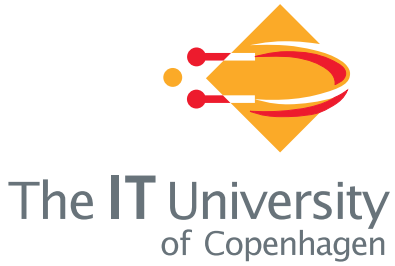

Verifying design patterns in Hoare Type Theory

Kasper Svendsen, Alexandre Buisse and Lars Birkedal

IT University Technical Report Series **TR-2008-112**

ISSN 1600–6100 **October 2008**

ISSN 1600–6100

ISBN 97887794918861

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: **+45 72 18 50 00**
Telefax: **+45 72 18 50 01**
Web **`www.itu.dk`**

Verifying design patterns in Hoare Type Theory

Kasper Svendsen, Alexandre Buisse and Lars Birkedal

Abstract

In this technical report we document our experiments formally verifying three design patterns in Hoare Type Theory.

1 Introduction

In [1, 2] Krishnaswami et al. defined a higher-order imperative language, Idealized ML, with an accompanying higher-order separation logic and used it to specify and informally verify three design patterns. In this technical report we describe our experiments translating these specifications into the language of Hoare Type Theory (HTT) and formally verifying the implementations in Ynot [3], the Coq implementation of Hoare Type Theory.

Idealized ML is a program logic with a separate type system and specification language, whereas specifications are integrated with types in HTT. Formalizing the examples thus requires a small translation. In the case of the Subject/observer pattern, the integrated types and specifications allows us to give a simple specification, however, the current type system appears to be too weak to allow us to give an implementation of this specification.

The Coq proof scripts can be downloaded at `http://www.itu.dk/people/kasv/patterns.tgz`.

2 Subject/observer pattern

The Subject/observer pattern is a well-known design pattern for object-oriented languages. An instance of this pattern consists of an object called the subject, and a number of objects called observers, which depend on the internal state of the subject. The subject provides a method for dynamically registering observers, by providing a callback method, to be called by the subject, whenever the subject's internal state changes.

The following is a specification of the subject-observer pattern in Idealized ML, for the special case where the

internal state of the subject is a natural number [2]:

$$
\begin{array}{l}
\exists sub : \tau_s \times \mathbb{N} \times \mathbf{seq}\ ((\mathbb{N} \Rightarrow prop) \times (\mathbb{N} \to \bigcirc 1)) \Rightarrow prop. \\
\exists newsub : \mathbb{N} \to \bigcirc \tau_s. \\
\exists register : \tau_s \times (\mathbb{N} \to \bigcirc 1) \to \bigcirc 1. \\
\exists broadcast : \tau_s \times \mathbb{N} \to \bigcirc 1. \\
\forall n.\{emp\}\ newsub(n)\ \{a : \tau_s.sub(a, n, [])\} \\
and \\
\forall s, m, n.\{sub(s, m, [])\}\ broadcast(s, n)\ \{sub(s, n, [])\} \\
and \\
\forall O : \mathbb{N} \Rightarrow prop.\ \forall f : \mathbb{N} \to \bigcirc 1. \\
\quad (\forall m, n.\{O(m)\}\ f(n)\ \{a : 1.O(n)\} \\
\quad implies \\
\quad\quad \forall s, n, os.\{sub(s, n, os)\}\ register(s, f)\ \{a : 1.\ sub(s, n, (O, f) :: os)\} \\
\quad\quad and \\
\quad\quad\quad (\forall s, m, n, os.\{sub(s, m, os) * obs(os)\}\ broadcast(s, n)\ \{sub(s, n, os) * obs\_at(os, n)\} \\
\quad\quad implies \\
\quad\quad\quad \{sub(s, m, (O, f) :: os) * obs((O, f) :: os)\} \\
\quad\quad\quad\quad broadcast(s, n) \\
\quad\quad\quad \{sub(s, n, (O, f) :: os) * obs\_at((O, f) :: os, n)\}))
\end{array}
$$

where

$$
\begin{aligned}
\tau_s &\equiv \mathbf{ref}\ \mathbb{N} \times \mathbf{ref}\ list(\mathbb{N} \to \bigcirc 1) \\
obs([]) &\equiv emp \\
obs((O, f) :: os) &\equiv (\exists i.O(i)) * obs(os) \\
obs\_at([], k) &\equiv emp \\
obs\_at((O, f) :: os, k) &\equiv O(k) * obs\_at(os, k)
\end{aligned}
$$

The specification asserts the existence of a representation predicate $sub(s, n, os)$, expressing that s is a subject with internal state n. The list os contains the notification computations currently registered with the subject, along with a predicate expressing how the state of the given observer depends on the state of the subject. The specification further asserts the existence of three computations: $newsub$, for creating a new subject, $register$, for registering a callback computation with the subject, and $broadcast$, for updating the internal state of the subject.

