Refinement to Imperative/HOL

Peter Lammich

Technische Universität München, lammich@in.tum.de

Abstract. Many algorithms can be implemented most efficiently with imperative data structures that support destructive update. In this paper we present an approach to automatically generate verified imperative implementations from abstract specifications in Isabelle/HOL. It is based on the Isabelle Refinement Framework, for which a lot of abstract algorithms are already formalized.
Based on Imperative/HOL, which allows to generate verified imperative code, we develop a separation logic framework with automation to make it conveniently usable. On top of this, we develop an imperative collection framework, which provides standard implementations for sets and maps like hash tables and array lists. Finally, we define a refinement calculus to refine abstract (functional) algorithms to imperative ones.
Moreover, we have implemented a tool to automate the refinement process, replacing abstract data types by efficient imperative implementations from our collection framework. As a case study, we apply our tool to automatically generate verified imperative implementations of nested depth-first search and Dijkstra's shortest paths algorithm, which are considerably faster than the corresponding functional implementations. The nested DFS implementation is almost as fast as a C++ implementation of the same algorithm.

1 Introduction

Using the Isabelle Refinement Framework (IRF) [13,17], we have verified several graph and automata algorithms (e. g. [23,14]), including a fully verified LTL model checker [6]. The IRF features a stepwise refinement approach, where an abstract algorithm is refined, in possibly many steps, to a concrete implementation. This approach separates the correctness proof of the abstract algorithmic ideas from the correctness proof of their implementation. This reduces the proof complexity, and makes larger developments manageable in the first place.

The IRF only allows refinement to purely *functional* code, while the most efficient implementations of (model checking) algorithms typically require imperative features like destructive update of arrays.

The goal of this paper is to verify *imperative* algorithms using a stepwise refinement approach, and automate the canonical task of replacing abstract by concrete data structures. We build on Imperative/HOL [2], which introduces a heap monad in Isabelle/HOL and supports code generation for several target platforms (currently OCaml, SML, Haskell, and Scala). However, the automation provided by Imperative/HOL is limited, and it has rarely been used for verification

projects so far. Thus, we have developed a separation logic framework that comes with a verification condition generator and some automation that greatly simplifies reasoning about programs in Imperative/HOL.

Based on the separation logic framework, we formalize imperative data structures and integrate them into an imperative collection framework, which, similar to the Isabelle Collection Framework [15], defines interfaces for abstract data types and instantiates them with concrete implementations.

Next, we define a notion of data refinement between an IRF program and an Imperative/HOL program, which supports mixing of imperative and functional data structures, and provide proof rules for the standard combinators (return, bind, recursion, while, foreach, etc.). We implement a tool which automatically synthesizes an imperative program from an abstract functional one, selecting appropriate data structures from both our imperative collection framework and the (functional) Isabelle Collection Framework.

Finally, we present some case studies. We can use already existing abstract formalizations for the IRF unchanged. In particular we use our tool to automatically synthesize an imperative implementation of a nested DFS algorithm from an existing abstract formalization, which is considerably faster than the original purely functional implementation, and almost as fast as a C++ implementation of the same algorithm. Our tool and the case studies are available at `http://www21.in.tum.de/~lammich/refine_imp_hol`.

The remainder of this paper is structured as follows: In Section 2 we present our separation logic framework for Imperative/HOL. In Section 3, we describe the imperative collection framework. The refinement from the IRF to Imperative/HOL and its automation is described Section 4. Section 5 contains the case studies, and, finally, Section 6, contains the conclusions, related work, and an outlook to future research.

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 of this, which did not include frame inference nor other automation, was formalized by Meis [19]. A more recent version is available in the Archive of Formal Proofs [16].

2.1 Basics

We formalize separation logic [25] along the lines of Calcagno et al. [4].

We define a type *pheap* for a *partial heap*, which describes the content of a heap at a specific set of addresses. An *assertion* is a predicate on partial heaps that

satisfies a well-formedness condition[1]. We define the type $assn \subset pheap \Rightarrow bool$ of assertions, and write $h \models P$ if the partial heap h satisfies the assertion P.

We define the basic assertions *true*, *false*, *emp* for the empty heap, $p \mapsto_r v$ for a heap storing value v at address p, and $p \mapsto_a l$ for a heap storing an array[2] with values $l :: \alpha\ list$ at address p. Moreover we define the *pure* assertion $\uparrow \Phi$, which holds if the heap is empty and the predicate Φ holds.

On assertions, we define the standard Boolean connectives, and show that they form a Boolean algebra. We also define universal and existential quantification. Moreover, we define the *separation conjunction* $P * Q$, which holds if the heap can be split into two disjoint parts, such that P holds on the first, and Q on the second part. Finally, we define entailment as $P \Longrightarrow_A Q$ iff $\forall h.\ h \models P \Longrightarrow h \models Q$.

We prove standard properties on assertions and use them to set up Isabelle's proof tools to work seamlessly with assertions. For example, the simplifier pulls existential quantifiers to the front, groups together pure assertions, and checks pointer assertions ($\mapsto_r$ and $\mapsto_a$) for consistency.

Example 1. 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 would have to point to two separate locations).

2.2 Hoare Triples

Having defined assertions, we are ready to define a separation logic on 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 $\alpha\ Heap = heap \Rightarrow (\alpha \times heap)\ option$. 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, program c returns a result x such that the new heap satisfies $Q\ x$.[3] 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 commands (allocation, load/store from/to pointer and array, get array length), rules for the standard combinators (return, bind, recursion, if, case, etc.), 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.

[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.

[2] The distinction between values and arrays is dictated by Imperative/HOL.

[3] 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.

2.3 Automation

Application of these rules can be automated. We implement a verification condition generator that transforms a Hoare triple to verification conditions, which are plain HOL propositions and do not contain separation logic. The basic strategy for verification condition generation is simple: Compute a strong enough postcondition of the precondition and the program, and show that it entails the postcondition of the Hoare triple. In Isabelle/HOL, this is implemented by using a schematic variable as postcondition (i. e. a unification variable that can be instantiated on rule application). However, there are two cases that cannot be fully automated: frame inference and recursion.

Frame Inference When applying a rule for a command c, say $\langle P\rangle\, c\langle Q\rangle$, the current goal has the form $\langle P'\rangle\, c\langle ?R\rangle$, where P' is the precondition describing the current heap, and $?R$ is the unification variable that shall take the postcondition. In order to apply the rule for c, we have to find a part of the heap that satisfies P. In other words, we have to find a *frame* F such that $P' \Longrightarrow_A P * F$. Then, we use the frame rule to prove $\langle P * F\rangle\, c\langle \lambda x.\ Q\ x * F\rangle$, and with the consequence rule we instantiate $?R$ to $\lambda x.\ Q\ x * F$. A detailed discussion about automating frame inference can be found in [28]. We implement a quite simple but effective method: After some initial simplifications to handle quantifiers and pure predicates, we split P and P' into $P = P_1 * \ldots * P_n$ and $P' = P'_1 * \ldots * P'_n$. 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, otherwise the heuristic fails and frame inference has to be performed manually.

Recursion Recursion over the heap monad is modeled as least fixed point over an adequate CCPO [10]. Proving a Hoare triple for a recursive function requires to perform induction over a well-founded ordering that is compatible with the recursion scheme. Coming up with an induction ordering and finding a generalization that is adequate for the induction proof to go through is undecidable in general. We have not attempted to automate this, although there exist some heuristics [3].

2.4 All-in-One Method

Finally, we combine the verification condition generator, frame inference and Isabelle/HOL's *auto* tactic into a single proof tactic *sep_auto*, which is able to solve many subgoals 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 this stuck state usually hints to missing lemmas. The *sep_auto* method allows for very straightforward and convenient proofs. For example, the original Imperative/HOL formalization [2] contains an example of in-place list reversal. The correctness proof requires about 100 lines of quite involved proof text. Using *sep_auto*, the proof reduces to 6 lines of straightforward proof text [16].

3 Imperative Collection Framework

We use our separation logic framework to implement an imperative collection framework along the lines of [15]: For each abstract data type (e. g. set, map) we define a locale that fixes a *refinement assertion* that relates abstract values with concrete values (which may be on the heap). This locale is polymorphic in the concrete data type $'s$, and is later instantiated for every implementation. For each operation, we define a locale that includes the locale of the abstract data type, fixes a parameter for the operation, and assumes a Hoare triple stating the correctness of the operation.

Example 2. The abstract set data type is specified by the following locale:

locale *imp_set* = **fixes** *is_set* :: $'a$ *set* $\Rightarrow$ $'s$ $\Rightarrow$ *assn*
  assumes *precise: precise is_set*

Here, the predicate *precise* describes that the abstract value is uniquely determined by the concrete value.

The insert operation on sets is specified as follows:

locale *imp_set_ins* = *imp_set* + **fixes** *ins* :: $'a$ $\Rightarrow$ $'s$ $\Rightarrow$ $'s$ *Heap*
  assumes *ins_rule:* $\langle$*is_set s p*$\rangle$ *ins a p* $\langle\lambda r.$ *is_set* $(\{a\} \cup s)$ $r\rangle_t$

Note that this specifies a destructive update of the set, as the postcondition does not contain the original set p any more.

Example 3. Finite sets of natural numbers can be implemented by bitvectors. We define a corresponding refinement assertion, and instantiate the set locale:

definition *is_bv* :: *nat set* $\Rightarrow$ *bool array* $\Rightarrow$ *assn* [...]
interpretation *bv: imp_set is_bv* [...]

Then, we define the insert operation, and instantiate the locale *imp_set_ins*:

definition *bv_ins* :: *nat* $\Rightarrow$ *bool array* $\Rightarrow$ *bool array Heap* [...]
interpretation *bv: imp_set_ins is_bv bv_ins* [...]

Using the approach sketched above, we have defined more standard data structures for sets and maps, including hash sets, hash maps and array maps.

4 Refinement to Imperative/HOL

In the last section, we have described how to formalize imperative collection data structures. In order to use these data structures in efficient algorithms, we develop a refinement technique that allows us to refine a formalization of the algorithm over abstract data types to one that uses efficient data structures.

With the Isabelle Refinement Framework [17], we have already developed a formalism to describe and prove correct algorithms on an abstract level and then refine them to use efficient *purely functional* data structures. With the Autoref tool [13], we have even automated this process. In this section, we extend these techniques to imperative data structures.

4.1 Isabelle Refinement Framework

We briefly review the Isabelle Refinement Framework. For a more detailed description, we refer to [17,12]. Programs are described via a nondeterminism monad over the type *'a nres*, which is defined as follows:

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

The type *'a 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** :: (*'a* $\Rightarrow$ *'b nres*) $\Rightarrow$ *'a* $\Rightarrow$ *'b 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. If *F* is not monotonic, **rec** *F* is defined to be **fail**:

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

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.

Functors that only use the standard combinators described above are monotonic by construction. This is also exploited by the Partial Function Package [10], which allows convenient specification of recursive monadic functions.

Building on the combinators described above, the IRF also defines **while** and **foreach** loops to conveniently express tail recursion and folding over the elements of a finite set.

Example 4. 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. [17,13]).

