Imperative Functional Programming with Isabelle/HOL

Lukas Bulwahn[1], Alexander Krauss[1], Florian Haftmann[1⋆],
Levent Erkök[2], John Matthews[2]

[1] Technische Universität München,
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
[2] Galois Inc., Beaverton, OR 97005, USA

Abstract. We introduce a lightweight approach for reasoning about programs involving imperative data structures using the proof assistant Isabelle/HOL. It is based on shallow embedding of programs, a polymorphic heap model using enumeration encodings and type classes, and a state-exception monad similar to known counterparts from Haskell. Existing proof automation tools are easily adapted to provide a verification environment. The framework immediately allows for correct code generation to ML and Haskell. Two case studies demonstrate our approach: An array-based checker for resolution proofs, and a more efficient bytecode verifier.

1 Introduction

A very common way of verifying programs in a HOL theorem prover is to use a shallow embedding and express the program as a set of recursive functions. Properties of the program can then be proved by induction. Despite some well-known limitations, shallow embeddings are widely used for verification. This success is due in part to the simplicity of the approach: A full-blown formal model of the operational or denotational semantics of the language is not required, and many technical difficulties (e.g. the representation of binders) are avoided altogether. Furthermore, the proof methods used are just standard induction principles and equational reasoning, and no specialized program logic is necessary. Code generation directly allows to turn such specifications into executable code.

Until recently, this approach has been used primarily for purely functional programs. As the notion of side-effect is alien to the world of HOL functions, programs with imperative updates of references or arrays cannot be expressed directly. However, there are many examples where for efficiency's sake imperative data structures are unavoidable to obtain performant executable programs.

We aim to permit Haskell's imperative specification style in Isabelle/HOL [10], where local state references and mutable arrays can be dynamically allocated without having to add their types to the enclosing function's type signature [5]. From such specifications we then generate efficient imperative functional code. Currently we need to restrict the contents of references and mutable arrays to first order values, but this is still sufficient for many applications.

Accordingly, the contributions of this paper are:

⋆ Supported by DFG project NI 491/10-1

1. A purely definitional polymorphic *heap* allowing to encode dynamic allocation of polymorphic first-order references and mutable arrays (§2).
2. A Haskell-style *heap monad* encapsulating the primitive heap operations and supporting abnormal termination through exceptions (§3); an adaption for Isabelle's code generator allows to generate monadic Haskell and imperative ML[3] code (§4).
3. A set of proof rules that allows to reason about such monadic programs (§5).
4. Two case studies (§6): an imperative MiniSat proof replay oracle and an imperative Jinja bytecode verifier.

1.1 Related Work

Since the seminal paper by Peyton Jones and Wadler [12], the use of monads to incorporate effects in purely functional programs is standard. However, up to now, no practically usable verification framework for such monadic programs exists.

For imperative programs, there are such tools: The Why/Krakatoa/Caduceus toolset [2] works by translating the source language into an intermediate language and using a verification condition generator to generate proof obligations. Schirmer [13] proposes a similar method, which is closely integrated with Isabelle/HOL, and whose metatheory is formally verified. These approaches rely on Hoare logic and a verification condition generator. The actual reasoning then happens on the generated verification conditions and is often outsourced to automatic provers. The user must provide enough annotations in the source code that the verification conditions can be solved by the automated prover. In our approach, reasoning happens on the source code level, which we find better suited for interactive use. Proof principles are similar to those used for purely functional programs, i.e. induction and equational reasoning.

Probably closest to our work is the concept of *single-threaded objects* [1] in the ACL2 prover. By declaring an object as single-threaded (and obeying rigorous syntactic restrictions), one instructs the prover to replace non-destructive updates by destructive ones. The rules ensure that referential transparency is not violated, and thus the code can be treated as purely functional in the reasoning phase. Our approach is similar in the sense that our heap can be seen as a single-threaded object. However, we allow the dynamic allocation of arrays and references, whereas in ACL2 imperative fields must be statically declared in a single record.

Imperative language features have previously been embedded in higher order logic via a *state monad* [7, 14]. As long as the monad primitives do not duplicate the state, the resulting programs are single threaded and can be safely transformed to monadic Haskell or imperative ML code. However, just like single-threaded objects, the state monad approach requires the state record be statically declared as part of the monad type itself, either fixed or as an explicit type parameter. This makes it difficult to write specifications that dynamically allocate new references or mutable arrays, or to compose monadic specifications that work over different state types.

Our heap model has some similarities to the one used by Tuch, Klein, and Norrish [15], especially concerning the use of type classes and phantom types to manage encodings. On the other hand, our model is slightly more abstract, since we are only dealing with functional languages instead of low level C code.

[3] in its two flavors SML and OCaml

Nanevski et al. [9] describe how Hoare logic can be integrated in dependent type theory, yielding *Hoare Type Theory*, with a sophisticated type system and program logic. However, it seems that this requires significant modifications to the theorem prover in order to support such a system. In contrast, our approach was developed on top of standard Isabelle/HOL.