Intuitively, if $os = [(O_1, f_1), ..., (O_n, f_n)]$ is the list of currently registered observers of the subject s, such that each notification computation f_i updates observer i's state in accordance with O_i, when called with the subject's new state, then we should be able to conclude that after running $broadcast(s, m)$ the heap should satisfy $sub(s, m, os) * O_1(m) * ... * O_n(m)$. This intuitive specification translates very naturally into the above specification, using implication between specifications to express the condition that each notification computation f_i should respect the accompanying predicate O_i in os.

2.1 HTT translation

In this section we discuss two possible translations of the Idealized ML specification into HTT. With the exception of implication between specifications most of the above specification translates directly into HTT. The first translation is very direct: implication between specifications is translated into arrow types in HTT. However, the resulting specification does not capture the Subject/observer pattern. The second translation is a more advanced translation, making use of HTT's more expressive types to quantify over lists of "proper" notification computations, to obtain a simpler specification.

2.1.1 Logical variables

Idealized ML and HTT handle logical variables differently. In Idealized ML the post-condition is expressed only in terms of the heap at termination and logical variables are expressed by universally quantifying over variables whose scope extends to both the pre- and post-condition. In HTT, the post-condition is expressed in terms of both the initial and terminal heap, which allows us to express logical variables by existentially quantifying them in the pre-condition and universally quantifying them in the post-condition.

We can thus translate an Idealized ML specification,

$$\forall x : \tau.\ \{P(x)\}\ comp\ \{a : 1.\ Q(x)\}$$

into the following HTT type:

$$comp : \{\lambda i : heap.\ \exists x : \tau.\ P\ x\ i\}\ a : 1\ \{\lambda i : heap.\ \lambda j : heap.\ \forall x : \tau.\ P\ x\ i \rightarrow Q\ x\ j\}$$

where i is the initial heap and j is the terminal heap. We will usually abbreviate this type as follows:

$$comp : \{i.\ \exists x.\ P\ x\ i\}\ a : 1\ \{i\ j.\ \forall x.\ P\ x\ i \rightarrow Q\ x\ j\}$$

2.1.2 Specification implication

In the Idealized ML specification, implication between specifications is used to "inductively" build up the specification of $broadcast$, to restrict the quantification of os to lists of pairs, (O, f), such that the notification computation, f, respects the predicate, O.

In HTT we can express implication between specifications using arrow types. Turning implications into arrow types, $register$ becomes a function that takes as argument a $broadcast$ function for broadcasting to a list of observers, os, a predicate, O, and a notification computation, f, respecting O, and returns a computation for broadcasting to $(O, f) :: os$:

$$\begin{array}{l}
\Pi l : loc.\ \Pi os : list\ T.\ \Pi O : \mathbb{N} \rightarrow Prop. \\
\Pi f : (\Pi m : \mathbb{N}.\{i.\ \exists k : \mathbb{N}.\ O\ k\ i\}\ a : 1\ \{i\ j.\ \forall k : \mathbb{N}.\ O\ k\ i \rightarrow O\ m\ j\}). \\
\Pi broad : (\Pi n : \mathbb{N}.\ \{i.\ \exists k.\ (sub\ (l, k, os) * obs\ os)\ i\} \\
\qquad\qquad r : 1 \\
\qquad\qquad \{i\ j.\ \forall k.\ (sub\ (l, k, os) * obs\ os)\ i \rightarrow (sub\ (l, n, os) * obs_at\ (os, n))\ j\}). \\
\quad \Pi n : \mathbb{N}.\ \{i.\ \exists k.\ (sub\ (l, k, (O, f) :: os) * obs\ ((O, f) :: os))\ i\} \\
\quad\ r : 1 \\
\quad \{i\ j.\ \forall k.\ (sub\ (l, k, (O, f) :: os) * obs\ ((O, f) :: os))\ i\ \rightarrow \\
\qquad\qquad (sub\ (l, n, (O, f) :: os) * obs_at\ ((O, f) :: os, n))\ j\}
\end{array}$$

Since O is free in the type of f we take T to be the following dependent sum type:

$$\begin{array}{l}
T \equiv \Sigma O : \mathbb{N} \rightarrow Prop. \\
\qquad (\Pi m : \mathbb{N}.\ \{i.\ \exists k : \mathbb{N}.\ O\ k\ i\}\ a : 1\ \{i\ j.\ \forall k : \mathbb{N}.\ O\ k\ i \rightarrow O\ m\ j\})
\end{array}$$

In the implementation of the Idealized ML specification, a list is maintained containing the currently registered notification computations; $register$ adds the notification computations it is called with to this list, and $broadcast$ iterates through the list, calling each notification computation.

