Refinement Calculus for Logic Programming in Isabelle/HOL

David Hemer[1], Ian Hayes[2], and Paul Strooper[2]

[1] Software Verification Research Centre
The University of Queensland, Australia
hemer@svrc.uq.edu.au
[2] School of Computer Science and Electrical Engineering
The University of Queensland, Australia

Abstract. This paper describes a deep embedding of a refinement calculus for logic programs in Isabelle/HOL. It extends a previous tool with support for procedures and recursion. The tool supports refinement in context, and a number of window-inference tactics that ease the burden on the user. In this paper, we also discuss the insights gained into the suitability of different logics for embedding refinement calculii (applicable to both declarative and imperative paradigms). In particular, we discuss the richness of the language, choice between typed and untyped logics, automated proof support, support for user-defined tactics, and representation of program states.

1 Introduction

In this paper we present a deep embedding of a refinement calculus for logic programs [8] based on the Isabelle/HOL logic [13,1]; this embedding forms part of the `Marvin` tool. The refinement calculus includes a wide-spectrum language, a declarative semantics for the language, and a collection of refinement laws. The wide-spectrum language includes executable constructs such as sequential conjunction, disjunction, existential quantification, procedures and recursion; the language also includes a number of specification constructs, including assumptions, specifications, parallel conjunction and universal quantification. The semantics of the language is described in terms of *executions*, which are mappings between initial and final states. Refinement laws have been proven in terms of the semantic framework, including: algebraic properties of the language; monotonicity rules; rules for lifting predicate connectives to the command level; rules for introducing commands; and rules for manipulating specifications and assumptions. These laws are applied by the user to refine programs to implementable code. `Marvin` lets the user write procedures, that can be called within other procedures or recursively. When introducing recursive calls, `Marvin` ensures the recursion terminates by insisting that the arguments to the recursive procedure are strictly decreasing with respect to a well-founded relation.

In embedding the refinement calculus we have two main choices : a *shallow* or a *deep* embedding. A shallow embedding avoids explicitly defining a separate

R.J. Boulton and P.B. Jackson (Eds.): TPHOLs 2001, LNCS 2152, pp. 249–264, 2001.
© Springer-Verlag Berlin Heidelberg 2001

syntax for the language. Rather it embeds the language directly in the host syntax and expresses the semantics of the language in terms of this representation. A deep embedding, on the other hand, explicitly represents an abstract syntax of the language and provides an explicit function to map the syntax to the semantics as defined in the host language. Thus it decouples issues of syntactic equality from semantic equality. In this paper we describe a deep embedding of a refinement calculus.

`Marvin` supports refinement in context, where information about the program environment and local assumptions are automatically passed down to subcommands. This context information can then be used in the refinement of these subcommands. The approach used in `Marvin` is similar to *program window inference*, developed by Nickson and Hayes [12]. This approach allows context information to be handled separately from the rest of the command. `Marvin` also includes several window-inference-like tactics that support step-wise refinement of programs, thereby greatly reducing the number and complexity of individual refinement steps.

In providing tool support for the refinement calculus we were initially faced with the question of what logic/proof tool to embed the calculus in. Prototype tools had been developed in several different logics prior to the version described in this paper. An earlier prototype [5] — based on a different semantic framework [7] — includes a shallow embedding of the calculus in the Ergo 4 theorem prover [19], a prover based on the window-inference paradigm. Two more embeddings (also based on the earlier semantics) were done in Ergo 5 [20] (a generic theorem prover based on sequent calculus) and Isabelle/ZF as part of a comparative study [9]. Finally, before settling on the current version, an embedding, based on the current semantics, was investigated in HOL-Z [11] (an embedding of the Z specification language in Isabelle/HOL).

At first, the choice of logic seemed rather arbitrary, but after experimenting with the different logics, we found several features of refinement calculi that have a fundamental effect on the choice of logics. Before choosing a logic, several questions need to be answered:

1. Can the constructs that model the syntax and semantics of the wide-spectrum language be easily represented in the logic?
2. Do the semantics include types?
3. To what extent does the proof tool automate proof of properties of the semantics and refinement laws, as well as the proof obligations that arise during refinement?
4. Can tactics be implemented to assist the user during the refinement process?
5. How readily can program states be represented in the logic?

In this paper, we discuss how the answers to these questions affect the choice of logic, using the embedding of the logic program refinement calculus as an example. Section 2 gives details of an embedding of the calculus in Isabelle/HOL. In Section 3 we illustrate the tool by showing the refinement of a simple program. We then show how this appears in the tool for the first couple of steps. In Section 4 we discuss how the questions raised above affect the choice of logic.

2 Refinement Calculus for Logic Programs in Isabelle/HOL

2.1 Preliminary Definitions

The semantics of our wide-spectrum language are defined in terms of a relationship between the initial and final states of a program. A state consists of a set of bindings, where a binding is a function that maps each variable to a value. The type *Val* represents the values that program variables can take; we wish to associate these with Isabelle types in some manner. To do this we represent *Val* as a type sum (using a datatype declaration):

```
Bnds == Var → Val
State == ℙ Bnds
datatype Val ==
     natV nat | listV (nat list) | setV (nat set) | pairV Val Val
```

The building blocks of our language are terms and predicates. A term is defined as either a variable or a function application to a list of terms. The functions *evallist* and *eval* evaluate a list of terms and a term respectively for a particular binding. Terms are evaluated to an element of *Val*. Predicates are modelled using a datatype; they are evaluated, with respect to a binding, by the function *evalp*, a boolean-valued result is returned.

```
datatype rterm == varT Var | funT (rterm list)

evallist : rterm list × Bnds → Val list
eval : rterm × Bnds → Val

datatype pred == pred ∧ pred | pred ∨ pred | pred ⇒ pred
     | pred ⇔ pred | rterm = rterm | rterm < rterm
     | ¬ pred | ∃ Var pred | ∀ Var pred | true | false
     | isNat Var | isList Var | isSet Var | isPair Var

evalp :: pred × Bnds → bool
```

2.2 Wide-Spectrum Language

The wide spectrum language includes both logic programming language and specification constructs. It allows constructs that may not be executable, allowing gradual refinement without the need for notational changes during the refinement process. The syntax of the commands in the wide-spectrum language is defined by the datatype *Cmd*.

```
datatype Cmd == ⟨ pred ⟩       -- specification
             | { pred }        -- assumption
             | Cmd ∧ Cmd       -- parallel conjunction
             | Cmd ∨ Cmd       -- parallel disjunction
             | Cmd , Cmd       -- sequential conjunction
             | ∃ Var • Cmd     -- existential quantification
             | ∀ Var • Cmd     -- universal quantification
             | PIdent(rterm)   -- procedure call
```

A specification $\langle P \rangle$ represents a set of instantiations of free variables of P that satisfy P. An assumption $\{A\}$ formally states the context in which a program is called. If an assumption does not hold, the program fragment in which it appears may abort. Aborting includes program behaviour such as nontermination and abnormal termination due to exceptions like division by zero (the latter being distinct from a procedure that fails if the divisor is zero), as well as termination with arbitrary results.

Sequential conjunction (S, T) evaluates S before T. The parallel conjunction $(S \wedge T)$ evaluates S and T independently and forms the intersection of their respective results on completion. The disjunction of two programs $(S \vee T)$ computes the union of the results of the two programs. Disjunction is generalised to an existential quantifier $(\exists X \bullet S)$, which computes the union of the results of S for all possible values of X. Similarly, the universal quantifier $(\forall X \bullet S)$ computes the intersection of the results of S for all possible values of X. A procedure can be called by referring to the procedure identifier and applying it to a term, e.g., $pid(T)$ applies the procedure named pid to the term T.

Parameterised commands are defined as a separate datatype `PCmd` consisting of two entries: non-recursive commands and recursive commands.

```
datatype PCmd == Var :- Cmd | re PIdent • PCmd er
```

A command can be parameterised over a variable; it has the form v`:-`S, where v is a variable and S is a wide-spectrum program. A parameterised command is instantiated by applying it to a term — the result is a command.

Recursive calls are introduced within a recursion block. A recursion block has the form **re** $id \bullet pc$ **er**, where id names the block and pc is a parameterised command. Within recursion blocks, recursive calls of the form $id(T)$ can be used, provided certain termination conditions are met. We associate a well-founded relation with the arguments of the recursive call, and prove that the arguments are strictly decreasing with respect to the relation.

Procedures are defined by associating a procedure identifier, pid, with a parameterised command, e.g., $pid \mathrel{\widehat=} v$`:-`$S$.

2.3 Execution Semantics

The semantics of the wide-spectrum language is defined in terms of the function `exec`, which maps each command to a partial mapping between initial and final states (so-called *executions*). The mapping is partial because programs may not defined for all initial states.

We begin by defining the type `StateMap`, representing the set of partial mappings between initial and final states. Partial mappings are defined in Isabelle/HOL in terms of total functions. The labels `Some` and `None` are used to distinguish defined and undefined values. More precisely, for a partial mapping f and element x, if x is in the domain of f then there exists a y such that `f(x) = Some y`, otherwise `f(x) = None`.

```
StateMap == State ⇸ State
```

We define the type *Exec*, as a subtype of *StateMap*, using Isabelle's *typedef* facility. The three conditions in the set comprehension correspond to the three healthiness conditions for executions [8].

```
Exec == {e: StateMap | dom(e) = ℙ {b. {b} ∈ dom(e)}
     ∧ ∀ s ∈ dom(e). the e(s) ⊆ s
     ∧ ∀ s ∈ dom(e). e(s) = Some {b. b ∈ s & (e({b}) = Some {b})}}
```

These conditions characterise useful properties of our wide-spectrum language that allow us to consider the effect of language constructs on a single binding rather than having to look at the effect on an entire state. The first condition states that the domain of an execution is equivalent to those states formed by merging states consisting of a single binding in the domain of the execution. Effectively the domain of an execution is closed under union and intersection (of states in the domain). The second condition states that executions are restricted to those mappings for which the output state is a subset of the input state. The third condition states that for an input state, the output state can be determined by considering the effect of the execution on each singleton binding in the input state, and then forming the union of the results.

The *typedef* facility defines *Exec* as a meta-level type, as well as an object level set. In defining a type in Isabelle/HOL, we must exhibit that the set is non-empty which is easily established by showing that the element $\{\varnothing \mapsto \varnothing\}$ is in the set.

The definition of *Exec* illustrates the overhead that is involved in representing partial functions in Isabelle/HOL. The second line of the definition requires the function "*the*", used in this case to strip away the label *Some* associated with *e(s)*. Conversely, in the third line we must include the label *Some*.

Programs are refined in the context of a program environment, describing any predefined procedures. The type *Env* is a mapping from procedure identifiers to their corresponding parameterised executions. Parameterised executions are defined as mappings from terms to executions, and provide the semantics of parameterised commands. Note that in this case *Exec* is used as a meta-level type.

```
PExec == rterm → Exec
Env == PIdent ↛ PExec
```

The function *exec*, used to define the semantics of the wide-spectrum language, is declared as a function mapping an environment and a command to a state mapping.

```
exec : Env × Cmd → StateMap
```

The translation of the *exec* into Isabelle/HOL was straightforward, with two exceptions. Firstly, to define *exec* for sequential conjunction, we needed to define composition of partial functions. Secondly, in defining *exec* for procedure calls we needed to perform a type coercion between the type *StateMap* (the expected type) and the type *Exec* (the actual type). This was done by introducing a call

to the auxiliary function *Rep_Exec* (created by the *typedef* facility), that maps an abstract type to its representative type.

Having defined the *exec* function, we prove that for any environment ρ and command c, *(exec ρ c)* is of type *Exec*, and thus satisfies the three healthiness properties above. This is proved by structural induction on the command c. Notice in this case we are using *Exec* as an object level set. Defining types using Isabelle/HOL's *typedef* facility allows this dual usage.

The number of proof steps for this theorem was significantly less than the same proof in an earlier embedding in HOL-Z. This can be attributed to the fact that the HOL-Z proof involved large amounts of type checking (the types in HOL-Z are modelled as sets at the object level, and therefore do not make use of Isabelle/HOL's built-in type checker), as well as the fact that the automated proof tactics were more effective in Isabelle/HOL.

2.4 Refinement

We define the refinement relations $\sqsubseteq$ and $\sqsupseteq$ for state mappings (and consequently for executions).

```
e1 ⊑ e2 == dom(e1) ⊆ dom(e2) ∧ (∀ s: dom(e1). e1(s) = e2(s))
e1 ⊒ e2 == (e1 = e2)
```

The above definition states that an execution can be refined by extending the set of initial states for which it is defined. Next we define the notion of refinement for commands, including the context in which the commands are refined. The context includes the environment, *env*, under which the commands are refined.

```
env ρ ⊩ c1 ⊒ c2 == (exec ρ c1) ⊒ (exec ρ c2)
env ρ ⊩ c1 ⊑ c2 == (exec ρ c1) ⊑ (exec ρ c2)
```

Refinement of a command is defined in terms of its underlying execution. A command is refined by extending the set of initial states for which we can associate a final state, effectively removing undefinedness. "$\sqsubseteq$" is a preorder and "$\sqsupseteq$" is an equivalence relation. These properties are exploited to perform step-wise refinement.

Refinement laws for the wide-spectrum language have been proven in terms of the underlying execution semantics. When performing program refinement, these refinement laws are used to prove the correctness of individual refinement steps, rather than apply the definition of refinement directly.

The collection of rules include: algebraic properties for the propositional constructs; monotonicity rules for the language constructs; rules for lifting predicate connectives appearing in specifications to corresponding program constructs; rules for introducing and eliminating program constructs; and rules for manipulating specifications and assumptions.

Monotonicity. The wide-spectrum language is monotonic with respect to refinement in that a command can be refined by refining one or more of its subcommands. Consider, for example the law sand-right-mono given as:

$$\frac{env\ \rho \Vdash c1 \sqsubseteq c2}{env\ \rho \Vdash c3\ ,\ c1 \sqsubseteq c3\ ,\ c2}$$

This law states that the command "$c3\ ,\ c1$" can be refined by refining the second command $c1$.

Predicate Lifting. Predicate lifting laws are used to lift predicate connectives contained in specifications to their program construct counterparts. For example the rule *lift-exists* lifts an existential quantifier within a specification to a command-level existential quantifier:

$$env\ \rho \Vdash \langle \exists\ V \bullet P \rangle \sqsubseteq \exists\ V \bullet \langle P \rangle$$

Note that the existential quantifier symbol $\exists$ is overloaded in the above law. The first occurence is the predicate quantifier, while the second occurence is the command quantifier.

