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

Automatic Refinement to Efficient Data Structures

A comparison of two approaches

Peter Lammich · Andreas Lochbihler

Received: date / Accepted: date

Abstract We consider the problem of formally verifying an algorithm in a proof assistant and generating efficient code. Reasoning about correctness is best done at an abstract level, but efficiency of the generated code often requires complicated data structures. Data refinement has been successfully used to reconcile these conflicting requirements, but usability calls for automatic tool support. In the context of Isabelle/HOL, two frameworks for automating data refinement have been proposed [23, 33]. In this paper, we present and compare the two frameworks and their underlying ideas in depth. Thereby, we identify the challenges of automating data refinement, highlight the similarities and differences of the two approaches, and show their strengths and limitations both from the implementer's and the user's perspectives. A case study demonstrates how to combine both frameworks, benefiting from the strengths of each.

Keywords data refinement · algorithm verification · code generation

1 Introduction

Nowadays, many formalisations in proof assistants yield executable verified functional implementations based on code generation. Examples range from language interpreters and compilers [30,31,34] over an LTL model checker [7,12] and a termination certificate checker [48] to a conference management system [20]. Early works [5,21,35] implemented containers (mostly sets and maps) inefficiently, in particular as

This article builds on the author's individual presentations of the two approaches at *Interactive Theorem Proving* 2013 [23, 33].

A. Lochbihler
Institute of Information Security, Department of Informatics, ETH Zurich
Universitätstrasse 6, CH-8092 Zurich, Tel.: +41 44 632 84 70, Fax: +41 44 632 11 72
E-mail: andreas.lochbihler@inf.ethz.ch

P. Lammich
Technische Universität München, Institut für Informatik
Boltzmannstr. 3, D-85748 Garching, Tel.: +49 89 289-17326, Fax: +49 89 289-17307
E-mail: lammich@in.tum.de

lists and closures, or burdened the formalisation with complex data structure details. Experience has shown that the latter makes the proofs more complicated, and may even render proofs of medium-complex algorithms unmanageable [22, 39]. Moreover, changing the implementation later means essentially redoing the whole formalisation.

A well known solution to this problem is stepwise refinement [49], where the specification of an algorithm is refined towards an efficient implementation in several correctness preserving steps. This simplifies the verification by introducing a natural modularisation: Each step focuses on a particular idea of the algorithm, and is independent of subsequent steps and mostly independent on former steps.[1]

A particular case of a refinement step is data refinement [17], where abstract data types are refined to implementations (e.g. sets to red-black trees), but the structure of the algorithm is preserved. Conceptually, a data refinement step is simple: Replace the operations on the abstract type by the corresponding operations of the implementation, and use the correctness theorems of the implementation to prove that the resulting algorithm refines the original one. Without dedicated support, however, this step requires a lot of effort [14, 28, 29, 34]. To automate this task in the context of Isabelle/HOL, two frameworks have been proposed: Autoref [23] and Containers [33]. The main difference is that Autoref performs the refinement in the logic based on relational parametricity whereas Containers exploits the existing refinement capabilities of Isabelle's code generator with higher-order rewrite systems as formal basis.

Contributions In this article, we present and compare the two frameworks and their underlying ideas in depth. Our main contributions are the following:

- We identify the challenges of automating the refinement step (Section 3). As both frameworks face the same challenges, these appear to be general challenges for data refinement in practice.
- We discuss how each framework addresses these challenges (Sections 4 and 5). This improves the clarity of presentation over the previously published papers [23, 33], which left some of the challenges and the limitations implicit.
- Our analysis highlights the similarities and differences of the two approaches and shows the strengths and limitations of each framework. We assume a setting where the user has verified an abstract algorithm and wants to refine it to use efficient data structures. In summary, the two frameworks are complimentary (see Section 6 for a detailed comparison): Autoref can handle a larger scope of specifications (in particular, it can resolve non-determinism), but requires more machinery and work. Containers needs less effort, but has a more limited scope.
- Our case study illustrates the usage of our frameworks in a larger formalisation context, and exemplifies the combination of both frameworks to leverage their individual advantages (Section 7).

Both frameworks are generic in the containers and their implementations. To be usable in practice, both frameworks are connected to libraries of verified container data structures [28,29,33] (we do not discuss the connection in detail). In that combination, both have been used successfully in several large applications (cf. Section 9). The implementations of Autoref and Containers are available in the Archive of Formal Proofs at `http://www.isa-afp.org/entries/Automatic_Refinement.shtml`

[1] For example, an implementation of an operation is independent of the overall correctness of the algorithm, but may require a proof that the algorithm guarantees the preconditions of the operation.

and `http://www.isa-afp.org/entries/Containers.shtml`. Both frameworks also come with documentation and collections of examples, which include those of this paper.[2]

We emphasise that neither framework requires any new axioms or any changes to the code generator. The trusted code base is therefore not enlarged when Autoref or Containers are used. In particular, it does not include the implementations of our tools, as Isabelle's kernel checks all definitions and proofs.

2 Running Example: 2SAT via Depth-First Search

We illustrate the use of both frameworks on checking the satisfiability of a boolean formula in conjunctive normal form with at most two literals per clause (2CNF) by analysing the associated implication graph. Figure 1 shows a canonical formalisation of the 2CNF formulas and satisfiability in Isabelle/HOL: Propositional variables *var* are identified by natural numbers. Literals *lit* are records with two fields *pos* and *var* where *pos* indicates whether the variable *var* occurs non-negated in the literal. We write $P\ x$ for the positive literal of variable x and $N\ x$ for the negative literal. A clause *clause* is an unordered pair of literals. Here, the type *'a uprod* from Isabelle/HOL's standard library has one non-free constructor $(\!|\_,\ \_|\!)$ of type $'a \Rightarrow 'a \Rightarrow 'a\ uprod$ that ignores the order of its arguments: $(\!|x,\ y|\!) = (\!|y,\ x|\!)$.[3] By using unordered pairs, we ensure that every clause consists of at most two literals whose order is irrelevant. In other words, a clause is a set of literals with cardinality 1 or 2. A formula *cnf* is a set of clauses. Formulas are interpreted over valuations σ of type *valuation* in the standard way. We overload satisfaction $\models$ for literals, clauses, and formulas. A formula F is satisfiable iff $\sigma \models F$ for some σ.

Satisfiability of 2CNF formulas (2SAT) can be decided by analysing the implication graph of the formula. The implication graph has one vertex for each literal, and every clause $(\!|l,\ l'|\!) \in F$ induces two edges: one from *negate l* to *l'* and one from *negate l'* to *l*, where *negate* negates a literal. We model the graph *gr F* as a set of edges, i.e., pairs of nodes. In the following, we assume that all clauses contain exactly two literals (predicate *is2cnf*). This can be ensured, e.g., by preprocessing the formula with unit propagation, which eliminates all unit clauses of the form $(\!|l,\ l|\!)$. A formula F is satisfiable iff there is no path in *gr F* from a literal $P\ x$ to $N\ x$ and back to $P\ x$ for any variable x in F (theorem *2SAT_graph*, where R^* denotes the transitive and reflexive closure of a binary relation R).

We now want to use this theorem to obtain an efficient satisfiability checker for 2CNF formulas. To that end, we focus on the reachability problem on directed graphs.[4] As before, we model a graph as a set of edges $E :: ('v \times 'v)\ set$, i.e., pairs of nodes. A node v can reach a node w (notation *reachable v w*) iff $(v,\ w) \in E^*$.

[2] A Tutorial on Autoref can be found at `http://www21.in.tum.de/~lammich/refine_tutorial.html`, documented examples are available in the Collections AFP entry in folder `Examples/Autoref`. A user guide with examples for Containers is contained in the AFP entry.

[3] Accordingly, pattern matching on $(\!|\_,\ \_|\!)$ is only well-defined if the result of the pattern match does not depend on the order of the components. Isabelle's function package, e.g., therefore asks the user to prove well-definedness.

[4] A more efficient algorithm can be obtained by looking at the strongly connected components (SCC) of the implication graph instead of reachability [2]. We use reachability because it is simpler and illustrates the challenges well. Both frameworks can be used in the same way with the SCC-based algorithm.

type_synonym *var* = *nat*
datatype *lit* = *Lit* (*pos: bool*) (*var: var*)
type_synonym *clause* = *lit uprod*
type_synonym *cnf* = *clause set*
type_synonym *valuation* = *var* ⇒ *bool*

abbreviation *P* = *Lit True*
abbreviation *N* = *Lit False*

$$\frac{\sigma\ (var\ l) = pos\ l}{\sigma \models l} \qquad \frac{\sigma \models l \vee \sigma \models l'}{\sigma \models (\!|l, l'|\!)} \qquad \frac{\forall c \in F.\ \sigma \models c}{\sigma \models F} \qquad \frac{\sigma \models F}{satisfiable\ F}$$

definition *negate* :: *lit* ⇒ *lit* **where** *negate* (*Lit p v*) = *Lit* (¬ *p*) *v*
definition *gr* :: *cnf* ⇒ (*lit* × *lit*) *set* **where** *gr F* = ⋃(|*l, l'*|)∈*F*. {(*negate l, l'*), (*negate l', l*)}

definition *is2cnf* :: *cnf* ⇒ *bool* **where** *is2cnf F* = ∀(|*l, l'*|)∈*F*. *l* ≠ *l'*
definition *vars* :: *cnf* ⇒ *var set* **where** *vars F* = ⋃(|*l, l'*|)∈*F*. {*var l, var l'*}

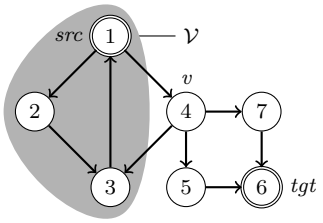
theorem *2SAT_graph:*
assumes *finite* (*vars F*) **and** *is2cnf F*
shows *satisfiable F* = (∀*x* ∈*vars F*. ¬ ((*P x, N x*) ∈ (*gr F*)* ∧ (*N x, P x*) ∈ (*gr F*)*))

Fig. 1: Formalisation of 2SAT and its implementation via graph reachability

input: graph given by a set of edges E
tgt node to be reached
output: *bool* whether *tgt* is reachable from v in E
state: $\mathcal{V}$ set of nodes, initially {}

```
1  dfs(v)
2    if v = tgt then return True;
3    V := insert v V
4    foreach successor v' of v do
5      if v' ∉ V ∧ dfs(v') then return True;
6    return False;
```

(a) Depth-first search algorithm in pseudo-code



(b) Snapshot of an example DFS execution started at node 1 taken during the visit to node 4

Fig. 2: The running example algorithm depth-first search and an example graph

Reachability can be implemented efficiently by a depth-first search (DFS) on the graph (see Figure 2a).

Figure 2b shows a snapshot of a DFS execution on an example graph. The search started at node 1 and the target node is 6. The **foreach** loop in line 4 over 1's successors visited node 2 before node 4. The snapshot has been taken during the visit to 4. From there, the search continues with the successors $\{3, 5, 7\}$ of 4, skipping already visited nodes (those in the grey area).

The first task is to obtain a verified and efficient implementation of this algorithm. In the verification part, we must prove that the DFS algorithm indeed implements the reachability test. To ease the verification, we assume that the algorithm is formulated in a functional style which uses mathematical sets. From there, we want to obtain an efficient implementation to which the correctness theorem extends. Such an implementation could look like the one in Figure 3, which has been written manually in Haskell. It differs from the pseudo-code in that it uses (unverified) efficient

```
import Data.Set (Set)
import qualified Data.Set as Set

type Graph a = a -> [a]

reachable :: forall a. Ord a => Graph a -> a -> a -> Bool
reachable succs src tgt = snd (dfs src Set.empty)
  where
    dfs :: a -> Set a -> (Set a, Bool)
    dfs x v
      | x == tgt  = (v, True)
      | otherwise = go (succs x) (Set.insert x v)

    go :: [a] -> Set a -> (Set a, Bool)
    go []     v = (v, False)
    go (x:xs) v
      | x `Set.member` v = go xs v
      | otherwise =
        let (v', b) = dfs x v
        in if b then (v', b) else go xs v'
```

Fig. 3: Haskell implementation of the DFS algorithm in Figure 2a

data structures (e.g., `Set` from the Haskell library), represents graphs by the successor function rather than as a set of edges, and fixes an iteration order for the **foreach** loop `go`. We will focus on the DFS algorithm in Sections 3–5 when we identify the challenges in automating the transition from formalisations of algorithms to executable programs using (verified) efficient data structures and show how the two frameworks address them.

The second task is to use the verified and efficient implementation in a larger formalisation context. In Section 7, we demonstrate how to obtain an efficient 2SAT checker based on the DFS algorithm by combining both frameworks and using the abstract theorem *2SAT_graph*.

3 Challenges on the Road to Efficient Data Structures

In this section, we identify four key challenges in automatically transforming a pseudo-code-like algorithm to an implementation which uses efficient data structures. The challenges appear to be general, as both frameworks face them, although they address them with different approaches. We assume that the frameworks have access to a library of verified data structures, e.g., from [28] or [37].

3.1 Identification of Types and Operations

The first challenge is to revert an important step of the formalisation process, namely the encoding of (mathematical) concepts in the language of logic. As the concepts shall be implemented with the available data structures from the library (see Section 3.2), we must (re-)identify the concepts first. Looking only at the HOL types does not suffice. In the DFS pseudo-code in Section 2, the concept *graph* has been

encoded as (*'v* × *'v*) *set*. But the same HOL type is also used for other concepts such as orders. Clearly, we should implement the former with efficient operations on graphs, but must not (and cannot) do so for the latter. Similarly, finite maps are usually modelled as *'a* ⇒ *'b option* in Isabelle/HOL, but ordinary functions that might fail and not return a result have the same type.

As we focus on implementing the concepts, it suffices to describe a concept with an abstract interface, i.e., what operations are used or available. We call this abstract interface the *conceptual type*. For example, a graph provides a function to retrieve the successors of a node, its predecessors, or the sets of all vertices and edges.

We have already seen that the HOL type does not suffice to determine the intended conceptual type. Moreover, the operations of a conceptual type do not always correspond to single constants in HOL. For example, the set of successors of a node v is conveniently written as E `` $\{v\}$ in HOL, where $R$ `` A denotes the image of the set A under the binary relation R and $\{x\}$ is just syntactic sugar for *insert x* {}. So, the abstract successor operation *succs* is expressed by the HOL term $\lambda E\ x.\ E$ `` *insert x* {}, but this term hardly ever appears literally in abstract programs, because HOL terms are automatically β-reduced.

Identifying the *conceptual types* in an abstract program is the first challenge in translating the program to a version that uses efficient data structures. Due to the ambiguities of encodings, full automation is not possible. Therefore, the Autoref and Containers framework provide heuristics that support the user in this task.

3.2 Selection of Data Structures

Suppose that the conceptual types and operations have been identified. Next, adequate data structures must be chosen. Ideally, the chosen data structures provide efficient implementations of all operations required by the program. For example, a conceptual graph can be implemented using a successor function, or an adjacency matrix, or a successor and a predecessor function. A successor function, which maps each vertex to the set of its successors, works fine if we only need the operation *succs*, but it supports a predecessor operation only inefficiently, if at all. The combination of successor and predecessor function supports both, and so do adjacency matrices when a conversion between indices and nodes is available. The DFS example needs only the successor operation, so all of the listed implementations can be used. The manual Haskell implementation in Figure 3 in fact represents graphs by the successor operation.

To relieve the user from having to choose implementations manually, both frameworks provide automation and heuristics. Automation selects the implementations that support all required operations and the heuristics then pick one of the selected ones. If necessary, the user can control the selection and override the heuristics.

Despite the different approaches to implementing the choice, the heuristics of the two frameworks are similar. In the remainder of this section, we discuss the two most important ones, namely (i) preferences for data structures and operations, and (ii) the homogeneity principle.

Preferences determine which data structures and which operations (on the same data structure) are preferred over others provided that both are applicable. For implementing a set, e.g., red-black trees are preferred over lists, but they need a total order on the elements. Accordingly, a set of integers becomes a red-black tree

and a set of complex numbers a list, unless we define a total order on complex numbers that is dedicated for red-black tree access. For operations, the situation is similar; the specialised union operation on red-black trees takes precedence over the generic implementation, which uses iteration and insertion.

The homogeneity principle says that the result of an operation should use the same data structure as its inputs, if possible. This avoids unnecessary conversions between data structures and requires less user annotations. For example, inserting an element into a set should not change the implementation of the set. Hence, if a set A of integers is implemented as a list, then *insert* n A also returns a list (and in particular does not convert A into a red-black tree first).

3.3 Non-Determinism

Algorithmic descriptions often use non-deterministic operations, e.g., when all possible interpretations are correct because the choice does not matter. In the DFS example in Figure 2a, the **foreach** loop in line 4 does not determine the order of iteration over the successors. In the situation of Figure 2b, *dfs* can visit node 5 or node 7 next. However, the order of iteration does affect the set of visited nodes. When *dfs* has terminated, depending on this choice, $\mathcal{V}$ will contain either node 5 or node 7. This non-determinism clashes with HOL functions being deterministic and total. Consequently, a functional implementation of DFS in HOL must refine this non-determinism.

For efficiency reasons, the implementation should exploit the freedom of how to resolve the choice. In Figure 3, the set of successors is implemented as a list, and the **foreach** loop `go` processes the elements in the order as they appear in the list. So there is no point in re-arranging the list into a distinguished order first (e.g., sorted). But different lists (e.g., $[5, 7]$ and $[7, 5]$) can represent the same set (e.g., $\{5, 7\}$). Thus, there are different iteration orders that cannot be distinguished in the logic on the abstract level of sets.

Hilbert's choice operator (written *SOME x. P x* in Isabelle) seems to offer an easy implementation of the **foreach** loop in the logic: just have it choose the iteration order. Yet, underspecified choice cannot be implemented at all in HOL, as discussed in [29, 34]. The reason is that the interpretation of the choice operator in a model of HOL is fixed (although underspecified). However, any implementation would have to commit to a specific choice, which cannot be proved to coincide with the one in the model due to the underspecification.

As we want to obtain an implementation in a deterministic functional language (the target programming languages of Isabelle's code generator in our case), any non-determinism has to be resolved on the way down to executable code. There are two ways how this can be achieved in HOL:

1. If the non-determinism does not affect the result of a function, it is sound to work with a function in Isabelle/HOL, and let the code generator resolve the non-determinism later. In the running example, we only care about reachability, not about the set of visited nodes, i.e., the Boolean result of *dfs* is deterministic. Hence, *dfs* can be formalised as a function. To that end, we build on the work by

Nipkow and Paulson [38]. They showed that in **foreach** loops, left-commutativity of the loop body ensures that the result does not depend on the order of iteration.[5]

2. Non-determinism can be modelled by relations between inputs and outputs. Lammich and Tuerk [29] have developed a language for formalising algorithms as relations and reasoning about them. This approach works for all non-deterministic specifications. For example, one could also return a path between the two nodes in case of reachability. Since the path depends on how the non-determinism is resolved ($1 \to 4 \to 5 \to 6$ vs. $1 \to 4 \to 7 \to 6$), this is not possible with the previous approach. Using a refinement calculus [3], the relational specification can be refined until it becomes deterministic, e.g., by replacing iteration over a set by iteration over a list representing the set. The Autoref tool automates such refinements.

3.4 Scalability in the Number of Data Structures

As the library of verified data structures grows, a naive framework can easily run into scalability problems. For example, sets can be implemented by characteristic functions, lists, distinct lists, various search trees, hashing, etc. The scalability issue shows up most quickly with binary operations such as $-$, $\cap$ and $\cup$. They must handle all combinations of different data structures in the parameters, which grows quadratically with the number of available ones. Therefore, we cannot have one dedicated implementation for every combination without sacrificing scalability.

Generic programming [36] offers a solution: Many operations on data structures can be implemented in terms of a few basic operations. For finite sets, e.g., the empty set $\{\}$, insertion *insert*, deletion *delete*, membership test $\in$, and folding *fold* suffice. For instance, set difference can be expressed as $A - B = \mathit{fold}\ \mathit{delete}\ A\ B$.

Clearly, $A - B$ can be implemented more efficiently. For example, the following takes the size of the sets (notation $|\ |$) into account—if B contains more elements than A, the resulting set is built from the empty set by inserting the remaining elements rather than removing B's from A.

$$A - B = \textbf{if}\ |B| > |A|\ \textbf{then}\ \mathit{fold}\ (\lambda x.\ \textbf{if}\ x \in B\ \textbf{then}\ \mathit{id}\ \textbf{else}\ \mathit{insert}\ x)\ \{\}\ A \\ \textbf{else}\ \mathit{fold}\ \mathit{delete}\ A\ B$$

Here, the **then** branch builds the result from the empty set by insertion. So, it is the generic implementation that must select the implementation for the result in this case. Of course, the same heuristics as in Section 3.2 apply. If both operands have the same implementation, homogeneity dictates that the result also uses this implementation (if possible). Otherwise, preferences guide the choice again.

Some combinations of data structures warrant a dedicated implementation for efficiency reasons. For example, if both A and B are implemented with red-black trees and of approximately equal size, one can implement $A - B$ in linear time following [1]. For modularity reasons, such special cases must not be coded into the generic implementation. Therefore, we set the preferences for operation selection such that generic algorithms are used only as a fall-back option when there is no specialisation. Each framework realizes generic programming and specialisation differently due to their different approaches; see Sections 4.5 and 5.7.

[5] A function f is left-commutative iff $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$ for all x, y, and z.

```
1  context fixes E :: ('v × 'v) set and src :: 'v and tgt :: 'v begin

2  definition dfs :: bool nres where
3    dfs = do {
4      (_, r) ← rec (λdfs (V, v).
5        if v = tgt then return (V, True)
6        else do {
7          let V = insert v V;
8          foreach (E `` {v}) (λ(V, brk). ¬ brk) (λv' (V, brk).
9            if v' ∉ V then dfs (V, v') else return (V, False)
10         ) (V, False)
11       })
12       ({}, src);
13     return r
14   }

15 end
```

Fig. 4: DFS written in the Monadic Refinement Framework

4 Autoref

The Autoref framework transfers an abstract program to its implementation by replacing abstract data types by concrete ones using ideas from relational parametricity (see Section 4.2). The result is (i) a concrete HOL program for which Isabelle/HOL can generate executable code, and (ii) a theorem that the concrete program correctly refines the original abstract one. Non-deterministic specifications are handled naturally thanks to parametricity.

4.1 Running DFS Example

The implementation of the DFS algorithm in Figure 4 lives in a context which fixes the edge relation E of the graph, the start node *src*, and the target node *tgt*. The algorithm *dfs* itself is defined in the non-determinism monad of the Monadic Refinement Framework [29] and follows the pseudo-code in Figure 2a.

First, *dfs* checks whether the current node v is the target, and if so, returns immediately (line 5). Otherwise, it adds the current node v to the visited set $\mathcal{V}$ (line 7) and invokes itself recursively for each of v's successors which have not yet been visited (line 9). The recursion is expressed with the recursion combinator **rec** (line 4) where the state consists of the set of visited nodes $\mathcal{V}$ and the current node v. The state of the **foreach** loop over the successors (line 8) consists of the set of visited nodes $\mathcal{V}$ and a break flag *brk* which interrupts the loop once the target node has been found. Recall from Section 3.1 that E `` $\{v\}$ denotes the set of v's successors.

Using the Monadic Refinement Framework, it is easy to show correctness:

lemma *dfs_correct:*
assumes $\forall v.\ (src,\ v) \in E^* \longrightarrow finite\ (E\ ``\ \{v\})$
shows $dfs \leqslant (\textbf{spec}\ r.\ r \longleftrightarrow (src,\ tgt) \in E^*)$

The assumption expresses that the reachable part of the graph is finitely branching, which ensures that the **foreach** loop is well-defined. The conclusion states that *dfs*

```
1  schematic_goal dfs_impl_refine_aux:
2    fixes succ_i and E :: ('a::linorder × 'a) set and src tgt :: 'a
3    assumes [autoref_rules]: (succ_i, E) ∈ ⟨Id⟩succg_rel
4    notes [autoref_rules] = IdI[of src] IdI[of tgt]
5    shows nres_of (?f::?'c dres) ≤ ⇓?R (dfs E src tgt)

6    unfolding dfs_def by autoref_monadic

7  concrete_definition dfs_impl for succ_i src tgt uses dfs_impl_refine_aux
8  prepare_code_thms dfs_impl_def
9  export_code dfs_impl checking SML OCaml? Haskell? Scala
```

Fig. 5: Refinement of DFS with Autoref

returns whether *tgt* is reachable from *src*. This is a partial correctness statement, i.e., the algorithm may not terminate. Note that the Isabelle code generator only ensures partial correctness for the generated code anyway. Yet, it is possible to prove termination of recursive definitions in the Monadic Refinement Framework. This guarantees that the code that stems from the recursive calls expressed in the Monadic Refinement Framework will terminate, but there are no guarantees that the overall generated code terminates, too.[6]

The next step synthesises an executable version of this algorithm by replacing the set data types by efficient implementations. Figure 5 shows how the Autoref tool is used for this task. We use a schematic goal statement to initiate the synthesis (line 1). The variables prefixed with *?* are instantiated during the proof. Thus, the proved theorem will have the shape indicated by the term after the **shows** keyword in line 5, where *?f::?'c dres* will have been replaced by the synthesized program and *?R* by the refinement relation. The **fixes** part in line 2 restricts the nodes to have a linear order. The **assumes** part (line 3) fixes the refinement for the edge relation: It is implemented by a successor function, which, for each node, returns a distinct list of successor nodes. The relation $\langle R\rangle$*succg_rel* relates successor functions to edge relations, given a relation R for the vertices of the graph. Here, we choose the identity relation, such that nodes are implemented by themselves.[7] (In detail, this part produces the theorems $(src, src) \in Id$ and $(tgt, tgt) \in Id$ and declares them to Autoref.) The program *?f::?'c dres* to be synthesised lives in a deterministic (executable) monad, and *nres_of* lifts it to the nondeterminism monad of the refinement framework, which is not executable.

We unfold the definition of *dfs* and invoke the *autoref_monadic* proof method in line 6. This instantiates *?f* by the concrete program and *?R* by the refinement relation and proves the theorem. The command **concrete_definition** defines a new constant for the synthesised program, where the argument order is specified in the **for**-clause, and the command **prepare_code_thms** converts the recursion combina-

[6] HOL has no notion of computation. It is therefore in principle impossible to formally prove termination of an algorithm shallowly embedded in HOL. One can only prove termination for the part of the computation that is (deeply) embedded into the monad.

[7] Autoref can also generate an implementation for an arbitrary polymorphic node type and refinement relation, if one provides a comparison operator. This general case is illustrated in the example theories accompanying this paper.

```
(succ_i, E) ∈ ⟨Id⟩succg_rel ⟹
nres_of
 (do {
    (_, r) ← rec (λdfs (V, v).
      if v = tgt then dRETURN (V, True)
      else
        let V = map2set_insert rbt_insert v V in
        foldli
          (succ_i v)
          (case_dres False False (λ(V, brk). ¬ brk))
          (λv' s. do {
            (V, brk) ← s;
            if ¬ map2set_memb (λk t. rbt_lookup t k) v' V
            then dfs (V, v') else dRETURN (V, False)
          })
          (dRETURN (V, False)))
      (rbt.Empty, src);
    dRETURN r
  })
≤ ⇓ bool_rel (dfs E src tgt)
```

Fig. 6: The refinement theorem generated by the proof in Figure 4

tor to recursive equations as required by the code generator. Finally, the command **export_code** generates code in several functional languages.

Figure 6 shows the refinement theorem proved by *autoref_monadic* (we replaced generated variable names by meaningful ones to make it more readable). The synthesised program uses a deterministic monad with three possible outcomes: a result *dRETURN x*, non-termination *dSUCCEED*, or assertion failure *dFAIL*. (Our program contains no run-time assertions.) The structure of the synthesised program is exactly the same as of the original program. Only some abstract operations have been replaced by concrete ones. For example, the membership query $v \in \mathcal{V}$ in the original program gets implemented as *map2set_memb* (λ*k t. rbt_lookup t k*) *v* $\mathcal{V}$.

Behind the scenes, the following has happened: First, the operation identification heuristics has identified a set membership operation. Second, the implementation type selection heuristics has decided to implement this set by a red-black tree. Third, the first synthesis step yielded an algorithm that uses the implementation data structures, but is still defined in the non-determinism monad. In Autoref, the set implementation by red-black trees is done via a generic algorithm, which converts map implementations with value type *unit* to set implementations. The generic algorithm for membership is called *map2set_memb*. As first argument, it takes the lookup operation of the map, in this case *rbt_lookup*. Fourth, a second synthesis step transfers the implementation from the non-determinism monad to the deterministic monad. The separation into two steps is done because the second step can also generate a plain HOL function (without any monad) provided that an unconditional termination proof is given for every non-tail recursion. These steps will be explained in more detail in Section 4.3, and are also illustrated in the example theories accompanying this paper.

The **concrete_definition** command extracts the synthesised code from the theorem as a new constant and generates the theorem *dfs_impl.refine*, which links the new constant to the original theorem:

$(succ_i,\ E) \in \langle succg\_rel \rangle list\_set\_rel$
$\longrightarrow nres\_of\ (dfs\_impl\ succ_i\ src\ tgt) \leqslant \Downarrow bool\_rel\ (dfs\ E\ src\ tgt)$

Combining this with the abstract correctness theorem *dfs_correct* from above, we get the following theorem:

theorem *dfs_code_correct:*
assumes $(succ_i,\ E) \in \langle Id \rangle succg\_rel$
shows case $dfs\_impl\ succ_i\ src\ tgt$ **of**
$dSUCCEED \Rightarrow True$
$\mid dRETURN\ r \Rightarrow r \longleftrightarrow (src,\ tgt) \in E^*$
$\mid dFAIL \Rightarrow False$

That is, the implementation may not terminate (yield *dSUCCEED*), but must not fail; if it returns a result r, this result indicates whether the target node is reachable.

4.2 Synthesis Based on Parametricity

In this section, we describe the theory behind the automatic refinement tool without going into the details of its implementation.

4.2.1 Relators

Relators offer a systematic way to construct relations for data refinement. We start with the simple case of trivial refinement and then consider increasingly complex relations. In general, an implementation and the corresponding conceptual type are related by a relator which takes relations as argument that describe the implementations for the type arguments of the conceptual type (e.g., a set type has unary relators and a map type has binary relators).

Some of Isabelle/HOL's data types should be implemented by themselves. For example natural numbers, integer numbers, and Booleans are supported directly by the code generator, and there is usually no need to refine them. Therefore, the refinement relations *nat_rel*, *int_rel*, and *bool_rel* are the identity relation on the corresponding types. For example, we write $(n_i,\ n) \in nat\_rel$ to express that n_i implements the natural number n (which is logically equivalent to $n_i = n$).

In the next paragraphs, we present the stepwise construction of a relator to implement sets by lists of disjoint elements, i.e., *distinct* lists. First, suppose we want to implement the elements of the set by themselves. We define a relation from lists to sets by

$list\_set\_rel\_aux = br\ set\ distinct$

where $br\ \alpha\ I = \{(c,\ a).\ I\ c \wedge a = \alpha\ c\}$ defines a refinement relation from an abstraction function and an invariant. Here, the abstraction function *set* converts lists to sets, and the invariant *distinct* ensures a distinct list. Note that not every list implements a set, nor does every set correspond to a list in this relation. The list $[1,1]$, e.g., is not distinct and thus does not implement a set. And the set $\{i.\ i > 5\}$ of natural numbers is infinite and thus cannot be represented by a list.

Consider the operation of inserting an element into a set. We define the implementation on lists by

insert_impl x l = (**if** $x \in$ *set l* **then** *l* **else** *x # l*)

and state (and prove) that *insert_impl* correctly implements inserting an element into a set represented by a distinct list by

$$\forall l\ s.\ (l,\ s) \in \mathit{list\_set\_rel\_aux} \longrightarrow (\mathit{insert\_impl}\ x\ l,\ \mathit{insert}\ x\ s) \in \mathit{list\_set\_rel\_aux}$$

This can be written more concisely as

$$(\mathit{insert\_impl},\ \mathit{insert}) \ \in\ \mathit{Id} \to \mathit{list\_set\_rel\_aux} \to \mathit{list\_set\_rel\_aux}$$

where *Id* is the identity relation and $\to$ denotes the *function relator* given by

$$R_1 \to R_2 = \{(f,\ g).\ \forall x\ y.\ (x,\ y) \in R_1 \longrightarrow (f\ x,\ g\ y) \in R_2\},$$

which combines a refinement relation R_1 for the argument and a refinement relation R_2 for the result into a refinement relation for the function. Note the syntactic similarity of the statement to the type declaration *insert :: 'a ⇒ 'a set ⇒ 'a set*.

Next, suppose we want to refine the elements of a list, but keep the list structure. For this purpose, we define *list_rel* to be the *natural relator on lists*. It lifts a relation for the elements to a relation on lists of the same length in which the corresponding elements are related. It is defined inductively by the following two rules:

$$([],\ []) \in \langle R\rangle \mathit{list\_rel} \qquad \frac{(x,\ y) \in R \qquad (xs,\ ys) \in \langle R\rangle \mathit{list\_rel}}{(x\ \#\ xs,\ y\ \#\ ys) \in \langle R\rangle \mathit{list\_rel}}$$

Now consider, e.g., the *append* operation. With respect to *list_rel*, it is implemented by itself, regardless of how the elements of the list are implemented. This fact is stated as

$$(\mathit{append},\ \mathit{append}) \in \langle R\rangle \mathit{list\_rel} \to \langle R\rangle \mathit{list\_rel} \to \langle R\rangle \mathit{list\_rel}$$

where R is a variable that can be instantiated with any relation, e.g., *list_set_rel_aux*. Again, note the similarity to the type *append :: 'a list ⇒ 'a list ⇒ 'a list*. Like for lists, a natural relator can be defined for any algebraic data type (sum of products), and this is even automated in Isabelle/HOL's data type package [6].

Finally, suppose we want to implement sets by distinct lists with some refinement relation R for the elements of the set. The appropriate refinement relation can be constructed as the composition of the relations described above: First, we use the natural relator on lists to map from a list of concrete elements to a list of abstract elements. Then, we map the list of abstract elements to a set. Formally, ($\bigcirc$ denotes relation composition)

$\langle R\rangle \mathit{list\_set\_rel} = \langle R\rangle \mathit{list\_rel} \bigcirc \mathit{list\_set\_rel\_aux}$

For the singleton set operation $\lambda x.\ \{x\}$, e.g., $(\lambda x.\ [x],\ \lambda x.\ \{x\}) \in R \to \langle R\rangle \mathit{list\_set\_rel}$ holds. The situation is more subtle for the refinement between *insert_list* and *insert*, as equality on abstract elements does not imply equality on concrete elements. For example, in a set of sets implemented by a list of lists, the two lists [*1,2*] and [*2,1*] both represent the same inner set {*1,2*}. To this end, we specify a concrete equality operation *eq* with $(\mathit{eq},\ \mathit{op}\ =) \in R \to R \to \mathit{bool\_rel}$ and generalise *insert_list* to *glist_insert :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a list ⇒ 'a list*, which takes the equality

operation as an argument. It inserts the element only if it is not *eq*-equal to any element in the list. Then, we can prove the conditional refinement theorem

$$\frac{(eq,\ op\ =) \in R \to R \to bool\_rel}{(glist\_insert\ eq,\ insert) \in R \to \langle R\rangle list\_set\_rel \to \langle R\rangle list\_set\_rel} \quad \text{(insert)}$$

where *op* turns infix operations into prefix operations, i.e., *op* = means $\lambda a\ b.\ a = b$.

Similarly, most other set operations require an equality operation for the element type. Note that an equality operation exists for a refinement relation R iff R is single-valued, i.e., every concrete value refines at most one abstract value. This makes $\langle R\rangle$*list_set_rel* more intuitive, as it excludes strange effects such as a non-distinct list of concrete elements refining a distinct list of abstract elements.

4.2.2 Synthesis Based on Relators

Next, we consider synthesis for a HOL term t which consists of constants, variables, function application, and lambda abstraction. To that end, we assume that for each constant or free variable c in t, we are given a parametricity rule of the form $(c', c) \in R$, where the implementation c' is related to c by the refinement relation R. Using these parametricity rules and the following rules for application and abstraction, we can derive a refinement theorem of the form $(t', t) \in R$.

$$\frac{(f', f) \in R_1 \to R_2 \quad (x', x) \in R_1}{(f'\ x', f\ x) \in R} \quad \text{(app-refine)}$$

$$\frac{\forall x\ x'.\ (x', x) \in R_1 \longrightarrow (t', t) \in R_2}{(\lambda x'.\ t', \lambda x.\ t) \in R_1 \to R_2} \quad \text{(abs-refine)}$$

If we select consistent refinement rules for the constants and free variables of t, we can even synthesise the term t' and the relator R by applying these rules according to the syntactic structure of t.

4.2.3 Side Conditions

Not all constants are parametric, though. For example, the function *hd :: 'a list* $\Rightarrow$ *'a* returns the first element of a non-empty list. We would like to have the theorem $(hd,\ hd) \in \langle R\rangle list\_rel \to R$. But we cannot prove $(hd\ [],\ hd\ []) \in R$, because the function *hd* is specified solely by the equation $hd\ (x \#\ xs) = x$, which does not say anything about *hd* []. In fact, *hd* is only parametric for non-empty lists, i.e., we have the conditional parametricity rule

$$\frac{l \neq [] \qquad (l',\ l) \in \langle R\rangle list\_rel}{(hd\ l',\ hd\ l) \in R}$$

In general, we can prove a function parametric only for those arguments for which their HOL specification uniquely determines the result. When using such a rule during synthesis, the side condition has to be discharged.

Moreover, there can be multiple rules for the same operation and the same implementation type, which only differ by the side condition. This makes it possible to base the choice of operations on side conditions. Consider, e.g., the insertion operation on sets implemented by distinct lists. If we can prove that the inserted element

is not in the set, the insertion can be implemented in constant time by *cons*ing. Otherwise, we have to use the linear-time operation *glist_insert*, which checks whether the element is already in the list, as expressed by the general rule (insert) above. The specialised rule with a side condition is

$$\frac{x \notin s \qquad (x',\ x) \in R \qquad (l,\ s) \in \langle R \rangle list\_set\_rel}{(x' \ \#\ l,\ insert\ x\ s) \in \langle R \rangle list\_set\_rel}$$

There are also synthesis rules that provide additional information to subsequent synthesis steps. For example, when synthesising the then-branch of an if-then-else expression, we may assume that the condition is true. Similarly, for the else-branch, we may assume that the condition is false. Thus, for if-then-else, we use the following congruence synthesis rule:[8]

$$\frac{(b',\ b) \in bool\_rel \qquad b \implies (t',\ t) \in R \qquad \neg b \implies (e',\ e) \in R}{(\textbf{if}\ b'\ \textbf{then}\ t'\ \textbf{else}\ e',\ \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ e) \in R}$$

This way we can handle terms like **if** l=[] **then** *0* **else** *hd l*, where the precondition needed for the refinement of *hd* is provided by the refinement of the enclosing if-then-else term.

Choosing which rules are applied if there are multiple possibilities is subject to some heuristics in the Autoref tool, which are described in Section 4.3.

4.2.4 Generic Algorithms

A precondition of a rule may also trigger the synthesis of a refinement for a constant which does not occur in the original term. In the previous section, the rule for inserting an element into a set implemented by a list triggers the synthesis of a refinement for the equality operator. Along the same lines, arbitrary generic algorithms can be defined. For example, the subset-or-equal operation can be implemented by bounded quantification and a membership operation, as expressed by the following rule:

lemma *generic_subset_by_ball_and_mem:*
fixes $R :: ('c \times 'a)\ set$
assumes $(ball_i,\ Ball) \in \langle R \rangle Rs_1 \to (R \to bool\_rel) \to bool\_rel$
assumes $(mem_i,\ op \in) \in R \to \langle R \rangle Rs_2 \to bool\_rel$
shows $(\lambda s_{1i}\ s_{2i}.\ ball_i\ s_{1i}\ (\lambda x.\ mem_i\ x\ s_{2i}),\ op \subseteq)$
$\in \langle R \rangle Rs_1 \to \langle R \rangle Rs_2 \to bool\_rel$

Note that this rule works for all set implementations where bounded quantification can be synthesized for the first set (implemented by relator Rs_1), and a membership operation can be synthesized for the second set (implemented by relator Rs_2).

[8] We call such rules congruence synthesis rules, because they generalise both the parametricity theorem (**if** _ **then** _ **else** _, **if** _ **then** _ **else** _) $\in bool\_rel \to R \to R \to R$ and the congruence rules known from contextual rewriting:

$$\frac{b = b' \qquad b \implies t = t' \qquad \neg b \implies e = e'}{(\textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ e) = (\textbf{if}\ b'\ \textbf{then}\ t'\ \textbf{else}\ e')}$$

4.3 The Automatic Refinement Tool

In the last subsection, we have presented the basic principles of parametricity-based synthesis. Here, we describe the implementation of these ideas in the Autoref tool.[9] Given a term over abstract data types as input, Autoref synthesises a corresponding term over implementation data types and the refinement theorem relating the two terms. The synthesis is done in three phases:

Operation Identification First, a heuristics tries to identify the conceptual types that occur in the abstract term (cf. Section 3.1). The term is rewritten such that every abstract operation is represented by a single constant. Every such constant can belong to only one conceptual type.

Implementation Selection The second phase selects implementation types for the conceptual types. The goal is to find a consistent selection of implementation types such that all required operations are available for these types. The result of this phase is an annotation of the abstract term that indicates over which concrete types to implement each operation.

The actual selection of implementation types is influenced by various heuristics (cf. Section 3.2) and configuration options specified by the user. For example, the user may specify **declare** *ty_REL*[**where** *R*=⟨*nat_rel*⟩*dflt_rs_rel, autoref_tyrel*] to instruct Autoref to implement sets of natural numbers by red-black trees, which are related to sets by the relator *dflt_rs_rel*. Note that such hints may be overridden by other heuristics, e.g., the homogeneity principle.

Translation The third phase transfers the operations from the abstract data types to the selected implementation types. In general, it has to select between multiple concrete operations available for the indicated implementation types. For this, it takes into account whether side-conditions can be proved, operations required by generic algorithms can be synthesised, and it prefers specialised operations over generic ones.

Example 1 Consider the synthesis of a successor operation on graphs. We set up the following synthesis goal:

schematic_goal
  fixes *succ :: nat ⇒ nat list* **and** *E :: (nat×nat) set* **and** *v :: nat*
  assumes [*autoref_rules*]*:* (*succ, E*) ∈ ⟨*nat_rel*⟩*succg_rel*
  notes [*autoref_rules*] = *IdI*[**of** *v*]
  shows (*?f::?'c, E `` {v}*) ∈ *?R*

We consider a graph E over natural numbers which is implemented by the successor function *succ* as indicated by the relator *succg_rel*. The node v is implemented by itself, which is declared to Autoref by the theorem *IdI*[**of** *v*], which expands to $(v, v) \in$ *nat_rel*. We want to synthesise an implementation for the abstract term E `` $\{v\}$.

First, the operation identification phase rewrites the term to a successor operation and inserts conceptual type annotations. The term E `` $\{v\}$ in the goal is rewritten to:

[9] The *autoref_monadic* method that we invoked in the example in Section 4.1 is a wrapper for the Autoref tool to be used with the Monadic Refinement Framework [29]. After the actual synthesis, it performs some deforestation optimisations and transfers the resulting program from the non-determinism monad of the Refinement Framework to a deterministic program for which code can be generated. Here, we describe the actual Autoref tool, which is independent from the Monadic Refinement Framework.

$(OP\ op\_succ :::_i \langle i\_nat\rangle i\_graph \to i\_nat \to \langle i\_nat\rangle i\_set)\ \$$
$(OP\ E :::_i \langle i\_nat\rangle i\_graph)\ \$\ (OP\ v :::_i i\_nat)$

For technical reasons (to control Isabelle's higher-order unification algorithm), function application is rewritten to an explicit constant $\$$, which is defined as $f\ \$\ x \equiv f\ x$, and operations are tagged with the constant $OP\ x \equiv x$. Each operation is annotated by its conceptual type (notation $:::_i$). Here, the operation identification heuristics has identified E to be of conceptual type $\langle i\_nat\rangle i\_graph$: this was inferred from the declaration $(succ,\ E) \in \langle nat\_rel\rangle succg\_rel$ and the fact that $succg\_rel$ is a relator for graphs.

Second, the implementation selection phase infers a consistent annotation of relations. The term is further rewritten to:

$(OP\ op\_succ ::: \langle nat\_rel\rangle succg\_rel \to nat\_rel \to \langle nat\_rel\rangle list\_set\_rel)\ \$$
$(OP\ E ::: \langle nat\_rel\rangle succg\_rel)\ \$\ (OP\ v ::: nat\_rel)$

moreover, the variable $?R$ is instantiated to $\langle nat\_rel\rangle list\_set\_rel$. Here, the conceptual type annotations have been replaced by compatible relator annotations (notation $:::$). None of the heuristics had to be applied here: a successor function is the only known implementation for E, so there is only one consistent annotation.

Finally, the translation phase instantiates $?f ::?'c$. The proved theorem is:

$(succ,\ E) \in \langle nat\_rel\rangle succg\_rel \longrightarrow (succ\ v,\ E\ ``\ \{v\}) \in \langle nat\_rel\rangle list\_set\_rel$

In the next subsections, we describe the three phases of Autoref in greater detail.

4.4 Identification of Operations

To identify the conceptual types and operations in a given term, Autoref uses type elaboration. The idea is to rewrite non-atomic subterms that might represent an operation to a single constant, and backtrack over rewriting until the term becomes typeable.

Formally, a conceptual type T is either a variable or a constructor applied to conceptual type arguments:

$$T ::= V \mid \langle T,\ldots,T\rangle\ C$$

The notation $c :_i T$ expresses that the constant c has the conceptual type T. For example, the conceptual type constructor for graphs is i_graph and the successor operation has the type $op\_succ :_i \langle V\rangle i\_graph \to V \to \langle V\rangle i\_set$, where V is the type variable for vertices and $\to$ and i_set denote the conceptual function and set type, respectively. In general, conceptual type constructors correspond to HOL type schemas, but not type constructors. Thus, conceptual types are more abstract than HOL types. For example, $\langle V\rangle i\_graph$ corresponds to the HOL type $('v \times 'v)\ set$ if V corresponds to $'v$. Hence, the conceptual type constructor i_graph itself corresponds to the HOL type schema $(\_ \times \_)\ set$.

Identification of conceptual operations uses a set of rewrite rules of the form $pat \equiv c\ x_1 \ldots x_n$, where c is a constant and the x_i are variables that also occur in pat. For the successor operation in the graph example, the rule $E\ ``\ \{x\} \equiv op\_succ\ E\ x$ replaces the HOL encoding $E\ ``\ \{x\}$ with the constant op_succ.

The actual operation identification is done by solving a type elaboration problem according to the rules in Figure 7. A type elaboration of the form $\Gamma \vdash t \rightsquigarrow t' : T$

$$ctxt\colon \frac{x :_i T \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : T} \qquad rew\colon \frac{t \equiv t' \quad \Gamma \vdash t' \rightsquigarrow t'' : T}{\Gamma \vdash t \rightsquigarrow t'' : T} \qquad const\colon \frac{c :_i T}{\Gamma \vdash c \rightsquigarrow c : T}$$

$$app\colon \frac{\Gamma \vdash t \rightsquigarrow t' : T_1 \quad \Gamma \vdash f \rightsquigarrow f' : T_1 \to T_2}{\Gamma \vdash f\ t \rightsquigarrow f'\ t' : T_2} \qquad abs\colon \frac{(x :_i T_1)\Gamma \vdash t \rightsquigarrow t' : T_2}{\Gamma \vdash (\lambda x.\ t) \rightsquigarrow (\lambda x.\ t') : T_1 \to T_2}$$

Fig. 7: Type elaboration rules for identifying operations and conceptual types

means that in context Γ, the term t can be rewritten to t' such that t' has type T. The rules are standard except for the *rew* rule, which, for a rewrite rule $t{\equiv}t'$, matches the current term against the left hand side t, replaces it by the right hand side t', and then elaborates the new term.[10] The elaboration algorithm first tries to apply a *rew* rule. Only if the matching *rew* rules do not lead to a valid typing, it backtracks to use the *ctxt*, *const*, *app*, or *abs* rules.

As Autoref's algorithm tries rewriting before the standard rules for abstraction and application, it will find an operation of a conceptual type for which a rewrite pattern is set up before it decomposes the compound HOL term into functions and arguments. Thus, if only operations compatible with a conceptual type are used, this type will be inferred, instead of the more elementary function type.

Example 2 Consider the following conceptual typing facts and patterns, which are part of Autoref's default setup for maps. (We omit the tags *OP* and $ for readability.)

op_map_empty $:_i$ $\langle K,V\rangle$*i_map*
op_map_lookup $:_i$ $K \to \langle K,V\rangle$*i_map* $\to \langle V\rangle$*i_option*
$\lambda\_.$ *None* $\equiv$ *op_map_empty*
m k $\equiv$ *op_map_lookup k m*

Moreover, consider the following synthesis setup:

schematic_goal
fixes m_i *::* (*nat* $\times$ *bool*) *list* **and** *m :: nat* $\Rightarrow$ *bool option* **and** *k :: nat*
assumes [*autoref_rules*]*:* (m_i*,* *m*) $\in \langle$*nat_rel,bool_rel*$\rangle$*list_map_rel*
notes [*autoref_rules*] *= IdI*[**of** *k*]
shows (*?f::?'c, m k*) $\in$ *?R*

Here, *list_map_rel* is the relator to refine maps by association lists. From the declarations, Autoref infers the additional typing facts *m* $:_i$ $\langle$*i_nat, i_bool*$\rangle$*i_map* and *k* $:_i$ *i_nat*. Therefore, on operation identification, the term *m k* is first rewritten to *op_map_lookup k m*, and this yields a consistent typing.

Now, consider the term $t \equiv$ **let** *m* $=\lambda x.$ *Some* (*x + 1*) **in** *m 3*. The rewrite rule for *op_map_lookup* matches in many positions, but none of these rewrites leads to a consistent typing. For example, consider the subterm *Some* (*x+1*). It could be rewritten to *op_map_lookup* (*x+1*) *Some*. However, this term is untypeable, as there is no rule to type *Some* as *i_map*. Thus, the heuristics backtracks, and *m* gets the conceptual type $K \to V$ *i_option*. Similarly, *m 3* gets rewritten to *op_map_lookup 3 m* first, before backtracking and interpreting *m 3* as function application.

In practice, this heuristics works quite well. In the rare cases when it fails, the term can be rewritten manually or annotations can be added. For example, the term $\lambda x.$ (*None* $:::_i$ $\langle A\rangle$*i_option*) gets typed as a function.

[10] We assume that application of a rule involves unification, as is standard in Isabelle. Thus, we do not explicitly indicate matching in the rules.

4.5 Selection of Implementations

After the conceptual types have been identified and the term has been rewritten such that each operation is represented by a single constant, the Autoref tool tries to find implementations of the conceptual types such that every operation can be realised. As described in Section 4.2, the synthesis is done by decomposing the abstract term using the refinement rules for application and abstraction as well as refinement rules for the constants and free variables. Typically, there are several refinement rules for a constant (e.g., there are multiple set implementations, and each of them comes with a refinement rule for the membership operation.). But only if we choose the refinement rules consistently within the abstract term, synthesis will succeed.

The implementation selection phase annotates each constant in the abstract term with a relation that restricts the applicable refinement rules to those that use the implementations indicated by the annotated relation. For example, if we annotate $op \in$ by $nat\_rel \to \langle nat\_rel \rangle list\_set\_rel \to bool\_rel$, we fix the implementation of the set to distinct lists and the implementation of the elements to natural numbers. The synthesis phase then tries only refinement rules compatible with this annotation.

Even with annotations, there may still be several compatible rules, e.g., a generic and a specialised algorithm for the same operation. The implementation selection phase only ensures that the relations are consistent, and that there is at least one implementation for the selected relations. While it considers operations introduced by generic algorithms, it ignores semantic side conditions. Thus, the final synthesis phase may fail due to unsolvable semantic side conditions. This incompleteness is a design decision. It helps to separate the phases and reduces the search space when backtracking, which makes the Autoref tool faster on large terms. We have not encountered practical problems with this incompleteness, as the semantic side conditions are usually the same for all implementations. In order to force a specific implementation, the user can add relator annotations to the term, which force the annotated subterms to be implemented with the specified relators.

The operation selection heuristics is, again, modelled as a type inference problem. From the given term, we first generate a set of initial typing constraints. For example, reconsider the term $m\ k$ from Example 2. The initial constraints are the following, where variables prefixed with $?$ can be instantiated later:

$$m : \langle ?R\_nat_1,\ ?R\_bool_1 \rangle ?R\_map$$
$$k : ?R\_nat_2$$
$$op\_map\_lookup : ?R\_nat_2 \to \langle ?R\_nat_1,\ ?R\_bool_1 \rangle ?R\_map \to \langle ?R\_bool_2 \rangle ?R\_option$$

The structure of the relations is derived from the conceptual type annotations from the operation identification phase. Note that the same conceptual type may be refined differently. For example, the Boolean in the map and the Boolean in the result of *op_map_lookup* get assigned different relator variables initially ($?R\_bool_1$ vs. $?R\_bool_2$). Similarly, the relator variable for k differs from the variable for the keys in the map.

The initial constraints are now processed by different heuristics, with the goal of assigning concrete relators to the variables. These quite complicated heuristics have evolved over many iterations of implementing and improving the framework and have proven useful in practice.

4.5.1 Homogeneity Rules

The first heuristics processes the constraints and tries to unify each constraint with homogeneity rules. There is a list of homogeneity rules, and the heuristics picks the first one that unifies, and instantiates the variables of the constraint accordingly. For example, there is the homogeneity rule

$op\_map\_lookup : ?Rk \to \langle ?Rk,\ ?Rv\rangle ?Rm \to \langle ?Rv\rangle option\_rel$

This unifies with the third constraint and we get the instantiation

$?R\_nat_2 \mapsto ?R\_nat_1 \quad \text{and} \quad ?R\_bool_2 \mapsto ?R\_bool_1 \quad \text{and} \quad ?R\_option \mapsto option\_rel$

Note that this heuristics limits the possible solutions, and even may render a solvable set of constraints unsolvable. However, in practice, we have not yet encountered problems with this heuristics, as there are typically rules for the homogeneous case. If there should occur problems in future extensions of the tool, the heuristics could easily be modified to be less aggressive, for example to apply homogeneity rules only if there remain applicable rules for the constraint.

4.5.2 Anti-Unification

The second heuristics instantiates the variables as far as no solutions are lost. Formally, this is realised by anti-unification [41] of the constraint with all rules declared to Autoref, i.e., the RHS of the constraint is replaced by the most specific instance that unifies with all rules it unified with before. This heuristics, obviously, does not limit the search space. However, it usually instantiates the relations for the primitive types, as well as the relations for parameter variables (like m and k). This is important for the next heuristics. In our running example, all remaining relator variables get instantiated, as they occur in the constraints for m or k. Thus, the constraint system now is:

$m : \langle nat\_rel,\ bool\_rel\rangle list\_map\_rel$
$k : nat\_rel$
$op\_map\_lookup : nat\_rel \to \langle nat\_rel,\ bool\_rel\rangle list\_map\_rel \to \langle bool\_rel\rangle option\_rel$

4.5.3 Type-Based Heuristics

The next heuristics tries to implement specific abstract types with specific relations. Its input is a set of *hint relations*, specialised to a specific abstract type. For example, the hint relation $\langle nat\_rel,\ ?Rv\rangle dflt\_rm\_rel$ indicates that maps from natural numbers shall be implemented by red-black trees with the default ordering ($\leqslant$).

The type-based heuristics tries to instantiate all relation variables in the constraint system whose abstract type matches a hint relation. Thus, an abstract type whose implementation was not fixed before (i.e., by explicit relation annotation, anti-unification, or homogeneity), may get instantiated according to the hints. Clearly, this heuristics may lose solutions. Assume, e.g., that anti-unification had not fixed *?R_map* in our running example. Then, the above-mentioned hint relation would instantiate it to *dflt_rm_rel*. This would render the constraint system unsolvable.

Typically, however, this heuristics causes no problems, as the types that are not instantiated by the previous heuristics do not depend on the parameters. However,

hint relations need to be carefully designed.[11] Again, this heuristics could be made less aggressive by not performing instantiations that lose all solutions to a constraint.

4.5.4 Solving

Finally, the constraints are solved based on the available rules: Each constraint $c ::: R$ must be instantiated by a relation R' such that R' is an instance of R, and such that there is a rule of the form $(c', c) \in R'$. Moreover, if the rule requires some new constants to be synthesised, there must be rules for these constants, too.

We try to solve the constraints by applying all possible rules with backtracking until a consistent solution is found. The rules are tried w.r.t. a *priority ordering*, which is a lexicographic ordering of the following components:

1. The *major priority* set by the user. The major priority is an integer number and defaults to 0.
2. The *homogeneity* of the rule, which is the number of different relators. Rules with fewer relators are preferred.
3. The *relator priority* is the sum of the priorities of the relators occurring in the rule. Relators can be arranged into an ordered list by the user, and the relator's priority is the position of the relator in the list. It defaults to zero for relators not in the list.
4. The *minor priority* set by the user. Again, the minor priority is an integer number and defaults to 0.

The current version of the collection framework does not use the major priority. Adjustments to the selected implementations are made via the relator priority. For example, the relator for implementing sets by red-black trees has a higher priority than the one for distinct lists. The distinction between generic algorithms, default implementation, and specialised implementations of an operation is made via the minor priority.

For example, a generic algorithm for the union of two sets may iterate over one set, and insert the elements into the other set. On distinct lists, we may want to implement the union of two sets directly. Moreover, if we can prove that the two sets are disjoint, union can be implemented by simply concatenating the lists. To implement these preferences, we set the priorities as follows: The generic set-union algorithm has a minor priority of -10, the default algorithm for union on distinct lists has a minor priority of 0, and the specialised algorithm for the case that the sets can be proved to be disjoint has a minor priority of 10. Note that the homogeneity and relator priority of the default and specialised algorithms are the same, and the homogeneity and relator priority of the generic algorithm is not higher: The generic algorithm supports different implementations for the two sets, so its homogeneity is smaller. Moreover, the relators in the generic algorithm are variables, which have no priority. Thus, its relator priority is lower.

In our running examples, there is only a unique solution with the available rules: There is only a single rule for m and k, and there is only a single rule to implement lookup on red-black trees.

[11] Actually, the default setup of the Collections Framework only hints at sets and maps of natural numbers to be implemented by red-black-trees, and sets of Booleans to be implemented by lists. Further hints are usually added locally for specific applications.

4.6 Translation Phase

The previous phases produce an abstract term with consistent relation annotations for every constant and free variable. The implementation selection has tried to ensure that there are actually rules to do the desired translation. To that end, it has considered the available rules and their prerequisite operations, but not the side conditions. Thus, translation may fail even if the previous phases were successful.

The translation phase recursively resolves the term with the available refinement rules: Abstraction is resolved by the default rule (abs-refine) from Section 4.2.2. An annotated constant is resolved by the first matching rule according to priority ordering for which all side conditions can be discharged and all prerequisite operations can be synthesised. Here, priority ordering ensures that specialised rules are tried before generic rules. As discussed in Section 4.2.3, there are also refinement rules that match on a constant with its arguments. Those rules are also applied in priority ordering. Only if no such rule can solve the subgoal, the default rule (abs-refine) is applied.

Moreover, the order in which the side conditions are solved does matter. Most side conditions are over the abstract term, e.g., we may require that the two sets be disjoint. Such side conditions can be solved before the synthesis of the operands is started. However, some rules impose side conditions on the synthesised term. For example, refinement for the recursion combinator only works if both, the abstract and the synthesised concrete function bodies are monotone. These side conditions must be discharged after the synthesis of the operands (the concrete function body in our example). Therefore, we tag side conditions to denote when they should be solved: The tags *PREFER* and *DEFER* on a side condition indicate that it is to be solved before or after the synthesis of operands.

5 Containers

The Containers framework exploits existing refinement capabilities of Isabelle's code generator [14, 16] (Section 5.2 introduces the necessary background). Consequently, this approach requires less manual work and dedicated automation than Autoref, as the refinement is done outside of the logic. However, it is not as expressive, because the approach is restricted to the limited refinement capabilities of the code generator.

Overall, the Containers framework is a usage pattern of Isabelle's code generator. So Containers does not need a large-scale package implementation like Autoref; a sequence of Isabelle declarations and some small tactics suffice. In this section, we explain the usage patterns and how they are used systematically.

5.1 Depth-First Search Declaratively

As before, we will refer to the DFS example to illustrate the concepts. First, we have to define DFS from Figure 2a as a *function* in Isabelle/HOL.[12] This is not straightforward, as we want to preserve the underspecification in the **foreach** loop. In detail, the pseudo-code in Figure 2a mutates state, namely the set of visited nodes.

[12] A relational specification does not suffice, as the code generator can handle only functions.

```
1  datatype 'a dfs_result = Reachable | Visited ('a set)

2  context fixes E :: ('a × 'a) set begin

3  definition dfs :: 'a ⇒ 'a ⇒ 'a set ⇒ 'a dfs_result where
4    dfs v tgt V =
5    (if tgt ∈ (E ↑ −V)* `` {src} then Reachable
6     else Visited (V ∪ (E ↑ −V)* `` {src}))

7  lift_definition dfs_body :: 'a ⇒ ('a, 'a dfs_result) comp_fun_commute
8    is λtgt v r. case r of Reachable ⇒ Reachable
9                        | Visited V ⇒ if v ∈ V then Visited V else dfs v tgt V
10   by standard (auto simp: dfs_def ...)

11 lemma dfs_code:
12   dfs v tgt V =
13     (if v = tgt then Reachable
14      else let S = E `` {src} in
15        if finite S then fold (dfs_body tgt) (Visited (insert v V)) S
16        else abort ''Infinite successor relation'' (λ_. dfs v tgt V))
17   ⟨proof⟩

18 definition reachable :: 'a ⇒ 'a ⇒ bool where reachable src tgt ⟷ (src, tgt) ∈ E*

19 lemma reachable_dfs: reachable src tgt = (dfs src tgt {} = Reachable)  ⟨proof⟩
20 end
```

Fig. 8: Declarative definition of reachability and DFS and derivation of the recursion equation

In HOL, we usually model the mutation as a state transformer, i.e., a function *dfs* of type *'a ⇒ 'a set ⇒ bool × 'a set* where the state of type *'a set* collects the visited nodes. With this approach, the function returns the set of visited nodes, which is non-deterministic due to the underspecification (as discussed in Section 3.3). A direct translation to an HOL function would therefore have to resolve this non-determinism.

Fortunately, a careful analysis reveals that the resulting state is only needed when the target node has not yet been reached during the call. In that case, however, the set of visited nodes is unique: *dfs v* $\mathcal{V}$ extends the visited nodes $\mathcal{V}$ with all the nodes that are reachable from the current node v without taking any edge to a node in $\mathcal{V}$. Thus, we can specify the depth-first search *declaratively* – without loops and recursion (see Figure 8). The idea is to return the set of visited nodes only if the target node has not been found. The type *'a dfs_result* formalises this insight (line 1). *Reachable* abstracts from the (underspecified) state $\mathcal{V}$ in the result (*True,* $\mathcal{V}$) of the state transformer, and *Visited* $\mathcal{V}$ corresponds to (*False,* $\mathcal{V}$).

For the following, we fix a graph given by a set E of edges, i.e., pairs of nodes. We write $E \upharpoonright A$ for the subgraph of E whose edges all end in a node in the set A. Thus, if the target *tgt* is reachable from v in the subgraph $E \upharpoonright (-\mathcal{V})$, which avoids all nodes in $\mathcal{V}$, then *dfs v tgt* $\mathcal{V}$ returns *Reachable* (line 5). Otherwise, the depth-first search terminates after having visited all nodes reachable from v without going through a node in $\mathcal{V}$. Hence, *dfs v tgt* $\mathcal{V}$ adds these nodes to $\mathcal{V}$ and returns them (line 6).

Clearly, this declarative definition is far from the algorithmic pseudo-code in Figure 2a. Fortunately, Isabelle can generate code from any equational theorem, not

only the defining equation. In lemma *dfs_code*, we show that our declarative definition satisfies the recursive specification which formalizes the pseudo-code from Figure 2a.

Two points are worth noting here. First, the loop construct *fold* is only well-defined if the set S is finite. However, the type (*'a* × *'a*) *set* also allows nodes with infinitely many successors. Therefore, the equation includes an additional check for finiteness in line 15; if it fails, the generated code will raise an error using the function *abort*.[13] Note that the finiteness check is just another operation on the conceptual type *'a set*.[14] In comparison to Autoref (Section 4.1), the run-time check replaces the logical finiteness assumption in Autoref's correctness theorem.

Second, recall from Section 3.3 that the *fold* function over finite sets is only well-defined for left-commutative functions. To that end, the Containers framework [32] introduces the type (*'a, 'b*) *comp_fun_commute* of left-commutative functions of type *'a* ⇒ *'b* ⇒ *'b*, and implements the *fold* operation for such functions following Nipkow and Paulson [38]. Consequently, the body *dfs_body* of the loop must be expressed as a value of type (*'a, 'a dfs_result*) *comp_fun_commute* (lines 7–9). This involves a proof that the body is in fact left-commutative (line 10). Thanks to *dfs*'s declarative definition, the proof is automatic.

Finally, we show that reachability can be implemented in terms of *dfs* (line 19). This follows directly from the declarative definition of *dfs*.

5.2 Background on Isabelle's Code Generator

Isabelle's code generator [14, 16] turns a set of equational theorems into a functional program with the same equational rewrite system. The translation guarantees partial correctness by construction, as one could simulate every execution step in the functional language by rewriting with the corresponding equational theorem in the logic. Thus, every theorem also holds for any terminating execution of the code. Conversely, in case of non-termination or termination with an exception, no guarantees are provided.

Since only equations matter (but not definitions or termination proofs), users can refine programs and data without affecting their formalisation globally. Program refinement separates code generation issues from the rest of the Isabelle formalisation. As any (executable) equational theorem suffices for code generation, the user may derive new (code) equations to use upon code generation (e.g., lemma *dfs_code* in Figure 8). In particular, by using *abort*, the equation may explicitly terminate the program with an error.

For data refinement, the user declares arbitrary constants to be the (pseudo-) constructors of a type and then derives equations that pattern-match on these (pseudo-)constructors. Neither need the (pseudo-)constructors be injective and pairwise disjoint, nor exhaust the type. Again, this is local as it affects only code generation, but not the logical properties of the refined type. Thus, one cannot exploit inside the logic the type's new structure for code generation. In Section 5.4, we use

[13] In the logic, the function *abort* is defined as *abort msg f* = *f* (). This ensures that the error cases in code equations can be proved by reflexivity. The code generator implements *abort* with an exception with the given message.

[14] We will later select an implementation for S which can only represent finite sets. Hence, the finiteness check will take at most constant time or be eliminated completely by the target language compiler.

```
1  typedef 'a graph = UNIV :: ('a × 'a) set set
2  setup_lifting type_definition_graph

3  definition successors :: ('a × 'a) set ⇒ 'a ⇒ 'a set where successors E x = E `` {x}
4  declare successors_def[containers_post, symmetric, containers_pre]

5  lift_definition succs :: 'a graph ⇒ 'a ⇒ 'a set is successors
6  lift_definition reachable_impl :: 'a graph ⇒ 'a ⇒ 'a ⇒ bool is reachable
7  lift_definition dfs_impl :: 'a graph ⇒ 'a ⇒ 'a ⇒ 'a set ⇒ 'a dfs_result is dfs
8  lift_definition dfs_body_impl :: 'a graph ⇒ 'a ⇒ ('a, 'a dfs_result) comp_fun_commute
9    is dfs_body

10 lemmas [containers_identify, code] = reachable_dfs dfs_code dfs_body.rep_eq
```

Fig. 9: Definition of the conceptual type *graph* with successor operation and identification of operations

data refinement to link conceptual types and their implementations. For example, the abstraction function *RSet :: 'a rbt ⇒ 'a set* from red-black trees to sets serves as a pseudo-constructor.

5.3 Identification of Operations

Having formalised the algorithm, we can now start to add efficient data structures. The first step is to identify the conceptual types and their operations.

In the Containers framework, every conceptual type must be represented by one type constructor in the logic. This restriction comes from the code generator: the data structures for the conceptual types will be connected via data refinement (see Section 5.4), which only works for type constructors [14]. Hence, a new type must be introduced if this requirement is not met. In the DFS example, there are two conceptual types: sets and graphs. Sets already have their own type constructor *set*, but graphs are modelled as the composite type *('a × 'a) set*. Thus, we define the type *'a graph* as a copy of *('a × 'a) set* (Figure 9, line 1).

Accordingly, operations of conceptual types must be constants and operate on the conceptual type. It is primarily the responsibility of the user to ensure that the program uses these operations. The lifting and transfer package [18] can assist in this task as follows.

In the first step, we define new temporary constants for all operations that consist of composite terms. In the DFS example, the code equation in Figure 8 only uses a successor operation on graphs. Since this is expressed as a composite term E `` $\{x\}$, we introduce a constant *successors* (Figure 9, line 3). We also register the connection between $E$ `` $\{x\}$ and *successors* with the operation identification procedure (line 4).

Next, the program must operate on the type *'a graph* rather than *('a × 'a) set*. For the successor operation *successors*, we define a counterpart *succs* on *'a graph* (line 5). As we have registered the type copy with the lifting package in line 2, the command **lift_definition** automatically inserts the necessary conversions. Similarly, all constants of our algorithm are lifted to the conceptual types, too (lines 6–9).

After these preparations, the actual operator identification part has to produce the code equations for the algorithm. In the example, we derive code equations for

the constants *reachable_impl*, *dfs_impl*, and *dfs_body_impl* from the ones for *reachable*, *dfs*, and *dfs_body* such that the new operation *succs* on *'a graph* is used. That is, we transfer the code equations *dfs_code* and *reachable_dfs*, and the defining equation for *dfs_body* from the type (*'a* × *'a*) *set* to *'a graph* by replacing E with a graph G, the successor operation E `` {_} with *succs* G, and the functions *reachable*, *dfs* and *dfs_body* by their counterparts defined in lines 6–9. The manual approach has been described in [14, §4.2]: (i) state the new code equations using *succs*, (ii) let the transfer package convert the equation to the original type, and (iii) prove the latter using the original equations.

Fortunately, a combination of rewriting and the transfer package can automate this process in many cases (line 10). The implementation *containers_identify* performs the following three steps. First, it replaces the composite operations with the temporary constants by folding their definitions (e.g., E `` $\{x\}$ becomes *successors* E x), which are registered as *containers_pre*. Second, it uses the transfer package to replace occurrences of the temporary constants with their conceptual counterparts, e.g., *successors* gets replaced with *succs*. As the replacement changes the type of the first argument from (*'a* × *'a*) *set* to *'a graph*, replacements must be done consistently for the whole equation. In particular, the old functions of the algorithm are also replaced with their counterparts from Figure 9. To that end, the transfer package sets up a constraint system and searches for a solution using so-called transfer rules with backtracking—see [18] for details; intuitively, transfer rules combine Autoref's type annotations (Section 4.5) with its refinement rules (Section 4.2). When the composite term of an operation occurs in the equation, but conceptually, this occurrence does not operate on the conceptual type (e.g., *subclass* `` $\{C\}$ has the same format as the successor operation $E$ `` $\{x\}$, but in a programming language context it computes the set of subclasses of C), the first step nevertheless replaces the occurrence with the temporary constant. In such a case, the transfer package leaves the temporary constant in the equation. Therefore, the third step eliminates the remaining temporary constants by unfolding the equations registered as *containers_post*, i.e., their definitions.

Automation fails if the transfer package is not able to consistently replace the operations. Then, the user must manually transfer the equations.

5.4 Linking Implementations to their Conceptual Types

The Containers framework uses data refinement in the code generator [14] to connect conceptual types with their data structures. Data refinement replaces the constructors of a data type by other constants and derives equations that pattern-match on these new (pseudo-)constructors. Neither need the new constructors be injective and pairwise disjoint, nor exhaust the type.

For every data structure that implements a conceptual type, there is such a pseudo-constructor, i.e., a function from the data structure to the conceptual type that abstracts from the representation details of the data structure. For example, Figure 10 implements a graph by a successor function. The constant *graph_of_succs* converts a successor function of type *'a* ⇒ *'a set* into a graph represented as a set of edges (lines 1–2, line 3 registers *graph_of_succs* with the operation identification procedure) and *Succ* is the counterpart on the conceptual graph type (line 4). The declaration in line 5 turns *Succ* into a pseudo-constructor for code generation. Con-

```
1  definition graph_of_succs :: ('a ⇒ 'a set) ⇒ ('a × 'a) set where
2    graph_of_succs S = {(v, w). w ∈ S v}
3  declare graph_of_succs_def[containers_pre, symmetric, containers_post]
4  lift_definition Succ :: ('a ⇒ 'a set) ⇒ 'a graph is graph_of_succs
5  code_datatype Succ
6  lemma [code]: succs (Succ s) = s
```

Fig. 10: Graph implementation by a successor function

sequently, the successor operation *succs* on *'a graph* is implemented for successor functions by pattern matching (line 6), expressed as a code equation derived from the definitions of *succs* and *Succ*.

Since the ranges of pseudo-constructors may overlap, different implementations of the same conceptual value are possible. For example, *'a set* has four implementations, i.e., one pseudo-constructor each for characteristic functions, lists with and without duplicates, and red-black trees.

char. function	$ChF :: ('a \Rightarrow bool) \Rightarrow 'a\ set$	$ChF\ P = \{x.\ P\ x\}$
monad-style list	$MSet :: 'a\ list \Rightarrow 'a\ set$	$MSet\ xs = \{x.\ memb\ xs\ x\}$
distinct list	$DSet :: 'a\ dlist \Rightarrow 'a\ set$	$DSet\ ds = \{x.\ dmemb\ ds\ xs\}$
red-black tree	$RSet :: 'a\ rbt \Rightarrow 'a\ set$	$RSet\ rs = \{x.\ rmemb\ rs\ x\}$

At run time, each value of a conceptual type is tagged with one pseudo-constructor. Accordingly, if an operation on the conceptual type is called, pattern matching dispatches to the operation of the implementation type. We implement this with a set of dispatch equation for each operation. For example, the membership operation $\in$ is implemented as follows.

$$x \in ChF\ P = P\ x \qquad x \in DSet\ ds = dmemb\ ds\ x$$
$$x \in MSet\ xs = memb\ xs\ x \qquad x \in RSet\ rs = rmemb\ rs\ x$$

Thus, the algorithm can continue to use the conceptual operations identified in Section 5.3. Pattern-matching selects the right implementation at run time.

5.5 Dealing with Sort Refinement

Most implementations of a data structure require additional operations on the elements such as equality or an ordering. The conceptual operations used by the algorithm cannot provide these, because the required operations depend on the implementation. So, this is up to the implementations. For efficiency reasons, it is not sensible to store these operations inside a data structure itself. Suppose we did. In the case of red-black trees, e.g., the implementation *RSet rs comp* of a set consists of a tree *rs* and a comparator *comp :: 'a ⇒ 'a ⇒ order* according to which the tree is sorted. As comparators are functions, we cannot compare them for equality. Thus, to decide set inclusion for two such trees, we cannot just iterate over the two trees, because we cannot decide whether they have used the same comparator. Thus, we have to use an inefficient generic algorithm for this.

To encode the invariant that all red-black trees of the same type use the same comparator, we use type classes. The operations on red-black tree retrieve the comparator from the type class instance of the element type. Thus, we statically know

that the two red-black trees use the same order, so subset comparisons can be performed more efficiently.

Unfortunately, sort refinement spoils the picture. To ensure that a suitable type class instance can always be found, the code generator enforces that every invocation of $\subseteq$ operates on sets whose element type provides a comparator. Similarly, if there is also a hash-based implementation, then all element types also have to have a hash function. Clearly, we cannot accept this, because some element type might not be able to provide all operations. For example, it is hard to linearly order the set of all finite graphs.[15]

Therefore, if a data structure requires certain operations on the element types, we introduce a new type class that wraps the operations in an *option* type. Hence, if a type cannot provide an operation, it can default to *None*. So, *any* type can be made an instance of these type classes and sort refinement is no longer a show-stopper. For example, the type class *ccompare* provides the linear order for elements of red-black trees (to make the overloading explicit, we write the type parameter as a superscript to type class parameters). Here, *order* consists of the values *Lt*, *Eq*, and *Gt*, and *comparator comp* predicates that the comparator *comp* implements a linear order on the elements of *'a*.

class *ccompare* = **fixes** $ccompare^{'a}$ :: $('a \Rightarrow 'a \Rightarrow order)$ *option*
assumes $ccompare^{'a} = Some\ comp \Longrightarrow comparator\ comp$

Sternagel's and Thiemann's **derive** tool [45] automates the boiler-plate instantiations of these classes. For example, the following Isabelle commands declare that *natural* numbers provide a linear order and *complex* ones do not.

derive (*compare*) *ccompare natural* **derive** (*no*) *ccompare complex*

Finally, we would like to exploit the invariant that if a data structure is used, then the parameter types do provide the operations, say $ccompare^{'a} \neq None$ whenever we have a *RSet rs* of type *'a set*. Unfortunately, Isabelle/HOL's type system can neither express nor enforce this invariant. Consequently, pattern matching in the code equations cannot exploit this invariant. Therefore, the dispatch equations must test whether all required operations are available. For example, the finiteness check for red-black tree is implemented as follows:

lemma [*code*]*:*
finite (*RSet rs*) = (**case** $ccompare^{'a}$ **of** *Some* _ ⇒ *True* | _ ⇒ *abort* ...)

The generated code raises an exception using *abort*, if no linear order is available. This is the only option, because the function *RSet* is unspecified in that case, i.e., we cannot prove that it returns a finite set.

Instead, we will make sure in the next section that a data structure will be chosen only if all required operations on the parameter types are supported.

[15] In theory, every type can be ordered linearly by the axiom of choice, but we cannot implement this order, so it is useless here. As the order is needed only for implementation purposes, one can design a specific linear order for virtually every finitely representable data object (in [33], we have done so for *'a set*). However, we do not want to burden the user with verifying and implementing such orders, especially if they are not needed at all for running the program.

5.6 Automatic Selection of Data Structures

It suffices to choose the data structure when a conceptual value is created (or modified), because the pseudo-constructors tag these values at run time. Thus, we now look at operations that *return* a conceptual type (the previous two sections considered operations that only take one). For example, consider the code equation *reachable_dfs* for *reachable* (Figure 8, line 19). It invokes the function *dfs* with {}, but it does not specify how the set of visited nodes should be implemented.

As discussed in Section 3.2, the user should not have to specify the data structure everywhere. This can be achieved statically or dynamically. In the static approach à la Autoref, the code equations are transformed such that {} is replaced by, say, *MSet* [], before or at the time of code generation. Yet, this selection scheme has two drawbacks. First, we must design and implement an annotation language and a transformation mechanism similar to Section 4.3. Second, conceptual operations cannot be used in code equations any more. For example, the generic implementation of set difference in Section 3.4 uses {}. Hence, we would have to commit to a specific implementation, say *DSet*. Yet, this violates the homogeneity rule, because the set difference of two *RSet*s becomes a *DSet* instead of an *RSet*. Thus, we need one copy of set difference for every possible result implementation. As calls to set difference must pick one of the copies, functions that use set difference must come in several copies, too. Ultimately, there are only specialised operations left, because the implementation selection and transformation of code equations happens statically.

In contrast, the dynamic approach can choose the implementation at run time according to heuristics specified at generation time. To that end, the Containers framework implements the heuristics itself using code equations. In principle, the heuristics can even take dynamic information such as the size of the input into account. Currently, we have only implemented heuristics based on static information such as types and user annotations. In the remainder of this section, we explain the basic idea using type information. In Section 5.8, we present a more elaborate selection scheme that supports user annotations.

First, we consider an operation on the conceptual type such as *insert* on sets. The operation dispatches to the different implementations by pattern matching. So, the code equations know the implementation type and can correctly tag the result. This ensures homogeneity. For example, the code equations for *insert* on *MSet* and *RSet* look as follows—the others for *DSet* and *ChF* are similar. For *RSet*, it checks that the linear order is available; otherwise, ... raises an error.

$$\begin{aligned} &\mathit{insert}\ x\ (\mathit{MSet}\ xs) = \mathit{MSet}\ (x \mathbin{\#} xs)\\ &\mathit{insert}\ x\ (\mathit{RSet}\ rs) = (\textbf{case}\ \mathit{ccompare}'^{a}\ \textbf{of}\ \mathit{Some}\ \_ \Rightarrow \mathit{RSet}\ (\mathit{rinsert}\ x\ rs) \mid \_ \Rightarrow \ldots) \end{aligned} \tag{1}$$

Next, we look at an operation that cannot extract the implementation choice from a parameter. The empty set {} is the prime example here, as it creates a set out of nothing. Our heuristics does not yet take context information into account. So, preferences guide the choice. Suppose that red-black trees are preferred over distinct lists and characteristic functions are to be avoided. Recall that the available operations on the parameter types of a conceptual type determine which implementations are possible, and type classes such as *ccompare* tell us which operations are available. Thus, the following HOL term implements the preference.

$$\textbf{case}\ \mathit{ccompare}'^{a}\ \textbf{of}\ \mathit{Some}\ \_ \Rightarrow \mathit{RSet}\ \mathit{rempty} \mid \mathit{None} \Rightarrow \mathit{DSet}\ \mathit{dempty} \tag{2}$$

This term selects a red-black tree if there is a linear order on the elements; otherwise, it picks distinct lists. Thus, red-black trees are only used for element types that do provide a comparator *ccompare*. Hence, the availability checks in the code equations for set operations always succeeds and the error cannot occur at run time.

5.7 Generic Programming

A generic algorithm implements an operation in terms of other operations. Recall the example of set difference from Section 3.4. As Containers works on the conceptual types, such an implementation is directly executable. In particular, no instantiations are needed in general. However, the homogeneity rule is violated in this case. The **then** branch creates an empty set, and the data structure of {} determines the one of the resulting set by the homogeneity rule for *insert*. Thus, if both A and B are implemented as, say, distinct lists, the heuristics for selecting {}'s data structure may nevertheless pick red-black trees.

Clearly, we must make the data structure information available to pick the same. This can be achieved in two ways. First, by pattern-matching on the pseudo-constructors—like in the code equations for *insert* in Section 5.6. This is achieved by instantiating the equation for every implementation. This is not ideal, as it currently must be done manually and it increases the size of the generated program.

Instead, reifying the different implementations of a conceptual type as values in the code leads to an automated solution. For sets, e.g., we introduce a type *set_impl* and a function *impl_of :: 'a set* ⇒ *set_impl* that extracts the data structure from the parameters (as sketched in [15]). Yet, as HOL cannot distinguish the implementations (e.g., *MSet* [] = *RSet rempty*), the function *impl_of* must return logically the same value independent of the implementation.

Data refinement can resolve this conflict as follows. The type *set_impl* is inhabited by just one value *SET_IMPL*, but implemented with a pseudo-constructor for each implementation: *CHF*, *DSET*, *RSET*, *MSET*. As they are only pseudo-constructors, one can add more for new implementations later. Then, *set_impl* is defined by *impl_of* _ = *SET_IMPL* in the logic and implemented by the following code equations.

$$\begin{array}{ll} \mathit{impl\_of}\ (\mathit{ChF}\ \_) = \mathit{CHF} & \mathit{set\_impl}\ (\mathit{RSet}\ \_) = \mathit{RSET} \\ \mathit{impl\_of}\ (\mathit{MSet}\ \_) = \mathit{MSET} & \mathit{set\_impl}\ (\mathit{DSet}\ \_) = \mathit{DSET} \end{array}$$

These equations are provable because all the pseudo-constructors denote the same value in the logic. Thus, *set_impl* extracts the implementation choice inside conceptual operations.

Next, we define a function *sempty* that chooses the desired implementation for empty sets. It is given logically by *sempty SET_IMPL* = {}, but the following code equations implement it as desired. Here, we now exploit that all implementations are logically indistinguishable.

$$\begin{array}{ll} \mathit{sempty}\ \mathit{CHF} = \mathit{ChF}\ (\lambda\_.\ \mathit{False}) & \mathit{sempty}\ \mathit{DSET} = \mathit{DSet}\ \mathit{dempty} \\ \mathit{sempty}\ \mathit{RSET} = \mathit{RSet}\ \mathit{rempty} & \mathit{sempty}\ \mathit{MSET} = \mathit{MSet}\ [] \end{array} \tag{3}$$

In summary, replacing {} with *sempty* (*impl_of* A) in the implementation of set difference in Section 3.4 ensures homogeneity.

Of course, we also want to use optimised implementations for special combinations of data structures. Suppose, e.g., that *rbt_diff* is a fast difference algorithm for red-black trees. So, *rbt_diff* should be preferred over the generic implementation whenever possible. This can be easily expressed using sequential pattern matching. The special cases are identified by pattern-matching in their own equation, and the generic equation for the operation is the last equation. Thus, the special cases take precedence. For example, the following two equations express the preference for *rbt_diff* (where ... raises an error as usual and *rbt_filter P r* retains only those elements in r that satisfy the predicate P).

RSet rs − RSet rs'
  = (**case** *ccompare'a* **of** *Some _* ⇒ *RSet* (*rbt_diff rs rs'*) | *_* ⇒ ...)
RSet rs − B
  = (**case** *ccompare'a* **of** *Some _* ⇒ *RSet* (*rbt_filter* (λ*x*. *x* ∈ *B*) *rs*) | *_* ⇒ ...)

5.8 Overriding the Selection Heuristics

In this section, we present a more sophisticated implementation selection heuristics based on the reification of implementations in Section 5.7. Users can override the heuristics per type (Section 5.8.1) or per instance in the generated code (Section 5.8.2). In case of overrides, however, it is the user's responsibility to ensure that the required operations are available (there are no static checks to enforce availability); if not, the generated code may raise an error at run time.

5.8.1 Type-Based Overrides

Type-based overrides globally change the implementations for a specific element type. For example, it is sometimes sensible to use distinct lists even if there is a linear order available – for *bool* with just two elements, e.g., distinct lists are faster than red-black trees.

For every type, a separate preference can be specified by an overloaded operation *set_impl*.

class *set_impl* = **fixes** *set_impl'a* :: *set_impl*

For modularity, we add another pseudo-constructor *AUTO* for automatic selection like in (2).

This preference overwrites the general preference as follows. The conceptual operation {} has the code equation

$$\{\} = sempty\ set\_impl'^{a}$$

and the function *sempty* from (3) picks the desired implementation. Following (2), we implement it for the new pseudo-constructor *AUTO* by

sempty AUTO = (**case** *ccompare'a* **of** *Some _* ⇒ *RSet empty* | *None* ⇒ *DSet empty*)

Thus, setting set_impl^{bool} to *DSET* encodes the preference for *DSet* for *bool set*.

$set\_impl^{bool} = DSET$ $set\_impl'^{a\ set} = AUTO$ $set\_impl'^{a\ option} = set\_impl'^{a}$

Element types with type variables can inherit the preference from the type parameter (e.g., *'a option*) or discard it (e.g., *'a set*). Automation with **derive** is also available.

Feature	Autoref	Containers
Identification of conceptual types	+	◯
Implementation selection	static	dynamic
Generic programming	static	dynamic
Underspecification	◯	◯
Non-determinism	+	—

Table 1: Comparison of Features

5.8.2 Instance-Based Overrides

Changing the choice of implementation at a specific point in the code does not require any special mechanism. The user can use the pseudo-constructor of the desired data structure directly in the code equation for that operation. For example, in *reachable*'s code equation (Figure 8, line 19), replacing {} with *MSet* [] enforces that the set of visited states is always implemented by a list with duplicates.

5.9 Non-Determinism

Containers hides all the implementation details from the logic, because the refinement happens in the code generator. Thus, truly non-deterministic specifications cannot be refined to deterministic ones. For iterations over sets, e.g., only left-commutative operators can be used. Fortunately, raising the level of abstraction and careful design often make it possible to achieve unique results, as the running example illustrates. Moreover, staying at conceptual types in the logic avoids all the non-determinism induced by different representations. In the generic algorithm for set difference in Section 3.4, e.g., the representations of the resulting set depends on the iteration order (as already noted in [28]). For Containers, the result is deterministic in the logic, because the pseudo-constructors abstract from the representation details.

6 Technical Comparison

After having presented the technical details of both frameworks, we now compare how well they address the challenges identified in Section 3 and distill the differences and similarities of the two frameworks. The features of each tool are summarised in Table 1, where + means fully supported, ◯ means partially supported, and — means not supported.

Identification of Conceptual Types Autoref identifies conceptual types in its first phase using a powerful heuristics based on type elaboration (cf. Section 4.4), which rarely fails in practice. In contrast, the Containers framework requires conceptual types to match HOL type constructors. If this is not the case, some manual preprocessing has to be done. In many cases, this preprocessing is straightforward by utilizing Isabelle/HOL's transfer package (cf. Section 5.3).

Implementation Selection Both tools implement a heuristics for automatically selecting the implementations which can be controlled by the user if desired. The heuristics are based on the same principles, namely the homogeneity principle and selection of implementations based on the abstract type. The main difference is when the selection happens: Autoref performs the selection statically while synthesizing the implementation, whereas the generated code from Containers selects the implementations dynamically at run time. In principle, Containers supports selection strategies that depend on dynamic run-time information (e.g., the size of the data structures), but such strategies have not yet been implemented.

Generic Programming Both tools support generic programming with specialization of algorithms. Like for the implementation selection, Autoref instantiates the generic algorithms during the synthesis whereas Containers generates dispatching code which executes at run time.

Underspecification Both tools support underspecified functions with some limitations. The Autoref tool must prove during the synthesis that every (underspecified) function is only applied to arguments for which its specification uniquely determines the result. For example, recall from Section 4.2.3 that the function *hd* is not specified for empty lists. This underspecification shows up as the side condition $l \neq []$ in the generalized parametricity rule for *hd l*. The synthesis succeeds only if Autoref can discharge these side conditions using the contextual information provided by congruence synthesis rules. When Autoref is used together with the Monadic Refinement Framework, assertions can be used to transport the required information from the correctness proof to the refinement proof. An important exception is when no actual refinement happens, i.e., the type of the list elements does not change, say a list of integers is implemented by a list of integers. Then, the trivial synthesis rule $(hd,\ hd) \in \langle int\_rel \rangle list\_rel \to int\_rel$ follows from reflexivity.

Underspecification affects the Containers framework only during the identification of conceptual types. If the type of an underspecified HOL constant changes when conceptual type constructors are introduced—for example, *hd* : $('a \times 'a)\ set\ list \Rightarrow ('a \times 'a)\ set$ becomes *hd : 'a graph list* $\Rightarrow$ *'a graph*—then the transfer to conceptual types must be done manually, because the underlying transfer package cannot handle conditional parametricity rules. This manual transfer typically involves a proof that the function is uniquely specified for the argument to which it is applied. Typically, this proof has already been done in the correctness proofs, but Containers does not provide any automation to ease its reuse. Conversely, if the HOL type remains the same, the trivial synthesis rule is used. In practice, this is the most frequent case, as most refinements take place in the code generator, outside of the logic. Thus, even when sets are implemented by red-black trees, the type *'a set* remains *'a set* and no proof need be done for *hd : 'a set list* $\Rightarrow$ *'a set*. In contrast, Autoref replaces *'a set* by *'a rbt* inside the logic and therefore requires the proof.

Non-Determinism In combination with the Monadic Refinement Framework, Autoref supports non-determinism natively. In contrast, the Containers framework does not support non-determinism at all. If the result of the overall algorithm is deterministic, but the algorithm itself contains non-deterministic parts, careful engineering may be used to transform the algorithm to a fully deterministic one (c.f. Section 5.1).

7 Using the Refinement in Context and Combining Both Frameworks

Recall Theorem *2SAT_graph* presented in Figure 1, which suggests a (naive) satisfiability algorithm by iterating over the variables and performing reachability checks.

In this section, we show how to obtain this algorithm in both frameworks, Containers (Section 7.1) and Autoref (Section 7.2). Actually, the Autoref-based implementation also uses the Containers-Framework, thus demonstrating how both frameworks complement each other (Section 7.3). This allows us to compare both frameworks from a user perspective (Section 7.4)

7.1 2SAT with Containers

In Section 5, we have obtained an efficient implementation *reachable_impl* for a non-executable function *reachable*. We now will use *reachable_impl* to obtain an efficient 2SAT checker. In general, using a Containers implementation in a larger context requires two steps:

1. Adapt the formalization to the conceptual types of the implementation, and
2. Instantiate the type classes such as *ccompare* and *set_impl* for the types that are used as elements of the data structures.

In the following, we will demonstrate these steps for the 2SAT example.

For the first step, recall that we have changed the type of graphs from a set of edges (*'a* × *'a*) *set* to the conceptual type *'a graph* (Figure 9) that is implemented by a successor function (Figure 10). Since the original 2SAT formalization in Figure 1 constructs the implication graph as a set of edges, we must provide an implementation via a successor function and use it in the code equations. Figure 11 shows the necessary steps. The function *sucs* in lines 4–5 takes a formula F and a literal l and returns the set of l's successor literals in the implication graph. This set is computed by iterating over all clauses $\langle\!|l_1, l_2|\!\rangle$ of F and collecting all l_{2-i} for which $l = \mathit{negate}\ l_i$ holds. The iteration is expressed as a fold over a set, which requires the loop body to be left-commutative (lines 1–3, cf. Section 5.1). The lemma in line 6 shows that this is a correct implementation of the implication graph for finite formulas. Similarly, we can express 2CNF satisfiability using the reachability operation *reachable* (lines 7–11). Note that we prove this characterization using the original theorem *2SAT_graph* from Figure 1.

To keep the proofs simple and well-automated, all these steps model the graph as a set of edges. So we must lift them to the conceptual type *lit graph*. Like in Figure 9, we define the implementation constant *gr_impl* for the implication graph (line 12) and transform the two code equations *gr_code* and *satisfiable_code* using Container's operation identification. In particular, the identification transforms *gr_code* into the following code equation for *gr_impl*

$$\mathit{gr\_impl}\ F = \textbf{if}\ \mathit{finite}\ F\ \textbf{then}\ \mathit{Succ}\ (\mathit{sucs}\ F)\ \textbf{else}\ \mathit{abort}\ \ldots \tag{4}$$

and replaces in *satisfiable_code* the functions *gr* and *reachable* with their implementations *gr_impl* and *reachable_impl*. Thus, we have obtained an implementation for *satisfiable* via *reachable*, where the input formula in particular is still of the same type, i.e., a set of unordered pairs of literals.

```
1  lift_definition sucs_loop :: lit ⇒ (lit uprod, lit set) comp_fun_idem is
2    λl (|l1, l2|) L. if l = negate l1 then {l2}∪L else if l = negate l2 then {l1}∪L else L
3    by auto

4  definition sucs :: cnf ⇒ lit ⇒ lit set where
5    sucs F l = fold (sucs_loop l) {} F

6  lemma gr_code: gr F = if finite F then {(l, l') | l' ∈ sucs F l} else abort ... ⟨proof⟩

7  lemma satisfiable_code: satisfiable F =
8    if finite F ∧ is2cnf F then
9      let E = gr F in ∀x ∈vars F. ¬ (reachable E (P x) (N x) ∨ reachable E (N x) (P x))
10   else abort ...
11   by(simp add: reachable_def 2SAT_graph ...)

12 lift_definition gr_impl :: cnf ⇒ lit graph is gr
13 lemmas [containers_identify, code] = gr_code satisfiable_code
```

Fig. 11: Construction of the implication graph using Containers and derivation of the executable 2SAT checker

In the second step, we now must specify which data structures (red-black trees, lists, etc.) to use by instantiating the relevant type classes (*ccompare* and *set_impl*) for the element types. Our 2SAT formalization uses two types for which the Containers library does not provide these instantiations: *lit* and *uprod*. As *lit* is defined as a datatype, we let the command **derive** do the instantiations. It orders negative literals before positive ones and then by the variable number and specify that sets of literals should use red-black trees:[16]

derive *linorder lit* **derive** *ccompare lit* **derive** (*rbt*) *set_impl lit*

For *uprod*, we instantiate the type classes manually as *uprod* is not a datatype and therefore out of **derive**'s scope. For example, given a comparator *comp* on *'a*, we obtain a comparator *comp_uprod* on *'a uprod* by

```
comp_uprod (|a, b|) (|c, d|) =
  let (x, y) = case comp a b of Lt ⇒ (a, b) | _ ⇒ (b, a);
      (x', y') = case comp c d of Lt ⇒ (c, d) | _ ⇒ (d, c)
  in case comp x x' of Lt ⇒ Lt | Gt ⇒ Gt | Eq ⇒ comp y y'
```

That is, we compare the smaller element x of $(|a, b|)$ with the smaller element x' of $(|c, d|)$ and compare the larger elements y and y' only if x and x' are equal. Note that the pattern matches on $(|a, b|)$ and $(|c, d|)$ are well-defined because *comp* is a comparator, i.e., it induces a total order.

Now, we already have an executable 2SAT checker. Before we generate the code, we should include one further optimisation, though. The graph construction in the code equation (4) does not supply all arguments to *sucs* (cf. line 5 in Figure 11). Consequently, when the graph E is computed in the **let** binding in line 9, the *Succ* constructor stores only a closure. At runtime, the graph is therefore recomputed

[16] Containers requires a few more type class instances for further operations such as set complement, which we have not discussed in this paper. We therefore do not show their instantiations either.

for every reachability check. We avoid this by tabulating *sucs F* when the graph is constructed. We accomplish this by simply proving an appropriate code equation for *sucs*. The tabulation is similar to the one we will present in Section 7.3, when we combine Autoref and Containers.

In the end, we generate an executable 2SAT checker in any of the target languages of Isabelle's code generator. For example, in Standard ML by the command

export_code *satisfiable* **in** *SML*

7.2 2SAT with Autoref

The first step for using the Autoref framework is to set up the DFS algorithm as an implementation for the reachability operation. We also configure operation identification to identify the pattern $(u,v) \in E^*$ as the reachability operation. This is accomplished in a few lines of boilerplate code, which we omit here.

To implement the satisfiability check, we first choose an implementation data type for formulas, and provide concrete operations for the set of variables and the successor function of the implication graph. Autoref then derives an implementation for the right hand side of Theorem *2SAT_graph*.

We implement a formula as a list of distinct pairs of literals (type *cnfi*). The refinement relation *cnfi_rel* is defined in terms of an abstraction function *cnf_α* and an invariant *is2cnfi*. The invariant corresponds to the abstract invariant *is2cnf*.

type_synonym *cnfi* = (*lit* × *lit*) *list*
definition *cnf_α* :: *cnfi* ⇒ *cnf* **where** *cnf_α Fi* = {(|*x*, *y*|) | (*x*, *y*) ∈ *set Fi* }
definition *is2cnfi* :: *cnfi* ⇒ *bool* **where** *is2cnfi Fi* = (∀(*l1*, *l2*) ∈ *set Fi*. *l1* ≠ *l2*)
definition *cnfi_rel* = *br cnf_α is2cnfi*
lemma *is2cnfi_correct: is2cnfi Fi* ⟷ *is2cnf* (*cnf_α Fi*)

Recall that *br α I* constructs a refinement relation from an abstraction function and an invariant. The next two lines register the conceptual type *i_cnf* of cnf formulas with Autoref.

consts *i_cnf* :: *interface*
lemmas [*autoref_rel_intf*] = *REL_INTFI*[**of** *cnfi_rel i_cnf*]

As required by our DFS algorithm, we implement the implication graph by its successor function (*gri*). The set of variables of a formula is implemented as a distinct list (*varsi*). We show the following refinement lemmas and declare them to Autoref:

lemma [*autoref_rules*]*:*
(*gri*, *gr*) ∈ *cnfi_rel* → ⟨*Id*⟩*succg_rel*
(*varsi*, *vars*) ∈ *cnfi_rel* → ⟨*Id*⟩*list_set_rel*

Our DFS algorithm requires the set of nodes reachable from the source node to be finite. Moreover, the characterisation of *satisfiable* by implication graphs (Theorem *2SAT_graph*) requires the set of variables to be finite, and the clauses to be proper. Formulas representable by our refinement relation fulfil these requirements. We derive our efficient algorithm in a two-step refinement process: First, we introduce a let-binding to only compute the implication graph once (*sat_refine*) and, second, we use Autoref to synthesise an implementation (*sati_refine*). Both refinement steps are

performed in a context where we assume (*cnfi, cnf*) ∈ *cnfi_rel*. In particular, this gives us finiteness and properness of the abstract formula *cnf*, which is required in the first refinement step.

context
  fixes *cnfi cnf*
  assumes [*autoref_rules*]*:* (*cnfi, cnf*) ∈ *cnfi_rel*
begin

lemma *sat_refine:*
  satisfiable cnf = (**let** *gr* = *imp_graph cnf* **in**
    ∀*x* ∈ *vars_of_cnf cnf*. ¬ ((*Pos x, Neg x*) ∈ *gr** ∧ (*Neg x, Pos x*) ∈ *gr**))

schematic_goal *sati_refine:* (*?f::?'c, satisfiable cnf*) ∈ *?R*
  unfolding *sat_refine* **by** *autoref*

end

concrete_definition *sati* **uses** *sati_refine*

Again, we extract a correctness statement that is independent of the refinement framework formalism, and also register our algorithm as implementation for the *satisfiable* operation, to be used by Autoref for refining more complex algorithms based on satisfiability:

lemma *sati_correct: is2cnfi Fi* ