Refinement to Imperative HOL

DOI:
10.1007/s10817-017-9437-1

Document Version
Accepted author manuscript

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Refinement to Imperative HOL

Peter Lammich

Received: date / Accepted: date

Abstract Many algorithms can be implemented most efficiently with imperative data structures. This paper presents Sepref, a stepwise refinement based tool chain for the verification of imperative algorithms in Isabelle/HOL.

As a back end we use Imperative HOL, which allows to generate verified imperative code. On top of Imperative HOL, we develop a separation logic framework with powerful proof tactics. We use this framework to verify basic imperative data structures and to define a refinement calculus between imperative and functional programs. We provide a tool to automatically synthesize a concrete imperative program and a refinement proof from an abstract functional program, selecting implementations of abstract data types according to a user-provided configuration. As a front end to describe the abstract programs, we use the Isabelle Refinement Framework, for which many algorithms have already been formalized. Our tool chain is complemented by a large selection of verified imperative data structures.

We have used Sepref for several verification projects, resulting in efficient verified implementations that are competitive with unverified ones in Java or C++.

1 Introduction

Using the Isabelle Refinement Framework (IRF) [19,26], we have verified several graph and automata algorithms [26,10,20,24], including a fully verified LTL model checker [12]. The IRF features a stepwise refinement approach [39], where an abstract algorithm is refined, in possibly many steps, to a concrete implementation. Organizing the proof into multiple steps reduces its complexity and makes larger developments manageable in the first place. However, the IRF only allows refinement to purely *functional* code, while the most efficient implementations of many algorithms require *imperative* features.

In this paper, we extend the IRF to imperative programs. We build on Imperative HOL [4], which introduces a heap monad in Isabelle/HOL and supports code

Peter Lammich
TU Munich, Boltzmannstr. 3, D-85748 Garching
Tel.: +49-89-289-17326
E-mail: lammich@in.tum.de

generation for several target platforms (currently OCaml, SML, Haskell, and Scala). Unfortunately, Imperative HOL has rarely been used for verification projects so far, mainly due to its limited proof support. We improve on that by developing a separation logic framework with powerful proof tools that greatly simplify reasoning about programs in Imperative HOL. (§2) We use this framework to verify basic algorithms and data structures,(§3) and as a basis of a refinement calculus [2] from IRF programs to Imperative HOL. Moreover, we implement the Sepref tool, which automatically generates an Imperative HOL program and a refinement proof from an IRF program, selecting appropriate implementations for the abstract data structures based on a configuration provided by the user.(§4) On top of Sepref, we implement the Imperative Collections Framework, which provides a library of reusable imperative data structures and convenience tools for data structure development. (§5) We have successfully used Sepref for numerous verification projects. The resulting implementations are considerably faster than previous functional versions, and, in some cases, competitive to unverified imperative implementations in Java and C++. (§6)

The Sepref tool and the Imperative Collections Framework are available as an entry in the Archive of Formal Proofs [21]. This entry also contains a user guide that describes the development cycle featured by Sepref, a reference manual, and several larger examples.

2 A Separation Logic for Imperative HOL

Imperative HOL provides a heap monad formalized in Isabelle/HOL, as well as a code generator extension to generate imperative code in several target languages (currently OCaml, SML, Haskell, and Scala). However, Imperative HOL itself only comes with minimalistic support for reasoning about programs. In this section, we report on our development of a separation logic framework for Imperative HOL. A preliminary version, which did not support frame inference nor other automation, was formalized by Meis [28]. The current version is available in the Archive of Formal Proofs [23].

2.1 Basics

We formalize separation logic [34] along the lines of Calcagno et al. [5]. A *partial heap* (type *pheap*) describes the content of a heap at a specific set of addresses. An *assertion* (type $assn \subset (pheap \Rightarrow bool)$) is a predicate on partial heaps that satisfies a well-formedness condition[1]. We write $h \models P$ if the partial heap h satisfies the assertion P.

The assertion *true* is satisfied by any heap, *false* is satisfied by no heap, and *emp* by the empty heap. A heap consisting of a single cell at address $p :: \alpha\ ref$ with value $v :: \alpha$ is described by $p \mapsto_r v$. Analogously, an array at address $p :: \alpha\ array$ holding the values $l :: \alpha\ list$ is described by $p \mapsto_a l$. Moreover, the *pure* assertion $\uparrow \Phi$ describes the empty heap if $\Phi :: bool$ holds, and no heap otherwise. It is used to embed propositions from HOL into assertions. We lift the standard Boolean

[1] For technical reasons, we formalize a partial heap as a full heap with an address range. Assertions must not depend on heap content outside this address range.

connectives to assertions and show that they form a Boolean algebra. We also provide lifted versions of the universal and existential quantifiers. The *separation conjunction* $P * Q$ is satisfied by heaps that can be split into two disjoint parts, such that one part satisfies P and the other part satisfies Q. Finally, we define the *entailment* $P \Longrightarrow_A Q$ as $\forall h.\ h \models P \Longrightarrow h \models Q$.

Example 1 Singly linked lists are a standard benchmark for separation logic tools. We first define a *node*, which contains a value and a next pointer:

datatype α *node* = *Node val:* α *next:* α *node ref option*

As Imperative HOL does not support null pointers, we use an option type for the next pointer, where *None* models the null pointer.

The assertion *lseg l p s* describes a linked list starting at the node pointed to by p, ending at the node pointed to by s (exclusive), and holding the data described by the HOL list l:

fun *lseg* :: α *list* $\Rightarrow$ α *node ref option* $\Rightarrow$ α *node ref option* $\Rightarrow$ *assn* **where**
lseg [] *p s* = $\uparrow$(*p*=*s*)
| *lseg* (*x*#*l*) (*Some p*) *s* = (∃*q*. *p* ↦$_r$ *Node x q* ∗ *lseg l q s*)
| *lseg* (_#_) *None* _ = *false*

Then, the assertion *os_list l p* ≡ *lseg l p None* describes an *open* singly linked list, i. e. one where the last element's next pointer is null.

2.2 Automation

One of the most important proof tools in Isabelle/HOL is the simplifier, which normalizes a term according to a configurable set of rewrite rules. It also supports simplifier procedures, which dynamically generate rewrite rules based on the current (sub)term to be rewritten. We configure the simplifier for handling assertions: Instantiating the type classes for *Boolean algebras* and *commutative monoids* with assertions already yields a basic simplifier setup. Additionally, we implemented a simplifier procedure for lists of assertions separated by $*$: Existential quantifiers are pulled to the front, pure assertions are summarized, and assertions that would force the same pointer to point to separate locations are rewritten to false.

Example 2 The simplifier rewrites the assertion $P * \uparrow \Phi * (\exists p.\ p \mapsto_r v * \uparrow \Psi)$ to $\exists p.\ P * p \mapsto_r v * \uparrow (\Phi \wedge \Psi)$, and the assertion $P * \uparrow \Phi * (\exists p.\ p \mapsto_r v * \uparrow \Psi * p \mapsto_r w)$ is rewritten to False (as p cannot point to two separate locations at the same time).

2.3 Hoare Triples

Having defined assertions, we are ready to define a separation logic for programs. Imperative HOL provides a shallow embedding of heap-manipulating programs into Isabelle/HOL. A program is encoded in a heap-exception monad, i. e. it has type α *Heap* = *heap* $\Rightarrow$ (α $\times$ *heap*) *option*. Intuitively, a program takes a heap and either produces a result of type α and a new heap, or fails.

We define the *Hoare triple* $\langle P\rangle\ c\ \langle Q\rangle$ to hold iff for all heaps that satisfy P, the program c returns a result x such that the new heap satisfies $Q\ x$.[2] When reasoning about garbage collected languages, one has to frequently specify that an operation may allocate some heap space for internal use. For this purpose, we define $\langle P\rangle\ c\ \langle Q\rangle_t$ as a shortcut for $\langle P\rangle\ c\ \langle\lambda x.\ Q\ x * true\rangle$.

For Hoare triples, we prove rules for the basic heap operations and monad combinators, as well as a consequence and a frame rule. Note that the frame rule, $\langle P\rangle\ c\ \langle Q\rangle \Longrightarrow \langle P * F\rangle\ c\ \langle\lambda x.\ Q\ x * F\rangle$, is crucial for modular reasoning in separation logic. Intuitively, it states that a program does not depend on the content of the heap that it does not access.

Example 3 We display the Hoare rules for return, bind, and array lookup:

$$\frac{-}{\langle P\rangle\ return\ x\ \langle\lambda r.\ P * \uparrow(r = x)\rangle} \qquad \frac{\langle P\rangle\ m\ \langle R\rangle \quad \forall x.\ \langle R\ x\rangle\ f\ x\ \langle Q\rangle}{\langle P\rangle\ x \leftarrow m; f\ x\ \langle Q\rangle}$$

$$\frac{i < |xs|}{\langle a \mapsto_a xs\rangle\ a!i\ \langle\lambda r.\ a \mapsto_a xs * \uparrow(r = xs!i)\rangle}$$

Note that these rules work in a forward manner, i. e. given a precondition and a command, they generate a (the strongest) postcondition. After a return, the heap is not changed, and, additionally, the result is the returned value. For a bind, we first generate a postcondition R for the first statement, and then a postcondition for the second statement, given that the argument satisfies R. Finally, for array lookup, we require the index to be in bounds.

2.3.1 Recursion

We do not provide explicit rules for recursion combinators, but rely on the standard Isabelle infrastructure, in particular on the partial function package [16]. However, in Section 4, we will provide rules to refine the recursion combinators of the IRF to corresponding recursion combinators of the heap monad.

Example 4 The following function implements in-place list reversal for open singly linked lists (cf. Example 1):

```
partial_function (heap) os_reverse_aux
  :: α os_list ⇒ α os_list ⇒ α os_list Heap
where
  os_reverse_aux q p = (case p of
    None ⇒ return q
  | Some r ⇒ do {
      v ← !r;
      r := Node (val v) q;
      os_reverse_aux p (next v) })
definition os_reverse p = os_reverse_aux None p
```

[2] Again, for technical reasons, we additionally check that the program does not modify addresses outside the heap's address range, and that it does not deallocate memory.

2.4 Verification Condition Generator

Given a Hoare triple $\langle P\rangle\ c\ \langle Q\rangle$, we can use the Hoare rules to compute a postcondition Q' with $\langle P\rangle\ c\ \langle Q'\rangle$, and then try to prove $Q' \Longrightarrow_A Q$.

While the rules for the combinators are set up to work with preconditions of any form, the rules for operations require the heap to contain the operands. Assume we have an operation c with the rule $\langle P'\rangle\ c\ \langle Q\rangle$. In order to compute the postcondition for a precondition P, we have to find a *frame assertion* F such that $P \Longrightarrow_A P' * F$. With the consequence and frame rules, we then get $\langle P\rangle\ c\ \langle\lambda x.\ Q\ x * F\rangle$.

The problem of finding an assertion F is called *frame inference*, and is undecidable in general. However, there are heuristics that work well in practice. For a detailed discussion on automating frame inference in HOL theorem provers we refer the reader to [38]. We implement a quite simple but effective heuristics: After some initial simplifications to handle quantifiers and pure predicates, we split P and P' into $P = P_1 * \ldots * P_n$ and $P' = P'_1 * \ldots * P'_m$. Then, for every P_i, we find the first P'_j that can be unified with P_i. If we succeed to match up all P_is, without using a P'_j twice, we have found a valid frame (F consists of exactly the unused P'_js), otherwise the heuristic fails and frame inference has to be performed manually.

2.5 All-in-One Method

We combine the verification condition generator, a heuristics to simplify goals of the form $Q' \Longrightarrow_A Q$, and Isabelle/HOL's *auto* tactic into a single proof tactic named *sep_auto*. This tactic is able to solve many goals involving separation logic completely automatically. Moreover, if it cannot solve a goal, it returns the proof state at which it got stuck. This is a valuable tool for proof exploration, as the stuck state usually hints to missing lemmas. The *sep_auto* tactic allows for very straightforward and convenient proofs, which are considerably smaller and simpler than the corresponding proofs carried out with the default rudimentary proof methods of Imperative HOL.

Example 5 The original Imperative HOL formalization [4] also contains an example of in-place list reversal (cf. Example 4). The correctness proof there requires about 100 lines of quite involved proof text. Using separation logic and the *sep_auto* tactic, the proof reduces to a few lines of straightforward proof text:

```
lemma aux: ⟨os_list xs p * os_list ys q⟩ os_reverse_aux q p ⟨os_list (rev xs @ ys)⟩
proof (induct xs arbitrary: p q ys)
  case Nil thus ?case by sep_auto
next
  case (Cons x xs) show ?case
    by (cases p; sep_auto heap: cons_pre_rule[OF _ Cons.hyps])
qed

corollary ⟨os_list xs p⟩ os_reverse p ⟨os_list (rev xs)⟩
  unfolding os_reverse_def using aux[where ys=[]]
  by auto
```

3 Simple Refinement to Imperative HOL

In the last section, we have reported on our separation logic framework for Imperative HOL and its powerful tools for proof automation. It can be used to verify simple algorithms (e. g. in-place list reversal). Also algorithms of medium complexity can be handled, using a lightweight but somewhat limited stepwise refinement approach. In this section, we first report on our formalization of a union-find data structure using this approach, and then discuss its limitations.

3.1 Refinement Based Development of a Union-Find Data Structure

Recall that a union-find data structure is used to model equivalence relations, in our case on an initial segment $\{0\ldots<N\}$ of the natural numbers. The union-find data structure is a forest over the nodes $\{0\ldots<N\}$. Each tree in the forest represents an equivalence class, the root node being the unique representative. The compare operation checks whether two elements are equivalent by comparing the root nodes of the elements' trees. The union operation joins the equivalence classes of two elements by attaching one tree to the root node of the other tree.

We also implemented *path compression* and the *size-based union heuristics*, which guarantee a quasi-constant amortized complexity of the compare operation. These are omitted here to keep the presentation simple.

A convenient implementation of the union-find data structure is by an array $[a_0\ldots a_{N-1}]$ of natural numbers, such that a_i is the parent node of i. Root nodes simply point to themselves.

In a first step, we model the array as a functional list. Isabelle/HOL's list library provides the function $|l|$ to obtain the length of a list l, the function $l!i$ to obtain the ith element, and the function $l[i := x]$ to obtain a list equal to l, except that the ith element is replaced by x. We define:

$$\mathit{find}_1\ l\ i = (\textbf{if}\ l!i = i\ \textbf{then}\ i\ \textbf{else}\ \mathit{find}_1\ l\ (l!i))$$
$$\mathit{invar}_1\ l = \forall i < |l|.\ \mathit{find}_1\_\mathit{dom}\ (l,i) \wedge l!i < |l|$$
$$\alpha_1\ l = \{(x,y).\ x < |l| \wedge y < |l| \wedge \mathit{find}_1\ l\ x = \mathit{find}_1\ l\ y\}$$
$$\mathit{union}_1\ l\ x\ y = l[\mathit{find}_1\ l\ x := \mathit{find}_1\ l\ y]$$

Here, the partial[3] function find_1 follows the parent pointers until a root node is reached. The predicate $\mathit{find}_1\_\mathit{dom}$ defines the arguments for which find_1 is defined (i. e. terminates). The invariant predicate states that the find_1 function must be defined on all elements, and that all parent pointers must point to valid elements. The abstraction function α_1 maps a list to the corresponding equivalence relation. Finally, the union_1 operation joins two equivalence classes as described above.

Using standard Isabelle reasoning, we show correctness of the union operation:

$$[\![\ \mathit{invar}_1\ l;\ x < |l|;\ y < |l|\ ]\!]$$
$$\Longrightarrow \alpha_1\ (\mathit{union}_1\ l\ x\ y) = \mathit{union}\ (\alpha_1\ l)\ x\ y \wedge \mathit{invar}_1\ (\mathit{union}_1\ l\ x\ y)$$

where *union* is the union operation on equivalence relations, and $[\![P_1;\ldots;P_n]\!] \Longrightarrow Q$ is syntactic sugar to summarize multiple premises of an implication.

In a second step, we implement the list by an array and define corresponding find and union functions in the Imperative HOL monad:

[3] The Isabelle/HOL function package [15] allows for convenient definition of such functions.

```
partial_function (heap) find₂ :: nat array ⇒ nat ⇒ nat Heap where
 find₂ p i = do {
   n ← Array.nth p i;
   if n=i then return i else find₂ p n }

 definition union₂ a i j = do {
   i ← find₂ a i; j ← find₂ a j;
   Array.upd i j a }
```

The following theorems state the correspondence of these functions to their list-based counterparts:

$$[\![\ invar_1\ l;\ i < |l|\ ]\!] \Longrightarrow \langle p \mapsto_a l\rangle\ find_2\ p\ i\ \langle\lambda r.\ p \mapsto_a l * \uparrow(r = find_1\ l\ i)\rangle$$
$$[\![\ invar_1\ l;\ i < |l|; j < |l|\ ]\!] \Longrightarrow \langle a \mapsto_a l\rangle\ union_2\ a\ i\ j\ \langle\lambda r.\ r \mapsto_a union_1\ l\ i\ j\rangle$$

Using the *sep_auto* tactic, they are easily proved with a few lines of proof text.

Finally, we combine the two refinement steps. We define an assertion $is\_uf\ R\ a$ that states that an array a represents an equivalence relation R:

$$is\_uf\ R\ a \equiv \exists l.\ a \mapsto_a l * \uparrow(invar_1\ l \wedge R = \alpha_1\ l)$$

Combining the correctness theorems for $union_1$ and $union_2$ yields a correctness theorem for the union operation on arrays wrt. equivalence relations:

$$[\![\ i \in Domain\ R; j \in Domain\ R\ ]\!] \Longrightarrow \langle is\_uf\ R\ a\rangle\ union_2\ a\ i\ j\ \langle is\_uf\ (union\ R\ i\ j)\rangle$$

Again, this theorem is easily shown with the *sep_auto*-tactic.

3.2 Limitations of the Simple Approach

Above, we have presented a refinement technique where an algorithm is first defined as a plain HOL function on standard HOL data types, and then refined to a monadic function on heap-based data types. This technique works well for simple algorithms and data structures. It even allows modularity: a data structure that has already been proved may be used as building block for more complex algorithms and data structures. However, in practice, one quickly encounters various problems for more complex algorithms:

- An implementation often requires some knowledge of the algorithm, in particular that it implies certain restrictions on the types to be implemented. These restrictions are usually obvious when proving the abstract algorithm correct. However, there is no simple way to transport them to the refinement proof where they are required. Instances of this problem already occur in the union-find data structure: In order to prove $union_2$ correct, one has to show that the index of the array update is within bounds. However, this index is the result of a call to $find_2$. In this particular case, the problem is easily discharged by showing that $find_1$ only returns valid elements, and using this fact after the result of $find_2$ has been related to the result of $find_1$. However, in more complex algorithms, substantial parts of the correctness proof may have to be repeated to get the properties required for the refinement. This blurs the separation between abstract and concrete levels, annihilating the positive effects of the stepwise refinement approach.

– The abstract formulation of an algorithm is often inherently nondeterministic. For example, when picking some element from a set, the element which is actually picked depends on the set's implementation, and cannot be determined on the abstract level. This is a common problem when developing algorithms in HOL, and there exists several workarounds, e. g. imposing a total ordering on the elements and returning the smallest one. However, it's not hard to run into cases where these simple tricks fail, e. g. when computing a path between two nodes in a graph.
– Many algorithms are naturally presented in an "imperative" style, using loops instead of recursive functions. An encoding of loops into recursive functions is possible, but tends to obfuscate the algorithms, and also the proofs, which have to be converted from invariant proofs to induction proofs.

4 Using the Isabelle Refinement Framework

The Isabelle Refinement Framework (IRF) provides a possible solution to the limitations sketched in the last section. Programs are described in a nondeterminism monad, shallowly embedded into Isabelle/HOL. This is a flexible but still lightweight way to describe nondeterministic algorithms in Isabelle/HOL. Moreover, assertions can be used to easily transport facts from the correctness proof to the refinement proof. Finally, the monad comes with loop combinators, which allow a more natural presentation of some algorithms.

In this section, we describe our approach to refinement from IRF programs to Imperative HOL programs. We start with a brief review of the IRF. For a more detailed description, we refer the reader to [26,18]. Programs are described via a nondeterminism monad over the type α *nres*, which is defined as follows:

datatype α *nres* = **res** (α *set*) | **fail**
fun $\leq$ :: α *nres* $\Rightarrow$ α *nres* $\Rightarrow$ *bool*
  where _ $\leq$ **fail** | **fail** $\not\leq$ **res** _ | **res** $X \leq$ **res** Y *iff* $X \subseteq Y$
fun **return** :: $\alpha \Rightarrow \alpha$ *nres* where **return** $x \equiv$ **res** $\{x\}$
fun **bind** :: α *nres* $\Rightarrow$ ($\alpha \Rightarrow \beta$ *nres*) $\Rightarrow$ β *nres*
  where **bind fail** $f \equiv$ **fail** | **bind** (**res** X) $f \equiv$ *SUP* $x \in X$. $f\ x$

The type α *nres* describes nondeterministic results, where **res** X describes the nondeterministic choice of an element from X, and **fail** describes a failed assertion. On nondeterministic results, we define the *refinement ordering* $\leq$ by lifting the subset ordering, setting **fail** as top element. The intuitive meaning of $a \leq b$ is that *a refines b*, i. e. results of a are also results of b. Note that the refinement ordering is a complete lattice with top element **fail** and bottom element **res** $\{\}$.

Intuitively, **return** x denotes the unique result x, and **bind** m f denotes sequential composition: Select a result from m, and apply f to it.

Non-recursive programs can be expressed by these monad operations and Isabelle/HOL's **if** and **case**-combinators. Recursion is encoded by a fixed point combinator **rec** :: $((\alpha \Rightarrow \beta$ *nres*$) \Rightarrow \alpha \Rightarrow \beta$ *nres*$) \Rightarrow \alpha \Rightarrow \beta$ *nres*, such that **rec** F is the greatest fixed point of the monotonic functor F, wrt. the flat ordering of result sets with **fail** as the top element. For non-monotonic F, **rec** F is set to **fail**:

rec $F\ x \equiv$ **if** (*mono*$'$ F) **then** (*flatf_gfp* $F\ x$) **else fail**

```
definition dfs :: (α × α) set ⇒ α ⇒ α ⇒ bool nres where
  dfs E s t ≡ do {
    (_,r) ← rec (λdfs (V,v).
      if v ∈ V then return (V,False)
      else do {
        let V = insert v V;
        if v = t then return (V,True)
        else foreach ({v'. (v,v') ∈ E}) (λ(_,brk). ¬brk)
               (λv' (V,_). dfs (V,v')) (V,False)
      }
    ) ({},s);
    return r
  }
```

Listing 1: Simple DFS algorithm formalized in the IRF

Here, $mono'$ denotes monotonicity wrt. both the flat ordering and the refinement ordering. The reason is that for functors that are monotonic wrt. both orderings, the respective greatest fixed points coincide, which is useful to show proof rules for refinement. Note that functors which only use the monad combinators described above are monotonic by construction [16].

Building on the basic combinators, the IRF also defines **while** and **foreach** loops to conveniently express tail recursion and folding over the elements of a finite set. Moreover, we define assertions by:

assert $\Phi \equiv$ **if** Φ **then return** () **else fail**

For assertions, we have the following rules:

$[\![\ \Phi;\ m \le m'\ ]\!] \Longrightarrow$ **do** { **assert** Φ; m } $\le m'$
$[\![\ \Phi \Longrightarrow m \le m'\ ]\!] \Longrightarrow m \le$ **do** { **assert** Φ; m' }

Here, we use a Haskell-like **do** notation as convenient syntax for bind operations. The first rule is used to show that a program m with assertion Φ refines the program m'. It requires to prove Φ, in addition to the refinement $m \le m'$. The second rule is used to show that a program m refines a program m' with an assertion. It allows one to assume Φ when proving the refinement $m \le m'$. This way, facts that are proved on the abstract level are made available for proving refinement.

Example 6 Listing 1 displays the IRF formalization of a simple depth-first search algorithm that checks whether a directed graph, described by a (finite) set of edges E, has a path from source node s to target node t. With the tool support provided by the IRF, it is straightforward to prove this algorithm correct and refine it to efficient functional code (cf. [26,19]).

4.1 Connection to Imperative HOL

With the Isabelle Refinement Framework we have developed various algorithms and refined them to efficient purely functional. In this section, we describe how to refine a program specified in the nondeterminism monad of the IRF to a program specified in the heap monad of Imperative HOL. The main challenge is to refine abstract data to concrete data that may be stored on the heap and updated destructively.

At this point, we have a design choice: One option is to refine the abstract functional program first to an abstract imperative program, and then to a concrete program. The second option is to skip the intermediate step, and directly refine abstract functional programs to concrete imperative ones.

Due to limitations of the logic underlying Isabelle/HOL, we need a single HOL type that can encode all types we want to store on the heap. In Imperative HOL, this type is chosen to be $\mathbb{N}$, and thus only countable types can be stored on the heap. As long as we store concrete data structures, this is no real problem. However, abstract data types are in general not countable, nor does there exist a type in Isabelle/HOL that could encode all other types. Thus, storing abstract types on the heap would lead to unnatural and clumsy restrictions, contradicting the goal of focusing the abstract proofs on algorithmic ideas rather than implementation details. Thus, we opted for not formalizing abstract imperative programs, and directly refine the abstract functional program to a concrete imperative one.

In our approach, the heap will always be described by assertions of the form $R_1\ a_1\ c_1 * \ldots * R_n\ a_n\ c_n$, where the R_i are *refinement assertions*, relating an abstract value a_i to its implementation c_i. Examples for refinement assertions are the primitive $\mapsto_a$ that relates lists to arrays (cf. Section 2.1), the *os_list* assertion for singly linked lists (cf. Example 1), and the *is_uf* assertion for union-find data structures. (cf. Section 3.1)

Note that a refinement assertion needs not necessarily relate heap content to an abstract value. It can also relate a concrete non-heap value to an abstract value. For a relation $R :: (\gamma \times \alpha)\ set$ we define:

$$pure\ R \equiv (\lambda a\ c.\ \uparrow((c,a) \in R))$$

This allows us to mix imperative data structures with functional ones. For example, the refinement assertion *pure nat_rel* describes the implementation of natural numbers by themselves, where $nat\_rel \equiv Id::(nat \times nat)\ set$.

To relate Imperative HOL programs to IRF programs, we define the predicate *hnr* (short for *heap-nres refinement*) as follows:

$$hnr\ \Gamma\ c\ \Gamma'\ R\ m \equiv$$
$$m \neq \textbf{fail} \longrightarrow \langle \Gamma \rangle\ c\ \langle \lambda r.\ \Gamma' * (\exists x.\ R\ x\ r * \uparrow(\textbf{return}\ x \leq m)) \rangle_t$$

Intuitively, $hnr\ \Gamma\ c\ \Gamma'\ R\ m$ states that on a heap described by assertion Γ, the Imperative HOL program c returns a value that refines the nondeterministic result m wrt. the refinement assertion R. Additionally, the new heap contains Γ'.

Example 7 The following refinement assertion refines lists of abstract elements to lists of concrete elements:

fun *list_assn* :: $('a \Rightarrow 'c \Rightarrow assn) \Rightarrow 'a\ list \Rightarrow 'c\ list \Rightarrow assn$ **where**
  $list\_assn\ P\ []\ []\ = emp$
| $list\_assn\ P\ (a\#as)\ (c\#cs)\ = P\ a\ c * list\_assn\ P\ as\ cs$
| $list\_assn\ \_\ \_\ \_\ = false$

Note that lists are implemented by (functional) lists, but the elements of the concrete list may be stored on the heap. The refinement theorem for the *Cons* operation (written as infix # in Isabelle) is:

$$hnr\ (list\_assn\ A\ l\ li * A\ a\ ai)$$
$$(\textbf{return}\ (ai\#li))$$

(*inv* (*list_assn A*) *l li* ∗ *inv A a ai*)
(*list_assn A*)
(**return** (*a*#*l*))

The precondition states that the abstract list l and the abstract element a to be prepended are refined by the concrete list li and the concrete element ai, respectively. The concrete operation $ai\#li$ prepends the concrete element to the concrete list, the abstract operation $a\#l$ prepends the abstract element to the abstract list. Afterwards, both the original list and the element are invalid (ownership has been transferred to the result list), and the result of the abstract operation is refined by the result of the concrete operation.

The *inv* assertion is defined as $inv\ R\ a\ c \equiv \uparrow (\exists h.\ h \models R\ x\ y) * true$. We have $pure\ R \Longrightarrow inv\ R\ a\ c \Longrightarrow_A R\ a\ c$, such that data not stored on the heap can be recovered. For example, a list of natural numbers is not stored on the heap. Thus, the original list remains valid even after prepending a new element to it. (Formally, we have *list_assn* (*pure nat_rel*) = *pure Id*)

In order to prove refinements, we derive a set of proof rules for the *hnr* predicate, including a frame rule, consequence rule, and rules relating the combinators of the heap monad with the combinators of the nondeterminism monad. For example, the consequence rule allows us to strengthen the precondition, weaken the postcondition, and refine the nondeterministic result:

$$[\![\Gamma_1 \Longrightarrow_A \Gamma_1';\ hnr\ \Gamma_1'\ c\ \Gamma_2\ R\ m;\ \Gamma_2 \Longrightarrow_A \Gamma_2';\ m \leq m']\!] \Longrightarrow hnr\ \Gamma_1\ c\ \Gamma_2'\ R\ m'$$

For recursion, we get the following rule:

assumes $\bigwedge f_c\ f_a\ x_a\ x_c.\ [\![$
$\bigwedge x_a\ x_c.\ hnr\ (R_x\ x_a\ x_c * \Gamma)\ (f_c\ x_c)\ (\Gamma'\ x_a\ x_c)\ R_y\ (f_a\ x_a)]\!]$
$\Longrightarrow hnr\ (R_x\ x_a\ x_c * \Gamma)\ (F_c\ f_c\ x_c)\ (\Gamma'\ x_a\ x_c)\ R_y\ (F_a\ f_a\ x_a)$
assumes $\bigwedge x.\ mono\_Heap\ (\lambda f.\ F_c\ f\ x)$
shows $hnr\ (R_x\ x_a\ x_c * \Gamma)\ (heap.fixp\_fun\ F_c\ x_c)\ (\Gamma'\ x_a\ x_c)\ R_y$ (**rec** $F_a\ x_a$)

The rule is specified in Isabelle's *long goal format*, which is more readable for large propositions. Moreover, $\bigwedge x_1 \ldots x_n$ is an Isabelle specific syntax for universal quantification. Intuitively, we have to show that the concrete functor F_c refines the abstract functor F_a, assuming that the concrete recursive function invocation f_c refines the abstract one f_a. The argument of the call is refined wrt. the refinement assertion R_x and the result is refined wrt. R_y. The additional heap before the call is described by Γ, and the additional heap after the call is described by $\Gamma'\ x_a\ x_c$. Here, the x_a and x_c that are attached to Γ' denote that the new heap may also depend on the argument to the recursive function. The second assumption requires the concrete function to be monotonic. This is always the case for functions using only monad combinators, and is discharged automatically by our tool.

4.2 Automation

Using the rules for *hnr*, it would be possible to manually prove refinement between an Imperative HOL program and a program in the Isabelle Refinement Framework, provided they are structurally similar enough.[4] However, this would be a tedious

[4] The control structures must be the same, and the abstract operations must match the corresponding concrete operations.

and quite canonical task, consisting of manually rewriting the program from one monad to the other, thereby unfolding expressions into monad operations if they depend on the heap. For this reason, we focused our work on automating this process: Given some hints which imperative data structures to use, we automatically synthesize an Imperative HOL program and its refinement proof. The Autoref tool [19] solves the analogous problem for purely functional programs, and we could reuse its ideas and even parts of its design for the imperative case.

In the rest of this section we describe our Sepref tool, which automatically synthesizes imperative programs from programs phrased in the Isabelle Refinement Framework. The synthesis consists of several consecutive phases: Identification of operations, monadifying, translation, and cleanup. Note that the implementation of the Sepref tool is not critical to the correctness of the generated theorems. As common for LCF style provers, all theorems are generated by the logical inference kernel. Thus, an error in a tool may lead to useless theorems, or theorems not being generated at all, but never to invalid theorems, provided the kernel is correct.

4.2.1 Identification of Operations

Given an abstract program in Isabelle/HOL, it is not always clear which abstract data types it uses. For example, maps are encoded as functions $\alpha \Rightarrow \beta$ *option*, and so are priority queues or actual functions. However, maps and priority queues are, also abstractly, quite different concepts. The purpose of this phase is to identify the abstract data types (e. g. maps and priority queues), and the operations on them. Technically, the identification is done by rewriting the operations to constants that are specific to the abstract data type. For example, (*f :: nat* $\Rightarrow$ *nat option*) *x* may be rewritten to *op_map_lookup f x*, provided that a heuristics identifies *f* as a map. If *f* is identified as a priority queue, the same expression would be rewritten to *op_get_prio f x*. The operation identification heuristic is already contained in the Autoref tool, and we slightly adapted it for our needs.

4.2.2 Monadifying

Once we have identified the operations, we flatten all expressions, such that each operation gets visible as a top-level computation in the monad. This transformation essentially fixes an evaluation order (which we choose to be left to right), and later allows us to translate the operations to heap-modifying operations in Imperative HOL's heap monad.

Example 8 Consider the program **do** { **let** *x = 1;* **return** {*x,x+1*} }. Note that {*x,y*} is syntactic sugar for (*insert x* (*insert y* {})). A corresponding Imperative HOL program might be:

do { **let** *x = 1; s* $\leftarrow$ *bv_new; s* $\leftarrow$ *bv_ins x s; bv_ins* (*x+1*) *s* }

Note that the *bv_new* and *bv_ins* operations modify the heap. Thus, they have to be applied as monad operations and cannot be nested into other expressions. For this reason, the monadify phase flattens all expressions, and thus exposes all operations as monad operations. It transforms the original program to[5]:

[5] We applied α-conversion to give the newly created variables convenient names.

do { $x \leftarrow$ **return** *1;* $y \leftarrow$ **return** *1;* $z \leftarrow$ **return** x+y;
$s \leftarrow$ **return** {}; $s \leftarrow$ **return** (*insert x s*); **return** (*insert z s*) }

Note that operations that are not translated to heap-modifying operations will be folded again in the cleanup phase.

4.2.3 Translation

Let a be the monadified program. We now synthesize a corresponding Imperative HOL program. Assume the program a depends on the abstract parameters $a_1 \ldots a_n$, which are refined to concrete parameters $c_1 \ldots c_n$ by refinement assertions $R_1 \ldots R_n$. We start with a proof obligation of the form:

$hnr\ (R_1\ a_1\ c_1 * \ldots * R_n\ a_n\ c_n)\ ?c\ ?\Gamma'\ ?R\ a$

Note that ? indicates schematic variables, which may be instantiated during resolution. We will maintain the invariant that the precondition contains an assertion $R_i\ a_i\ c_i$ for every variable a_i that is in scope. If the corresponding concrete value c_i has been destroyed, we set $R_i = inv\ R'_i$, where R'_i is the original assertion for the variable.

We now repeatedly resolve with a set of syntax directed rules for the *hnr* predicate (cf. Section 4.1). Apart from *hnr*-predicates, which trigger recursive resolution, the premises of a rule may contain other side conditions: Frame inference, merge goals, constraints on the refinement assertions, monotonicity goals, and semantic side conditions.

Frame Inference A goal of the form

$\Gamma \Longrightarrow_A\ ?F * R_1\ a_1\ c_1 * \ldots * R_n\ a_n\ c_n$

triggers frame inference: Sepref tries to instantiate $?F$ such that the implication holds. Moreover, it will try to recover invalidated assertions as necessary. While frame inferences have to be explicit premises of combinator rules, Sepref generates them automatically for operator rules.

Merge Goals After translating an **if** or **case** combinator, we have to merge the descriptions of the heaps after the different branches. We use a goal of the form:

$R_1\ a_1\ c_1 * \ldots * R_n\ a_n\ c_n \vee R'_1\ a_1\ c_1 * \ldots * R'_n\ a_n\ c_n \Longrightarrow_A\ ?\Gamma$

The merging is done element-wise by the following rules:

$R\ a\ c \vee R\ a\ c \Longrightarrow_A R\ a\ c$
$inv\ R\ a\ c \vee R\ a\ c \Longrightarrow_A inv\ R\ a\ c$
$R\ a\ c \vee inv\ R\ a\ c \Longrightarrow_A inv\ R\ a\ c$

Note that the first rule also covers the case $inv\ R\ a\ c \vee inv\ R\ a\ c$.

Example 9 The rule for the if combinator is:

assumes *P:* $\Gamma \Longrightarrow_A \Gamma_1 * pure\ bool\_rel\ a\ a'$
assumes *RT:* $a \Longrightarrow hnr\ (\Gamma_1 * pure\ bool\_rel\ a\ a')\ b'\ \Gamma_{2b}\ R\ b$
assumes *RE:* $\neg a \Longrightarrow hnr\ (\Gamma_1 * pure\ bool\_rel\ a\ a')\ c'\ \Gamma_{2c}\ R\ c$
assumes *MERGE:* $\Gamma_{2b} \vee_A \Gamma_{2c} \Longrightarrow_A \Gamma'$
shows $hnr\ \Gamma$ (**if** a' **then** b' **else** c') $\Gamma'\ R$ (**if** a **then** b **else** c)

Intuitively, it works as follows: We start with a heap described by the assertion Γ. First, the refinement for the condition a is extracted from Γ using frame inference. (Premise P) Then, the *then* and *else* branches are translated, (Premises RT and RE) producing new heaps described by the assertions Γ_{2b} and Γ_{2c}, respectively. Finally, these assertions are merged to form the assertion Γ' for the resulting heap after the if statement. (Premise MERGE)

Constraints Another type of side conditions are constraints on the refinement assertions. For example, some data structures require the refinement relation for their elements to be pure. Also, recovery of an invalidated operand is only possible for a pure refinement assertion. However, when the corresponding rules are applied, the refinement assertion may not be completely known, but (parts of) it may be schematic and only instantiated later. We defer constraints over schematic assertions and solve them after the assertions have been instantiated.

Monotonicity Goals Monotonicity goals occur when synthesizing recursion combinators (cf. Section 4.1). They are solved by an automatic procedure which is provided by the Partial Function Package [16].

Semantic Side Conditions Rules may have semantic side conditions, e. g. that an array index is in bounds. The resulting goals are solved by applying Isabelle's *auto* tactic. If solving a side condition fails, the synthesis procedure backtracks over the application of the rule that produced the side condition.

Choosing between Implementations The resolution is directed by the syntax of the abstract program: For each *combinator*, i. e. a function with arguments that describe a monadic computation, there is exactly one rule. However, there may be multiple rules for *operators*, i. e. functions without monadic arguments. They correspond to the implementations of the operator for different data structures. If an operator does not construct a new data structure, the corresponding rule can be uniquely determined by the refinement assertions for the operands. However, if an operator constructs a data structure, multiple rules may apply. In this case, we rely on disambiguation by the user: We define specific constants for each possible implementation as synonyms for the abstract operator, and force the user to rewrite the operator to the constant corresponding to the desired implementation.

Example 10 Lists can be implemented by open singly linked lists (cf. Example 1) or by HOL lists (cf. Example 7). Consider the prepend operation. The *hnr* rules for HOL lists and open singly linked lists are:

hnr (*list_assn A l li* $*$ *A a ai*)
(**return** (*ai*#*li*))
(*inv* (*list_assn A*) *l li* $*$ *inv A a ai*)
(*list_assn A*)
(**return** (*a*#*l*))

hnr (*osll.assn A l li* $*$ *A a ai*)
(*os_prepend ai li*)
(*inv* (*osll.assn A*) *l li* $*$ *inv A a ai*)
(*osll.assn A*)
(**return** (*a*#*l*))

For a proof obligation of the form

hnr Γ *?c* *?*Γ' *?R* (*a*#*l*)

```
dfs_impl Ei si ti ≡ do {
  V ← bv_new;
  (_,r) ← heap_rec (λdfs (V,v). do {
    visited ← bv_memb v V;
    if visited then return (V,False)
    else do {
      V ← bv_ins v V;
      if v = ti then return (V,True)
      else do {
        succ_list ← succi Ei v;
        imp_nfoldli succ_list (λ(_, brk). return (¬ brk))
          (λv (V,_). dfs (V,v)) (V,False)
      }
    }
  }) (V,si);
  return r
}
```

Listing 2: Imperative DFS algorithm generated by SEPREF.

only one of the rules will match, depending on the refinement assertion for l in Γ,

Now consider the empty list operation. The rules are:

hnr emp (**return** []) *emp* (*list_assn A*) (**return** [])
hnr emp os_empty emp (*osll.assn A*) (**return** [])

For a proof obligation of the form

hnr Γ ?c ?Γ′ ?R []

both rules do match, and there is no obvious way to choose a rule. To resolve this ambiguity, we define

op_HOL_list_empty ≡ [] *op_os_empty* ≡ []

and use the rules

hnr emp (**return** []) *emp* (*list_assn A*) (**return** *op_HOL_list_empty*)
hnr emp os_empty emp (*osll.assn A*) (**return** *op_os_empty*)

To choose the implementation, the user has to rewrite the [] to *op_HOL_list_empty* or *op_os_empty* in the *hnr* goal, right before invoking the repeated resolution.

4.2.4 Cleanup

After we have generated the imperative version of the program, we apply some rewriting rules to make it more readable. They undo the flattening of expressions performed in the monadify phase at those places where it was unnecessary, i. e. the heap was not accessed. Technically, this is achieved by using Isabelle/HOL's simplifier with an adequate setup.

Example 11 Recall the DFS algorithm from Example 6. With less than ten lines of straightforward Isabelle text, SEPREF generates[6] the imperative algorithm

[6] Again, we applied α-conversion, to make the generated variable names more readable.

displayed in Listing 2. From this, Imperative HOL generates verified code in its target languages (currently OCaml, SML, Haskell, and Scala). Moreover, Sepref proves the following refinement theorem:

hnr (*is_graph nat_rel E Ei* ∗ *pure nat_rel s si* ∗ *pure nat_rel t ti*)
(*dfs_impl Ei si ti*)
(*is_graph nat_rel E Ei* ∗ *pure nat_rel s si* ∗ *pure nat_rel t ti*)
(*pure bool_rel*)
(*dfs E s t*)

If we combine this with the correctness theorem of the abstract DFS algorithm *dfs*, we immediately get the following theorem, stating total correctness of our imperative algorithm:

corollary *dfs_impl_correct:*
finite (*reachable E s*) ⟹
⟨*is_graph nat_rel E Ei*⟩
dfs_impl Ei s t
⟨λ*r*. *is_graph nat_rel E Ei* ∗ ↑(*r* ⟷ (*s*,*t*) ∈ E^*)⟩$_t$

4.3 Limitations of the Automation

The Autoref [19] tool uses elaborate heuristics to choose implementation data structures that support all the required operations. However, these heuristics are difficult to debug and may silently yield undesired (inefficient) results. Thus, for Sepref, the user has to unambiguously fix all the implementations (cf. Example 10), and is responsible for choosing implementations that support all the required operations. In practice, choosing the implementations is easily done using the *rewrite* tool [37], and debugging of errors due to missing operations is simple.

A previous version of the Sepref tool contained a linearity analysis to automatically synthesize a copy operation when invalidating an operand that is still required. For data not stored on the heap, the copy operation is simply the identity function. For data stored on the heap, the user has to define a custom copy operation. However, setup of the linearity analysis was quite complicated, and the analysis itself was incomplete. Moreover, our experiments indicated that copy operations for impure operands are rarely necessary in practice. Thus, we decided to drop the linearity analysis, and replaced it by automatic recovery of invalidated pure operands. In the rare cases where an impure operand needs to be copied, the user has to manually insert the copy operation.

The most severe limitations of Sepref are due to its simplistic handling of structured data. For example, the operation *hd* :: α *list* ⇒ α returns the first element of a non-empty list. However, the assertion for the resulting element describes a part of the heap that is also described by the assertion for the list itself. Thus, we cannot combine these assertions with a separation conjunction. The only simple solution is to invalidate the original list, preventing further access to all its elements[7]. For this reason, most of our collection data structures require their

[7] A workaround for this particular example is to simultaneously return the head and tail of the list. However, for more complex operations, e. g. returning the first element that satisfies a predicate P, this technique gets unwieldy.

elements to be refined by pure assertions. A somewhat related problem occurs for binary decision diagrams (BDDs), which store functions from vectors of Booleans to Booleans. The representations of the different functions use shared data, such that we cannot define refinement assertions for the single functions stored in the BDD, as would be required for automatic refinement.

We conclude that, despite these limitations, the Sepref tool is powerful enough to verify efficient implementations of substantial algorithms. (cf. Section 6)

5 Imperative Collections Framework

In the last section, we have reported on the translation from programs phrased in the Isabelle Refinement Framework to Imperative HOL programs. We have described the Sepref tool, which performs this translation automatically, refining abstract data types (e. g. sets) to efficient imperative implementations (e. g. hash tables). In order to use this tool in practice, a library of efficient imperative data structures is required. In this section, we briefly report on this library, which we call the *Imperative Collections Framework*. For further reference, including recipes for modular design of more complex data structures, we refer the reader to [22].

5.1 Notation for Refinement

For two n ary functions f and g, we write $(f,g) \in R_1 \times \ldots \times R_n \to R$, if f refines g for argument relations R_i and result relation R, i. e.

$$\forall a_1\ c_1\ \ldots\ a_n\ c_n.\\ (c_1,a_1) \in R_1 \wedge \ldots \wedge (c_n,a_n) \in R_n \Longrightarrow (f\ c_1 \ldots c_n, g\ a_1 \ldots a_n) \in R$$

For an Imperative HOL function f and an IRF function g, we write $(f,g) \in A_1^{x_1} \times \ldots \times A_n^{x_n} \to A$. Here, the A_i are the refinement assertions for the arguments, and A is the refinement assertion for the result. The x_i indicate whether f destroys the argument ($x_i = d$) or not ($x_i = k$), i. e. the above notation is defined as:

$$\forall a_1\ c_1\ \ldots\ a_n\ c_n.\\ hnr\ (A_1\ c_1\ a_1 * \ldots * A_n\ c_n\ a_n)\ (f\ c_1 \ldots c_n)\\ \quad (A_1'\ c_1\ a_1 * \ldots * A_n'\ c_n\ a_n)\ A\ (g\ a_1 \ldots a_n)$$

such that $A_i' = A_i$ if $x_i = k$ and $A_i' = inv\ A_i$ if $x_i = d$.

Example 12 Correctness of the append operation on singly linked lists (cf. Example 1) is specified as follows:

$$(os\_append, append) \in os\_list^d \times os\_list^k \to os\_list$$

where $append :: \alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list$ is the append operation on HOL lists. That is, the append operation destroys the first list, but the second list remains valid.

5.2 Formalizing Imperative Data Structures

We use two complementary approaches to design data structures for Imperative HOL. The first, direct approach is described in Section 3.1: The operations are manually implemented in Imperative HOL, and proved correct using our separation logic tools. Stepwise refinement is possible up to a limited degree, as neither nondeterminism nor assertions are supported. This approach is well-suited to develop simple data structures, and necessary to reason about pointer manipulations, which have no convenient abstract functional model.

The second approach uses the Isabelle Refinement Framework: The data structure and its operations are first described in the nondeterminism monad of the IRF, and then refined to Imperative HOL using the Sepref tool. This approach is well suited to develop complex data structures, which can be described in terms of simpler data structures.

Example 13 For the implementation of priority queues by heapmaps [22] we require a data structure for distinct lists, which supports efficient index query, i. e. to return the position of an element in the list. We may assume that the elements to be stored in the list are natural numbers less than N. Such a data structure can be implemented by a list l of elements, and a list $p = [p_0 \dots p_{N-1}]$, such that p_i is the position of element i in l, or a special value if i is not in l.

In a first step, we phrase the operations in a purely functional fashion, using standard HOL lists for both lists. We define an invariant that asserts well-formedness of the data structure:

locale *invar* = **fixes**
  N :: nat
and *l :: nat list*
and *p :: nat list*
assumes *l_set: set l* $\subseteq$ {*0..<N*}
assumes *l_distinct: distinct l*
assumes *p_len:* $|p| = N$
assumes *p_map:* $\forall x < N.\ p!x =$ (**if** $x \in$ *set l* **then** *List_Index.index l x* **else** *N*)

The assumptions *l_set* and *l_distinct* state that l contains only elements less than N, and no duplicates. The assumptions *p_len* and *p_map* state that p is a list of fixed size N that actually maps elements of l to their indexes. Elements not in l are mapped to the value N (which cannot be a valid index in l, as the invariant implies $|l| \leq N$). Using the invariant, we define the relation R, which relates our abstract data structure to lists:

$R\ N \equiv \{\ ((l,p),l').\ l' = l \wedge invar\ N\ l\ p\ \}$

Then, we define the operations on the abstract data structure, using the Isabelle Refinement Framework. We let each operation assert its precondition, such that the preconditions are available for later refinement.

For example, the operation to retrieve the index of an element which is *contained in the list* is defined as follows:

index $\equiv \lambda(l,p)\ x.$ **do** {
  assert $(x \in$ *set l*$)$;
  $i \leftarrow$ *lst_op_get p x*;

return i }

We first assert the precondition $x \in set\ l$, and then return the x th element of p, which stores the index of x. Note that *lst_op_get* $l\ i$ returns the i th element of a list l, asserting that the index i is in bounds:

lst_op_get $l\ i \equiv$ **do** { **assert** $(i < length\ l)$; **return** $(l!i)$ }

With the verification condition generator of the Isabelle Refinement Framework, it is straightforward to show correctness of the above operation:

(*index, lst_op_index*) $\in R\ N \times nat\_rel \to nat\_rel$

where *lst_op_index* is defined as:

lst_op_index $l\ x \equiv$ **do** { **assert** $(x \in set\ l)$; **spec** $i.\ i < |l| \wedge l!i = x$ }

In a next step, we use the Sepref tool to refine the operations to Imperative HOL. The list l of elements is implemented by an array list of maximum capacity N, i. e. an array of size N and a counter to indicate the actual length of the list. The list p is implemented by an array of length N. The Sepref tool generates a new constant *index_impl*, and the theorem:

(*index_impl,index*) $\in (is\_arl\ N \times (\mapsto_a))^k \times (pure\ nat\_rel)^k \to pure\ nat\_rel$

Here, the assertion *is_arl* N denotes implementation of lists by array lists of maximum capacity N, and $\mapsto_a$ relates an array to the list of its elements.

Finally, combination of the refinement theorems for *index_impl* and *index* yields:

(*index_impl, lst_op_index*) $\in (is\_ial\ N)^k \times (pure\ nat\_rel)^k \to pure\ nat\_rel$

where *is_ial* is the refinement assertion for our *indexed array list* data structure:

is_ial $N\ (l,p)\ l' \equiv \exists \hat{l}\ \hat{p}.\ is\_arl\ N\ \hat{l}\ l * p \mapsto_a \hat{p} * \uparrow(((\hat{l},\hat{p}),l') \in R\ N)$

Note that the Sepref tool provides some automation for the steps sketched above, in particular for combination of refinement theorems. A more detailed description can be found in [22], and in the Sepref user guide [21].

We conclude with an overview of the current data structures in the Imperative Collections Framework, which are listed in Table 1. Note that the amount of supported operations varies from implementation to implementation, and some implementations have been done in an ad hoc manner to exactly fit the needs of a particular algorithm.

6 Case Studies

In this section, we present two case studies: We apply Sepref to a nested depth-first search algorithm and Dijkstra's shortest paths algorithm. Both algorithms have already been formalized within the Isabelle Refinement Framework [32,31,12], and we were able to reuse the existing abstract algorithms and correctness proofs. The resulting Imperative HOL algorithms are considerably faster than the original functional versions. We also briefly report on current projects that use Sepref.

Data Structure	Abstract Type	Remark
Circular Linked List	$\alpha\ list$	
Open Linked List	$\alpha\ list$	cf. Example 1
Array List	$\alpha\ list$	dynamic resize
Maximum Capacity Array List	$\alpha\ list$	
Fixed Size Array	$\alpha\ list$	
Indexed Array List	$nat\ list$	cf. Example 13
Array Map	$nat \to \alpha\ option$	dynamic resize
Hash Map	$\alpha \to \beta\ option$	with rehashing
Array Set	$nat\ set$	dynamic resize
Hash Set	$\alpha\ set$	with rehashing
Union-Find	$(nat \times nat)\ set$	cf. Section 3.1
Heap	$\alpha\ multiset$	cf. [22]
Heap Map	$\alpha \to \beta\ option$	cf. [22]
Square Matrix	$nat \times nat \to \alpha$	by array, row major
Graphs	$(nat \times nat)\ set$	by adjacency list
Edge Weighted Graph	$\alpha\ set \times (\alpha \times \beta \times \alpha)\ set$	by adjacency list
Binary Decision Diagram	$bool\ list \to bool$	limited automation

Table 1: Data Structures Formalized with our Approach

6.1 Nested Depth-First Search

For the CAVA model checker [12], we have verified various nested depth-first search algorithms [35]. Here, we pick a version from the examples that come with the Isabelle Collections Framework [17]. It contains an improvement by Holzmann et al. [13], where the search already stops if the inner DFS finds a path back to a node on the stack of the outer DFS.

From the existing abstract formalization, it takes about 160 lines of mostly straightforward Isabelle text to arrive at the generated SML code and the corresponding correctness theorem, relating the imperative algorithm to its specification.

We compile the generated code with MLton [29] and benchmark it against the original functional refinement and an unverified implementation of the same algorithm in C++, taken from material accompanying [35]. The algorithm is run on state spaces extracted from the BEEM benchmark suite [33]: dining philosophers and Peterson's mutual exclusion algorithm. We have checked for valid properties only, such that the search has to explore the whole state space. The results are displayed in the table below:

Model	Property	#States	Fun	Fun*	Imp	Imp*	C_{O3}	C_{O0}
phils.4	ϕ_1	353668	975	75	70	63	48	66
phils.5		517789	1606	120	113	108	83	112
phils.4	$G(true)$	287578	740	59	53	46	40	54
phils.5		394010	1156	83	77	71	64	85
peterson.3	ϕ_2	58960	119	9	7	5	5	7
peterson.4		1120253	2476	184	142	110	111	158
peterson.3	$G(true)$	29289	55	4	3	2	3	4
peterson.4		576156	1314	88	70	55	54	78

where $\phi_1 = G(one_0 \implies one_0\ W\ eat_0)$ and $\phi_2 = G(wait_0 \implies F(wait_0) \vee G(\neg ncs_0))$

The first column displays the name of the model, the second column the checked property, and the third column displays the number of states. The remaining columns show the time in ms required by the different implementations,

on a 2.2GHz i7 quadcore processor with 8GiB of RAM. Fun denotes a purely functional implementation with red-black trees. Fun* denotes a purely functional implementation, relying on an unverified array implementation similar to Haskell's *Array.Diff*. Imp denotes the verified implementation generated by Sepref, which uses array lists. Imp* denotes a verified implementation generated after hinting Sepref to preinitialize the array lists to the correct size, such that no array reallocation occurs during the search. Finally, the C columns denote the unverified C++ implementation, which uses arrays of fixed size. It was compiled using gcc 4.8.2 with (C_{O3}) and without (C_{O0}) optimizations.

The results are quite encouraging: Our Sepref tool generates code that is more than one order of magnitude faster than the purely functional code. We are also faster than the Fun*-implementation, which depends on an unverified component, and faster than the unoptimized C++ implementation. For the philosopher models, we come close to the optimized C++ implementation, and for the Peterson models, we even catch up.

6.2 Dijkstra's Shortest Paths Algorithm

We have performed a second case study, based on an existing formalization of Dijkstra's shortest paths algorithm [32]. The crucial data types in the existing formalization are a priority queue, a map from nodes to current paths and weights, and the adjacency map of the graph. It took us about 130 lines of straightforward Isabelle text to set up Sepref to produce an imperative version of Dijkstra's algorithm, using arrays for the maps and heap maps for the priority queue. We benchmark our implementation (Imp) against the original functional version (Fun), and a reference implementation in Java (Java), taken from Sedgewick et al. [36]. The inputs are complete graphs with random weights and 1300 and 1500 nodes (cl1300, cl1500), as well as two examples from [36] (medium, large). The required times in ms are displayed in Table 1. The results show a significant speedup wrt. the purely functional version, and our implementation is only a factor 4 to 6 slower than the Java reference implementation.

Test	Fun	Imp	Java
cl1300	240	127	23
cl1500	325	171	29
medium	1	$\ll 1$	2
large	38746	4068	1218

Fig. 1: Dijkstra benchmark

6.3 Other Applications

More recently, we have used the IRF and Sepref to develop a verified implementation [25] of the Edmonds-Karp algorithm for finding maximum flows in networks [11], which is competitive with a Java reference implementation by Sedgewick et al. [36]. We also extended this formalization to push-relabel algorithms.[8] In another project,

[8] Paper under review at the time of writing, the formalization is available at `https://www21.in.tum.de/~lammich/max_flow/`.

we have developed a verified SAT solver certification tool, which is as efficient as the current (unverified) state of the art tool.[9]

7 Conclusion

We have presented an Isabelle/HOL based approach to automatically refine functional programs specified over abstract data types to imperative ones using heap-based data structures. Not only the program, but also the refinement proof is generated, such that we get imperative programs verified in Isabelle/HOL.

The main components of our approach are:

- A separation logic based verification framework for Imperative HOL.
- A refinement calculus from the Isabelle Refinement Framework (IRF) to Imperative HOL, along with the Sepref tool to automatically synthesize Imperative HOL programs from IRF programs.
- The Imperative Collections Framework, which provides a large and extensible library of efficient imperative data structures.

In several case studies we could obtain verified algorithms that were competitive with unverified reference implementations.

7.1 Current and Future Work

Currently we are using Sepref to verify a model checker for timed automata [1]. Another current project is to retarget the Sepref tool to a fragment of the C programming language. Apart from being able to create more efficient implementations and excluding Isabelle's code generator from the trusted code base, we hope to be able to formally reason about the time complexity and resource usage of the generated programs.

An interesting topic for future research is to allow more general imperative container data structures. Currently, the element types of most container data structures must be refined to purely functional data types. (cf. Section 4.3) Charguéraud [7] presents a technique to elegantly encode refinement assertions where the elements are also represented on the heap, owned by their container. His technique could probably be adapted to our formalization. We also hope to be able to support references to elements of data structures at some point, which is required for automatic refinement of Boolean functions using BDDs. (cf. Section 4.3) One step further would be to allow more elaborate ownership models, e. g. elements shared between containers, which are required to model (limited forms of) concurrency. Fractional permissions [3] might be the right tool to achieve this.

7.2 Related Work

We are not aware of interactive theorem prover based tools to automatically refine functional to imperative programs.

[9] Paper under review at the time of writing, the tool and further information is available at `https://www21.in.tum.de/~lammich/grat/`.

Separation logic has been implemented for various interactive theorem provers, e. g. [30, 14, 38, 27]. The work closest to ours is probably the Ynot project [30], which provides a heap monad, a separation logic, and imperative data structures in Coq. Their code generator targets Haskell. However, we are not aware of any performance benchmarks. For Isabelle/HOL, another separation logic framework [14] has been developed independently. In contrast to our framework, it can be instantiated to various heap models. However, it provides less powerful automation. The HOLFoot tool [38] implements a separation logic framework in HOL4. While it provides more powerful automation than our framework, its simplistic imperative language is less convenient for formalizing complex algorithms. In Coq, various imperative OCaml programs and data structures, including Dijkstra's shortest paths algorithm and a union find data structure, have been verified with *characteristic formulas* [6, 8]. Apart from the genuine characteristic formula technique, the main difference to our work is that we use a top-down approach, refining an abstract algorithm down to executable code, while they use a bottom-up approach, starting with a translation of the OCaml code to characteristic formulas. Moreover, they support reasoning about time complexity.

Delaware et al. [9] present the Fiat tool for Coq, which supports synthesis of executable code from abstract specifications. Currently, their approach is limited to specialized abstract specifications (SQL like queries) and purely functional code, but they are planning to extend it to support imperative code.

Acknowledgements We thank Rene Meis for formalizing the basics of separation logic for Imperative HOL. Moreover, we thank Thomas Tuerk for interesting discussions about automation of separation logic. Finally, we thank Max Haslbeck, Simon Wimmer, and the anonymous reviewers for useful comments on draft versions of this paper.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**, 183–235 (1994)
2. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, pp. 259–270. ACM (2005)
4. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: TPHOL, *LNCS*, vol. 5170, pp. 134–149. Springer (2008)
5. Calcagno, C., O'Hearn, P., Yang, H.: Local action and abstract separation logic. In: LICS 2007, pp. 366–378 (2007)
6. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: ICFP, pp. 418–430. ACM (2011)
7. Charguéraud, A.: Higher-order representation predicates in separation logic. In: Proc. of CPP, pp. 3–14. ACM (2016)
8. Charguéraud, A., Pottier, F.: Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation, pp. 137–153. Springer (2015)
9. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: Proc. of POPL, pp. 689–700. ACM (2015)
10. Eberl, M.: Efficient and verified computation of simulation relations on NFAs. Bachelor's thesis, Technische Universität München (2012)
11. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM **19**(2), 248–264 (1972)

12. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: CAV, *LNCS*, vol. 8044, pp. 463–478. Springer (2013)
13. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: SPIN, *Discrete Mathematics and Theoretical Computer Science*, vol. 32, pp. 23–32. American Mathematical Society (1996)
14. Klein, G., Kolanski, R., Boyton, A.: Mechanised separation algebra. In: ITP, pp. 332–337. Springer (2012)
15. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. Journal of Automated Reasoning **44**(4), 303–336 (2009). DOI 10.1007/s10817-009-9157-2. URL `http://dx.doi.org/10.1007/s10817-009-9157-2`
16. Krauss, A.: Recursive definitions of monadic functions. In: Proc. of PAR, vol. 43, pp. 1–13 (2010)
17. Lammich, P.: Collections framework. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Collections.shtml` (2009). Formal proof development
18. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Refine_Monadic.shtml` (2012). Formal proof development
19. Lammich, P.: Automatic data refinement. In: ITP, *LNCS*, vol. 7998, pp. 84–99. Springer (2013)
20. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: ITP, *LNCS*, vol. 8558, pp. 325–340. Springer (2014)
21. Lammich, P.: The imperative refinement framework. Archive of Formal Proofs (2016). `http://isa-afp.org/entries/Refine_Imperative_HOL.shtml`, Formal proof development
22. Lammich, P.: Refinement based verification of imperative data structures. In: CPP, pp. 27–36. ACM (2016)
23. Lammich, P., Meis, R.: A separation logic framework for Imperative HOL. Archive of Formal Proofs (2012). `http://afp.sf.net/entries/Separation_Logic_Imperative_HOL.shtml`, Formal proof development
24. Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms. In: CPP '15, pp. 137–146. ACM, New York, NY, USA (2015)
25. Lammich, P., Sefidgar, S.R.: Formalizing the Edmonds-Karp algorithm. In: Proc. of ITP, pp. 219–234 (2016)
26. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Proc. of ITP, *LNCS*, vol. 7406, pp. 166–182. Springer (2012)
27. Marti, N., Affeldt, R.: A certified verifier for a fragment of separation logic. In: PPL-Workshop (2007)
28. Meis, R.: Integration von Separation Logic in das Imperative HOL-Framework (2011). Master Thesis, WWU Münster
29. MLton Standard ML compiler. URL `http://mlton.org/`
30. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: ICFP (2008)
31. Neumann, R.: A framework for verified depth-first algorithms. In: Workshop on Automated Theory Exploration (ATX 2012), pp. 36–45 (2012)
32. Nordhoff, B., Lammich, P.: Formalization of Dijkstra's algorithm. Archive of Formal Proofs (2012). `http://afp.sf.net/entries/Dijkstra_Shortest_Path.shtml`, Formal proof development
33. Pelánek, R.: Beem: Benchmarks for explicit model checkers. In: Model Checking Software, LNCS, pp. 263–267. Springer (2007)
34. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc of. Logic in Computer Science (LICS), pp. 55–74. IEEE (2002)
35. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: TACAS, *LNCS*, vol. 3440, pp. 174–190. Springer (2005)
36. Sedgewick, R., Wayne, K.: Algorithms. Addison-Wesley (2011). 4th edition
37. Traut, C., Noschinski, L.: Pattern-based subterm selection in Isabelle. In: Proceedings of Isabelle Workshop 2014 (2014)
38. Tuerk, T.: A separation logic framework for HOL. Tech. Rep. UCAM-CL-TR-799, University of Cambridge, Computer Laboratory (2011). URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-799.pdf`
39. Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4) (1971)