To give an implementation of the above HTT type, we do not need to maintain a list of registered notification computations, as $register$ can simply return a computation that runs the given $broadcast$ computation followed by the new notification computation. Thus, the sub predicate does not need to take the list of registered notification computations as an argument and we obtain the following HTT type for the entire Idealized ML specification:

$$\begin{array}{lll}
S_1 \equiv \Sigma \alpha : Type.\ \Sigma sub : \alpha \times \mathbb{N} \rightarrow Prop. & & \\
\qquad \Pi n : \mathbb{N}.\{i.\ emp\ i\}\ a : \alpha\ \{i\ j.\ emp\ i \rightarrow sub\ (a, n)\ j\}\ \times & \texttt{/* newsub */} & \\
\qquad \Pi l : loc.\ bspec(l, [])\ \times & \texttt{/* broadcast */} & \\
\qquad \Pi l : loc.\ \Pi os : list\ T.\ \ Pit : T.\ \Pi broad : bspec(l, os).\ bspec(l, t :: os) & \texttt{/* register */} &
\end{array}$$

where

$$
\begin{array}{l}
bspec(l : loc, os : list\ T) \equiv \\
\quad \Pi n : \mathbb{N}.\ \{i.\ \exists k.\ (sub\ (l,k) * obs\ os)\ i\} \\
\qquad r : 1 \\
\quad \{i\ j.\ \forall k.\ (sub\ (l,k) * obs\ os)\ i \rightarrow (sub\ (l,n) * obs\_at\ (os,n))\ j\}
\end{array}
$$

This is clearly not a specification of the Subject/observer pattern, as the currently registered notification computations are no longer part of the subject's state, but have to be passed around manually between observers.

2.1.3 Quantifying over notification computations

In [1], Krishnaswami et al. give an alternative specification for the Subject/observer pattern, where the restriction to proper notification computations is moved into the sub predicate. This results in a simpler specification for $broadcast$, as the specification no longer has to be built up "inductively", using implication between specifications:

$$
\begin{array}{l}
\forall s, i, os, k. \\
\quad \{sub(s,i,os) * obs(os)\}\ broadcast(s,k)\ \{a : 1.\ sub(s,k,os) * obs\_at(os,k)\}
\end{array}
$$

Since os is universally quantified, an implementation of the above specification could for instance take $sub(s,i,os) = notify(os) \wedge ...$, where

$$
\begin{array}{rl}
notify([]) & \equiv True \\
notify((O,f) :: l) & \equiv (\forall m, n.\ \{O(m)\}\ f(n)\ \{a : 1.\ O(n)\}\ \mathbf{valid}) \wedge notify(l)
\end{array}
$$

to restrict the quantification to "proper" notification computations.

The above $notify$ predicate does not translate directly into HTT, as HTT does not have an assertion of the form S **valid**, to express that the specification S holds. However, since the type T defined above is the type of pairs of predicate and notification computations, such that the notification computation respects the accompanying predicate, we can simply let os quantify over $list\ T$:

$$
\begin{array}{l}
\Pi a : \alpha.\ \Pi m : \mathbb{N}. \\
\quad \{i.\ \exists n : \mathbb{N}, os : list\ T.\ (sub\ (a,n,os) * obs\ os)\ i\} \\
\qquad r : 1 \\
\quad \{i\ j.\ \forall n : \mathbb{N}, os : list\ T.\ (sub\ (a,n,os) * obs\ os)\ i \rightarrow (sub\ (a,m,os) * obs\_at\ (os,m))\ j\}
\end{array}
$$

We can thus express the Subject/observer pattern, where the currently registered notification computations are a part of the subject's state with the following HTT type:

$$
\begin{array}{lll}
S_2 \equiv \Sigma \alpha : Type.\ \Sigma sub : \alpha \times \mathbb{N} \times list\ T \rightarrow Prop. & & \\
\quad \Pi n : \mathbb{N}.\ \{i.\ emp\ i\} a : \alpha \{i\ j.\ sub\ (a,n,[])\ j\} \times & & \texttt{/* newsub */} \\
\quad \Pi a : \alpha.\ \Pi t : T. & & \texttt{/* register */} \\
\qquad \{i.\ \exists n : \mathbb{N}, os : list\ T.sub\ (a,n,os)\ i\} & & \\
\qquad\quad r : 1 & & \\
\qquad \{i\ j.\ \forall n : \mathbb{N}, os : list\ T.\ sub\ (a,n,os)\ i \rightarrow sub\ (a,n,t :: os)\ j\} \times & & \\
\quad \Pi a : \alpha.\ \Pi m : \mathbb{N}. & & \texttt{/* broadcast */} \\
\qquad \{i.\ \exists n : \mathbb{N}, os : list\ T.\ (sub\ (a,n,os) * obs\ os)\ i\} & & \\
\qquad\quad r : 1 & & \\
\qquad \{i\ j.\ \forall n : \mathbb{N}, os : list\ T.\ (sub\ (a,n,os) * obs\ os)\ i & & \\
\qquad\qquad \rightarrow (sub\ (a,m,os) * obs\_at\ (os,m))\ j\} & &
\end{array}
$$

This specification still differs slightly from the Idealized ML version in its handling of the O predicate. In the Idealized ML version $register$ takes the notification computation as argument and the specification allows us to choose any O predicate that the notification computation respects, when deriving the specification for $broadcast$. In the HTT version, the O predicate has to be given as an argument along with the notification computation to $register$.