1.2 A trivial example: Array reversal

To illustrate imperative functional programming in monadic HOL, we define two simple functions, one for swapping two elements in an array, and one for reversing an array:

$$
\begin{array}{l}
\mathit{swap} :: (\alpha{::}\mathit{hrep})\ \mathit{array} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{unit\ Heap} \\
\mathit{swap}\ a\ i\ j = \mathsf{do}\ x \leftarrow a[i]; \\
\qquad\qquad\qquad\quad\ y \leftarrow a[j]; \\
\qquad\qquad\qquad\quad\ a[i] := y; \\
\qquad\qquad\qquad\quad\ a[j] := x; \\
\qquad\qquad\qquad\quad\ \mathsf{return}\ () \\
\\
\mathit{rev} :: (\alpha{::}\mathit{hrep})\ \mathit{array} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{unit\ Heap} \\
\mathit{rev}\ a\ i\ j = (\mathsf{if}\ i < j\ \mathsf{then\ do}\ \mathit{swap}\ a\ i\ j; \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{rev}\ a\ (i+1)\ (j-1) \\
\qquad\qquad\quad \mathsf{else\ return}\ ())
\end{array}
$$

This idiom is well-known from Haskell: Manipulations of imperative arrays are monadic actions (of type $\alpha\ \mathit{Heap}$), and they can be composed into more complex actions using the sequencing operation provided by the monad. Other language constructs (conditionals, recursion, data types) are taken from the functional part of the language.

Let us prove a lemma that describes the behavior of swap:

$$
\begin{array}{l}
(h, h', r) \in [\![\ \mathit{swap}\ a\ i\ j\ ]\!] \Longrightarrow \\
\mathit{get\text{-}array}\ a\ h'\ !\ k = \\
(\mathsf{if}\ k = i\ \mathsf{then}\ \mathit{get\text{-}array}\ a\ h\ !\ j \\
\ \mathsf{else\ if}\ k = j\ \mathsf{then}\ \mathit{get\text{-}array}\ a\ h\ !\ i\ \mathsf{else}\ \mathit{get\text{-}array}\ a\ h\ !\ k)
\end{array}
$$

The notation $[\![\ c\ ]\!]$ stands for the big-step semantics of a command c, which is a ternary relation: $(h, h', r) \in [\![\ c\ ]\!]$ holds iff the computation c started on heap h does not generate an exception and yields the result r and the updated heap h'. The lemma expresses how the entries of the updated array are related to original entries. The function *get-array* returns the contents of an array on a given heap as a list, and infix ! denotes list indexing.

The lemma is proved by unfolding the call to swap and applying standard rules for the semantics of the monad operations and the basic commands, which can easily be automated using existing Isabelle technology. Now let us turn to the reversal function:

$$
\begin{array}{l}
(h, h', r) \in [\![\ \mathit{rev}\ a\ i\ j\ ]\!] \Longrightarrow \\
\mathit{get\text{-}array}\ a\ h'\ !\ k = \\
(\mathsf{if}\ k < i\ \mathsf{then}\ \mathit{get\text{-}array}\ a\ h\ !\ k \\
\ \mathsf{else\ if}\ j < k\ \mathsf{then}\ \mathit{get\text{-}array}\ a\ h\ !\ k\ \mathsf{else}\ \mathit{get\text{-}array}\ a\ h\ !\ (j - (k - i)))
\end{array}
$$

Since *rev* is defined recursively, we proceed by induction. The proof is as one would expect: In the step case, we distinguish the cases $k < i$, $k = i$, $i < k < j$, $k = j$ and $j < k$, and apply the induction hypothesis and the lemma about swap.

1.3 Dynamic allocation: Linked lists

The ability to explicitly allocate memory is another fundamental technique in imperative programming. To illustrate how this idiom can be coded in HOL, we will show how to build and traverse a dynamically allocated linked list.

Linked lists are represented by a recursive datatype, where the tail of the list is a mutable reference.

datatype α *node* = *Empty* | *Node* α (α *node ref*)

To convert a HOL list of elements to a linked list, we simply recurse over each tail, allocating the nodes as we go along by calling the *ref* function:

make-llist :: (α::*hrep*) *list* $\Rightarrow$ α *node Heap*
make-llist [] = **return** *Empty*
make-llist (x·*xs*) = **do** *tl* $\leftarrow$ *make-llist xs*;
next $\leftarrow$ *ref tl*;
return (*Node x next*)

In the other direction, we can traverse a linked list as follows:

traverse :: (α::*hrep*) *node* $\Rightarrow$ α *list Heap*
traverse Empty = **return** []
traverse (*Node x r*) = **do** *tl* $\leftarrow$!*r*;
xs $\leftarrow$ *traverse tl*;
return (x·*xs*)

Note that the definitions of *make-llist* and *traverse* operationally mimic their equivalents in Haskell using the state monad, or in ML using imperative features.[4]

For reference, here is the *traverse* function as rendered by the code generator of our framework in ML:[5]

```
datatype 'a node = Node of 'a * 'a node ref | Empty;

fun traverse A_ (Node (x, r)) =
  let
    val tail = ! r;
    val xs = traverse A_ tail;
  in
    (x :: xs)
  end
  | traverse A_ Empty = [];
```

[4] Technical details on the definition of *traverse* can be found in §7.3

[5] The `A_` argument denotes the dictionary, which is not used in this particular example. See [3]

2 Modeling a polymorphic heap

In the following two sections we present our definitional model of a typed heap and the monad we are using. We present the theory in a bottom-up manner, and explain the most important design decisions.