4.2 Connection to Imperative/HOL

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-exception monad

```
definition dfs :: ('v × 'v) set ⇒ 'v ⇒ 'v ⇒ 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

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 formalize a nondeterministic heap-exception monad, in which we encode an abstract program with a heap containing abstract data. In a second step, this program is refined to a deterministic program with concrete data structures. The other option is to omit the intermediate step, and directly relate abstract nondeterministic programs to concrete deterministic 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 as $\mathbb{N}$, 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. This would lead to unnatural and clumsy restrictions on abstract data types, contradicting the goal of focusing the abstract proofs on algorithmic ideas rather than implementation details.

Thus, we opted for directly relating nondeterministic results with heap-modifying programs. We define the predicate *hnr* (short for *heap-nres refinement*) as follows:

$$\begin{aligned}&\mathit{hnr}\ \Gamma\ c\ \Gamma'\ R\ m \equiv\\ &\quad m \neq \mathbf{fail} \longrightarrow \langle\Gamma\rangle\ c\ \langle\lambda r.\ \Gamma' * (\exists x.\ R\ x\ r * \uparrow(\mathbf{return}\ x \leq m))\rangle_t\end{aligned}$$

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

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:

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

For recursion, we get the following rule[4]:

assumes $\bigwedge cf\ af\ ax\ px.\ [\![$
$\bigwedge ax\ px.\ \mathit{hnr}\ (Rx\ ax\ px * \Gamma)\ (cf\ px)\ (\Gamma'\ ax\ px)\ Ry\ (af\ ax)]\!]$
$\Longrightarrow \mathit{hnr}\ (Rx\ ax\ px * \Gamma)\ (Fc\ cf\ px)\ (\Gamma'\ ax\ px)\ Ry\ (Fa\ af\ ax)$
assumes $(\bigwedge x.\ \mathit{mono\_Heap}\ (\lambda f.\ Fc\ f\ x))$
assumes *precise Ry*
shows $\mathit{hnr}\ (Rx\ ax\ px * \Gamma)\ (\mathit{heap.fixp\_fun}\ Fc\ px)\ (\Gamma'\ ax\ px)\ Ry\ (\mathbf{rec}\ Fa\ ax)$

Intuitively, we have to show that the concrete functor *Fc* refines the abstract functor *Fa*, assuming that the concrete recursive function *cf* refines the abstract one *af*. The argument of the call is refined by the refinement assertion *Rx* and the result is refined by *Ry*. Additionally, the heap may contain Γ, and is transformed to $\Gamma'\ ax\ px$. Here, the *ax* and *px* that are attached to Γ' denote that the new heap may also depend on the argument to the recursive function.

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 $R :: ('c \times 'a)\ set$ we define:

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

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

4.3 Automation

Using the rules for *hnr*, it is possible to manually prove refinement between an Imperative/HOL program and a program in the Isabelle Refinement Framework, provided they are structurally similar enough. However, this is usually a tedious and quite canonical task, as it essentially consists 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 the Imperative/HOL program and the refinement proof. The idea is similar to the Autoref tool [13], which automatically synthesizes efficient functional programs, and, indeed, we could reuse parts of its design for our tool.

In the rest of this section, we describe our approach to automatically synthesize imperative programs, focusing on the aspects that are different from the Autoref tool. The process of synthesizing consists of several consecutive phases: Identification of operations, monadifying, linearity analysis, translation, and cleaning up.

[4] Specified in Isabelle's *long goal format*, which is more readable for large propositions.

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 $'a \Rightarrow 'b\ option$, 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 heuristic 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.

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 5. Consider the program **let** *x = 1;* **return** {*x,x*}.[5] Note that {*x,x*} is syntactic sugar for (*insert x* (*insert x* {})). A corresponding Imperative/HOL program might be:

let *x = 1; s* ← *bv_new; s* ← *bv_ins x s; bv_ins x s*

Note that the *bv_new* and *bv_ins* operations modify the heap, and thus have to be applied as monad operations and cannot be nested into a plain HOL expression. For this reason, the monadify phase flattens all expressions, and thus exposes all operations as monad operations. It transforms the above program to[6]:

x ← **return** *1; s* ← **return** {}*; s* ← **return** (*insert x s*)*;* **return** (*insert x s*)

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

Linearity Analysis In order to refine data to be contained on the heap, and destructively updated, we need to know whether the value of an operand may be destroyed. For this purpose, we perform a program analysis on the monadified program, which annotates each operand (which is a reference to a bound variable) to be *linear* or *nonlinear*. A linear operand is not used again, and can safely be destroyed by the operation, whereas a nonlinear operand needs to be preserved.

Example 6. Consider the program from Example 5. Linearity analysis adds the following annotations, where $\cdot^L$ means linear, and $\cdot^N$ means nonlinear:

x ← **return** *1; s* ← **return** {}*; s* ← **return** (*insert* x^N s^L)*;* **return** (*insert* x^L s^L)

That is, the insert operations may be translated to destructively update the set, while at least the first insert operation must preserve the inserted value.

[5] Inserting x twice is redundant, but gives a nice example for our transformations.
[6] We applied α-conversion to give the newly created variables meaningful names.

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

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

Recall that ? indicates schematic variables, which are instantiated during resolution. We now repeatedly try to resolve with a set of syntax directed rules for the *hnr* predicate. There are rules for each combinator and each operator. If a rule would destroy an operand which is annotated as nonlinear, we synthesize code to copy the operand. For this, the user must have defined a copy operation for the operand's concrete data type.

Apart from *hnr*-predicates, the premises of the rules may contain frame inference and constraints on the refinement assertions. Frame inference is solved by a specialized tactic, which assumes that the frame consists of exactly one refinement assertion per variable. The rules are designed to preserve this invariant. If the content of a variable is destroyed, we still include a vacuous refinement assertion *invalid* for it, which is defined as $invalid \equiv \lambda\_\ \_.\ true$.

Apart from standard frame inference goals, which have the form

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

we also have to solve goals of the form

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

These goals occur when merging the different branches of if or case combinators, which may affect different data on the heap. Here, we keep refinement assertions with $R_i = R'_i$, and set the others to *invalid*.

Example 7. 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 concrete value for the condition a is extracted by a frame rule (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).

Another type of side conditions are constraints on the refinement assertions. For example, some rules require a refinement assertion to be precise (cf. Section 3). When those rules are applied, the corresponding refinement assertion may not be completely known, but (parts of) it may be schematic and only instantiated

later. For this purpose, we keep track of all constraints during translation, and solve them as soon as the refinement assertion gets instantiated.

Using the resolution with rules for *hnr*, combined with frame inference and solving of constraints, we can automatically synthesize an Imperative/HOL program for a given abstract program. While there is only one rule for each combinator, there may be multiple rules for operators on abstract data types, corresponding to the different implementations. In the Autoref tool [13], we have defined some elaborate heuristic how to select the implementations. In our prototype implementation here we use a very simplistic strategy: Take the first implementation that matches the operation. By specifying the refinement assertions for the parameters of the algorithm, and declaring the implementations with specialized abstract types, this simplistic strategy already allows some control over the synthesized algorithm. In future work, we may adopt some more elaborate strategies for implementation selection.

Sometimes, functional data structures are more adequate than imperative ones, be it because they are accessed in a nonlinear fashion, or because we simply have no imperative implementation yet. Pure refinement assertions allow for mixing of imperative and functional data structures, and our tool can import rules from the Isabelle Collection Framework, thus making a large amount of functional data structures readily available.

Cleaning Up 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 is not modified. Technically, this is achieved by using Isabelle/HOL's simplifier with an adequate setup.

Example 8. Recall the DFS algorithm from Example 4. With a few (<10) lines of straightforward Isabelle text, our tool generates[7] the imperative algorithm displayed in Listing 2. From this, Imperative/HOL generates verified code in its target languages (currently OCaml, SML, Haskell, and Scala). Moreover, our tool 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*)
(*pure nat_rel t ti* ∗ *invalid s si* ∗ *is_graph nat_rel E Ei*)
(*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*⟩

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