2.2 Ynot implementation

In the version of HTT presented in [4], dependent sums are predicative, i.e., for $\Sigma x : A.B$ to be a monotype, both A and B have to be a monotypes. Since the type of heaps, $heap$, is defined as a subset of the type $\mathbb{N} \times \Sigma\alpha : \mathbf{mono}.\alpha$ and $\mathbf{mono}$ is not a monotype, it follows that the T type defined above is not a monotype either and that values of type T cannot be stored in the heap.

In the implementation given in [1], a list of registered notification computations is stored in the heap. Since types and specifications are separate in Idealized ML, the type of these computations can be very weak, $\mathbb{N} \to \bigcirc 1$, because the specification language allows us to express that if these are proper notification computations then the broadcast computation will do a broadcast when performed. Since types and specifications are integrated in HTT, these notification computations have to be stored with a much stronger type, as the $broadcast$ computation must be able to infer from their type that they are proper notification computations when it retrieves them from the heap. Since values of type T cannot be stored in the heap, it is unclear whether this is possible in the predicative version of HTT.

Since Ynot is based on the predicative version of HTT [4], the same holds for Ynot: trying to store a value of type T in the heap causes a universe inconsistency error.

The impredicative version of HTT [5] has an impredicative sum type, $\Sigma^T x : A.B$, which is a monotype if B is. Hence, in the impredicative version of HTT, we can store values of type T, by using impredicative sums. We conjecture that the implementation in [1] has the type S_2 in impredicative HTT.

For the other HTT specification of the functional Subject/observer pattern, S_1, we do not have to store the notification computations in the heap: $register$ simply returns a new broadcast computation, which runs the old broadcast computation followed by the given notification computation. We have formally verified that this gives an implementation of the type S_1 in Ynot.

The Subject/observer pattern thus provides an example of a potential weakness in the predicative version of HTT and suggests that future work should include an implementation of impredicative HTT.

3 Flyweight

The Flyweight pattern is a design pattern used for reducing memory consumption, by caching objects. A Flyweight for a given class consists of an object table and a method, new, for constructing objects of the given class. The new method checks the object table to see if any objects of the given class has already been constructed with the given parameters, in which case it returns the object in the table, and otherwise creates a new object and inserts it into the table.

Krishnaswami et al. give the following specification for a Flyweight factory for constructing Flyweights for caching pairs of characters and fonts.

$$
\begin{array}{l}
\exists make\_flyweight : \\
\quad font \to \bigcirc((char \to \bigcirc\mathbf{ref}\ (char \times font))\ \times (\mathbf{ref}\ (char \times font) \to \bigcirc(char \times font))). \\
\qquad \forall f.\ \{emp\}\ make\_flyweight(f)\ \{a.\ \exists I : Prop.\ I \wedge Flyweight(I, fst\ a, snd\ a, f)\ \mathbf{valid}\}
\end{array}
$$

where

$$
\begin{array}{l}
Flyweight(I : Prop, newchar : char \to \bigcirc\mathbf{ref}\ (char \times font), \\
\qquad\qquad getdata : \mathbf{ref}\ (char \times font) \to \bigcirc(char \times font), f : font) \equiv \\
\quad \exists glyph : \mathbf{ref}\ char \times char \times font \to Prop. \\
\qquad \forall c, S.\ \{I \wedge chars(S)\}\ newchar(c)\ \{a : \mathbf{ref}\ (char \times font).\ I \wedge chars(\{(a, (c, f))\} \cup S)\} \\
\qquad and \\
\qquad \forall l, c, f, P.\ \{glyph(l, c, f) \wedge P\}\ getdata(l)\ \{a : char \times font.\ glyph(l, c, f) \wedge P \wedge a = (c, f)\} \\
\qquad and \\
\qquad \{\forall l, l', c, c'.\ I \wedge glyph(l, c, f) \wedge glyph(l', c', f') \to (l = l' \leftrightarrow (c = c' \wedge f = f'))\}
\end{array}
$$

and

$$chars(\emptyset) \equiv \top$$
$$chars(\{(l, (c, f))\} \cup S) \equiv glyph(l, c, f) \wedge chars(S)$$

Since Idealized ML is not an object oriented language and therefore cannot return a reference to an object, the new computation just returns a reference to the character and font pair. The Flyweight further defines a $getdata$ computation, which returns the actual character and font for a given reference, to simulate an object method.