Essentially, our heap will be a mapping $h :: \mathbb{N} \Rightarrow \mathcal{V}al$ from addresses to values. However, since values can generally have arbitrary types, this is difficult to model in a simply typed language. Since there is no HOL type that can contain all types, we are facing the problem what type to choose for $\mathcal{V}al$.

This problem could be solved using dependent types, but we want to stay in the simply typed framework, so we make a draconian restriction: We decide that functions cannot be stored on the heap, and use the natural numbers as value type, in which all first-order data objects will be encoded. We'll use phantom types (§2.2) to safely omit these encodings from the generated code.

Obviously this restrictive design decision excludes a fair number of relevant programs. But even then our model allows for interesting applications. Possibilities for lifting this restriction are discussed in §7.2.

2.1 Representable Types

Using encodings to circumvent restrictions in the type system seems very awkward at first, but we can make this transparent to the user by defining an axiomatic type class *countable*, with an axiom stating that the type can be encoded into the natural numbers:

axclass *countable* $\subseteq$ *type*
$\exists\,(enc :: \alpha \Rightarrow nat).\ inj\ enc$

Obviously, basic types like *nat*, *int* and all finite types are countable, and the well-known constructions can be used to show that if α and β are countable then so are $\alpha \times \beta$ and α *list*. In fact, such instance proofs are straightforward for first-order recursive data types and could be automated. The overloaded encoding and decoding functions are called *to-nat* and *from-nat*:

to-nat :: $(\alpha$::*countable*$) \Rightarrow$ *nat*
from-nat :: *nat* $\Rightarrow (\alpha$::*countable*$)$

2.2 Typed References

References are just addresses, i.e. natural numbers

datatype α *ref* $=$ *Ref nat*

with the projection *addr-of* $(Ref\ n) = n$. Here, unlike above, the type system is again a useful tool instead of just a handicap: The *phantom type* α, which does not occur on the right hand side of the definition, allows us to view references as typed objects as we know them from ML, although the underlying representation is untyped.

Reference equality We will certainly need to reason about reference (in)equality. For example, we would expect the following simplification rule to hold, where r and s are references and h is a heap:

$$r \neq s \Longrightarrow \textit{get-ref}\ r\ (\textit{set-ref}\ s\ x\ h) = \textit{get-ref}\ r\ h$$

Indeed, when we write down and prove this lemma, everything seems to work. Only if we look at the inferred types, there is an unpleasant surprise: Because of the equality in the premise, the references r and s have the same type, and we have thus only proved a special case. Of course we want to perform the same simplification if we have references of different types, and ideally, we want the condition $r \neq s$ to be immediate, when the references have different types.

The solution is to define a heterogeneous inequality relation for references, which just strips away the phantom type and compares the bare addresses:

$$r \ncong s \leftrightarrow (\textit{addr-of}\ r \neq \textit{addr-of}\ s)$$

If r and s have the same type, the relation $\ncong$ coincides with $\neq$.

2.3 Type reflection

Comparing references of different types is a little artificial. In a typed language, aliasing between e.g. an integer and a boolean reference is not possible, and the above rewrite rule should be applicable unconditionally. Our model will be built in such a way that we can automatically derive $r \ncong s$, whenever r and s have different types:

We define a type *typerep* and a type class *typeable* to reflect the syntax of (monomorphic) types back into the language of terms:

datatype $\textit{typerep} = \textit{Typerep}\ \textit{string}\ (\textit{typerep}\ \textit{list})$

class $\textit{typeable} = \textit{type} +$
fixes $\textit{typerep} :: \alpha\ \textit{itself} \Rightarrow \textit{typerep}$

The predefined type α *itself* comes with a singleton term written $\textit{TYPE}(\alpha)$ which is used to embed types into terms. We write $\textit{RTYPE}(\alpha)$ for $\textit{typerep}\ (\textit{TYPE}(\alpha))$. The overloaded function *typerep* constructs a concrete syntactic representation of a type name. Its definition for concrete types is completely schematic (and easily automated):

$$\begin{aligned}
&\textit{RTYPE}(\textit{nat}) = \textit{Typerep}\ ''\textit{nat}''\ []\\
&\textit{RTYPE}(\textit{bool}) = \textit{Typerep}\ ''\textit{bool}''\ []\\
&\textit{RTYPE}(\alpha\ \textit{list}) = \textit{Typerep}\ ''\textit{list}''\ [\textit{RTYPE}(\alpha)]
\end{aligned}$$

The result of this exercise (which is also common in the Haskell world) is that we can now compare types for equality explicitly. For example: $\textit{RTYPE}(\textit{nat}) \neq \textit{RTYPE}(\textit{bool})$ and $\textit{RTYPE}(\textit{char list}) = \textit{RTYPE}(\textit{string})$ are theorems[6], however $\textit{RTYPE}(\alpha) \neq \textit{RTYPE}(\beta)$ is not, since α and β could later be instantiated to the same type.