```
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 our tool.

$$\langle \lambda r.\ \mathit{is\_graph}\ \mathit{nat\_rel}\ E\ \mathit{Ei} * \uparrow(r \longleftrightarrow (s,t)\in E^*\ )\rangle_t$$

5 Case Studies

In this section, we present two case studies: We apply our method to a nested depth-first search algorithm and Dijkstra's shortest paths algorithm. Both algorithms have already been formalized within the Isabelle Refinement Framework [23,22,6], and we were able to reuse the existing abstract algorithms and correctness proofs unchanged. The resulting Imperative/HOL algorithms are considerably more efficient than the original functional versions.

5.1 Nested Depth-First Search

For the CAVA model checker [6], we have verified various nested depth-first search algorithms [26]. Here, we pick a version from the examples that come with the Isabelle Collection Framework [11]. It contains an improvement by Holzmann et al. [8], 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. The main part of the required Isabelle text consists of declaring parametricity rules for specific algebraic data types defined by the abstract formalization, and could be automated.

We compile the generated code with MLton [20] and benchmark it against the original functional refinement and an unverified implementation of the same algorithm in C++, taken from material accompanying [26]. The algorithm is run on state spaces extracted from the BEEM benchmark suite [24]: 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 $\phi_1 = G(one_0 \Longrightarrow one_0\ W\ eat_0)$ and $\phi_2 = G(wait_0 \Longrightarrow F(wait_0) \vee G(\neg ncs_0))$

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 our tool, which uses array lists. Imp* denotes a verified implementation generated after hinting our tool to preinitialize the array lists to the correct size (which required 5 extra lines of Isabelle text), 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 (O3) and without (O0) optimizations.

The results are quite encouraging: Our 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.

5.2 Dijkstra's Shortest Paths Algorithm

We have performed a second case study, based on an existing formalization of Dijkstra's shortest paths algorithm [23]. The crucial data types in the existing formalization are a priority queue, a map from nodes to current paths and weights, and a map from nodes to outgoing edges that represents the

graph. It took us about 130 lines of straightforward Isabelle text to set up our tool to produce an imperative version of Dijkstra's algorithm, using arrays for the maps. Currently, we do not have an imperative priority queue data structure, so we reused the existing functional one which is based on finger trees [7], demonstrating the capability of our tool to mix imperative and functional data structures. We benchmark our implementation (Imp) against the original functional version (Fun), and a reference implementation in Java (Java), taken from Sedgewick et al. [27]. The inputs are complete graphs with random weights and 1300 and 1500 nodes (cl1300, cl1500), as well as two examples from [27] (medium, large). The required times in ms are displayed in Table 1:

Name	Fun	Imp	Java
cl1300	278	167	28
cl1500	378	219	29
medium	2	2	3
large	45606	28861	1490

Fig. 1: Dijkstra benchmark

The results show that a significant speedup (factor 1.5–2) can be gained by replacing only some of the functional data structures by imperative ones. However, we are still one order of magnitude slower than the reference implementation in Java. Our profiling results indicate that most of the time in the Imperative/HOL implementation is spent to manage the finger tree-based priority queue, and we are currently formalizing an array-based min-heap — the same data structure as used in the Java implementation.