Table 1 gives a comparison of the number of steps required to prove various refinement laws in Isabelle/HOL and HOL-Z. The table shows the reduction in proof steps in Isabelle/HOL, which can be attributed to the built-in type checker and improved automated proof support.

Table 1. Comparison of Proof Steps to Prove Refinement Laws.

Law	Isabelle/HOL	HOL-Z
lift-exists	4	75
por-mono	26	37
distribute-pand-over-por	4	8
left-distribute-sand-over-por	3	122
weaken-assume	5	54

2.5 Assumption Context

We have already seen that commands are refined in the context of a program environment ρ. In this section we describe a second kind of context, the so-called *assumption context* (based on the "**pre**" context used in the program window inference of Nickson and Hayes [12]), assumptions are introduced with the keyword "*assume*". Consider the command "$\{A\}\ ,\ S$". In refining S and its subcommands we may use the assumption A. Rather than having to propagate the assumption throughout S and its subcommands, we add the assumption A to the context of the refinement. Refinement of commands in the context of assumptions extends the definition of command refinement given in Section 2.4.

$$env\ \rho\ assume\ A \Vdash c1 \sqsubseteq c2 == env\ \rho \Vdash \{A\}\ ,\ c1 \sqsubseteq \{A\}\ ,\ c2$$

Extending refinement to include assumption contexts allows us to prove refinement laws that use assumption contexts to manipulate specifications and assumptions. The equivalent-spec law states that a specification can be replaced by one that is equivalent under the assumptions in context:

$$\frac{\texttt{A} \Rrightarrow (P \Leftrightarrow Q)}{\texttt{env}\ \rho\ \texttt{assume}\ \texttt{A} \Vdash \langle P \rangle \sqsubseteq \langle Q \rangle}$$

where $\Rrightarrow$ is the entails operator, mapping two predicates to a boolean, defined as:

$$(p1 \Rrightarrow p2) \Leftrightarrow (\forall\, b \bullet p1(b) \Rightarrow p2(b))$$

For example, under the assumption `X = []`, the specification "$\langle Y = lenX \rangle$" can be refined to "$\langle Y = 0 \rangle$", by observing that `X = []` $\Rrightarrow$ `Y = len X` $\Leftrightarrow$ `Y = 0`.

We have also proved laws for extending the assumption context. For example, the assumption context can be extended by using the law assumpt-in-context, a specialisation of the (right) monotonicity law for sequential conjunction.

$$\frac{\texttt{env}\ \rho\ \texttt{assume}\ A \wedge B \Vdash \texttt{c1} \sqsubseteq \texttt{c2}}{\texttt{env}\ \rho\ \texttt{assume}\ \texttt{A} \Vdash \{B\}, c1 \sqsubseteq \{B\}, c2}$$

Similarly the law spec-in-context states that specifications can be used to extend the assumption context:

$$\frac{\texttt{env}\ \rho\ \texttt{assume}\ A \wedge P \Vdash \texttt{c1} \sqsubseteq \texttt{c2}}{\texttt{env}\ \rho\ \texttt{assume}\ \texttt{A} \Vdash \langle P \rangle, c1 \sqsubseteq \langle P \rangle, c2}$$

2.6 Supporting Window Inference

The window-inference proof paradigm [14] (on which Ergo 4 is based) provides support for stepwise refinement. To support window inference in Isabelle/HOL we have implemented four tactics — `optac`, `close`, `transtac` and `trans_inst_tac` — which can all be applied regardless of the underlying refinement relation. Each tactic operates on the first subgoal, with any newly created subgoals moving to the top of the subgoal stack. These tactics are described in more detail separately below.

Tactic `optac` supports monotonic refinement, where a command is refined by refining one of its components. The subcommand to be refined is indicated by a list of numbers; for example, for the command "$S, (T \wedge U)$", `[1]` refers to the subcommand `S` and `[2,2]` refers to the subcommand `U`. Tactic `optac` updates the assumption context by calling rules such as `assumpt_in_context` and `spec_in_context` where appropriate. Consider the subgoal:

```
1. env ρ assume A ⊩ ⟨P⟩, (T, U) ⊑ ?x
```

where `?x` is a meta-variable used to represent the as yet to be determined refined command. Focusing on the subcommand `T` (by calling `optac [2,1]`), adds `P` to the context for `T`. The refinement of `T` becomes the first subgoal, with `T` being replaced by the program it is refined to in the original command, i.e.,

```
1. env ρ assume A ∧ P ⊩ T ⊑ ?y
2. env ρ assume A ⊩ ⟨P⟩ , (?y , U) ⊑ ?x
```

Tactic `close` applies an appropriate reflexivity rule to instantiate an unknown on the right-hand side of a refinement relation to the command on the left-hand side, effectively removing the first subgoal. For example, applying *close* to

```
1. env ρ assume A ⊩ S ⊑ ?x
```

removes subgoal 1 from the subgoal stack, and instantiates ?x to S.