Now we can refine the definition of reference inequality as follows:

[6] Note that in Isabelle, *string* just abbreviates *char list* on the surface syntax level

$$(r :: \alpha\ ref) \ncong (s :: \beta\ ref) \leftrightarrow RTYPE(\alpha) \neq RTYPE(\beta) \vee addr\text{-}of\ r \neq addr\text{-}of\ s$$

From this immediately follows that references of different types are always unequal. Hence, aliasing prove obligations like $r \ncong s$ can be solved automatically, if r and s are of different types.

2.4 The Heap

The heap is modelled as a mapping from type representations and addresses to *nat*-encoded values. We use two separate mappings for references and arrays (which are mapped to lists of encoded values). Additionally, we use a counter which bounds the currently used address space. It is incremented when new references are created.

record *heap* =
 refs :: *typerep* ⇒ *addr* ⇒ *nat*
 arrays :: *typerep* ⇒ *addr* ⇒ *nat list*
 lim :: *addr*

We can now define the basic heap operations, such as allocation, reading and writing of references. Note how the embeddings and projections are used here to convert the stored values into their respective encodings and back[7]:

get-ref :: (α::*hrep*) *ref* ⇒ *heap* ⇒ α
get-ref (*Ref r*) (*h*(|*refs* := *f*|)) = *from-nat* (*f RTYPE*(α) *r*)

set-ref :: (α::*hrep*) *ref* ⇒ α ⇒ *heap* ⇒ *heap*
set-ref (*Ref r*) *x* (*h*(|*refs* := *f*|)) = *h*
(|*refs* := *f*(*RTYPE*(α) := (*f RTYPE*(α))(*r* := *to-nat x*))|)

new-ref :: *heap* ⇒ (α::*hrep*) *ref* × *heap*
new-ref (*h*(|*lim* := *l*|)) = (*Ref l*, *h*(|*lim* := *Suc l*|))

The operations for arrays are analogous. From these definitions, we can now easily prove the expected lemmas, expressing the interaction of the operations, e.g.:

get-ref r (*set-ref r x h*) = *x*
r ≇ *s* ⟹ *get-ref r* (*set-ref s x h*) = *get-ref r h*

Since arrays and references occupy different heap areas, the corresponding heap operations always commute:

get-array a (*set-ref r v h*) = *get-array a h*
get-ref r (*heap-upd a i v h*) = *get-ref r h*

[7] The *hrep* type class just merges the classes *countable* and *typeable*.

3 The Heap Monad

We now define a monad which characterizes computations affecting the heap. An imperative program with return type α will be logically represented as a value of type α *Heap*. Essentially, our monad is a state-exception monad, where the state is the heap from the previous section:

$$\begin{array}{l}\textbf{datatype}\ \alpha\ \mathit{Heap} = \mathit{Heap}\ (\mathit{heap} \Rightarrow (\alpha + \mathit{exception}) \times \mathit{heap}) \\ \mathit{heap}\ f \qquad\qquad = \mathit{Heap}\ (\lambda h.\ \mathsf{let}\ (x, h') = f\ h\ \mathsf{in}\ (\mathit{Inl}\ x, h')) \\ \mathit{execute}\ (\mathit{Heap}\ f) = f\end{array}$$

Exceptions are essentially strings generated by $\mathit{error} :: \mathit{string} \Rightarrow \mathit{exception}$ and are not caught inside the monad; they are a mere device to introduce a notion of abnormal termination. The monad operations *return*, $(\gg\!=)$ and *raise* are defined as expected:

$$\begin{array}{l}\mathsf{return}\ x = \mathit{heap}\ (\lambda h.\ (x, h)) \\ f \gg\!= g \ = \mathit{Heap} \\ \qquad\qquad (\lambda h.\ \mathsf{case}\ \mathit{execute}\ f\ h\ \mathsf{of}\ (\mathit{Inl}\ x, h') \Rightarrow \mathit{execute}\ (g\ x)\ h' \\ \qquad\qquad\quad \mid (\mathit{Inr}\ e, h') \Rightarrow (\mathit{Inr}\ e, h')) \\ \mathsf{raise}\ s \ = \mathit{Heap}\ (\lambda h.\ (\mathit{Inr}\ (\mathit{error}\ s), h))\end{array}$$

Isabelle's syntax facilities allow for Haskell-style do-notation. Lifting the heap operations into the monad is straightforward:

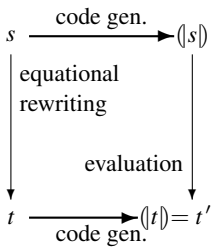
$$\begin{array}{l}\mathit{ref}\ x \ = \mathit{heap}\ (\mathit{heap\text{-}ref}\ x) \\ !r \qquad = \mathit{heap}\ (\lambda h.\ (\mathit{get\text{-}ref}\ r\ h, h)) \\ r := x = \mathit{heap}\ (\lambda h.\ ((), \mathit{set\text{-}ref}\ r\ x\ h))\end{array}$$

