Guest Logic Type Checking

In Subsection 4.6 we have seen that type checking the guest logic cannot be done if we use the functional representation of states. With deep embedding (Section 6) this becomes possible, though it will require the type checker of the guest logic to be embedded in HOL too. The advantage of record representation is that we can simply use HOL's own type checker to type check the guest logic's sentences. For example, the type incorrect VSPL program in Example 4.18 is now represented like this:

```
IF (\s. s.x)
   THEN (ASG (\s. s with x := F))
   ELSE (ASG (\s. s with x := s.x + 1))
```

which will be rejected by HOL's type checker because (assuming we use `State_A` to represent states) the first assignment tries to assign a boolean value to an integer record field.

5.5 Program Transformation with Variable Constraints

Program transformations such as those in Examples 4.20 and 4.21 can still be expressed, however transformations as the one in Example 4.22 cannot be expressed because they contain a constraint on the set of variables used by the program in question. In a record representation of program states the set of variables of a program is specified by the set of field names of some record. However, this information cannot be retrieved from HOL's logic because field names of HOL records are not first class HOL values.

6 Deep Embedding

Deep embedding extends shallow embedding by representing the grammar of the guest logic in the host logic. It is known that a context free grammar can

[7]The use of extendible record may solve the problem. Reader may refer to [19].

be represented by a set of (typically mutual recursive) functional data types [8]. This kind of data types are available in HOL. Values of those data types fully represent the syntactic structures of sentences of the represented grammar. This allows us to analyze the syntactic structures of the guest logic from HOL and proves various property about the analysis itself.

As an example, we will show a deep embedding of the logic VSPL from Figure 1. We will assume that the deep embedding will be built on top of the functional state semantics given in Section 4. Record based semantics (Section 5) gives a problem –we will say more about this later.

The following definition provides the definition of the HOL data types we use to represent VSPL sentences. Notice the similarity in the structure of the data types and that of VSPL grammar. Because of the similarity, these data types representation is also called *abstract syntax* in literature [8].

Definition 6.1 : VSPL's Abstract Syntax

```
datatype A_expr
    =
    Add         of A_expr => A_expr
  | Equal       of A_expr => A_expr
  | Conj        of A_expr => A_expr
  | Imp         of A_expr => A_expr
  | Not         of A_expr
  | Boolean     of bool
  | Integer     of int
  | Var         of string

datatype A_stmt
  =
    Skip
  | Asg         of string => A_expr
  | IfThenElse  of A_expr => A_stmt => A_stmt

datatype A_spec
  =
    Hoare       of A_stmt => A_expr => A_expr
  | Pred        of A_expr
```

□

VSPL grammars have three syntactic categories, namely *Expr*, *Stmt*, and *Spec*, each is represented by a HOL data type. For each category we will have to define a semantic function that assign a semantics to a given sentence of the category. The semantics is a value from the semantic domain specified in Definition 4.2. We provide three functions:

```
MEx : A_expr -> Expr
```

```
MSt : A_stmt -> Stmt
MSp : A_spec -> Spec
```

Those are, respectively, the semantic function for VSPL expressions, statements, and specifications. They are defined as follows:

Definition 6.2 : VSPL Semantics

```
MEx (Add   e f)  =  fromInt  o (\s. (toInt o MEx e) s + (toInt o MEx f) s)
MEx (Equal e f)  =  fromBool o (\s. MEx e = MEx f)
MEx (Conj  e f)  =  fromBool o ((toBool o MEx e) AND (toBool o MEx f))
MEx (Imp   e f)  =  fromBool o ((toBool o MEx e) IMP (toBool o MEx f))
MEx (Not   e  )  =  fromBool o (NOT (toBool o MEx e))
MEx (Boolean b)  =  fromBool o (\s. b)
MEx (Integer i)  =  fromInt  o (\s. i)
MEx (Var     x)  =  (\s. s "x")

MSt Skip         = SKIP
MSt (Asg x e)    =  x ASG (MSt e)
MSt (IfThenElse g A B) = IF (toBool o MEx g) THEN MSt A ELSE MSt B

MSp (Hoare A P Q)  = HOA (MSp A) (toBool o MEx P) (toBool o MEx Q)
MSp (Pred P)       = VALID (toBool o MEx P)
```

□

Notice that had we used record representation of states –as in Section 5– defining the meaning of `Var x` and `Asg x e` would be problematic.

6.1 Syntax Extension

Changing the syntax of the guest logic means that we have to change it's HOL data type representation. For example, extending the syntax of VSPL with sequential composition as in Example 4.12 has to be matched by the following change in the HOL representation of statements:

```
datatype A_stmt
  =
    Skip
  | Asg         of string => A_expr
  | IfThenElse  of A_expr => A_stmt => A_stmt
  | Seq         of A_stmt => A_stmt
```

The semantic function for statements has to be extended too:

```
MSt Skip               = ... /* as before */
MSt (Asg x e)          = ... /* as before */
MSt (IfThenElse g A B) = ... /* as before */
MSt (Seq A B)          = SEQ (MSt A) (MSt B)
```

Unfortunately because some definitions are changed, older proofs are no longer valid in HOL. They have to be re-executed. Some proofs may even have to be re-coded. For example, if a proof involves an induction over the structure of `A_stmt`, then adding a new alternative into the data type will cause the generation of an extra sub-goal at the induction.

6.2 Extending Value Space

Adding new types of guest logic's values, e.g. adding floats or lists, is already problematic in the shallow embedding level (Subsection 4.5.2). This problem is passed on to the deep embedding extension. Adding new types may also require some extension of the guest logic's syntax, in which case we are also confronted with the problem mentioned in the previous subsection.

6.3 Guest Logic Type Checking

Recall that in functional state representation HOL's own type checker cannot be reused to type check the guest logic's sentences (Subsection 4.6). With deep embedding we can, if we decide to, implement the guest logic's type checker in HOL.

Figure 3 shows a type chekcer for VSPL. `A_type` represents the type language of VSPL. `tiEx` is a function that infers the type of a VSPL expression. The `tc-` functions are functions to type check VSPL expressions, statements, and specifications. Those functions are coded as HOL definitions, but they can be executed like normal functions by repeatedly calling HOL rewrite mechanism. However, this way of doing a syntax analysis is somewhat overkill, since we can simply code those functions in, say, ML and have the ML interpreter execute them much more efficiently. The necessity to embed a type checker, or other kinds of syntax analysis, in HOL comes, for example, when:

- We want to prove properties about the checker itself.
- We want to prove that a given program transformation preserves type correctness.

7 Hybrid Embedding

We have seen that embedding in higher order logic, despite various approaches, remains problematic. To circumvent the problems we are currently experimenting with an alternative approach which we call *hybrid embedding*. It is actually not a pure embedding. In (pure) embedding, a logic is mechanized by representing it as formulas of some already mechanized host logic. In hybrid embedding of a logic, the semantics of the logic is embedded in some host logics. However, the syntactic structure of the logic and the semantic functions translating

```
datatype A_type = TyCons of string

tiEx _ (Add   _ _)  =  TyCons "int"
tiEx _ (Equal _ _)  =  TyCons "bool"
tiEx _ (Conj  _ _)  =  TyCons "bool"
tiEx _ (Imp   _ _)  =  TyCons "bool"
tiEx _ (Not   _  )  =  TyCons "bool"
tiEx _ (Boolean _)  =  TyCons "bool"
tiEx _ (Integer _)  =  TyCons "int"
tiEx c (Var     x)  =  (getType c x)

isInt  (TyCons t)   =  (t = "int")
isBool (TyCons t)   =  (t = "bool")

tcEx c (Add   e f)  =  isInt (tiEx c e) /\ isInt (tiEx c f)
tcEx c (Equal e f)  =  (tiEx c e = tiEx c f)
tcEx c (Conj  e f)  =  isBool (tiEx c e) /\ isBool (tiEx c f)
tcEx c (Imp   e f)  =  isBool (tiEx c e) /\ isBool (tiEx c f)
tcEx c (Not   e  )  =  isBool (tiEx c e)
tcEx _  _           =  true

tcSt _ Skip               =  true
tcSt c (Asg x e)          =  tcEx c e /\ (getType c x = tiEx c e)
tcSt c (IfThenElse g A B) =  tcEx c e /\ tcSt c A /\ tcSt c B

tcSp c (Hoare A P Q)  = tcSt c a /\ tcEx c P /\ tcEx c Q
                        /\
                        isBool (tiEx c P) /\ isBool (tiEx c Q)

tcSp c (Pred P)       = tcEx c P /\ isBool (tiEx c P)
```

Figure 3: VSPL Type Checker

sentences of the logic into their semantics are directly implemented (without embedding).

For example, consider again the VSPL logic given in Figure 1. Subsection 4.2 gives a functional state semantics of VSPL, embedded in HOL. Section 6 shows a deep embedding of VSPL. It provides a set of HOL datatypes representating VSPL grammar (Definition 6.1) and the corresponding semantic functions (Definition 6.2) to translate VSPL sentences to their semantics. In an hybrid embedding, the data types in Definition 6.1 and the semantic functions in Definition 6.2 are not written in HOL. Rather they are coded directly in an ordinary programming language. For example, we use ML whose style of definition resembles that of HOL.

Section 6 shows a combination of deep embedding and functional state semantics. This is a powerful combination, but recall that they have problems. These problems are solved in hybrid embedding:

1. **Readability**

   Problem Embedded sentences of a language L tend to be cluttered since they are encoded as sentences of another language. Large sentences can be quite unreadable. This is actually a problem of all (pure) embedding approaches. This can be solved by providing a set of translators between L and embedded L, but this means that we still need this intermediate layer, not to mention that making such translators may require in-depth knowledge of the implementation of the host logic.

   Solution In hybrid embedding, sentences of the guest logic are not embedded, so there is no readability problem[8] and there is also no need for any intermediate layer.

2. **Syntax Update**

   Problem In deep embedding, changing the syntax of the guest logic also changes the HOL data types representing it, along with its semantic functions This may invalidate some proofs, so they have to be rebuilt. Some may even need recoding.

   Solution The syntax and the semantic functions are not embedded in HOL, consequently updating them has no impact on the existing HOL proofs.

3. **Value space extension**

   Problem In HOL, the functional state semantics cannot represent polymorphism in the guest logic.

[8] Well, we represent those sentences as values of functional data types. For reading or producing the actual sentences one still need to write a set of parsers and pretty printers. We will not go into this, but the required techniques are well known –see for example [8]. There are also tools that greatly simplify the task of writing them.

Solution Subsection 4.5.2 mentions that the problem can be solved if we can treat the types of the host logic as first class values, and hence they can be freely manipulated. Unfortunately, types are not first class in HOL. However HOL is implemented in ML. We can build a hybrid embedding on ML too, hence representing, for example, VSPL grammar, including its type system, in ML. It follows that HOL values, VSPL values, HOL types, and VSPL types are all just plain ML values. So in ML they are all first class, and hence we are saved from the problem mentioned above.

In addition to solving the above mentioned problems, the decision not to embed the grammar of the guest logic makes mechanization by hybrid embedding considerably more programable:

1. We have the option to implement a proof strategy directly rather than embedding it in a host logic. This is unsafe. Left unembedded the correctness of the strategy cannot be verified. On the other hand the soundness of the strategy may be difficult (expensive) to prove and we may want to delay its actual verification until we have enough resources to do so.

   Another point is that a directly implemented proof runs much faster. So we may even want to keep using them even after it has been succesfully embedded and verified in the host logic.

2. For real examples, syntax directed analyses and transformations are very helpful. Think of things like type checking, invariant generation, or adding auxiliary variables. Since the syntax of the guest logic is directly implemented, syntax driven operations have to be directly implemented too (with some exceptions –Subsection 4.7). Although embedding them has an advantage (will be elaborated later), there are also a number of serious objections:

   (a) Embedding them (semantically) is very complicated –see for example an attemp to encode program refinement in shallow embedding by Vos [25].

   (b) They typically are not really where we want the logic to be focuses on. So, including them in the semantics (in the embedding) can seriously clutter the logic.

   (c) As remarked earlier, embedded analyses or transformations run sufficiently slower.

   (d) Some analyses or transformations are either trivial enough or well known that they can be directly and reliably implemented.

3. We can interface to different host logics. Even in a simple case, this is very useful. Consider again the VSPL specification in Example 4.6. Repeatedly

applying VSPL rules will reduce the specification to the following:

$$(b \wedge \neg(x = y) \wedge (x = y) \;\Rightarrow\; b \wedge \neg(x + 1 + 1 = y))$$
$$\wedge$$
$$(b \wedge \neg(x = y) \wedge \neg(x = y) \;\Rightarrow\; b \wedge \neg(x + 1 = y + 1))$$

To prove this we translate the formula to HOL. The translation is carried out by the semantic functions given in Definition 6.2 (though in hybrid embedding these functions are directly implemented rather than embedded). The result is an embedded representation of the above formula –see Code 4.9. However we can also use another set of semantic functions that translate the above formula to a plain HOL formula (one without any embedding information), which is:

```
(b /\ ~(x=y) /\ (x=y)   ==>  b /\ ~(x+1+1=y))
/\
(b /\ ~(x=y) /\ ~(x=y)  ==>  b /\ ~(x+1=y+1))
```

which looks much cleaner, and more importantly, can be proven without having to resolve the state abstraction and the `to-` and the `from-` functions.

Though both translations targets HOL, we actually target different logics embedded in HOL. Targeting an entirely new host logic can be done in the same way, namely by defining a new set of semantics function.

For all the advantages of hybrid embedding mentioned above, some price has to be paid. Because the grammar and the semantics functions of a guest logic are not embedded, it means that properties about them are left unverified. Syntax driven operations are also not embedded, so they are left unverified too. As remarked above, this does not mean that they cannot be verified, since once embedded, they can be verified too. However, we have the option to delay the verification. In an industrial setup this can be an advantage since one does not always have the resources available to do a total verification. Hybrid embedding is also not totally unsafe, since the semantic part of the guest logic is still embedded. For many logics, the semantics is the most delicate part, and hence its embedding is urgent.

8 Conclusion

We have shown how embedding can be done using several methods. Each method has it own weaknesses and strengths. Table 1 summaries those[9].

The shallow and deep embedding approaches have their own limitations. The weakest point of both is the lack of the extension features (i.e. syntax

[9]The symbol '-' and '+' indicate the weak point and the strong point. The double symbol '--' and '++' indicate weaker and stronger point.

Criteria	Shallow I	Shallow II	Deep	Hybrid
Readability	−−	−	+	++
Soundness	+	+	++	−
Syntax Extension	−	−	−−	++
Value Space Extension	−	−	−	+
Type check	−	++	+	+
Program Transformation	+	−	++	++

Table 1: Comparison of the embedding methods

extension and value space extension) that allow models and theories to be reused easily when we add some new parts to them. Readability is also important because less readable models will give less confidence to the user when proving them. Hybrid embedding shows a better result on those criteria since it does not embed all in the theorem prover.

Embedding actually specializes the theorem prover. It is built as a part of the theorem prover and does not modify the theorem prover in any way. Hybrid embedding provides another layer between the user and the theorem prover in order to do the embedding. It forms an interface between a host logic and a guest logic. It is also possible to have several (guest) logics implemented in the hybrid embedding which interface with multiple host logics. Therefore, hybrid embedding combines not only the capability of several computer aided verification tools, but it can also enhance the tactical capability of its basic theorem prover.

There are choices to make in representing a formal semantic, such as trading between completeness and elegance, and trading between completeness and practicality. Hybrid embedding comes from a pragmatic point of view. In favour of practicality, it chooses not to define the complete formal semantic of the programming logic in the theorem prover. The soundness check of the used tools is postponed to provide a clear separation between proving the core problem (input program) and proving the reliability of the used tools. This helps the developer to concentrate on the issues which are directly related to his/her products.

8.1 Future work

This paper reports on a preliminary investigation on embedding. The proposed method of Hybrid Embedding still needs to be improved and used in handling concrete industrial cases to have a better experience. Below are some interesting directions for future research.

- Framework. On verifying a system, a methodological-support is also an important issue beside the basic prover engine [18]. Therefore, having a complete framework that uses hybrid embedding will show how the approach can fit into the current methodology in order to be used in the industrial world.

- Tools combination. We believe that there is no panacea on verification tools. Therefore having a framework that allows the combination of many specific tools would be an advantage. Hybrid embedding allows us to link to several host logics. However, how different results can be composed still requires deeper investigation.

- Reliability. Hybrid embedding is more secure than mechanization without embedding at all, but it is still less secure than pure embedding. There are at least three alternatives to improve the security. The first one is to generate un-embedded simplifiers and transformations from some higher level specification (rather than writing them by hand). The second one is to extend un-embedded functions with some codes that generate proof steps whenever the functions are executed. The generated proof steps can be examined to validate the execution. The third way, and the most difficult one, is to formally specify those functions and then verify them by other tools. We may also want to borrow the idea of *reflection* in theorem prover community[10] while keeping aware not to violate the Gödel's Second Incompleteness Theorem[7].

- Case study. Realistic case study will raise confidence. There are several case studies formulated as challenges to the formal method community. In [20] a verification of cache buffer algorithm using PVS is presented. This is a sub part of an Operating system module. In [15] a description of 'production cell' as one of the manufacturing system is presented. Another known case study for formal method is the RPC memory specification problem, as presented in [4].

References

[1] Sten Agerholm.
Mechanizing program verification in HOL.
Master's thesis, Computer Science Department Aarhus University, 1992.

[2] Flemming Andersen.
A Theorem Prover for UNITY in Higher Order Logic.
PhD thesis, Technical University of Denmark, 1992.

[3] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel.
Experience with embedding hardware description languages in HOL.
In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Nijmegen, 1992. North-Holland.

[4] M. Broy and L. Lamport.
The RPC-Memory specification problem — problem statement.
Lecture Notes in Computer Science, 1169:1–??, 1996.

[5] K.M. Chandy and J. Misra.
Parallel Program Design.

Addison-Wesley, Austin, Texas, May 1989.

[6] Edmund M. Clarke and Jeannette M. Wing.
Formal methods: State of the art and future directions.
Technical Report CMU-CS-96-178, Carnegie Melon University, Pittsburgh, USA, September 1996.

[7] Martin Davis.
Computability and Unsolvability.
McGraw-Hill, 1958.

[8] J. Fokker.
Functional parsers.
Lecture Notes in Computer Science, 925, 1995.

[9] M.J.C. Gordon and T.F. Melham.
Introduction to HOL.
Cambridge University Press, 1993.

[10] John Harrison.
Metatheory and reflection in theorem proving: A survey and critique.
Technical Report CRC-053, SRI Cambridge, UK, February 1995.

[11] Peter V. Homeier and David F. Martin.
Mechanical verification of total correctness through diversion verification conditions.
In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, 1998.

[12] Marieke Huisman.
Java Program Verification in Higher-Order Logic with PVS and Isabelle.
PhD thesis, University of Nijmegen, The Netherlands, 2001.

[13] Bart Jacobs and Erik Poll.
A logic for the Java Modelling Language (JML).
Technical Report CSI-R0018, Catholic University of Nijmegen, November 2000.

[14] Twan Laan.
The Evolution of Type Theory In Logic And Mathematics.
PhD thesis, Technische Universiteit Eindhoven, 1997.

[15] T. Lindner.
Task description.
Lecture Notes in Computer Science, 891:7–??, 1995.

[16] Savitri Maharaj.
A Type-Theoretic Analysis of Modular Specifications.
PhD thesis, University of Edinburgh, 1996.

[17] M.J.C. Gordon.
Mechanizing programming logics in Higher-order logic.
In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings*

of the Workshop on Hardware Verification), pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[18] César Mu noz and John Rushby.
Structural embeddings: Mechanization with method.
In Jeannette Wing and Jim Woodcock, editors, *FM99: The World Congress in Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 452–471, Toulouse, France, sep 1999. Springer-Verlag.

[19] Lawrence C. Paulson.
Mechanizing UNITY in Isabelle.
ACM Transactions on Computational Logic, 1(1):3–32, July 2000.

[20] N. S. Pendharkar and K. Gopinath.
Formal verification of an O.S. submodule.
Lecture Notes in Computer Science, 1530:197–??, 1998.

[21] I.S.W.B. Prasetya.
Mechanically Supported Design of Self-stabilizing Algorithms.
PhD thesis, Utrecht University, 1995.

[22] I.S.W.B. Prasetya, T. Vos, S.D. Swierstra, and B. Widjaja.
A theory for composing distributed components, based on mutual exclusion.
Draft. Presently available via : www.cs.uu.nl/people/wishnu.

[23] Donald Sannella and Andrzej Tarlecki.
Algebraic methods for specification and formal development of programs.
ACM Computing Surveys, 31(3es), 1999.

[24] J. von Wright and K. Sere.
Program transformations and refinements in HOL.
In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 231–241, Los Alamitos, CA, USA, August 1992. IEEE Computer Society Press.

[25] Tanja Vos.
Unity in Diversity: A Stratified Approach to the Verification of Distributed Algorithm.
PhD thesis, Utrecht University, 2000.