3.1 HTT translation

Besides Hoare triples, Idealized ML's specification language contains specifications of the form $\{P\}$, for asserting that P is true. In the above specification this is used to express that calling $getdata$ with the same character multiple times, produces the same reference. In HTT we can express that an arbitrary proposition P is true by returning an element of the subset type, $\{x : 1 \mid P\}$, where x is not free in P.

The assertion language of Idealized ML also contains an expression for asserting that a given specification holds. In the above example this is used in the post-condition of $make\_flyweight$, to assert that the code returned implements a Flyweight. In HTT, we can express the same by simply giving a more precise type for the return value of the $make\_flyweight$ computation.

We have further generalized the specification, such that the computation can generate a Flyweight for simulated objects consisting of a value of an arbitrary monotype. The Flyweight factory computation therefore also has to take as an argument, a function, α_{eq}, for deciding propositional equality between α values.

The rest of the specification can be translated almost directly into HTT, however, we have made a few changes, to simplify the formal verification of the implementation in Ynot.

- In the specification of $newchar$, instead of using a set to associate arguments with objects, we have used a partial function (i.e., a total function from α to $option\ loc$).
- In the above specification the predicate I has to specify the representation of both the object table and the simulated objects. We have split I into two predicates, $table$ and $refs$, and changed the precondition of $newchar$ to the HTT equivalent of $table(...) * (refs(...) \wedge chars(S))$, to make it explicit that the object table and the simulated objects are in separate subheaps, to simplify verification.

The final HTT type of the Flyweight factory thus looks as follows:

$\Pi\alpha : \mathbf{mono}.\ \Pi\alpha_{eq} : (\Pi x : \alpha.\ \Pi y : \alpha.\ \{z : 1 \mid x = y\} + \{z : 1 \mid x \neq y\}).$
$\{i.\ emp\ i\}$
$r : \Sigma table : (\alpha \to option\ loc) \to heap \to Prop.$
$\Sigma refs : (\alpha \to option\ loc) \to heap \to Prop.$
$\Sigma objat : loc \to \alpha \to heap \to Prop.$
$\Sigma prf_1 : \{x : 1 \mid \forall h, l, l', a, a', f.\ objat\ l\ a\ h \wedge objat\ l'\ a'\ h \wedge refs\ f\ h \to (l = l' \leftrightarrow a = a')\}.$
$\Pi a : \alpha.$ /* new */
$\{i.\ \exists f.\ (table\ f * (\lambda h.\ allobjat(\alpha, objat, f, h) \wedge refs\ f\ h))\ i\}$
$l : loc$
$\{i\ j.\ \forall f.\ (table\ f * (\lambda h.\ allobjat(\alpha, objat, f, h) \wedge refs\ f\ h))\ i \to$
$((\forall l'.\ f\ a = Some\ l' \to l = l') \wedge$
$(table\ f[a \mapsto l] * (\lambda h.\ allobjat(\alpha, objat, f[a \mapsto l], h) \wedge refs\ f[a \mapsto l]\ h))\ j)\} \times$
$\Pi l : loc.$ /* get */
$\{i.\ \exists a : \alpha,\ objat\ l\ a\ i\}\ r : \alpha\ \{i\ j.\ \forall a : \alpha,\ objat\ l\ a\ i \to (i = j \wedge r = a)\}$
$\{i\ j.\ ((fst\ r)\ []\ * (\lambda h.\ allobjat(\alpha, fst\ (snd\ (snd\ r)), [], h) \wedge (fst\ (snd\ r))\ []\ h))\ j\}$

where

$$allobjat(\alpha, objat, f, h) \equiv \forall l : loc, o : \alpha.\ f\ o = Some\ l \to (objat\ l\ o * (\lambda h.\ True))\ h$$

and $[] \equiv (\lambda x.\ None)$.

3.2 Ynot implementation

In the case of the Flyweight pattern, we were able to formally verify that the implementation given in [1] has the above HTT type in Ynot.

