Modelling a refinement calculus in Isabelle/HOL

David Hemer

February 10, 2010
Outline

- Background
- Motivating example
- A formal model
- Semantics
- Refinement
- Conclusions
Introduction
Software verification - are we building the software right?

- does the software satisfy its specification?

Formal verification - proving that the software satisfies its specification.

- rigorous proofs done by hand (pencil and paper proofs)
- fully formal proofs machine checked (using theorem prover)
Proving correctness - first approach

- start with code that has already been written in languages such as Ada
- requires full formal semantics of the programming language
- specification properties often added as annotations to the code
- verification conditions generated to prove that the program satisfies its specification
- verification conditions could be checked by hand or checked in theorem prover
Proving correctness - second approach

- start with a high level specification of the problem
- transform or \textit{refine} the specification in a step-wise manner
- at each step use a correctness preserving refinement rule
- refinement rules have been pre-proven within theorem prover
- continue until the refined program can be translated directly into executable code
Motivating example
GNU Prolog 1.3.0
By Daniel Diaz
Copyright (C) 1999–2007 Daniel Diaz

?- member(2, [1,2]).
true ? ;

no
| ?- member(X, [1,2]).
X = 1 ? ;
X = 2 ? ;
(1 ms) no
We can specify the member predicate at a high level

\[(X, L) : - \text{isList}(L), \langle X \in \text{elems}(L) \rangle\]

- *assertion* stating that \(L\) is a list
- *specification* stating that \(X\) is in the set of elements in the list \(L\)
- cannot be translated directly to a logical programming language
- need to *refine* before code can be generated
The *member* predicate can be refined to

\[
\text{re } \text{Member } \bullet (X,L) :- \\
(\exists \ H, \ T \bullet \langle L = \text{cons}(H, T) \rangle, \ (\langle X = H \rangle \lor \\
\text{Member}(X,T)))
\]

- now fairly straightforward to translate to executable code
- refinement occurs via a series of correctness preserving steps
- at each stage refinement laws are used to transform the program
A formal model
Refinement calculus language and semantics represented in Isabelle/HOL
- Enables us to prove properties about the language
- Refinement laws can be proved using these properties
- Example refinements can be carried out in the theorem prover with any required assumptions formally proven
- We start right from the basics of terms and predicates and build up – a deep embedding
A *binding* maps the variables in a program to values

\[ Bnds == Var \rightarrow Val \]

A *state* consists of a set of bindings

\[ State == \mathbb{P} Bnds \]

(Logic programs are non-deterministic)

The semantics of our language will be modelled as a relationship between initial and final states of a program.
Predicates are used in specifications and assertions. Modelled as a data type in Isabelle

\[
\text{datatype } \text{pred} = \text{pred} \land \text{pred} \\
\quad | \quad \text{pred} \lor \text{pred} \\
\quad | \quad \text{pred} \Rightarrow \text{pred} \\
\quad | \quad \text{pred} \Leftrightarrow \text{pred} \\
\quad | \quad \text{rterm} = \text{rterm} \\
\quad | \quad \text{rterm} < \text{rterm} \\
\quad | \quad \neg \text{pred} \\
\quad | \quad \exists \ \text{Var} \ \text{pred} \\
\quad | \quad \forall \ \text{Var} \ \text{pred} \\
\quad | \quad \text{ptrue} \\
\quad | \quad \text{pfalse} \\
\quad | \quad \text{isNat Var}
\]

A predicate is evaluated by providing a binding

\[
\text{evalp :: pred} \times \text{Bnds} \rightarrow \text{bool}
\]
Logic language contains both non-executable and executable constructs

Modelled using a data type in Isabelle/HOL

```plaintext
datatype Cmd = ⟨pred⟩
             | { pred }
             | Cmd ∧ Cmd
             | Cmd ∨ Cmd
             | Cmd , Cmd
             | ∃ Var • Cmd
             | ∀ Var • Cmd
             | PIdent(rterm)
```
Semantics
The semantics of the language will be defined in terms of a relationship between initial and final state

\[
\text{StateMap} = \text{State} \rightarrow \text{State}
\]