$$\begin{array}{l}\mathit{array}\ n\ x \ = \mathit{heap}\ (\mathit{heap\text{-}array}\ n\ x) \\ \mathit{length}\ \mathit{arr} = \mathit{heap}\ (\lambda h.\ (\mathit{heap\text{-}length}\ \mathit{arr}\ h, h)) \\ a[i] \qquad = \mathsf{do}\ \mathit{len} \leftarrow \mathit{length}\ a; \\ \qquad\qquad\quad (\mathsf{if}\ i < \mathit{len}\ \mathsf{then}\ \mathit{heap}\ (\lambda h.\ (\mathit{heap\text{-}nth}\ a\ i\ h, h)) \\ \qquad\qquad\quad\ \mathsf{else}\ \mathsf{raise}\ ''\mathit{array\ lookup}\text{: }\mathit{index\ out\ of\ range}'') \\ a[i] := x \ = \mathsf{do}\ \mathit{len} \leftarrow \mathit{length}\ a; \\ \qquad\qquad\quad (\mathsf{if}\ i < \mathit{len}\ \mathsf{then}\ \mathit{heap}\ (\lambda h.\ ((), \mathit{heap\text{-}upd}\ a\ i\ x\ h)) \\ \qquad\qquad\quad\ \mathsf{else}\ \mathsf{raise}\ ''\mathit{array\ update}\text{: }\mathit{index\ out\ of\ range}''); \\ \qquad\qquad\quad \mathsf{return}\ a\end{array}$$

These are the necessary foundations to write stateful programs like in §1.2.

4 Execution

When we consider some parts of HOL as the shallow embedding of a programming language, then the inverse of that embedding is called *code generation*. Isabelle's code generator [3] can produce SML, OCaml and Haskell code from executable HOL specifications. In a first approximation, the executable fragment of HOL consists of datatype



(a) Partial correctness

HOL	ML	Haskell
α *Heap*	α	*ST* ξ α
$t \gg= (\lambda x.\, f)$	*let* $x = t$ *in* f *end*	$t \gg= (\lambda x.\, f)$
return t	t	*return* t
α *ref*	α *ref*	*STRef* ξ α
ref x	*ref* x	*newSTRef* x
$!r$	$!r$	*readSTRef* r
$r := t$	$r := x$	*writeSTRef* r x
α *array*	α *array*	*STArray* ξ *Integer* α
array n x	*Array.array* (n, x)	*newArray* $(0, n)$ x
$a[i]$	*Array.sub* (a, i)	*readArray* a i
$a[i] := x$	*Array.update* (a, i, x)	*writeArray* a i x
length a	*Array.length* a	*liftM snd* (*getBounds* a)
raise s	*raise Fail* s	*error* s

(b) Translating monadic constructs to ML and Haskell

Fig. 1. Code generation

and function definitions, which are simply translated to their counterparts. This guarantees partial correctness: if $(\!|s|\!)$ denotes the generated code from term s, then each abstract evaluation step from $(\!|s|\!)$ to some t' in the target language corresponds to an equational rewrite step $s = t$ in HOL, such that $(\!|t|\!) = t'$ (cf. Fig. 1(a)).

The reference and array operations are mapped to the target language as given in Fig. 1(b). Since ML expressions may already contain side effects, the monad collapses to the identity on types; $\gg=$ degenerates to a let clause (which is always sequential in ML) and *return* is again the identity.

For Haskell we use the built-in *ST* state monad, together with the corresponding *STRef* and *STArray* types. Recall that our HOL programs only raise exceptions but never handle them – instead of dealing with them inside the monad, we treat them as partiality, using the *error* primitive.

Note that the extended executable fragment of HOL does not include the constructions that were used to define the heap monad: If we break the monad abstraction (e.g. by writing *heap* $(\lambda h.\, (h, h))$), the results are no longer executable and trying to generate code for them causes an error, just like for other purely logical notions like quantifiers.

5 Verification

Having defined the model of execution for our stateful programs, we need verification tools which can be used to prove an individual program correct. Our model does not force us to use a particular technique: We can choose any calculus that is sound with respect to the semantics we have defined. After a bit of experimenting, we opted for a very simple method, which seems to fit well with the structured proof language Isabelle/Isar.

We use the relational description of the big-step semantics we have already seen in §1.2. The relation is defined by:

$$(h, h', r) \in [\![c]\!] = ((\mathit{Inl}\ r, h') = \mathit{execute}\ c\ h)$$

We can prove rules which connect this relation to the different basic commands. Here is the rule for the bind operation.

$$(h, h'', r') \in [\![f \gg= g]\!] \Longrightarrow$$
$$(\bigwedge h'\ r.\ (h, h', r) \in [\![f]\!] \Longrightarrow (h', h'', r') \in [\![g\ r]\!] \Longrightarrow P) \Longrightarrow P$$

Note the elimination rule format. Since the $(\ldots) \in [\![\ \ldots\ ]\!]$ relation usually lives in the premise of a statement, we use elimination rules to manipulate it: if our goal has a premise of the form $(h, h'', r) \in [\![f \gg= g]\!]$, we can obtain the intermediate heap h' and the new assumptions $(h, h', a) \in [\![f]\!]$ and $(h', h'', r) \in [\![g\ a]\!]$. These elimination rules allow us to systematically decompose compound statements into primitive steps. Here are some other rules:

$$(h, h', r) \in [\![\mathsf{return}\ x]\!] \Longrightarrow (r = x \Longrightarrow h = h' \Longrightarrow P) \Longrightarrow P$$

$$(h, h', r) \in [\![a[i]]\!] \Longrightarrow$$
$$(r = \mathit{get\text{-}array}\ a\ h\ !\ i \Longrightarrow h = h' \Longrightarrow i < \mathit{heap\text{-}length}\ a\ h \Longrightarrow P) \Longrightarrow P$$

$$(h, h', r) \in [\![a[i] := v]\!] \Longrightarrow (r = a \Longrightarrow h' = \mathit{heap\text{-}upd}\ a\ i\ v\ h \Longrightarrow P) \Longrightarrow P$$

By feeding these rules into Isabelle's *auto* method, we obtain a reasonable *ad-hoc* automation, which makes proofs quite short.