The implementation given in [1] assumes the existence of an implementation of a table data-structure with the following Idealized ML specification:

$$
\begin{array}{l}
\exists table : A_t \times (B \rightarrow^{fin} C) \rightarrow Prop. \\
\exists newtable : 1 \rightarrow \bigcirc\, A_t. \\
\exists update : A_t \times B \times C \rightarrow \bigcirc\, 1. \\
\exists lookup : A_t \times B \rightarrow \bigcirc(option\ C). \\
\quad \{emp\}\ newtable()\ \{a : A_t : table(a, [])\} \\
\quad and \\
\quad \forall t, f, b, c.\ \{table(t, f)\}\ update(t, b, c)\ \{a : 1.\ table(t, f[b \mapsto c])\} \\
\quad and \\
\quad \forall t, b, f.\ \{table(t, f)\}\ lookup(t, b)\ \{a : option\ C.\ table(t, f) \wedge \\
\qquad ((b \in dom(f) \wedge a = Some\ f(b)) \vee (b \notin dom(f) \wedge a = None))\}
\end{array}
$$

For the formalization in Ynot we found it necessary to use a table implementation with a slightly stronger specification: In particular, $f' = f[b \mapsto c]$ should satisfy that,

$$
f'\ x = \begin{cases} c & \text{if } b = x \\ f\ x & \text{if } b \neq x \end{cases}
$$

where $b = x$ denotes propositional equality. To define such an f', we need a function for deciding propositional equality on B values. Similarly, to define the $update$ and $lookup$ computations, we need a computation or function for deciding propositional equality on B values. We further had to extend the specification with two preciseness properties, $prf1$ and $prf2$ in the HTT type below, to make the proofs go through. At the moment it is unclear whether these preciseness properties were truly needed to prove that the Flyweight implementation had the expected HTT type or whether they are a consequence of the current Ynot implementation. Section 5 contains a discussion of the preciseness issues we encountered with the Ynot formalization.

$$
\begin{array}{l}
\Pi\alpha : \mathbf{mono}.\ \Pi\alpha_{eq} : (\Pi x : \alpha.\ \Pi y : \alpha.\ \{z : 1 \mid x = y\} + \{z : 1 \mid x \neq y\}). \\
\Sigma table : loc \rightarrow (\alpha \rightarrow option\ loc) \rightarrow heap \rightarrow Prop. \\
\Sigma prf_1 : \{x : 1 \mid \forall l : loc,\ precise\ (\lambda h.\ \exists f : obj \rightarrow option\ loc,\ table\ l\ f\ h)\}. \\
\Sigma prf_2 : \{x : 1 \mid \forall l, f, f', h.\ table\ l\ f\ h \wedge table\ l\ f'\ h \rightarrow f = f'\}. \\
\quad \{i.\ emp\ i\}\ r : loc\ \{i\ j.\ table\ r\ (\lambda x.\ None)\ j\} \times \qquad \texttt{/* newtable */} \\
\quad \Pi t : loc.\ \Pi k : \alpha. \qquad \texttt{/* lookup */} \\
\qquad \{i.\ \exists f : (\alpha \rightarrow option\ loc).\ table\ t\ f\ i\} \\
\qquad\quad r : option\ loc \\
\qquad \{i\ j.\ \forall f : (\alpha \rightarrow option\ loc).\ table\ t\ f\ i \rightarrow (i = j \wedge r = f\ k\} \times \\
\quad \Pi t : loc.\ \Pi k : \alpha.\ \Pi v : loc. \qquad \texttt{/* update */} \\
\qquad \{i.\ \exists f : (\alpha \rightarrow option\ loc).\ table\ t\ f\ i\} \\
\qquad\quad r : 1 \\
\qquad \{i\ j.\ \forall f : (\alpha \rightarrow option\ loc).\ table\ t\ f\ i \rightarrow table\ t\ f[k \mapsto Some\ v]\ j\}
\end{array}
$$

where

$$
\begin{array}{c}
precise(P) \equiv \forall h, h_1, m_1, h_2, m_2 : heap.\ splits\ h\ h_1\ m_1 \rightarrow splits\ h\ h_2\ m_2 \rightarrow \\
P\ h_1 \rightarrow P\ h_2 \rightarrow (h_1 = h_2 \wedge m_1 = m_2)
\end{array}
$$

where $splits\ h\ h_1\ h_2$ iff h_1 and h_2 are disjoint and h is the union of h_1 and h_2.

4 Iterators

The Iterator pattern is a design pattern used to provide a uniform interface for enumerating elements in a collection. Instead of manipulating a collection directly, one uses an iterator, which provides a computation, $next$, which produces the next element in the collection. The instance of the Iterator pattern we consider further provides two ways of constructing new iterators from old ones: A filter iterator, $Filter\ p\ i$, which returns the elements of the underlying iterator i that satisfies the predicate p, and a map iterator, $Map2\ f\ i_1\ i_2$, which applies the function f to the elemenets of two underlying iterators i_1 and i_2.

If a destructive operation is performed on an underlying collection of an iterator, the iterator will be considered to be in an invalid state. We will thus forbid such operations, but still allow non-destructive ones, such as querying for the size of the collection.

Figure 1 contains the Idealized ML specification of the instance of the iterators pattern from [1].

4.1 HTT translation and Ynot implementation

The translation of the this specification into HTT is straightforward, with some very minor modifications, such as letting S be a list instead of a finite subset.