Tactic *transtac* implements step-wise refinement; it transforms the current program to an intermediate program by applying a single refinement law. The context information is passed to the new intermediate program. If the refinement law includes any assumptions, then a proof obligation is generated to show that the assumptions are satisfied in the current context.

Consider the subgoal

```
1. env ρ assume A ⊩ S ⊑ ?x
```

applying *transtac* to a law of the form

$$\frac{C(\rho, A)}{\texttt{env } \rho \texttt{ assume } A \Vdash S \sqsubseteq T}$$

replaces the above subgoal with

```
1. C(ρ, A)
2. env ρ assume A ⊩ T ⊑ ?x
```

Tactic *transtac* applies Isabelle/HOL's simplification tactics to the proof obligation in the first subgoal. In most cases, where the proof obligation is valid, the proof obligation is discharged automatically.

Often the refinement law that we want to apply involves $\sqsupseteq$ instead of $\sqsubseteq$. In this case, *transtac* will weaken the law by replacing $\sqsupseteq$ by $\sqsubseteq$. Similarly, in some cases we want to apply a law containing $\sqsupseteq$ in the reverse direction, in this case *transtac* will attempt to apply such a rule in both directions.

Tactic *trans_inst_tac* is similar to *transtac*, except that it also allows the user to instantiate some or all of the meta-variables that occur in the law.

3 Example

In this section, we illustrate `Marvin` by showing the refinement of a simple example for testing list membership by recursively traversing the list. The refinement process begins by supplying a top-level program:

$$(X, L) \texttt{ :- } \{isList(L)\} , \langle X \in elems(L) \rangle$$

The program, parameterised over the variables X and L, includes assumptions that L is a list. The specification states that X is in the set of elements that appear in the list.

The individual refinement steps shown here correspond to those used in `Marvin`. However, we use a more readable notation, based on the structured calculational proof notation described in [2]. The notation is indented to represent the depth of the subcommand that is currently being refined; 'n •' and '.' represent the beginning and end respectively of a refinement of a subcommand (where n is the depth of the subcommand). The notation $\llcorner S \lrcorner$ represents opening on the subcommand S (using `optac`). Application of refinement laws (using `transtac` or `trans_inst_tac`) is represented by a refinement relation ($\sqsubseteq$ or $\sqsubseteq\!\!\!\!-$) followed by the name of the rule. The individual assumptions that make up the assumption context are labelled as they are introduced. Closing subcommands (using `close`) is represented by removal of indentation. Appendix A gives listing of the laws used in this example that have not been described earlier.

The refinement of the membership program starts by setting up the framework for introducing recursive calls. We apply the recursion-intro law to introduce a recursion block, `Member`. Included is an assumption stating that specifications of the form $\langle X1 \in elems(L1)\rangle$ can be refined to the procedure call `Member(X1,L1)`, provided $L1$ is a list, and the length of $L1$ is strictly less than the length of L. The last condition ensures the recursive procedure terminates.

(X, L) **:-** $\{isList(L)\}$, $\langle X \in elems(L)\rangle$
$\sqsubseteq\!\!\!\!-$ recursion-intro*: $<$ is well-founded on* $\mathbb{N}$
re *Member* • (X, L) **:-**
$\{\forall X1, L1 \bullet$ *env* ρ *ass* $\#L1 < \#L \wedge isList(L1)$
$\Vdash \langle X1 \in elems(L1)\rangle \sqsubseteq Member(X1, L1)\}$,
$\{isList(L)\}$, $\llcorner\langle X \in elems(L)\rangle\lrcorner$
er

Focusing on the specification, we can assume the assumptions prior to the specification:

Assumption 1: $\forall X1, L1 \bullet$ *env* ρ *ass* $\#L1 < \#L \wedge isList(L1)$
$\Vdash \langle X1 \in elems(L1)\rangle \sqsubseteq Member(X1, L1)$
Assumption 3: $isList(L)$
1 • $\langle X \in elems(L)\rangle$
$\sqsubseteq$ intro-spec
$\llcorner\langle L = [\,] \vee \exists H, T \bullet L = cons(H, T)\rangle\lrcorner \wedge \langle X \in elems(L)\rangle$
2 • $\langle L = \langle\,\rangle \vee \exists H, T \bullet L = cons(H, T)\rangle$
$\sqsubseteq$ lift-por
$\langle L = [\,]\rangle \vee \langle \exists H, T \bullet L = cons(H, T)\rangle$
. $\langle L = [\,]\rangle \vee \langle \exists H, T \bullet L = cons(H, T)\rangle \wedge \langle X \in elems(L)\rangle$
$\sqsubseteq$ right-distribute-pand-over-por
$\llcorner\langle L = [\,]\rangle \wedge \langle X \in elems(L)\rangle\lrcorner \vee \langle \exists H, T \bullet L = cons(H, T)\rangle \wedge \langle X \in elems(L)\rangle$

Next we focus on the subcommand on the left of the disjunction, firstly converting the parallel conjunction to sequential conjunction, then replacing the specification $\langle X \in elems(L)\rangle$ with $\langle false\rangle$ using the assumption context.

2 • $\langle L = [\,]\rangle \wedge \langle X \in elems(L)\rangle$
$\sqsubseteq$ pand-to-sand

$\langle L = [\,] \rangle \,, \llcorner \langle X \in elems(L) \rangle \lrcorner$
`Assumption 4:` $L = [\,]$
3 • $\langle X \in elems(L) \rangle$
$\sqsubseteq$ equivalent-spec $L = [\,] \Rrightarrow X \in elems(L) \Leftrightarrow false$
$\langle false \rangle$
. $\langle L = [\,] \rangle \,, \langle false \rangle$
$\sqsubseteq$ sand-false
$\langle false \rangle$

We now focus on the subcommand on the right of the disjunction. We begin by lifting the existential quantifiers in the specification to the command level, then expand the scope of these quantifiers over the parallel conjunction (the rules lift-existsx2 and expand-scope-existsx2 correspond to lift-exists and expand-scope-exists except they apply to programs quantified over two variables). The parallel conjunction is then replaced by sequential conjunction.

2 • $\llcorner \langle \exists\, H, T \bullet L = cons(H, T) \rangle \lrcorner \wedge \langle X \in elems(L) \rangle$
3 • $\langle \exists\, H, T \bullet L = cons(H, T) \rangle$
$\sqsubseteq$ lift-existsx2
$\exists\, H, T \bullet \langle L = cons(H, T) \rangle$
. $(\exists\, H, T \bullet \langle L = cons(H, T) \rangle) \wedge \langle X \in elems(L) \rangle$
$\sqsubseteq$ expand-scope-existsx2
$\exists\, H, T \bullet \llcorner \langle L = cons(H, T) \rangle \wedge \langle X \in elems(L) \rangle \lrcorner$
4 • $\langle L = cons(H, T) \rangle \wedge \langle X \in elems(L) \rangle$
$\sqsubseteq$ pand-to-sand
$\langle L = cons(H, T) \rangle \,, \llcorner \langle X \in elems(L) \rangle \lrcorner$

The specification $\langle X \in elems(L) \rangle$ is replaced by noting that under the assumption $L = cons(H, T)$, either X equals the first element of L, or X is an element of the remainder of the list. The introduced disjunction is then lifted to the command level. Finally we replace the specification $\langle X \in elems(T) \rangle$ by $Member(X, T)$ using Assumption 1, and noting that from the assumption $L = cons(H, T)$ it follows that $\#T < \#L$.

`Assumption 5:` $L = cons(H, T)$
5 • $\langle X \in elems(L) \rangle$
$\sqsubseteq$ equivalent-spec, `assumption 5`
$\langle X = H \vee X \in elems(T) \rangle$
$\sqsubseteq$ lift-por
$\langle X = H \rangle \vee \llcorner \langle X \in elems(T) \rangle \lrcorner$
6 • $\langle X \in elems(T) \rangle$
$\sqsubseteq$ `Assumption 1`
$Member(X, T)$
. $\langle X = H \rangle \vee Member(X, T)$

Putting this all together, removing assumptions, using remove-assumpt, and removing the first specification $\langle false \rangle$, using por-false we get:

re $Member \bullet (X, L)$ **:-**
$(\exists H, T \bullet \langle L = cons(H, T) \rangle , (\langle X = H \rangle \lor Member(X, T)))$
er

The resulting program is now at a sufficiently concrete level to be readily translated into code in a language such as Prolog or Mercury.

4 Discussion

There are a number of different embeddings of refinement calculi in theorem provers [3]. While these embeddings differ in many ways, there are a number of fundamental issues for each. In this section we present these issues, comparing the solutions presented in this paper to solutions presented elsewhere.

4.1 Richness of Language

The first issue relates to the richness of the syntax of the logic. Can the syntax and semantics of the refinement language be conveniently expressed in the logic? In particular, when the language has already been specified using some formal language (such as Z), can this specification be readily translated (by hand or indeed automatically) into the notation used by the logic?

The semantics of our language [8] are formally specified using the Z specification language [15]; this in turn is based on Zermelo Frankel set theory. Therefore it was relatively straightforward to translate the semantics into HOL-Z and Isabelle/ZF notation. However, while the translation to Isabelle/HOL notation required a little more effort, we found that in reference to the other issues raised below, the extra effort up front was more than rewarded later on. We conclude here that we need to be wary of selecting a logic primarily because of its richness without considering the other factors. Also note that while in this case ZF and HOL-Z were closer to the specification language than HOL, this is not necessarily always the case. For example users of higher-order logics might specify their calculus using a language that is closer to HOL.

4.2 Typed versus Untyped Logics

The second issue involves the choice of an untyped or typed logic. Embedding refinement calculi in logics such as HOL-Z or Isabelle/ZF requires that any types in the semantics be expressed at the object level (as sets). The result of this is that large parts of proofs involve discharging type constraints. These same type constraints are handled automatically by a built-in type checker in the typed Isabelle/HOL logic.