6 Case studies

6.1 A SAT Checker

Our first case study is motivated by the wish to integrate SAT solvers into Isabelle in a scalable way, such that they can be used to solve large propositional proof obligations.

We aim at a compromise between performing a full replay of the proof within Isabelle and trusting the SAT solver completely. The first approach was taken by Weber and Amjad [16] and gives the usual high assurance of the LCF principle, but is computationally expensive. On the other end of the spectrum, trusting the external tool is obviously cheap but unsatisfactory.

A reasonable compromise is to run the external proof (a standard propositional resolution proof) through a checker, which is itself proved correct in Isabelle. This gives a good balance between assurance and cost, since unlike the SAT solver, the checker is formally verified, and checking a proof is about an order of magnitude faster than replaying the inferences in Isabelle.

Usually, for such a reflective approach, the checker would need to be purely functional. Using our framework, we can implement a checker that uses destructive arrays instead, which gives us another 30% speedup over a purely functional implementation with balanced trees.

The core of our checker operates on a table that stores the clauses that have already been derived. Clauses are modeled (purely functionally) as sorted lists of integers, where a negative number signifies a negated variable:

```
types                                  datatype ProofStep =
  idx        = nat                       Root idx clause
  lit        = int                     | Resolve idx resolvants
  clause     = lit list                | Delete idx
  resolvants = idx × (lit × idx) list
```

A proof step can either (a) add a new so-called *root clause* to the array, (b) derive a new clause from existing clauses and store it in the array, or (c) delete a clause from the array, to free some memory.

The root clauses are the initial clauses from which a contradiction is derived. It is a specialty of the MiniSAT proof format that root clauses may be added any time during the proof, hence our checker must accumulate all root clauses it encounters in a list. Then, if the checker succeeds in deriving the empty clause, the root clauses it has collected must be inconsistent.

A *Resolve* step derives a new clause in a series of resolutions: *Resolve i* (j, rs) starts with clause no. j and resolves it with the clause/variable pairs in rs. In the end, the result is stored at position i. The *Delete* proof step removes a clause from the array. This weakening step is simply an optimization to reduce memory usage of the checker by removing clauses that are no longer needed.

With clauses modeled as sorted lists, resolution is essentially a merge operation and can be done in just one traversal. However, the operation may fail if the literal does not occur in the clause. It is convenient to let the monad deal with such failures, even if no heap access is required. Hence our *resolve* operation has the following type (for brevity, we omit the implementation, which does not contain surprises):

resolve :: *lit* ⇒ *clause* ⇒ *clause* ⇒ *clause Heap*

The function *get-clause* retrieves a clause from the array. It fails if it sees a *None*:

get-clause :: *clause option array* ⇒ *idx* ⇒ *clause Heap*

The heart of our checker is the function *step*, which processes a single proof step, collecting root clauses in the accumulator list *rcs*:

```
step :: clause option array ⇒ ProofStep ⇒ clause list ⇒ clause list Heap

step a (Root cid clause) rcs = do a[cid] := Some (remdups (sort clause));
                                  return (clause·rcs)
step a (Resolve saveTo (i, rs)) rcs =
do cli ← get-clause a i;
   result ← foldM (λ(l, j) c. get-clause a j ≫= resolve l c) rs cli;
   a[saveTo] := Some result;
   return rcs
step a (Delete cid) rcs = do a[cid] := None;
                             return rcs
```

Finally, a wrapper function *checker* just allocates an array of a given size, folds the step function over a list of proof steps, and finally checks for the empty clause at some given position. Our main result is the following partial correctness theorem:

$$(h, h', cs) \in [\![\ checker\ n\ p\ i\ ]\!] \Longrightarrow inconsistent\ cs$$

Integration Since we have verified our checker, we may now choose to use it to import proofs into Isabelle. This can be done using a generic *monadic evaluation oracle*, which implements the following inference rule:

$$\frac{\bigwedge h\ h'.\ (h, h', r) \in [\![\ c\ ]\!] \Longrightarrow P\ r}{P\ r}\ \text{(if } c\text{, when executed in ML, evaluates to } r\text{)}$$

Thus we can discharge the premise of a partial correctness theorem by just running the generated code in ML.

The soundness of this rule relies on the fact that the semantics of ML is compatible with our model of monadic programs, a claim that we have not formalized.

However, such a generic reflection mechanism, which provides a clearly defined way to extend the theorem prover by reflected imperative proof tools, still provides higher assurance than an ad-hoc extension, since the monadic code is verified, and no additional "glue code" is required for the integration.

In particular, nothing in our particular development of the SAT checker needs to be trusted.

6.2 A Jinja Bytecode verifier

Our second case study is a modification of the Jinja bytecode verifier. Jinja [6] is a complete formal model of a Java-like language, which includes a formal semantics, type system, virtual machine model, compiler, and bytecode verifier.

Essentially, the bytecode verifier performs an abstract interpretation of the bytecode instructions, keeping track of the abstract state, that is, the types of values in registers and on the stack. The central data structure is a mapping that assigns such an abstract state to every bytecode instruction. Then, this information is propagated to the successors of the instruction until a fixed point is reached.

In the existing implementation, this mapping is represented by a list of fixed length. In our modification, we use an imperative array instead, with the obvious advantages: constant-time access and no garbage.

Fortunately, the bytecode verifier is modeled in a very abstract framework using a semilattice (type σ), which hides all the technical details of the virtual machine. Later, the "real thing" can be obtained by instantiation. A bytecode method is modelled by a function $step :: nat \Rightarrow \sigma \Rightarrow (nat \times \sigma)\ list$ that maps a given program position and an abstract machine state to a list of possible successor positions and states. Additional requirements for the *step* function (e.g. monotonicity) are detailed in [6].

Figure 2 shows the pure version of the bytecode verifier together with its monadic counterpart. This side-by-side comparison shows that the differences between the two versions are small. Consequently, proving partial correctness of *kildallM* wrt. *kildall* is straightforward:

$$\tau s \in \mathit{list}\ n\ A \Longrightarrow (h, h', \tau s') \in [\![\ \mathit{kildallM}\ \tau s\ ]\!] \Longrightarrow \tau s' = \mathit{kildall}\ \tau s$$

This shows that it is relatively easy to move from a purely functional specification to a monadic one, which can then be executed efficiently.

```
propa [] τs w = (τs, w)
propa ((q, τ)·qs) τs w =
(let u = τ ⊔ τs ! q;
    w' = if u = τs ! q then w else {q} ∪ w
 in propa qs (τs[q := u]) w')

iter (τs, w) =
(if w = ∅ then τs
 else let p = SOME p. p ∈ w
      in iter
           (propa (step p (τs ! p)) τs (w − {p})))

kildall τs = iter (τs, unstables τs)
```

```
propaM [] τs w = return w
propaM ((q, τ)·qs) τs w =
do τ' ← τs[q];
   let u = τ ⊔ τ';
   let w' = (if u = τ' then w else {q} ∪ w);
   τs[q] := u;
   propaM qs τs w'

iterM τs w =
(if w = ∅ then list-of τs
 else let p = SOME p. p ∈ w
      in do v ← τs[p];
            w' ← propaM (step p v) τs
                   (w − {p});
            iterM τs w')

kildallM τs =
do a ← of-list τs;
   iterM a (unstables τs)
```

Fig. 2. Pure vs. monadic versions of the bytecode verifier

7 Problems and Limitations

7.1 No Monad Polymorphism

Of course, one would like to specify a monad as a constructor class, and see our heap monad just as a particular instance of the general concept. However, for this we would need type constructor polymorphism, which is not supported in HOL. We must be satisfied with the possibility of defining concrete instances of monads.

Huffman, Matthews, and White [4] describe how to simulate constructor classes in an extension of HOL, but their embedding does not seem practical for our application.

7.2 Heap model

Our simple heap model prohibits storing any kind of function value in mutable references. Although many applications can live with this limitation, it may be painful in other situations. One can think of different ways to improve this situation:

Encoding types of order n. Just like we now encode all first-order values in $\mathbb{N}$, one could also encode all functions on such values by $\mathbb{N} \Rightarrow \mathbb{N}$, and all functions that take such arguments by $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$, and so on. For any given order, we can encode all "smaller" types in a single type. Again, this can be made transparent using type classes. Probably, order 3 or 4 would be enough for most practical purposes.

Dependent types In a dependently typed system, one could do without explicit encodings, and represent heap values as a dependent pair of a type and a value. In such a system, the type $heap$ would live in some higher universe than the types used in a program.

ZF extension HOLZF [11] is a consistent extension of HOL which declares a set-theoretic universe Z, in which all HOL types can be embedded. In such a system we could store the full tower of (pure, monomorphic) higher order functions over the naturals, since our heap function could take values in Z.

However, even these extensions will not allow us to store *monadic* functions in the heap. The collection of heap monad functions has at least the cardinality of $heap \Rightarrow heap$, which is strictly larger than $heap$ itself in classical HOL.

One avenue of escape would be to limit ourselves to the constructive portion of HOL and build some kind of impredicative datatype facility to represent the heap. A more pragmatic option is to store only a representable subset of the full function space in the heap, for example just the continuous functions as is done in Isabelle/HOLCF[8]. We would retain the full power of classical HOL while still allowing to store all (partially) executable functions, which are the only ones we are really interested in.

