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

April 17, 2016

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:

$$\textbf{datatype}\ 'a\ Heap = Heap.Heap\ (heap \Rightarrow ('a \times heap)\ option)$$

Unwrapping of this monad type happens through

$$execute :: 'a\ Heap \Rightarrow heap \Rightarrow ('a \times heap)\ option$$
$$execute\ (Heap.Heap\ f) = f$$

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*:

$$(\bigwedge x\ h'.\ execute\ f\ h = Some\ (x,\ h') \Longrightarrow P) \Longrightarrow (execute\ f\ h = None \Longrightarrow P) \Longrightarrow P$$

Monadic expression involve the usual combinators:

$return :: 'a \Rightarrow 'a\ Heap$
$op \gg= :: 'a\ Heap \Rightarrow ('a \Rightarrow 'b\ Heap) \Rightarrow 'b\ Heap$
$raise :: char\ list \Rightarrow 'a\ Heap$

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:

$assert :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ Heap$
$assert\ P\ x = (\mathsf{if}\ P\ x\ \mathsf{then}\ return\ x\ \mathsf{else}\ raise\ ''assert'')$

2 Relational reasoning about *Heap* expressions

To establish correctness of imperative programs, predicate

$effect :: 'a\ Heap \Rightarrow heap \Rightarrow heap \Rightarrow 'a \Rightarrow bool$

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

$success :: 'a\ Heap \Rightarrow heap \Rightarrow bool$

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

execute, *effect*, *success* and $op \gg=$ are related by rules *execute-bind-success*, *success-bind-executeI*, *success-bind-effectI*, *effect-bindI*, *effect-bindE* and *execute-bind-eq-SomeI*.

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:

$Array.alloc :: 'a\ list \Rightarrow heap \Rightarrow 'a\ array \times heap$
$Array.present :: heap \Rightarrow 'a\ array \Rightarrow bool$
$Array.get :: heap \Rightarrow 'a\ array \Rightarrow 'a\ list$
$Array.set :: 'a\ array \Rightarrow 'a\ list \Rightarrow heap \Rightarrow heap$
$Array.length :: heap \Rightarrow 'a\ array \Rightarrow nat$
$Array.update :: 'a\ array \Rightarrow nat \Rightarrow 'a \Rightarrow heap \Rightarrow heap$
$op =!!= :: 'a\ array \Rightarrow 'b\ array \Rightarrow bool$

Monad operations:

$Array.new :: nat \Rightarrow 'a \Rightarrow 'a\ array\ Heap$
$Array.of\text{-}list :: 'a\ list \Rightarrow 'a\ array\ Heap$
$Array.make :: nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow 'a\ array\ Heap$
$Array.len :: 'a\ array \Rightarrow nat\ Heap$
$Array.nth :: 'a\ array \Rightarrow nat \Rightarrow 'a\ Heap$
$Array.upd :: nat \Rightarrow 'a \Rightarrow 'a\ array \Rightarrow 'a\ array\ Heap$
$Array.map\text{-}entry :: nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a\ array \Rightarrow 'a\ array\ Heap$
$Array.swap :: nat \Rightarrow 'a \Rightarrow 'a\ array \Rightarrow 'a\ Heap$
$Array.freeze :: 'a\ array \Rightarrow 'a\ list\ Heap$

3.2 References

Heap operations:

$Ref.alloc :: 'a \Rightarrow heap \Rightarrow 'a\ ref \times heap$
$Ref.present :: heap \Rightarrow 'a\ ref \Rightarrow bool$
$Ref.get :: heap \Rightarrow 'a\ ref \Rightarrow 'a$
$Ref.set :: 'a\ ref \Rightarrow 'a \Rightarrow heap \Rightarrow heap$
$op =!= :: 'a\ ref \Rightarrow 'b\ ref \Rightarrow bool$

Monad operations:

$ref :: 'a \Rightarrow 'a\ ref\ Heap$
$Ref.lookup :: 'a\ ref \Rightarrow 'a\ Heap$
$Ref.update :: 'a\ ref \Rightarrow 'a \Rightarrow unit\ Heap$
$Ref.change :: ('a \Rightarrow 'a) \Rightarrow 'a\ ref \Rightarrow 'a\ Heap$

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 *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 *SML-imp*, *Ocaml-imp* and *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

[1] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs '08: Proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 2008.

[2] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.