Isabelle/HOL is a typed logic, meaning that types used in our semantics can be modelled at the meta-level. Isabelle/HOL employs a built-in type checker that automatically checks type constraints, therefore simplifying proofs. New types are introduced using either type synonyms or type definitions. The type system

available in Isabelle/HOL is not as rich as those used in the untyped logics (where types are modelled as sets). However it does include the basic types required to model our semantics; including natural numbers, lists, sets and functions.

4.3 Automated Proof Support

The third issue relates to the fact that embeddings of refinement calculi involve a large amount of proof. An embedding will typically involve establishing some general properties of the language and proving refinement laws. Automated proof support to take care of the mundane proof steps is useful here. However it is worth noting that embedding the calculus is a one-off process usually performed by somebody with expertise in theorem proving; of more importance is automatically discharging proof obligations that arise during refinement of programs. The extra importance here is twofold: firstly refinement of programs is an ongoing process, and secondly the user of the tool is not necessarily an expert in theorem proving.

We found that Isabelle/HOL gave us a better level of automation (and consequently fewer proof steps) than the other logics, in both setting up and using the tool. This is despite the fact that Isabelle/ZF and HOL-Z both employ the same tactics as Isabelle/HOL. The decreased performance for these two logics can be partly explained by the fact that the tactics struggled to discharge type constraints[1]. For HOL-Z, we also note that the tactics have been set up for the HOL logic, but have not been extended to provide support for the HOL-Z theories.

An earlier report [9] shows that the tactics employed by Ergo 4 and Ergo 5 were in turn significantly less effective than those used in Isabelle/ZF.

4.4 User-Defined Tactics

In relation to tactics, we observe that significant reductions in both the number and complexity of refinement steps can be achieved by using a window-inference paradigm. Some theorem provers, most notably Ergo 4 [19], provide native support for window inference. The majority, however, are based on other proof paradigms, in which case it is important to be able to emulate the window-inference paradigm through the use of tactics [6,16]. Only those provers that offer user-extensionality, usually in the form of tactics, are suitable in this case.

As we have seen earlier in the paper, it was a straightforward task to define tactics in Isabelle/HOL that emulate the window inference paradigm. Indeed we were able to write (or adapt) tactics that emulate window inference in all of the logics that we have trialed. The approach used here is similar to that used by [16] in Isabelle/ZF, except we provide better support for context, based on program window inference [12].

[1] We used an earlier version of Isabelle/ZF. The most recent version includes tactics for discharging type constraints and we have not evaluated the effectiveness of these tactics.

4.5 Representing Program State

Finally, we need to consider how we can model the state of programs within a particular logic. More precisely, given that a state associates program variables with values, we need to consider what range of values can be modelled, i.e., how do we model the type of values? Earlier embeddings tended to ignore the issue by only allowing the set of natural numbers as values. However, this severely restricts the application of the tool.

States are represented in our embedding as sets of bindings, which are in turn mappings from variables to values. The type of values is represented as a sum of Isabelle types (including lists, sets, naturals and pairs). This sum of types is static, therefore introducing a new value type can only be done by reworking the semantics. This would be of no use if we wanted to introduce data abstraction, where users can introduce new value types on the fly.

Von Wright [21] represents states as a tuple of values, with each state component (program variable) corresponding to a value in the tuple. Each state is assigned a product of types, with one type for each variable. Staples [17] notes some limitations of this approach, in particular the fact that variables are not explicitly named.

Staples's [17,18] solution to these problems is to represent states as dependently typed functions from variable names to under-specified families of types. This function is dynamically extendible, meaning that arbitrary abstract types for local program variables can be introduced during the refinement.

5 Conclusions

Using an embedding of a refinement calculus in a theorem prover leads to greater assurance that any proof obligations for individual refinement steps have been satisfied. The more mundane proof obligations can often be discharged automatically by the prover. In this paper we presented a deep embedding of a refinement calculus for logic programs in the Isabelle/HOL logic, including support for parameterised procedures and recursion. The embedding includes support for refinement in context, based on program window inference [12]. In comparison to other embeddings of (imperative) refinement calculi [6,18,4,21], the embedding presented here makes use of Isabelle/HOL's datatypes, primitive recursive definitions and type definitions to provide a deep embedding.

In developing `Marvin`, we formulated a number of criteria that are fundamental in choosing a logic for embedding refinement calculi (or other similar semantic models [10]). These criteria were gathered by tracing the problems encountered during the embedding back to deficiencies in the logics. Against these criteria, Isabelle/HOL scored well. In particular, we found:

- Isabelle/HOL notation was rich enough to represent our Z specified semantics with only minimal adaptations required.
- The type system significantly reduced the amount of proof in comparison to the untyped logics (cf. Table 1).

– Automated proof support performed better than for the other provers.
– Window-inference-like tactics could be easily written using Isabelle's tactic language.

One remaining concern is with representing program state. Our solution only allows state variables to range over a fixed product of types.