7.3 Recursive Functions

Monadic functions can be defined recursively just like any other function by using the available packages in Isabelle. However, proving termination of the functions can sometimes be tricky, as the following example demonstrates:

$$
\begin{aligned}
&f :: \mathit{nat\ ref} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat\ Heap} \\
&f\ r\ n = \mathsf{do}\ x \leftarrow !r; \\
&\qquad (\mathit{if}\ x = 0\ \mathsf{then\ return}\ n\ \mathsf{else\ do}\ r := x - 1; \\
&\qquad\qquad f\ r\ (\mathit{Suc}\ n))
\end{aligned}
$$

Since there is no wellfounded order for which $(r, \mathit{Suc}\ n) \prec (r, n)$ holds, we cannot hope to define f by wellfounded recursion on its arguments. Instead, the recursion happens on the heap itself, which is not an explicit argument of the function. To define f, we must first break the monad abstraction and define a function $f' :: \mathit{nat\ ref} \Rightarrow \mathit{nat} \Rightarrow \mathit{heap} \Rightarrow (\mathit{nat} + \mathit{exception}) \times \mathit{heap}$, which explicitly recurses over the heap. Then f can be defined in terms of f', deriving the above recursion equation.

Another issue is that when building pointer structures on the heap, many functions are actually partial, since the structures can become cyclic. The attentive reader may have noticed that the *traverse* function in §1.3 is in fact such an example.

However, it turns out that even nonterminating recursive functions are definable if the recursion happens within the heap monad, since such definitions always have a total model. This argument is similar to the observation that tail-recursive functions can always be defined in HOL (e.g. using a *while* combinator). The details are beyond the scope of this paper, so we just mention that *traverse* was defined using a monadic recursion combinator *MREC*, which satisfies the following recursion equation:

$$\begin{array}{l} \mathit{MREC} :: (\alpha \Rightarrow (\beta + \alpha)\ \mathit{Heap}) \Rightarrow (\alpha \Rightarrow \alpha \Rightarrow \beta \Rightarrow \beta\ \mathit{Heap}) \Rightarrow \alpha \Rightarrow \beta\ \mathit{Heap} \\ \mathit{MREC}\ f\ g\ x = \\ \mathsf{do}\ y \leftarrow f\ x; \\ \quad (\mathsf{case}\ y\ \mathsf{of}\ \mathit{Inl}\ r \Rightarrow \mathsf{return}\ r \mid \mathit{Inr}\ s \Rightarrow \mathsf{do}\ z \leftarrow \mathit{MREC}\ f\ g\ s; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g\ x\ s\ z) \end{array}$$

In the future, we plan to provide automated packages to facilitate the definition of such recursive functions.

7.4 External I/O

Another practical limitation is that our heap monad does not support any kind of interaction with the outside world. This means, for example, that the full sequence of MiniSat proof steps needs to be passed into our SAT checker from the start. This becomes a problem for long-running proofs where the number of steps may exceed the total size of Isabelle's memory.

However, if our monad supported IO actions then we could incrementally ask MiniSat to supply us just the next portion of the proof to check, and never have to represent the entire proof at once. Supporting IO would require us to extend our heap model to include relevant aspects of the outside system, plus some kind of nondeterminism for IO actions to take into account that we can never model the world in its entirety.

8 Conclusion

We presented a lightweight approach to reuse our favorite theorem prover for verifying monadic programs that manipulate a state. Our shallow embedding of imperative constructs in HOL is a continuation of the traditional way of modeling programs and systems by recursive functions, which can be translated to "real" programs by a code generator. Although there are still some limitations (see §7), our case studies show that it is already quite useful in its current form. Equipped with that, we want to tackle specification, verification and prototypic code generation for compute-intensive applications like e.g. microprocessor models. Another important application is the extension of the Isabelle system itself by means of verified monadic proof procedures as we have sketched it in §6.1.

Future work will also focus on alleviating current limitations, most notably to allow a broader range of heap-representable types, monadic I/O, and more automation for defining monadic recursive functions.

9 Acknowledgments

We would like to thank David Hardin, Joe Hurd, Matt Kaufmann, Dylan McNamee, Tobias Nipkow, Konrad Slind, and Tjark Weber for their useful discussions, encouragement and feedback on our work.

References

[1] R. S. Boyer and J. S. Moore. Single-threaded objects in ACL2. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 9–27, London, UK, 2002. Springer.

[2] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, LNCS, Berlin, Germany, July 2007. Springer.

[3] F. Haftmann and T. Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, 08 2007.

[4] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle/HOLCF. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 147–162. Springer, 2005.

[5] S. P. Jones and J. Launchbury. Lazy functional state threads. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, 1994.

[6] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[7] S. Krstić and J. Matthews. Verifying BDD algorithms through monadic interpretation. In *VMCAI '02: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 182–195, London, UK, 2002. Springer.

[8] O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[9] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 62–73, New York, NY, USA, 2006. ACM.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[11] S. Obua. Partizan games in Isabelle/HOLZF. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC*, volume 4281 of *LNCS*, pages 272–286. Springer, 2006.

[12] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

[13] N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452, pages 398–414. Springer, 2005.

[14] C. Sprenger and D. A. Basin. A monad-based modeling and verification toolbox with application to security protocols. In K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 302–318. Springer, 2007.

[15] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, Jan. 2007.

[16] T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 2007. To appear.