Imperative HOL – a leightweight framework for imperative data structures in Isabelle/HOL

August 27, 2014

*Imperative HOL* is a leightweight framework for reasoning about imperative data structures in *Isabelle/HOL* [2]. Its basic ideas are described in [1]. However their concrete realisation has changed since, due to both extensions and refinements. Therefore this overview wants to present the framework “as it is” by now. It focusses on the user-view, less on matters of construction. For details study of the theory sources is encouraged.

## 1 A polymorphic heap inside a monad

Heaps (*heap*) can be populated by values of class *heap*; HOL’s default types are already instantiated to class *heap*. Class *heap* is a subclass of *countable*; see theory *Countable* for ways to instantiate types as *countable*.

The heap is wrapped up in a monad `'a Heap` by means of the following specification:

```
datatype `'a Heap = Heap.Heap (heap ⇒ (`'a × heap) option)
```

Unwrapping of this monad type happens through

```
execute :: `'a Heap ⇒ heap ⇒ (`'a × heap) option
execute (Heap.Heap f) = f
```

This allows for equational reasoning about monadic expressions; the fact collection *execute-simps* contains appropriate rewrites for all fundamental operations.

Primitive fine-granular control over heaps is available through rule *Heap-cases*:

```
(∀x h'. execute f h = Some (x, h') ⇒ P) ⇒⇒ (execute f h = None ⇒⇒ P) ⇒⇒ P
```

Monad expression involve the usual combinators:
return :: 'a ⇒ 'a Heap

op ≫= :: 'a Heap ⇒ ('a ⇒ 'b Heap) ⇒ 'b Heap

raise :: char list ⇒ 'a Heap

This is also associated with nice monad do-syntax. The string argument to raise is just a codified comment.

Among a couple of generic combinators the following is helpful for establishing invariants:

assert :: ('a ⇒ bool) ⇒ 'a ⇒ 'a Heap
assert P x = (if P x then return x else raise "assert")

2 Relational reasoning about Heap expressions

To establish correctness of imperative programs, predicate

effect :: 'a Heap ⇒ heap ⇒ heap ⇒ 'a ⇒ bool

provides a simple relational calculus. Primitive rules are effectI and effectE, rules appropriate for reasoning about imperative operations are available in the effect-intros and effect-elims fact collections.

Often non-failure of imperative computations does not depend on the heap at all; reasoning then can be easier using predicate

success :: 'a Heap ⇒ heap ⇒ bool

Introduction rules for success are available in the success-intro fact collection.

3 Monadic data structures

The operations for monadic data structures (arrays and references) come in two flavours:

• Operations on the bare heap; their number is kept minimal to facilitate proving.

• Operations on the heap wrapped up in a monad; these are designed for executing.
Provided proof rules are such that they reduce monad operations to operations on bare heaps.
Note that HOL equality coincides with reference equality and may be used as primitive executable operation.

3.1 Arrays

Heap operations:

- `Array.alloc :: 'a list ⇒ heap ⇒ 'a array × heap`
- `Array.present :: heap ⇒ 'a array ⇒ bool`
- `Array.get :: heap ⇒ 'a array ⇒ 'a list`
- `Array.set :: 'a array ⇒ 'a list ⇒ heap ⇒ heap`
- `Array.length :: heap ⇒ 'a array ⇒ nat`
- `Array.update :: 'a array ⇒ nat ⇒ 'a ⇒ heap ⇒ heap`
- `op =!= :: 'a array ⇒ 'b array ⇒ bool`

Monad operations:

- `Array.new :: nat ⇒ 'a ⇒ 'a array Heap`
- `Array.of-list :: 'a list ⇒ 'a array Heap`
- `Array.make :: nat ⇒ (nat ⇒ 'a) ⇒ 'a array Heap`
- `Array.len :: 'a array ⇒ nat Heap`
- `Array.nth :: 'a array ⇒ nat ⇒ 'a Heap`
- `Arrayupd :: nat ⇒ 'a ⇒ 'a array ⇒ 'a array Heap`
- `Array.map-entry :: nat ⇒ ('a ⇒ 'a) ⇒ 'a array ⇒ 'a array Heap`
- `Array.swap :: nat ⇒ 'a ⇒ 'a array ⇒ 'a Heap`
- `Array.freeze :: 'a array ⇒ 'a list Heap`

3.2 References

Heap operations:

- `Ref.alloc :: 'a ⇒ heap ⇒ 'a ref × heap`
- `Ref.present :: heap ⇒ 'a ref ⇒ bool`
- `Ref.get :: heap ⇒ 'a ref ⇒ 'a`
- `Ref.set :: 'a ref ⇒ 'a ⇒ heap ⇒ heap`
- `op =!= :: 'a ref ⇒ 'b ref ⇒ bool`

Monad operations:

- `ref :: 'a ⇒ 'a ref Heap`
- `Ref.lookup :: 'a ref ⇒ 'a Heap`
- `Ref.update :: 'a ref ⇒ 'a ⇒ unit Heap`
- `Ref.change :: ('a ⇒ 'a) ⇒ 'a ref ⇒ 'a Heap`
4 Code generation

Imperative HOL sets up the code generator in a way that imperative operations are mapped to suitable counterparts in the target language. For Haskell, a suitable $\text{ST}$ monad is used; for SML, Ocaml and Scala unit values ensure that the evaluation order is the same as you would expect from the original monadic expressions. These units may look cumbersome; the target language variants $\text{SML-imp}$, $\text{Ocaml-imp}$ and $\text{Scala-imp}$ make some effort to optimize some of them away.

5 Some hints for using the framework

Of course a framework itself does not by itself indicate how to make best use of it. Here some hints drawn from prior experiences with Imperative HOL:

- Proofs on bare heaps should be strictly separated from those for monadic expressions. The first capture the essence, while the latter just describe a certain wrapping-up.

- A good methodology is to gradually improve an imperative program from a functional one. In the extreme case this means that an original functional program is decomposed into suitable operations with exactly one corresponding imperative operation. Having shown suitable correspondence lemmas between those, the correctness prove of the whole imperative program simply consists of composing those.

- Whether one should prefer equational reasoning (fact collection execute-simps or relational reasoning (fact collections effect-intros and effect-elims) depends on the problems to solve. For complex expressions or expressions involving binders, the relation style usually is superior but requires more proof text.

- Note that you can extend the fact collections of Imperative HOL yourself whenever appropriate.

References