Refinement Based Verification of Imperative Data Structures

Peter Lammich
Technische Universität München, Germany
lammich@in.tum.de

Abstract

In this paper we present a stepwise refinement based top-down approach to verified imperative data structures. Our approach is modular in the sense that already verified data structures can be used for construction of more complex data structures. Moreover, our data structures can be used as building blocks for the verification of algorithms. Our tool chain supports refinement down to executable code in various programming languages, and is fully implemented in Isabelle/HOL, such that its trusted code base is only the inference kernel and the code generator of Isabelle/HOL.

As a case study, we verify an indexed heap data structure, and use it to generate an efficient verified implementation of Dijkstra's algorithm.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification

Keywords Data Structures, Interactive Theorem Proving, Isabelle/HOL, Stepwise Refinement, Separation Logic

1. Introduction

When verifying algorithms that should also come with verified efficient implementations, it is essential to have a library of reusable standard data structures, which can be used as building blocks for the efficient implementation.

The Isabelle Collection Framework [14] provides such a library for purely functional data structures. However, the most efficient implementations of many algorithms require imperative data structures. With the Sepref tool [13], we have recently established an infrastructure for the verification of imperative algorithms. In this paper, we describe our approach to the development of verified, efficient imperative data structures, building on and extending the Sepref tool.

Based on the Isabelle Refinement Framework [16], we use a stepwise refinement approach to separately prove the algorithmic ideas of the data structures, and their low-level implementations. This separation of concerns greatly simplifies the proofs. Moreover, it allows for reuse of already verified basic data structures in the context of more complex data structures.

Our approach is based on Isabelle/HOL, whose LCF architecture guarantees that every proof must go through a trustworthy and relatively small logical inference kernel. In particular, implementation bugs in our tool cannot compromise the validity of a proved correctness theorem, but only the ability to prove the theorem.

We illustrate our approach with the development of an indexed heap (*heapmap*) data structure, which is then used in a verified version of Dijkstra's shortest paths algorithm [6]. We build on an existing verification of a functional version of Dijkstra's algorithm [19]. Exploiting our tools, replacement of the original purely functional priority queue by the imperative heapmap implementation is straightforward, and, thanks to the refinement based approach, the existing formalization can be reused without modification or duplication.

1.1 Contributions

This paper describes a stepwise refinement approach to the verification of imperative data structures. Although presented for Isabelle/HOL, our methods should also be usable in other refinement based program verification settings.

In previous research, we have already explored refinement approaches for algorithms [12, 13, 16], and we also have explored verification of purely functional data structures [14, 17, 18]. However, the verification of complex imperative data structures poses some new challenges:

- Complex imperative data structures are often composed from simpler building blocks, and the complex data structure can be proved correct over an abstract view on its building blocks. Then, refinement is used to implement the abstract building blocks, and thus obtain an implementation of the complex data structure.

  For example, in our heapmap data structure, we have a list of keys, which is implemented by an array and a lookup table from keys to array indexes. The lookup table, in turn, is implemented by an array.

  This paper describes how to systematically handle such implementation hierarchies, exploiting the automation infrastructure provided by our Sepref tool.

- Imperative data structures are often implemented with fixed capacities, e. g. our heap is implemented as a (fixed size) array, whose capacity is specified on allocation time. This poses additional proof challenges when refining from an abstract version of the data structure which has no such limitation. This paper explores techniques to semi-automatically do such refinements in certain cases, exploiting parametricity arguments.

  For example, Dijkstra's algorithm uses a priority queue to store nodes of a graph, represented by numbers from $\{0..<N\}$. If we know that the graph operations only return nodes less than N, and that the algorithm does not fiddle with the node representation (e. g. incrementing nodes), we can conclude that pushing nodes to the priority queue is always within the capacity limit N.

- Data structures usually have element types. If the data structure is used in an algorithm, one usually wants to refine both the representation of the data structure and the representation of the element type. However, for proving correctness of an implementation, it is much simpler to assume that the element types are not refined. This paper shows how to exploit parametricity arguments to get from an implementation that does not support element refinement to one that does. While we have already used such techniques for a purely functional setting in our Autoref tool [12], we transfer them to an imperative setting, and add some automation which simplifies their application.

To enable the techniques described above, we had to slightly extend the Sepref tool itself. Apart from extensions that simplify usability of the tool, this paper describes the FCOMP tool, which puts the different notions of refinement from our previous work (functional to functional, functional to imperative) into a unified framework and supports composition of refinement relations.

Finally, we have applied our techniques in a case study to verify the heap and heapmap data structures, and use them in a verified version of Dijkstra's shortest paths algorithm [6]. While heaps and heapsort algorithms are a standard verification benchmark, we believe to be the first to verify the considerably more complex heapmap data structure. The abstract version of Dijkstra's algorithm stems from earlier work of the author [19], where a purely functional priority queue data structure was used.

The rest of this paper is organized as follows: In Section 2, we briefly introduce some notational conventions, the Isabelle Refinement Framework, and the basics of refinement to imperative programs. Section 3 describes our methods for the implementation of imperative data structures, and illustrates them by simple examples. Section 4 presents our case study, the implementation of heapmaps and Dijkstra's algorithm. Finally, related work and conclusions are presented in Section 5.

The Isabelle source code of this project is available at `http://www21.in.tum.de/~lammich/heapmaps`, and we also plan to submit it to the Archive of Formal Proofs. (`http://afp.sf.net`)

2. Prerequisites

In this section we first fix some notational conventions. Then, we briefly introduce the theory behind the Isabelle Refinement Framework and the Sepref tool. For a more complete description, we refer the reader to [12, 13, 16].

2.1 General Notations

We use some notations derived from Isabelle/HOL syntax. Function arguments are separated by whitespace, i. e. we write $f\ a\ b$ to apply f to arguments a and b. Moreover, we identify curried and uncurried functions, and functions to Boolean values and sets. For example, we may write the proposition that a and b are related by relation R as $R\ a\ b$ or $(a,b)\in R$.

Type arguments are applied in prefix notation, e. g. *int list* for a list of integers. Type variables are prefixed by a tick, e. g. $'a$. Datatypes are defined by their constructors, separated by $|$. For example, lists are defined as:

$$'a\ list \equiv [] \mid 'a\ \#\ ('a\ list)$$

where *x#xs* is infix notation for *Cons x xs*. Constants are defined using $\equiv$, where we separate pattern matches by $|$. For example, concatenation of two lists is defined as:

$$[]@ys \equiv ys \mid (x\#xs)@ys \equiv x\#(xs@ys)$$

where *xs@ys* is infix notation for *append xs ys*. To explicitly declare the type of a constant, we use *::*, e. g. *Cons* $:: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$.

For a monad, we use a semicolon for bind. If the bound value is not used, we may omit it. I. e. we define the notation:

$$x \leftarrow m; f\ x \equiv \textbf{bind}\ m\ f$$
$$m; f \equiv \textbf{bind}\ m\ (\lambda\_.\ f)$$

Note that we use underscores (_) as wildcards.

2.2 The Isabelle Refinement Framework

The Isabelle Refinement Framework [12, 16] is a set of theories and tools that support the development of verified programs inside Isabelle/HOL, featuring a stepwise refinement approach [2, 25]. Recently, we have extended the framework to support refinement to imperative programs [13].

A program in the Isabelle Refinement Framework is described as a function which returns a result of type $'a\ nres \equiv \textbf{res}\ 'a\ set \mid \textbf{fail}$. Intuitively, **res** X describes that the program may nondeterministically return any value from the set X, and **fail** means that an assertion in the program fails. We lift the subset ordering to *nres*, with **fail** being the greatest element: $\textbf{res}\ X \le \textbf{res}\ Y \equiv X \subseteq Y \mid \_ \le \textbf{fail}$. Note that this ordering forms a complete lattice over *nres*. We define a set-exception monad over *nres* as follows:

$$\textbf{return}\ x \equiv \textbf{res}\ \{x\}$$
$$\textbf{bind}\ (\textbf{res}\ X)\ f \equiv \mathit{Sup}\ x{\in}X.\ f\ x \mid \textbf{bind fail}\ \_ \equiv \textbf{fail}$$

Moreover, we define recursion by a fixed point construction:

$$\textbf{rec}\ F\ x \equiv \textbf{assert}\ (\mathit{mono}\ F);\ \mathit{gfp}\ F\ x$$

where $\textbf{assert}\ \Phi \equiv \textbf{if}\ \Phi\ \textbf{then return}\ ()\ \textbf{else fail}$.

This gives us a shallowly embedded nondeterministic programming language in Isabelle/HOL, in which we can elegantly express abstract algorithms and their specifications. We define the shortcut $\textbf{spec}\ x.\ \Phi\ x \equiv \textbf{res}\ \{x.\ \Phi\ x\}$ for all values that satisfy the predicate Φ. Then, correctness of a program f wrt. precondition pre and postcondition $post$ is specified as

$$\mathit{pre\ args} \Longrightarrow f\ \mathit{args} \le \textbf{spec}\ r.\ \mathit{post\ args\ r}$$

i. e. if the precondition holds for the arguments, the possible results satisfy the postcondition. (which also depends on the arguments.)

Sometimes we want to prove additional properties about a program of that we already know that it does not fail. We define $m \le_n m' \equiv m{\neq}\textbf{fail} \Longrightarrow m \le m'$. When proving refinement wrt. $\le_n$, assertions on the left hand side may be assumed, i. e. we have the rule:

$$[\![\ \Phi \Longrightarrow m \le_n m'\ ]\!] \Longrightarrow \textbf{assert}\ \Phi;\ m \le_n m'$$

Data refinement is specified by a *refinement relation* between *concrete* and *abstract* values. The function $\Downarrow$, which is defined as

$$\Downarrow R\ (\textbf{res}\ X) \equiv \textbf{res}\ (R^{-1}\ X) \mid \Downarrow R\ \textbf{fail} \equiv \textbf{fail}$$

returns the greatest concrete result which refines the given abstract result wrt. the refinement relation R. Thus, the proposition $(a_i,a) \in R \Longrightarrow f_i\ a_i \le \Downarrow S\ (f\ a)$ states that f_i *refines* f wrt. relation R for the arguments and S for the result. For convenience, we include an additional precondition for the abstract arguments and define the notation:

$$(f_i,f)\text{: } [\lambda a.\ \mathit{pre}\ a]\ R \to S \equiv \forall (a_i,a){\in}R.\ \mathit{pre}\ a \Longrightarrow f_i\ a_i \le \Downarrow S\ (f\ a)$$

If the precondition is always true, we just write $(f_i,f)\text{: } R \to S$. Moreover, we identify curried and uncurried functions, and write $(f_i,f)\text{: } R_1 \to \ldots \to R_n \to S$ for functions with n arguments that are refined by $R_1 \ldots R_n$.

2.3 Imperative Programs

We represent imperative programs using the heap monad of Imperative/HOL [3]. Its type *'a Heap* represents a deterministic operation that modifies the heap and returns a result of type *'a*.

To reason about heap-modifying programs we use separation logic [4, 22], which we have formalized on top of Imperative/HOL [13, 15]. In order to relate data structures on the heap with plain values, we use refinement relations of type $'c \Rightarrow 'a \Rightarrow assn$, where *assn* is the type of separation logic assertions. For example, the assertion $a \mapsto_a l$ states that a points to an array that contains the values in list l, thus relating the array and the list.

Refinement between a program c in the deterministic heap monad and a program a in the nondeterministic nres monad wrt. refinement relation S is expressed by the Hoare triple:

$$\{\}\ c\ \{r_i.\ \exists r.\ S\ r_i\ r * \textbf{return}\ r \le a\ \}$$

I. e. the result r_i, which may contain pointers to the heap, must be related to a plain value r, which is among the possible results of the abstract program. We extend this to arguments, and define the following notation:

$$\begin{array}{l}(f_i,f):\ [\lambda a.\ pre\ a]\ (R,R') \to S\ \equiv\ \forall a_i\ a.\\ \ \{pre\ a * R\ a_i\ a\}\\ \ \ f_i\ a_i\\ \ \{r_i.\ R'\ a_i\ a * \exists r.\ S\ r_i\ r * \textbf{return}\ r \le f\ a\}\end{array}$$

As the function may have side effects on the heap, we use R to describe the refinement of the arguments *before* f_i is executed, and R' for the state *after* execution of f_i. Again, we may omit vacuous preconditions, and write $(f_i,f):\ (R_1,R_1') \to \ldots \to (R_n,R_n') \to S$ for multiple arguments. Note that, for a single argument, we usually have either $R_i' = R_i$ (the function does not change the argument) or $R_i' = \lambda\_\ \_.\ true$ (the function invalidates the argument). Thus, we define the notations $R^k \equiv (R,R)$ and $R^d \equiv (R,\lambda\_\ \_.\ true)$ to express that an argument is kept or destroyed.

Consider, for example, array update and list update:

$$\begin{array}{l}a\_upd :: 'a\ array \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ array\ Heap\\ l\_upd :: 'a\ list \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ list\ nres\end{array}$$

They satisfy the following refinement theorem:

$$(a\_upd,l\_upd):\ [\lambda l\ i\ v.\ i<length\ l]\ (\mapsto_a)^d \to Id^k \to Id^k \to (\mapsto_a)$$

Note that we define $Id\ x\ y \equiv x{=}y$, i. e. the values x and y are not compared wrt. the heap content. We call relations that do not depend on the heap *pure*, and identify them with the refinement relations for values that we used in Section 2.2.

2.4 The Sepref Tool

The Sepref tool [13] automates the refinement from a functional program in the nres monad to an imperative program in the heap monad. Given a program in the nres monad, it automatically generates a program in the heap monad and proves a corresponding refinement theorem. Thereby, it replaces abstract data types by efficient imperative data structures. A linearity analysis of the input program identifies opportunities for destructive update, and an operation identification heuristics tries to identify abstract data types and operations on them in the input program.

2.5 Composition and Nested Refinement

We define $(R_1\ O\ R_2)\ a\ c \equiv \exists b.\ R_1\ a\ b * R_2\ b\ c$ to be the composition of refinement relations R_1 and R_2.

When we refine data structures, they often have *element types*, e. g. a set has an element type, and a map has types for keys and values. Typically, we also want to refine the element types[1]. For this, we parameterize the refinement relations with relations for the element types.

A special case are refinement relations that preserve the structure of the outer type, and only refine the element types. For example, the relation $\langle R\rangle list\_rel$ relates two lists of the same length, whose elements at corresponding indices are related by R:

$$\langle R\rangle list\_rel\ \equiv\ \{([c_1,\ldots,c_n],[a_1,\ldots,a_n]) \mid \forall i.\ (c_i,a_i) \in R\ \}$$

Note that we use, similar to type arguments, a prefix notation for the parameter relations.

An elegant way to construct refinement relations that support refinement of elements is by composing a refinement relation for the structure of the data type with a structure preserving one for the element types. For example, $\mapsto_a\ O\ \langle R\rangle list\_rel$ is a refinement between arrays and lists, such that the elements of the list are refined by R. Operations which are related to themselves wrt. a structure preserving refinement relation are called *parametric* [21]. For example, we have:

$$\begin{array}{l}(l\_upd,l\_upd):\\ \ [\lambda l\ i\ v.\ l<length\ l]\ \langle R\rangle list\_rel \to Id \to R \to \langle R\rangle list\_rel\end{array}$$

For parametric operations, it is straightforward to extend an implementation that does not support refinement of the element types to one with refinement for the element types (cf. Section 3.4).

3. Implementing Data Structures

In the last section, we gave a brief overview of the Isabelle Refinement Framework. In this section, we describe our techniques to implement efficient and verified imperative data structures, which can be used as building blocks to construct efficient verified algorithms.

3.1 Interfaces

Intuitively, an interface describes the operations on an abstract data type. Operations may come with preconditions, and may be nondeterministic.

Formally, an interface consists of a data type T, and a collection of operations related to T. The operations are specified in the nres monad, such that preconditions can be expressed via assertions. A typical operation specification has the form *f args* = **assert** *pre args;* **spec** *r. post args r*, where *pre* is the precondition, and *post* is the postcondition of the operation.

Example 3.1. The list interface has type $T = \alpha\ list$. The deterministic operation $nil\_op \equiv$ **return** [] returns the empty list; the deterministic operation $hd\_op\ l \equiv$ **assert** $l{\neq}[]$**;** **return** $(hd\ l)$ returns the first element of a non-empty list; and the nondeterministic operation

$$idx\_of\_op\ l\ x\ \equiv\ \textbf{assert}\ x{\in}set\ l;\ \textbf{spec}\ i.\ i<length\ l \wedge l!i = x$$

returns an index of element x, which must be in the list at least once.

Apart from the data type and operations, an interface contains some setup for the Sepref tool. This ensures that Sepref will correctly recognize operations of this interface, and translate standard Isabelle functions to operations of the interface, based on its operation identification heuristics. For example, also [] may be recognized as empty list operation. Similarly, *hd* may be recognized as operation to return the first element of a list. However, this requires a proof that the list is not empty, which only succeeds if the program contains enough information for an automatic proof, e. g. via assertions or conditionals.

In general, when implementing abstract algorithms, there is a trade off between using abstract operations that come with preconditions, and plain Isabelle functions. The former choice ensures that the preconditions are readily available on refinement, while, for

[1] Currently, our Sepref tool only supports pure refinements for element types.

the latter option, enough assertions to get refinement through have to be added manually. On the other hand, the abstract operations are monad operations, and thus cannot be composed to complex expressions as concisely as plain functions.

Example 3.2. For example, the program **return** *hd* (*hd l*) returns the second element of the list l. In order to refine this, one has to add an assertion, i. e. **assert** *length l >= 2;* **return** *hd* (*hd l*). If one forgets this assertion, the program can still be proved correct, but refinement will fail. On the other hand, using monad operations, the same program reads as *t* ← *hd_op l; hd_op t*, i. e. one has to make explicit the intermediate result, which can be cumbersome for more complex expressions. However, refinement of this program always succeeds, as *hd_op* already contains the assertion of the precondition.

3.2 Implementations

An implementation of an interface with type T provides a concrete data type T', and a refinement relation $R : T' \to T \to assn$. Moreover, an implementation defines concrete operations on T' for the abstract operations on T, and proves that they are related via R. Optionally, R may be parameterized with refinement relations for inner types.

An implementation may restrict its interface in several ways:

- It needs not implement all operations. This allows for specialized implementations that focus on certain operations, without the formal requirement of defining a new interface for them.
- It may restrict the representable abstract data. For example, a set implementation may be restricted to finite sets.
- It may restrict the element types of the abstract data type. For example, a list implementation may be restricted to lists of natural numbers.
- It may add further preconditions to operations. For example, an insert operation may add the precondition that the size of the set is below a maximum size.

As for interfaces, an implementation comes with a setup of the Sepref tool, such that the implementation can be used by automatic synthesis. The setup declares the refinement theorems for the operations to Sepref. Moreover, it defines custom versions of the abstract constructor operations, i. e. those that return a type containing T but where the argument types do not contain T. These custom constructor operations, which are specific to the implementation, are used to control the implementation selection during synthesis: The user has to rewrite the constructor operations in the abstract program to the specific constructors for the desired implementation[2]. Alternatively, there are default implementations for certain interfaces, which are associated with the original constructor operations.

Example 3.3. A list of fixed length can be implemented by an array. The refinement relation is $\mapsto_a$. In this implementation, we allow no refinement of the element type. Moreover, we only implement operations which do not change the size of the list, e. g. indexing and length. Note that operations like *hd* can be implemented based on indexing.

For example, the refinement lemma for indexing into a list is

$$(\textit{Array.nth,lst\_op\_get}) : (\mapsto_a)^k \to \textit{nat\_rel}^k \to \textit{Id}$$

As we do not refine the element type, the refinement relation for the result is the identity relation *Id*.

The implemented constructor operations are the empty list [], and *replicate n v*, which returns a list of n vs. We define the specific synonyms *array_empty* and *array_replicate* for these operations, and register the refinement lemmas with these synonyms. Moreover, we also define the lemma *hnr_array_dflt* to be the refinement lemmas for the original operations. Now we can, locally, declare *hnr_array_dflt* to the Sepref tool to use arrays as default list implementation.

3.2.1 Fixed Capacity Implementations

A notable special class of implementations are those where an initial capacity is specified on construction, and the data structure can only grow up to this capacity. In imperative programs, such data structures are quite common, as they require no dynamic resizing on overflow.

To map those data structures to our approach, we have to specify the capacity as a parameter of the refinement relation. Moreover, operations that grow the data structure get the additional precondition that the data structure is not yet full.

Making the refinement relation dependent on an argument passed to the constructor has, however, a caveat: If an operation computes an initial size, and then returns the data structure with this initial size, its refinement relation may depend on an arbitrary complex function in the operation's arguments.

Currently, Sepref is quite limited in what it can synthesize: The maximum size must be a constant or an argument of the function. Otherwise, we have to do the refinement manually. Note that Sepref can *use* dependent refinement theorems during a synthesis, it only cannot synthesize them itself. Thus, this restriction did not complicate the refinements in our case study (cf. Section 4.3), where we use data structures with fixed capacity. We leave it to future work to explore ways to automatically synthesize dependent refinement relations.

Example 3.4. A list can be implemented by a length counter n and an array a, such that the list is formed by the initial n elements of the array. If we do not want to dynamically reallocate the array on over/underflow, the list cannot grow beyond the initial capacity of the array. However, this initial capacity is not visible in the abstract data type, which is just a plain list. Thus, we have to make it visible in the refinement relation: We define:

$$\textit{msl\_rel}\ N\ (n,p)\ l \equiv \exists l'.\ p \mapsto_a l' * n \leq N * l = \textit{take}\ n\ l' * \textit{length}\ l' = N$$

Here, the additional parameter N to the refinement relation denotes the capacity of the list. Operations have additional preconditions based on N, e. g. for appending an element to the list, we have:

$$(\textit{msl\_ap, lst\_op\_ap}): [\lambda l.\ \textit{length}\ l < N]\ (\textit{msl\_rel}\ N)^d \to \textit{Id}^k \to \textit{msl\_rel}\ N$$

3.3 Reusing Data Structures

When implementing complex data structures, it can be profitable to make use of already implemented data structures for the implementation itself. In many cases, the new data structure can be abstractly represented using some interfaces first, and in a second step, this abstract implementation can be refined to a concrete one, using implementations of the used interfaces. Usually, the second step, i. e. implementing the used interfaces, is canonical and can be completely automated by Sepref.

This approach has the advantage of proof modularization and reuse: In the first step, one can focus on the abstract idea of the data structure, while the second step focuses on its implementation, reusing existing implementations of the used interfaces.

Note that this approach works well for array-based data structures, as they always have an abstract representation using lists. However, pointers do not have meaningful abstract representations outside the heap monad. As Sepref currently cannot perform refinements inside the heap monad, but only from the nres to the heap monad, we loose the advantage of automation in those cases.

[2] Using the recent rewrite tool [24], this is usually a straightforward task.

Example 3.5. In Example 3.4, we sketched the implementation of a list by a number n for the length and an array. We can also verify this implementation on a number and a fixed length list first, and then implement the list by the array implementation described in Example 3.3.

For the first refinement step, we define msl_rel_1 as follows:

$msl\_rel_1\ N \equiv br\ (\lambda(n,l).\ take\ n\ l)\ (\lambda(n,l).\ n \leq N \wedge length\ l = N)$

Note that $br\ \alpha\ I$ defines a refinement relation from an abstraction function and an invariant:

$br\ \alpha\ I \equiv \{\ (c,\ \alpha\ c)\ |\ I\ c\ \}$

The append operation is implemented as follows:

$msl\_ap_1\ (n,l)\ x \equiv l \leftarrow lst\_op\_set\ l\ n\ x;$ **return** $(n+1,\ l)$

Clearly, updating the element at index n via *lst_op_set* adds the precondition $n<length\ l$. Thus, we prove the following refinement:

$apref_1$: $(msl\_ap_1,\ lst\_op\_ap)$:
$[\lambda l\ x.\ length\ l<N]\ msl\_rel_1\ N \rightarrow Id \rightarrow msl\_rel_1\ N$

Using the array implementation of the list interface described in Example 3.3, Sepref automatically synthesizes refinements for the operations. For example, it synthesizes msl_ap_2 with the refinement theorem:

$apref_2$: $(msl\_ap_2,\ msl\_ap_1)$: ${msl\_rel_2}^d \rightarrow Id^k \rightarrow msl\_rel_2$

where $msl\_rel_2 = nat\_rel \times \mapsto_a$. Note that we use $\times$ as the natural relator of products here, i. e.

$((c_1,c_2),(a_1,a_2)) \in R_1 \times R_2 \equiv (c_1,a_1) \in R_1 \wedge (c_2,a_2) \in R_2$

Combining the theorems $apref_1$ and $apref_2$ yields

apref: $(msl\_ap_2,\ lst\_op\_ap)$:
$[\lambda l\ x.\ length\ l<N]\ msl\_rel\ N \rightarrow Id \rightarrow msl\_rel\ N$

where $msl\_rel = msl\_rel_2\ O\ msl\_rel_1$.

Note that the algorithmic idea of the data structure, i. e. representing a variable length list by a fixed length one and a counter, was proved in Theorem $apref_1$. In this proof, we could use lists for which there is a well-developed Isabelle/HOL library. The implementation idea, i. e. using arrays to implement fixed length lists, was proved in Theorem $apref_2$. By reusing an already existing implementation for fixed length lists, this was fully automatic in our case.

3.3.1 FCOMP Tool

In Example 3.5 we had to combine two refinement lemmas to yield a new one. This is possible due to transitivity of refinement, i. e. we have[3]:

(f_1,f_2): $U_1 \rightarrow R_1 \wedge (f_2,f_3)$: $U_2 \rightarrow R_2$
$\Longrightarrow (f_1,f_3)$: $(U_1\ O\ U_2) \rightarrow (R_1\ O\ R_2)$

After application of this rule, we get the argument relation $U_1\ O\ U_2$. However, for n arguments, Sepref expects the relation to have the form $A_1 \times \ldots \times A_n$. Fortunately, we have $(A \times B)\ O\ (A' \times B') = (A\ O\ A') \times (B\ O\ B')$. Thus we can use rewriting to bring the relation into the expected form.

We have implemented this composition and rewriting, together with some normalization of the combined preconditions, as an Isabelle attribute. Thus, obtaining Theorem *apref* (cf. Example 3.5) is as easy as writing:

lemmas $apref = apref_2[FCOMP\ apref_1]$

[3] Note that we have skipped handling of preconditions and preservation information for the sake of simplicity.

3.4 Exploiting Parametricity

Note that we have not refined the element types of the implemented data structures in our previous examples. This greatly reduced the complexity of the refinement proofs. However, ultimately, we also want to specify refinements for the element types.

If the interface operations are parametric, refinement of element types can simply be added by composing the original refinement theorems with the parametricity theorems.

Note that interface operations are typically parametric wrt. relators that support some refinement for the element types.

Example 3.6. The operations of the list interface are parametric wrt. the natural relator on lists. Only the *idx_of* operation, which returns the index of an element, requires that the relation for the elements preserves equality.

For example, we have:

$(lst\_op\_ap,\ lst\_op\_ap)$: $\langle R\rangle list\_rel \rightarrow R \rightarrow \langle R\rangle list\_rel$

Composing this with the theorem *apref* from Example 3.5 yields:

$(msl\_ap_2,\ lst\_op\_ap)$:
$[\lambda l\ x.\ length\ l<N]\ \langle R\rangle msl\_rel'\ N \rightarrow R \rightarrow \langle R\rangle msl\_rel'\ N$

where $\langle R\rangle msl\_rel'\ N \equiv msl\_rel\ N\ O\ \langle R\rangle list\_rel$. Again, we can use the FCOMP tool to automatically obtain the new refinement lemmas[4].

4. Case Study: Heap and Heapmap

In the last section, we have introduced our refinement based development methods for verified imperative data structures along with simple examples. In this section, we present a case study for a more complex example: We formalize a heap data structure, extend it to a heapmap (also called indexed heap), and finally use our data structure to obtain an efficient implementation of Dijkstra's shortest paths algorithm [6]. This case study shows that our methods are applicable for the development of more complex data structures, and that the resulting data structures are usable as building blocks for verified algorithms.

4.1 Heaps

A heap (cf. [23, Chap. 2.4]) is a binary tree whose nodes have priorities, such that each node has smaller or equal priority than its children. Clearly, the root node has minimal priority. Heaps are a standard data structure to implement priority queues. For the imperative version of heaps, the binary tree is usually encoded as a list (implemented by an array), such that the root is at index 1, and the left and right children of a node at index i are at indexes $2i$ and $2i+1$ respectively.

Removing an element with minimal priority is achieved by copying the last element of the list to the root node, shortening the list by one, and then performing the *sink* (or sift down) procedure to restore the heap property: as long as the element is not smaller or equal than its children, the sink procedure exchanges the element with its smaller child. Thus, the element „sinks down" until the heap property is restored.

Symmetrically, new elements are inserted at the end of the list, i. e. at a leaf node position, and then the *swim* (or sift up) procedure restores the heap property: as long as the parent's priority is bigger, it exchanges the element with its parent. Thus, the element „swims up" until the heap property is restored.

[4] Currently, some cases require straightforward massaging of the generated precondition, e. g. to transport the *length* function over the list relator. However, this could also be added to the FCOMP tool in a future version.

Roughly following the presentation of Sedgewick et al. [23], we base the heap operations on a few basic operations that actually access the data structure:

empty: $'a$ *heap* — The empty heap.
val_of: $'a$ *heap* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ — Get element at **valid** index.
exch: $'a$ *heap* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *heap* — Exchange elements at two **valid** indices.
butlast: $'a$ *heap* $\Rightarrow$ $'a$ *heap* — Remove last element of **non-empty** heap.
append: $'a$ *heap* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *heap* — Append new element.

The preconditions of the operations are printed in bold face.

This architecture is well suited to refinement, as only the basic operations need to be implemented in order to get an implementation of the other operations automatically.

4.1.1 Priority Queue Interface

The type of our priority queue interface is $'a$ *multiset*. Moreover, we assume a fixed function *prio:* $'a \Rightarrow 'b$*::linorder*, which maps each element to its priority, where priorities have a linear order. The operations are the following:

empty: $'a$ *multiset* — Empty priority queue.
is_empty: $'a$ *multiset* $\Rightarrow$ *bool* — Test for emptiness.
insert: $'a$ *multiset* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *multiset* — Insert element.
peek_min: $'a$ *multiset* $\Rightarrow$ $'a$ — Return a minimal element a of **non-empty** queue.
pop_min: $'a$ *multiset* $\Rightarrow$ ($'a$ *multiset* $\times$ $'a$) — Remove and return a minimal element of a **non-empty** queue.

4.1.2 Implementation on Lists

In order to implement this interface by an efficient array data structure, we take a two-step approach: In a first step, we model the heap as a list, and in a second step, we implement the list by a length counter and an array (cf. Example 3.5). The first step performs the correctness reasoning of the heap data structure. For this task, lists, which are backed by a well-developed Isabelle theory, are appropriate. The second step is mostly automatic: We reuse the array-based list implementation depicted in Example 3.5, and let Sepref synthesize the implementation for us.

For the first step, we define the following refinement relation, which abstracts the list to the multiset of its elements, and captures the heap property described above:

$$\begin{array}{l} \mathit{heap1\_}\alpha \;\equiv\; \mathit{multiset\_of} \\ \mathit{heap1\_invar}\ h \;\equiv\; \forall i.\ 1{<}i{\le}\mathit{length}\ h \\ \quad\longrightarrow \mathit{prio}\ (\mathit{val\_of}\ h\ (i/2)) \le \mathit{prio}\ (\mathit{val\_of}\ h\ i) \\ \mathit{heap1\_rel} \;\equiv\; \mathit{br}\ \mathit{heap1\_}\alpha\ \mathit{heap1\_invar} \end{array}$$

Then, we define the basic operations in terms of the list interface, and set up the VCG to produce verification conditions over plain lists, which are convenient to prove. In this case, the setup is particularly simple, as each operation corresponds to exactly one operation of the list interface, only the indices have to be converted from 1-based indexing into the heap to 0-based indexing into lists.

For example, the exchange operation is defined as follows:

$$\mathit{exch\_op}\ l\ i\ j \;\equiv\; \textbf{assert}\ (i{>}0 \wedge j{>}0);\ \mathit{lst\_op\_swap}\ l\ (i-1)\ (j-1)$$

Moreover, we declare the following rule to the VCG, to map the operation to the plain list operation *swap*:

$$\begin{array}{l} [\![\mathit{valid}\ l\ i;\ \mathit{valid}\ l\ j]\!] \\ \quad\Longrightarrow \mathit{exch\_op}\ l\ i\ j \le \textbf{return}\ (\mathit{swap}\ l\ (i{-}1)\ (j{-}1)) \end{array}$$

where $\mathit{valid}\ l\ i \equiv 0 < i \le \mathit{length}\ l$ denotes a valid index.

The next step is to define the auxiliary operations sink and swim, based on the *val_of* and *exch* operations. We follow the definitions of [23], but use explicit tail recursion instead of while loops. Moreover, we chose to use the basic operations in monadic style, such that they contain assertions of their preconditions. While this makes the code slightly larger, it is more amenable to later refinement (cf. Example 3.2).

```
swim1_op h i ≡
 rec (λswim (h,i). do {
  assert (valid h i);
  if has_parent h i then do {
   vpi ← val_of_op h (parent i);
   vi ← val_of_op h i;
   if (¬ (prio vpi ≤ prio vi)) then do {
    h ← exch_op h i (parent i);
    swim (h, parent i)
   } else
    return h
  } else
   return h
 }) (h,i)
```

```
private boolean greater(int i, int j) {
 if (comparator == null) {
  return ((Comparable<Key>) pq[i]).compareTo(pq[j]) > 0;
 } else {
  return comparator.compare(pq[i], pq[j]) > 0;
 }
}

private void swim(int i)  {
 while (i > 1 && greater(i/2, i)) {
  exch(i, i/2);
  i = i/2;
 }
}
```

Figure 1: Isabelle and Java implementation of swim.

In Figure 1, we display our implementation of the swim function and the corresponding Java implementation from [23][5]: Note that we did not define an own function for the comparison of two keys: Though this would make the algorithm slightly more readable, it has no positive effect on the generated VCs.

For our swim operation, we prove two correctness conditions: (1) It does not change the multiset abstraction of the list and (2) it restores the heap invariant if it is locally violated at index i, which is expressed by *swim_invar h i*. We prove the following theorem and declare it to the VCG:

$$\begin{array}{l} \mathit{swim\_invar}\ h\ i \Longrightarrow \mathit{swim1\_op}\ h\ i \\ \quad\le \textbf{spec}\ h'.\ \mathit{heap1\_}\alpha\ h' = \mathit{heap1\_}\alpha\ h \wedge \mathit{heap1\_invar}\ h' \end{array}$$

The proof is fairly straightforward, the main work goes into proving that *swim_invar* is an invariant.

Finally, we implement the priority queue operations and show them correct wrt. the multiset based interface. For example, the insert operation is implemented as follows:

```
insert1_op h v ≡ do {
 assert (heap1_invar h);
 h ← append1_op h v;
 swim1_op h (length h)
}
```

[5] We slightly changed the layout, and renamed the argument of swim, to get the same naming as in the Isabelle source

To show its correctness, we first prove that appending an element to the heap only violates the heap property locally:

lemma *swim_invar_insert:*
$heap1\_invar\ l \Longrightarrow swim\_invar\ (l@[x])\ (length\ l + 1)$
unfolding *swim_invar_def heap1_invar_def valid_def parent_def val_of_def*
by (*fastforce simp: nth_append*)

Thanks to the powerful Isabelle list library, this proof is fairly straightforward: We first unfold some definitions to convert the proposition to one over standard list functions, and then complete the proof by Isabelle's *fastforce* tactic with one additional hint. Having proved the above lemma, showing correctness of the insert operation is straightforward, using the VCG:

lemma *insert_op_refine:*
$(insert1\_op, insert) \in heap1\_rel \to Id \to heap1\_rel$
unfolding *insert1_op_def[abs_def] heap1_rel_def*
by *refine_vcg* (*auto simp: swim_invar_insert in_br_conv*)

4.1.3 Implementation with Arrays

Refining the list-based heap implementation to arrays is straightforward, and provides no surprises. First, we agree on a refinement relation: We use the list implementation from Example 3.5, and define $heap2\_rel\ N \equiv msl\_rel\ N$. Here, N is the capacity of the heap. Then, we use the Sepref tool to automatically synthesize implementations of the basic operations, and, after declaring these implementations to Sepref, it can also synthesize implementations of the remaining operations. Finally, we use the FCOMP tool to combine the refinement lemmas from array-based to list-based, and from list-based to multiset-based, resulting in an implementation of the multiset-based priority queue interface, wrt. the refinement relation:

$$heap\_rel\ N \equiv heap2\_rel\ N\ O\ heap1\_rel$$

For example, Sepref synthesizes a function *insert_impl* with:

$(insert\_impl,\ insert1\_op)$:
$[\lambda l\ x.\ length\ l{<}N]\ (heap2\_rel\ N)^d \to Id^k \to heap2\_rel\ N$

Combining this with Lemma *insert_op_refine*, we get:

$(insert\_impl,\ insert)$:
$[\lambda m\ x.\ size\ m{<}N]\ (heap\_rel\ N)^d \to Id^k \to heap\_rel\ N$

4.2 Heapmaps

In the last section we have implemented a simple priority queue interface with heaps. For the verification, we used two main techniques: First, we restricted the access to the actual data structure to a few basic operations. This makes further refinement simpler, as from a refinement of the basic operations, a refinement of the other operations can be generated automatically. Second, we used a two-step approach, first proving the idea of the heap data structure on lists, and then implementing the lists with arrays. This stepwise refinement approach separates the proof of the algorithmic idea of the data structure from the proof of its implementation. Moreover, it allowed us to reuse an existing implementation of the list interface.

4.2.1 Priority Map Interface

The multiset-based priority queue interface does not support access to other elements than a minimal one. For this we define the *priority map* interface over the type $('k,'v)pm \equiv 'k \Rightarrow 'v\ option$, which is the standard Isabelle type for partial functions (called *maps*), and comes with a large library and good proof automation setup. As for priority queues, we fix a function $prio\text{: } 'v \Rightarrow 'b{::}linorder$, which maps values to priorities, In the following, we display some selected operations of the priority map interface:

- $empty\text{: } ('k,'v)pm$ — Empty map.
- $is\_empty\text{: } ('k,'v)pm \Rightarrow bool$ — Emptiness check.
- $insert\text{: } (('k,'v)pm) \Rightarrow 'k \Rightarrow 'v \Rightarrow (('k,'v)pm)$ — Update value at **unmapped** key.
- $decr\text{: } (('k,'v)pm) \Rightarrow 'k \Rightarrow 'v \Rightarrow (('k,'v)pm)$ — **Decrease** priority of **existing** entry.
- $remove\text{: } (('k,'v)pm) \Rightarrow 'k \Rightarrow (('k,'v)pm)$ — Remove mapping for **existing** key.
- $lookup\text{: } (('k,'v)pm) \Rightarrow 'k \Rightarrow 'v\ option$ — Retrieve value for key.
- $pop\_min\text{: } (('k,'v)pm) \Rightarrow (((('k,'v)pm) \times 'k \times 'v)$ — Remove and return a minimal element from **non-empty** map.

4.2.2 Implementation with List and Map

In [23], a data structure for *indexed priority queues* is described: It is based on a heap that contains the keys, and a map from keys to values. The data structure maintains an additional map from keys to their indexes on the heap, which is required to locate the element on the heap if it is removed or its priority changes.

The keys are restricted to natural numbers less than N, where N is an initially fixed capacity. This allows an implementation of the maps as arrays.

In order to implement this data structure, we again use stepwise refinement: First, we model the heap as a list, and the map from keys to values as a map of type $'k \rightharpoonup 'v\ option$. Note that we do not need to model the map from keys to heap indices, as this information can be obtained by looking up the key in the heap. In a second step, we add the map to heap indices, and implement the maps and the heap by arrays.

When verifying this implementation, we want to reuse the verification that we have already done for heaps. However, we cannot use the multiset-based priority queue interface to describe our heapmap data structure, as we require indexing into the heap. Instead, we define a relation between our heapmap data structure and the list-based heap data structure, and show that the heapmap operations refine the corresponding heap operations. From this, we get preservation of the heap invariant. Moreover, we can reuse lemmas that we used to prove the heap operations correct in the correctness proofs of our heapmap operations. The refinement relation maps the list of keys to their corresponding values. Moreover, it ensures integrity of our implementation, i. e. the list of keys contains exactly the keys of the map, without duplicates:

$$hmh\_\alpha\ (pq,m) \equiv map\ (the\ o\ m)\ h$$
$$hmh\_invar\ (pq,m) \equiv distinct\ pq \wedge set\ pq = dom\ m$$
$$hmh\_rel \equiv br\ hmh\_\alpha\ hmh\_invar$$

Additionally, we define a refinement relation between heapmaps and the priority map interface:

$$hm1\_\alpha\ (pq,m) \equiv m$$
$$hm1\_invar\ (pq,m) \equiv hmh\_invar\ (pq,m) \wedge h.heap1\_invar\ (hmh\_\alpha\ (pq,m))$$
$$hm1\_rel \equiv br\ hm1\_\alpha\ hm1\_invar$$

Note that we prefix operations from the heap implementation with „*h.*". The invariant combines both the structural integrity of our implementation *hmh_invar*, and the heap property of the abstracted heap. Note that this splitting of invariants is necessary, as we want to show refinements between the basic operations on heapmaps and heaps, which do not preserve the heap invariant, but must preserve the structural integrity of our implementation. For example, the implementation of the append operation should append an entry to the heap, and update the map accordingly. However, in order to restore the heap property, a subsequent swim operation is required.

Next, we define the basic operations on heapmaps, and show that they refine the corresponding basic operations on heaps. For example, we define the exchange operation as:

exch_op (pq,m) i j ≡ **assert** (*i>0* ∧ *j>0* ∧ *hmh_invar (pq,m)*);
pq ← *lst_op_swap pq (i − 1) (j − 1)*; **return** (*pq,m*)

Then, we show the following refinement theorem (note that *h.exch_op* is the corresponding operation on heaps):

(*exch_op,h.exch_op*): *hmh_rel* → *nat_rel* → *nat_rel* → *hmh_rel*

Additionally, we need to show what effect the operations have on the abstraction to priority maps. For *exch_op* we show that it does not change the abstraction to priority maps. Note that we cannot get this information from the relation to the *h.exch* operation, as we do not see the keys on this abstraction level, but only a list of values.

After we have defined the basic operations, we implement swim and sink. Their refinement proof is straightforward, as they only use the basic operations, for which we have already shown refinement. The proof that they preserve the abstraction to priority maps is also straightforward, using the already proved facts about the basic operations.

Finally, we have to implement the priority map interface. The refinement is shown wrt. the *hm1_rel* relation, which also contains the heap invariant. However, as the heapmap operations relate to heap operations, we can reuse the lemmas we have proved for the heap operations. This proof principle is formalized as follows:

lemma *heapmap_nres_rell′:*
assumes *1:* (*hm,h*)*:* *hmh_rel*
assumes *2: h* ≤ **spec** *h.heap1_invar*
assumes *3: hm* ≤$_n$ **spec** (λ*hm′*. **return** (*hm1_α hm′*) ≤ *m*)
shows (*hm, m*)*:* *hm1_rel*

Intuitively, the lemma states that, if (1) the heapmap operation *hm* is related to a heap operation *h*, which (2) ensures the heap property, and moreover, (3) assuming that *hm* does not fail, its result abstracted to maps is described by priority map operation *m*, then *hm* is related to *m*.

Note that this formulation makes it easy to transport assertions from the heap level to the heapmap level (in the proof of assumption (1)). These can then be assumed when proving correct behavior wrt. the map abstraction (3). However, assertions newly introduced on the heapmap level have to be proved in the refinement proof (1), although one might want to prove them in (3). As we did not encounter such a case in our implementation, we did not investigate more powerful alternative proof principles, but leave this to future research.

4.2.3 Implementation with Arrays

The implementation of heapmaps again uses stepwise refinement: We first refine the map to a list of values, restricting the keys to natural numbers less than N. Checking whether a key is contained in the map is done by checking whether it is contained in the heap. The refinement relation is:

hm2_α (pq,ml) ≡ (*pq,*λ*k.* **if** *k*∈*set pq* **then** *Some* (*ml!k*) **else** *None*)
hm2_invar N (pq,ml) ≡ *hmh_invar (hm_α (pq,ml))*
∧ *set pq* ⊆ {*0..<N*} ∧ (*length ml = N*)
hm2_rel N ≡ *br hm2_α (hm2_invar N)*

Note that we only have to implement the basic operations for this intermediate refinement step. In a second step, we implement the first list by two arrays and a length counter, the first array holding the heap, and the second array holding the indices of the keys in the heap, or N if the key is not in the heap[6]. We call the corresponding relation *hm3_rel*.

Note that, for the implementation of the heap list by two arrays, we reuse an implementation of the list interface, which we have developed independently (again, using stepwise refinement and the *msl* implementation).

After we have combined the two refinement steps for the basic operations, we obtain a refinement wrt. the relation:

hm32_rel N ≡ *hm3_rel N O hm2_rel N*

The other operations are transferred automatically over this relation, resulting in a complete implementation of heapmap. Combining these refinement lemmas with the refinement of heapmap to priority map, we obtain a priority map implementation wrt. the relation:

hm321_rel N ≡ *hm32_rel N O hm1_rel*

Finally, we compose the implementation with parametricity lemmas on the priority map interface, to obtain a final implementation wrt. the relation:

⟨*K,V,P*⟩*hm_rel N* ≡ *hm321_rel N O* ⟨*K,V,P*⟩*prio_map_rel*

where K, V, P are the refinement relations for the keys, values, and priorities. However, the last refinement comes with some restrictions: As we use equality on keys and comparison on priorities in the specification of priority maps, K and P have to preserve these operations. Moreover, we have to specify a new $prio'$ function, which refines the original one. Finally, for the fixed capacity implementation, K has to preserve comparison with a maximum size. Although we have proved the parametricity lemmas in this generality, in practice, we use the constraints $K \subseteq Id$, $V = P = Id$, and $prio' = prio$. This is sufficient for our examples, and there are nice syntactic rules to discharge such constraints automatically during the synthesis process.

4.3 Reality Check: Dijkstra's Algorithm

In the last section we have described our implementation of priority maps by the heapmap data structure. Apart from stepwise refinement and a small set of basic operations, we also used refinement techniques to reuse results from the already verified heap data structure.

In this section, we present an application of our heapmap data structure: We use it as priority map data structure for Dijkstra's algorithm [6]. Recall that Dijkstra's algorithm iteratively updates a distance map, using a priority queue to select a node with current minimal distance. After processing a node, the minimal distances of its neighbors may change. This requires a decrease-key operation — exactly what is provided by our priority map interface.

We have used Dijkstra's algorithm as an example for Sepref already in [13]: From an existing functional implementation [19], Sepref can automatically synthesize an imperative implementation. However, we only had a functional heap data structure at hand, which was rather slow. Here we describe what was required to set up the synthesis with our imperative heapmap data structure.

The main challenge is the fixed capacity of our heapmap implementation, which must be initialized on construction. Here, we use the number of nodes in the input graph. However, originally, we refined the nodes, which are represented by natural numbers, by the identity relation. This, however, does not carry enough information to prove that the node numbers are small enough to fit into the heapmap. One solution is to modify the original algorithm by adding assertions of the form $v \in V$, whenever a node v is accessed.

[6] As we use natural numbers, we could not use -1 to denote non-existing keys, which would be more standard.

These assertions have to be proved in the correctness proof of the algorithm, and can be assumed during the refinement proof.

However, as we want to avoid implementation specific modifications to the abstract algorithm, we choose an alternative solution: We strengthen the refinement relation for nodes to *node_rel N* $\equiv \{(u,u).\ u<N\}$. As nodes are not changed by Dijkstra's algorithm, but just stored and retrieved[7], this only requires to change the refinement of our graph implementation accordingly, which is straightforward.

With this setup, the Sepref tool automatically synthesizes an imperative implementation of the abstract monadic version of Dijkstra's algorithm. It creates a constant *dijkstra_imp*, and the theorem:

(*dijkstra_imp N, mdijkstra*): $[\lambda G\ v_0.$ *Dijkstra G* $v_0]$
$(\langle Id\rangle graph\_rel\ N)^k \to (node\_rel\ N)^k \to drmap2\_rel\ N$

The precondition *Dijkstra* is the precondition for Dijkstra's algorithm, i. e. the graph must be well-formed and finite, the start node must be a node of the graph, and the weights must not be negative. The relation $\langle Id\rangle graph\_rel\ N$ denotes the implementation of graphs with N nodes numbered from $0 \ldots < N$ and weights refined by *Id*. The relation *drmap2_rel N* is the relation for the result of Dijkstra's algorithm, basically a map from nodes to shortest paths and their weights. Note that the constant *mdijkstra*, which we used as starting point of refinement, is, itself, a refined version of Dijkstra's algorithm. We have:

(*mdijkstra, shortest_path_map*):
$[\lambda G\ v_0.$ *Dijkstra G* $v_0]$ *Id* $\to$ *Id* $\to$ *drmap1_rel*

Here, *drmap1_rel* removes the weights from the shortest paths, such that we get a map from nodes to paths, and *shortest_path_map* is the specification of a correct map, i. e. one that associates exactly the reachable nodes to shortest paths.

The combination of these lemmas still contains the precondition *Dijkstra*. However, our graph implementation can only represent finite graphs, and the well-formedness is already contained in the refinement relation *graph_rel*. Exploiting this information, it is straightforward to prove the following correctness theorem for the implementation of Dijkstra's algorithm:

(*dijkstra_imp N, shortest_path_map*):
$[\lambda G\ v_0.\ v_0 \in nodes\ G \wedge (\forall(\_,w,\_)\in edges\ G.\ 0\leq w)]$
$(graph\_rel\ N\ Id)^k \to (node\_rel\ N)^k \to drmap\_rel\ N$

where *drmap_rel N* $\equiv$ *drmap2_rel N O drmap1_rel*.

4.4 Benchmarks

We have benchmarked our heapmap implementation, as well as our implementation of Dijkstra's algorithm, against the reference implementations from [23] and the purely functional implementation based on 2-3-finger trees [8, 18] that we originally used. The results are shown in Tables 1 and 2.

For the heapmap benchmark, we have created a heapmap of capacity N, inserted N random values at keys from 0 to $N-1$, changed all keys to a new random value, and then flushed the queue by repeatedly removing minimal values. Our imperative implementation is roughly a factor of 4 faster than the purely functional one. This is a clear efficiency improvement for verified priority map data structures. However, depending on N, it is between 2 and 5 times slower than the unverified Java implementation. We have not yet fully investigated the reason for that. First profiling results indicate that the generated code wastes too much time on arbitrary precision integers, which seems to be a problem with the code generator. We get similar results when benchmarking Dijkstra's algorithm. Here, the test inputs are complete graphs with random weights over N nodes (cl$N$) and the medium and large examples from [23].

[7] In other words: the algorithm is parametric in the node type.

N	Fun	Imp/HOL	Java
10000	72	18	9
100000	947	195	38
1000000	12816	2790	873
10000000	162893	39118	15487

Table 1: Heapmap Benchmark. (Times in ms)

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

Table 2: Dijkstra Benchmark. (Times in ms)

5. Conclusion

In this paper we have presented a method for stepwise refinement based development of verified imperative data structures, and illustrated our methods via the implementation of the priority map interface by a heapmap data structure.

We have presented techniques to modularly build complex data structures from simpler ones. Using the infrastructure of our Sepref tool, we could automatically synthesize the implementations from abstract versions of the operations.

In our heapmap case study, we presented a quite complex setting for our techniques: On the abstract heapmap data structure, we identify a set of basic operations, on which all other operations can be built. This allows us to automatically derive implementations of the other operations from implementations of the basic operations. To implement the basic operations, we use stepwise refinement, decomposing the abstract heapmap data structure in simpler data structures first, and then reusing existing implementations of these simpler data structures. Finally, we related the heapmap data structure to an independently verified heap data structure. Although we cannot express priority maps using priority queues, we still could reuse parts of the correctness proof for heaps (implementing priority queues) in the proof for heapmaps (implementing priority maps). Finally, we added support for refinement of the element types, by exploiting parametricity of the priority map interface.

The whole verification process was supported by the Sepref tool [13], which automatically replaces abstract by concrete operations, and the FCOMP tool, which composes the refinement theorems of multiple steps.

The techniques illustrated in this paper are not specific to heapmaps, but work for all imperative data structures which have nice abstract representations inside Isabelle/HOL.

5.1 Related Work

There are tools that generate verification conditions for different programming languages, and use interactive or automated theorem provers to discharge them, e. g. [5, 7, 10]. One example is the Jahob tool [10, 11], which combines several analysis and proving tools, including Isabelle/HOL, to prove properties about Java programs. It uses data abstraction as a refinement technique, to reason about data structures in terms of their abstract interfaces. The main differences to our approach are that Jahob and other VCG based tools use a bottom-up approach, i. e. the final implementation (Java program)

is developed first, then annotated with specification information, and proved correct. In contrast, we develop the abstract algorithm first, and then refine it, in possibly multiple steps, towards the final implementation. Thereby, we may perform more complex refinements than just a single data refinement step, which roughly corresponds to the data abstraction available in Jahob.

Another important difference is the trusted code base, i. e. the code that is critical to the correctness of the tool. Apart from the compiler, the machine, and the operating system, our tool's correctness only depends on the small logical inference kernel of Isabelle, the axiom scheme of HOL, and the correctness of the Isabelle Code Generator, which generates the final source text. In contrast, VCG based tools like Jahob usually combine different tools to discharge the VCs, and the VCG itself has to correctly map the target language's semantics, which is usually more complex than the mapping done by the Isabelle/HOL code generator.

The heap data structure has been used as an example in several program verification tools: e. g. heapsort is verified in at least [1, 10, 20]. A top-down refinement based approach to heap and selection sort is presented in [20]. However, their refinement is focused to derive the sorting algorithm from some very generic specification. Their refinement stops at an abstract tree data structure, while we prove correctness of a heap data structure embedded in a list directly, and focus our refinement on the implementation.

A priority queue implemented by a heap in OCaml has been verified using characteristic formulas [5], a bottom-up approach to verify imperative programs. We are not aware of previous verifications of the (more complex) heapmap data structure.

The Transfer and Lifting tool [9] performs a task similar to our Sepref tool: It transports definitions and lemmas along relations. However, it is restricted to purely functional programs, and the concrete type is constructed by renaming the top-level type name of the abstract type.

5.2 Future Work

One direction is to explore how our techniques work with data structures that rely on pointer modification, like balanced trees. As pointers have no abstract counterpart in Isabelle/HOL, our approach of reusing simple data structures for the implementation of more complex ones has to be adapted if pointers need to be visible on the abstract level.

Another direction of future work is to improve the available automation. For example, capturing the grouping of operations into interfaces and implementations within a tool would allow us to perform operations like composition of refinements for whole sets of operations at once, avoiding to repeat boilerplate code for each single operation.

Finally, we have observed a great effect of low-level peephole optimizations on the generated Imperative/HOL programs: Some simple inlining and deforestation that we manually performed on our heapmap implementation (inside the logic!) already had a significant impact on performance. We would like to systematically explore and automate those optimizations.

References

[1] R.-J. Back and J. Eriksson. An exercise in invariant-based programming with interactive and automatic theorem prover support. In *THedu*, volume 79 of *EPTCS*, pages 29–48, 2011.

[2] R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction*. Springer, 1998.

[3] 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.

[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] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[7] J.-C. Filliâtre and A. Paskevich. Why3 – Where Programs Meet Provers. In *ESOP*, volume 7792. Springer, Mar. 2013.

[8] R. Hinze and R. Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

[9] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL, 2012. Isabelle Users Workshop 2012.

[10] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[11] V. Kuncak and M. Rinard. An overview of the Jahob analysis system: Project goals and current status. In *NSF Next Generation Software Workshop*, 2006.

[12] P. Lammich. Automatic data refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.

[13] P. Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.

[14] P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In *Proc. of ITP*, volume 6172 of *LNCS*, pages 339–354. Springer, 2010.

[15] 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.

[16] 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.

[17] R. Meis, F. Nielsen, and P. Lammich. Binomial heaps and skew binomial heaps. *Archive of Formal Proofs*, Oct. 2010. `http://afp.sf.net/entries/Binomial-Heaps.shtml`, Formal proof development.

[18] B. Nordhoff, S. Körner, and P. Lammich. Finger trees. *Archive of Formal Proofs*, Oct. 2010. `http://afp.sf.net/entries/Finger-Trees.shtml`, Formal proof development.

[19] 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.

[20] D. Petrovic. Verification of selection and heap sort using locales. *Archive of Formal Proofs*, Feb. 2014. `http://afp.sf.net/entries/Selection_Heap_Sort.shtml`, Formal proof development.

[21] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[22] 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.

[23] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 2011. 4th edition.

[24] C. Traut and L. Noschinski. Pattern-based subterm selection in isabelle. In *Proceedings of Isabelle Workshop 2014*, 2014.

[25] N. Wirth. Program development by stepwise refinement. *ACM*, 14(4):221–227, Apr. 1971.