Acknowledgements

We would like to thank Robert Colvin for his comments on earlier drafts of this paper. The work reported in this paper is supported by Australian Research Council grant number A49937007 : *Refinement Calculus for Logic Programming.*

References

1. Isabelle home page. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html`.
2. Ralph Back, Jim Grundy, and Joakim von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9(5–6):469–483, 1997.
3. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A review of existing refinement tools. Technical report 94-08, Software Verification Research Centre, The University of Queensland, Brisbane 4072. Australia, June 1994.
4. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10:97–124, 1998.
5. R. Colvin, I. Hayes, R. Nickson, and P. Strooper. A tool for logic program refinement. In D. J. Duke and A. S. Evans, editors, *Second BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer Verlag, 1997.
6. J. Grundy. A window inference tool for refinement. In C.B. Jones, R.C. Shaw, and T. Denvir, editors, *Fifth Refinement Workshop*, Workshops in Computing, pages 230–254. BCS FACS, Springer-Verlag, 1992.
7. I. Hayes, R. Nickson, and P. Strooper. Refining specifications to logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 1996*, volume 1207 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag, 1997.
8. I. Hayes, R. Nickson, P. Strooper, and R. Colvin. A declarative semantics for logic program refinement. Technical Report 00-30, Software Verification Research Centre, The University of Queensland, October 2000.
9. D. Hemer. Building tool support for a refinement calculus for logic programming: A comparison of interactive theorem provers. Technical Report 00-06, Software Verification Research Centre, The University of Queensland, March 2000.
10. P. Homeier and D. Martin. Mechanical verification of mutually recursive procedures. In M.A. McRobbie and J.K. Slaney, editors, *Proceedings of the 13th International Conference on Artificial Deduction (CADE-13)*, number 1104 in Lecture Notes in Artificial Intelligence, pages 201–215. Springer-Verlag, 1996.
11. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, LNCS 1125, pages 283–298. Springer Verlag, 1996.

12. R. Nickson and I. Hayes. Supporting contexts in program refinement. *Science of Computer Programming*, 29:279–302, 1997.
13. L.C. Paulson. *Isabelle - A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
14. P.J. Robinson and J. Staples. Formalising the hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, February 1993.
15. J.M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall, New York, 1989.
16. M. Staples. Window inference in Isabelle. In L. Paulson, editor, *Proceedings of the First Isabelle User's Workshop*, volume 379 of *University of Cambridge Computer Laboratory Technical Report*, pages 191–205, September 1995.
17. M. Staples. *A Mechanised Theory of Refinement*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.
18. M. Staples. Representing WP Semantics in Isabelle/ZF. In *TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 239–254, September 1999.
19. M. Utting and K. Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, The University of Queensland, Brisbane, QLD 4072, Australia, March 1994. Describes Version 4.0 of Ergo.
20. Mark Utting. The Ergo 5 generic proof engine. Technical Report 97-44, Software Verification Research Centre, The University of Queensland, 1997.
21. J. von Wright. Program refinement by theorem proving. In D. Till, editor, *Sixth Refinement Workshop*, Workshops in Computing, pages 121–150. BCS FACS, Springer-Verlag, 1994.

A Refinement Laws

This section contains a listing of rules used in the example that have not been described earlier.

The law `introduce_recursion` lets the user introduce recursion by replacing a parameterised command with a recursion block, and states conditions under which a recursive call can be made.

$$\begin{array}{l}\texttt{env}\ \rho\ \texttt{ass}\ A \Vdash pc \sqsupseteq \\ \quad \textbf{re}\ pid\ .\ (V)\ \texttt{:-}\ \{\forall y \bullet \texttt{env}\ \rho\ \texttt{ass}\ y \prec V \Vdash pc(y) \sqsubseteq pid(y)\}\ ,\ pc(V) \\ \textbf{er}\end{array}$$

To ensure termination, we must show that the recursive argument is strictly decreasing with respect to a well-founded relation $\prec$.

expand-scope-exists	$\dfrac{X \ \mathsf{nfi}\ c_2}{(\exists X \bullet c_1) \wedge c_2 \sqsupseteq (\exists X \bullet c_1 \wedge c_2)}$
remove-assumpt	$\{A\}\ ,\ c \sqsubseteq c$
por-false	$\langle false \rangle \vee c \sqsupseteq c$
sand-false	$c\ ,\ \langle false \rangle \sqsubseteq \langle false \rangle$
pand-to-sand	$c_1 \wedge c_2 \sqsubseteq c_1\ ,\ c_2$
right-distribute-pand-over-por	$(c1 \vee c2) \wedge c3 \sqsupseteq (c1 \wedge c3) \vee (c2 \wedge c3)$
lift-por	$\langle c1 \vee c2 \rangle \sqsupseteq \langle c1 \rangle \vee \langle c2 \rangle$
intro-spec	$\dfrac{A \Rrightarrow P}{\texttt{env}\ \rho\ \texttt{ass}\ A \Vdash c \sqsupseteq \langle P \rangle \wedge c}$