These relationships are expected to obey three properties:

\[
\text{Exec} == \{ e : \text{StateMap} \mid \text{dom}(e) = \mathbb{P} \{ b. \{b\} : \text{dom}(e) \} \\
\quad \wedge \ \forall s \in \text{dom}(e). \text{the } e(s) \subseteq s \\
\quad \wedge \ \forall s \in \text{dom}(e). \ e(s) = \text{Some} \{ b. b:s \ & \ (e(\{b\}) = \text{Some} \{ b \}) \} \} \\
(\text{Exec\_witness})
\]
Semantics now defined using `exec` function

- Defined using primitive recursion for each of the commands

```plaintext
consts
  exec :: "[Env,Cmd] => StateMap"

primrec
  "exec p (spec x) = (%s . Some (s Int bnds(x)))"
  "exec p (c1 pand c2) = ((exec p c1) PInt (exec p c2))"
  "exec p (c1 por c2) = ((exec p c1) PUn (exec p c2))"
  "exec p (c1 sand c2) = ((exec p c1) o; (exec p c2))"
  ...
```
Parallel conjunction defined in terms of pointwise intersection

Defined as:

```plaintext
consts
  PInt :: "('a set => 'a set) => ('a set => 'a set) => ('a set => 'a set)"
    (infixl 120)
defs
  PInt_def: "f PInt g == %x. if (x: dom(f) Int dom(g)) then (Some ((the (f x)) Int (the (g x)))) else None"
```
Having defined the pointwise operators, a number of properties are proven stated as lemmas, e.g.

```
lemma PInt_empty [simp]: "(m PInt empty) = empty";
```

and

```
lemma PInt_commute : "(f PInt g) = (g PInt f)";
```

These properties will later be used in proving algebraic refinement laws
Refinement
Refinement relations

- Refinement is a correctness preserving transformation
- Defined firstly for state mappings (and executions)
  \[ e_1 \sqsubseteq e_2 = (\text{dom}(e_1) \subseteq \text{dom}(e_2) \land (\forall s:\text{dom}(e_1). \; e_1(s) = e_2(s))) \]
  \[ e_1 \sqsubset e_2 = e_1 = e_2 \]
- Then define refinement for commands
  \[ \text{refeq}_\text{cmd}_\text{def} \ "\text{env} \; \rho \; \vdash \; c_1 \sqsubseteq c_2 \; \equiv \; (\text{exec} \; \rho \; c_1) \sqsubseteq (\text{exec} \; \rho \; c_2)" \]
  \[ \text{refsto}_\text{cmd}_\text{def} \ "\text{env} \; \rho \; \vdash \; c_1 \sqsubseteq c_2 \; \equiv \; (\text{exec} \; \rho \; c_1) \sqsubseteq (\text{exec} \; \rho \; c_2)" \]
the wide-spectrum language is monotonic with respect to refinement

- a number of rules are proven showing that a command can be refined by refining one or more subcommands
- for example sequential conjunction is monotonic on the right hand side

\[
\text{env } \rho \models c_1 \sqsubseteq c_2 \\
\text{env } \rho \models c_3 \text{ sand } c_1 \sqsubseteq c_3 \text{ sand } c_2
\]
predicate connectives contained in specifications can be lifted to their program construct counterparts

for example an existential quantifier can be lifted using the following law

\[ \text{env } \rho \models \text{spec}(\exists V \cdot P) \sqsubseteq \exists V \cdot \text{spec}(P) \]
parallel conjunction and disjunction are commutative and associative
- sequential conjunction is associative (but not commutative)
- parallel conjunction distributive over parallel disjunction
- etc.
- rules are proven using earlier properties and definitions
- (some properties have not yet been proven - stated as axioms)
- having proven a number of refinement laws, we can now work through the refinement of the *member* example
- refinement demo
Wrapping up
Interactive theorem provers can be used for formal modelling
Model the abstract syntax and semantics of the language
Properties and rules can be derived and proven from the model
Minimise definitions and axioms - conservative theory extensions
A high level problem description can be transformed to a lower level implementation using the correctness preserving rules
Modelling using a theorem prover gives greatest guarantee that properties are correct

- no hand wavy proofs - theorem prover forces you to spell out detail
- but must keep definitions and axioms to a minimum

Abstract models relatively easy to model in theorem prover
- Already contain theories for rings/semi rings etc, so may get some properties for “free”
- May be possible to derive concrete models from abstract ones using notion similar to refinement
- Alternatively use theorem proving to apply static analysis on existing models