Similarly, the formal verification in Ynot of the implementation of the above specification given in [1] is straightforward, with the exception of the $Map2$ iterator. Since the $next$ computation has to perform two recursive calls to $next$ on two subheaps of the initial heap in the case of the $Map2$ iterator, we have to prove some preciseness properties about $iter$ predicate, to get the proof to go through. This is explained in greater detail in Section 5. Without the $Map2$ iterator, we have formally verified that the implementation given in [1] has the expected HTT type, however, so far we have not been able to prove the required preciseness properties to finish the formal proof with the $Map2$ iterator.

5 Logical Variables and preciseness

As previously mentioned, in carrying out the formalization of the Flyweight implementation and the Iterators implementation in Ynot, we encountered some issues with preciseness. More specifically, that we often had to prove preciseness for a predicate to complete the proof, where the same proof could have been completed on paper in the impredicative version of HTT (which also uses unary post-conditions), without proving preciseness.

To illustrate the problem, assume that we have an implementation, add, of the following HTT type:

$$\Pi l : loc.\ \{i.\ \exists n.\ inv\ l\ n\ i\}\ r : 1\ \{i\ j.\ \forall n.\ inv\ l\ n\ i \rightarrow inv\ l\ (n+1)\ j\}$$

where $inv(l, n, i)$ holds iff i is the singleton heap with n stored at location l. Then we should clearly we able to prove that the computation,

$$add_2 = \lambda l_1.\ \lambda l_2.\ add\ l_1; add\ l_2$$

has the type,

$$\begin{aligned} T = \Pi l_1 : loc.\ \Pi l_2 : loc.\ & \\ & \{i.\ \exists n_1, n_2.\ (inv\ l_1\ n_1 * inv\ l_2\ n_2)\ i\} \\ & \quad r : 1 \\ & \{i\ j.\ \forall n_1, n_2.\ (inv\ l_1\ n_1 * inv\ l_2\ n_2)\ i \rightarrow (inv\ l_1\ (n_1+1) * inv\ l_2\ (n_2+1))\ m\} \end{aligned}$$

without having to prove any preciseness properties about inv (or unfolding the definition of inv). In the current version of Ynot, if we use the $nextvc$ tactic, which is an important part of the automation provided by Ynot for simplifying proof obligations, then we cannot prove that add_2 has the desired type, without a preciseness property about inv, such as:

$$\forall l : loc.\ precise(\lambda h.\ \exists n : \mathbb{N}.\ inv\ l\ n\ h)$$

However, without the $nextvc$ tactic the proof goes through without having to prove preciseness and the same holds when doing a paper proof using the proof rules of the impredicative version of HTT.

To prove that add_2 has the type T, using $nextvc$, we have to prove the precondition of $add\ l_1$ and $add\ l_2$ holds, under the assumption that we have a heap i for which $(inv\ l_1\ n_1 * inv\ l_2\ n_2)$ holds. By the assumption

$A_c = \mathbf{ref}\ list\ \mathbb{N}.$
$A_i = Coll\ of\ \mathbf{ref}\ A_c \mid Filter\ of\ ((\mathbb{N} \to bool) \times A_i) \mid Map2\ of\ ((\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \times A_i \times A_i).$

$\exists\, coll : A_c \times \mathbf{seq}\ \mathbb{N} \times Prop \Rightarrow Prop.$
$\exists\, iter : A_i \times \mathcal{P}^{fin}(A_c \times \mathbf{seq}\ \mathbb{N} \times Prop) \times \mathbf{seq}\ \mathbb{N} \Rightarrow Prop.$
$\exists\, newcoll : \mathbf{1} \to \bigcirc A_c.$
$\exists\, size : A_c \to \bigcirc \mathbb{N}.$
$\exists\, add : A_c \times \mathbb{N} \to \bigcirc \mathbf{1}.$
$\exists\, newiter : A_c \to \bigcirc A_i.$
$\exists\, filter : (\mathbb{N} \to bool) \times A_i \to \bigcirc A_i.$
$\exists\, map2 : (\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \times A_i \times A_i \to \bigcirc A_i.$
$\exists\, next : A_i \to \bigcirc(option\ \mathbb{N}).$

$\{emp\}\, newcoll()\, \{a : A_c.\exists\, P.coll(a, P, \epsilon)\}.$
and
$\forall\, c : A_c,\ xs : \mathbf{seq}\ \mathbb{N},\ P : Prop,$
$\quad \{coll(c, xs, P)\}\, size(c)\, \{a : \mathbb{N}, coll(c, xs, P) \wedge a = |xs|\}.$
and
$\forall\, c : A_c,\ x : \mathbb{N},\ xs : \mathbf{seq}\ \mathbb{N},\ P : Prop,$
$\quad \{coll(c, xs, P)\}\, add(c, x)\, \{a : 1.\exists, Q.coll(c, x :: xs, Q)\}.$
and
$\forall\, c : A_c,\ xs : \mathbf{seq}\ \mathbb{N},\ P : Prop,$
$\quad \{coll(c, xs, P)\}\, newiter(c)\, \{a : A_i.coll(c, xs, P) * iter(a, \{(c, xs, P)\}, xs)\}.$
and
$\forall\, p : \mathbb{N} \to bool,\ i : A_i,\ S : \mathcal{P}^{fin}(A_c \times \mathbf{seq}\ \mathbb{N} \times Prop),\ xs : \mathbf{seq}\ \mathbb{N},$
$\quad \{iter(i, S, xs)\}\, filter(p, i)\, \{a : A_i.iter(a, S, filter\ p\ xs)\}.$
and
$\forall\, (f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}),\ (i\, i' : A_i),\ (S\, S' : \mathcal{P}^{fin}(A_c \times \mathbf{seq}\ \mathbb{N} \times Prop)),\ (xs\, xs' : \mathbf{seq}\ \mathbb{N}),$
$\quad \{iter(i, S, xs) * iter(i', S', xs') \wedge S \cap S' = \varnothing\}\, map2(f, i, i')$
$\quad \{a : A_i.iter(a, S \cup S', map\ f\ (zip\ xs\ xs'))\}.$
and
$\forall\, i : A_i,\ S : \mathcal{P}^{fin}(A_c \times \mathbf{seq}\ \mathbb{N} \times Prop),$
$\quad \{colls(S) * iter(i, S, \epsilon)\}\, next(i)\, \{a : option\ \mathbb{N}.colls(S) * iter(i, S, \epsilon) \wedge a = None\}.$
and
$\forall\, i : A_i,\ S : \mathcal{P}^{fin}(A_c \times \mathbf{seq}\ \mathbb{N} \times Prop),\ x : \mathbb{N},\ xs : \mathbf{seq}\ \mathbb{N},$
$\quad \{colls(S) * iter(i, S, x :: xs)\}\, next(i)\, \{a : option\ \mathbb{N}.colls(S) * iter(i, S, xs) \wedge a = Some\ x\}.$

Figure 1: The Idealized ML specification given in [1] of the Iterators pattern.

about i, we know that i can be split into two disjoint subheaps i_1 and i_2, for which $inv\ l_1\ n_1$ and $inv\ l_2\ n_2$ holds, respectively. i_1 can be used to prove the precondition of $add\ l_1$ and i_2 to prove the precondition of $add\ l_2$. Then we have to prove the post-condition in T,

$$\forall n_1, n_2.\ (inv\ l_1\ n_1 * inv\ l_2\ n_2)\ i \rightarrow (inv\ l_1\ (n_1 + 1) * inv\ l_2\ (n_2 + 1))\ m$$

under the assumption that the post-condition of $add\ l_1$ hold, instantiated with i_1 as initial heap and m_1 as final heap, and the post-condition of $add\ l_2$ instantiated with i_2 and m_2, where m splits into the two disjoint subheaps m_1 and m_2.

Hence, we have to prove that $(inv\ l_1\ (n_1' + 1) * inv\ l_2\ (n_2' + 1))$ holds for m under the further assumption that $(inv\ l_1\ n_1' * inv\ l_2\ n_2')$ holds for i. Since the post-condition of $add\ l_1$ has already been instantiated with i_1 as initial heap, we need a preciseness property about the inv predicate to be able to conclude that the subheap of i for which $inv\ l_1\ n_1'$ holds is i_1, to be able to use the post-condition of $add\ l_1$ with the new assumptions. The assumption that $inv\ l_1\ n_1$ holds for i_1 allows us to prove that $inv\ l_1\ (n_1 + 1)$ holds for m_1, but this is not very useful unless we know that $n_1 = n_1'$, which also requires a preciseness property about the inv predicate, to prove.

It is unclear whether the preciseness problem encountered with the $Map2$ Iterator is a limitation of binary post-conditions in general or the current Ynot implementation, as we have been unable to finish the proof with and without $nextvc$ for $Map2$ (without $nextvc$ the proof became too long for us to finish by hand).

Acknowledgements

We thank Bastien Maubert for his assistance with conducting some of the experiments in Ynot during his time as an intern at the IT University of Copenhagen.

References

[1] Neelakantan R. Krishnaswami and Jonathan Aldrich. Verifying object-oriented patterns with higher-order separation logic.

[2] Neelakantan R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Modular verification of the subject-observer pattern via higher-order separation logic.

[3] A. Nanevski, G. Morrisett, A. Shinnar, P. Goverau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *Proceedings of ICFP 2008*, Sep 2008.

[4] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of ICFP'06*, 2006.

[5] R.L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A Realizability Model of Impredicative Hoare Type Theory. In *Proceedings of ESOP 2008*, 2008.