6 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.

Our approach is based on the Isabelle Refinement Framework, for which many formalized algorithms already exist. These can now be refined to imperative implementations, without redoing their correctness proofs.

We have implemented a prototype tool, which we applied to generate a verified nested DFS algorithm, which is almost as efficient as an unverified implementation of the same algorithm in C++. Moreover, our approach can mix refinements to functional and imperative data structures, which we demonstrated by a refinement of Dijkstra's algorithm, where the priority queue is functional, but the graph representation and some maps are imperative. We gained a speedup of factor 1.5–2 wrt. the purely functional version, but are still an order of magnitude slower than an unverified implementation in Java.

Apart from extending the imperative collection framework by more data structures, future work includes improving the automation. Another interesting direction is to allow sharing of read-only data on the heap, which also would allow refinement of nested abstract data types, e. g. sets of sets to arrays of pointers to arrays. Fractional permissions [1] may be the right tool to achieve this.

6.1 Related Work

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

Separation logic has been implemented for various interactive theorem provers, e. g. [21,9,28,18]. The work closest to ours is probably the Ynot project [21]. They formalize 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, there is a second separation logic framework [9], which has been developed independently of ours. It can be instantiated to various heap models, while ours is specialized to Imperative/HOL. However, the provided automation is less powerful than ours.

The HOLFoot tool [28] 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, including Dijkstra's shortest paths algorithm, have been verified with *characteristic formulas* [5]. 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.

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.

References

1. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.
2. L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *TPHOL*, volume 5170 of *LNCS*, pages 134–149. Springer, 2008.
3. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL '09*, pages 289–300, 2009.
4. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS 2007*, pages 366–378, July 2007.
5. A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*, pages 418–430. ACM, 2011.
6. J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. In *CAV*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.
7. R. Hinze and R. Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.
8. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *SPIN*, volume 32 of *Discrete Mathematics and Theoretical Computer Science*, pages 23–32. American Mathematical Society, 1996.

9. G. Klein, R. Kolanski, and A. Boyton. Mechanised separation algebra. In *ITP*, pages 332–337. Springer, Aug 2012.
10. A. Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43, pages 1–13, 2010.
11. P. Lammich. Collections framework. In *Archive of Formal Proofs*. `http://afp.sf.net/entries/Collections.shtml`, Dec. 2009. Formal proof development.
12. P. Lammich. Refinement for monadic programs. In *Archive of Formal Proofs*. `http://afp.sf.net/entries/Refine_Monadic.shtml`, 2012. Formal proof development.
13. P. Lammich. Automatic data refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.
14. P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *ITP*, volume 8558 of *LNCS*, pages 325–340. Springer, 2014.
15. P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In *Proc. of ITP*, volume 6172 of *LNCS*, pages 339–354. Springer, 2010.
16. P. Lammich and R. Meis. A separation logic framework for imperative hol. *Archive of Formal Proofs*, Nov. 2012. `http://afp.sf.net/entries/Separation_Logic_Imperative_HOL.shtml`, Formal proof development.
17. P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.
18. N. Marti and R. Affeldt. A certified verifier for a fragment of separation logic. In *PPL-Workshop*, 2007.
19. R. Meis. Integration von Separation Logic in das Imperative HOL-Framework, 2011. Master Thesis, WWU Münster.
20. MLton Standard ML compiler. http://mlton.org/.
21. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP*, 2008.
22. R. Neumann. A framework for verified depth-first algorithms. In *Workshop on Automated Theory Exploration (ATX 2012)*, pages 36–45, 2012.
23. B. Nordhoff and P. Lammich. Formalization of Dijkstra's algorithm. *Archive of Formal Proofs*, Jan. 2012. `http://afp.sf.net/entries/Dijkstra_Shortest_Path.shtml`, Formal proof development.
24. R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Model Checking Software*, LNCS, pages 263–267. Springer, 2007.
25. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc of. Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.
26. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *TACAS*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.
27. R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 2011. 4th edition.
28. T. Tuerk. A separation logic framework for HOL. Technical Report UCAM-CL-TR-799, University of Cambridge, Computer Laboratory, June 2011.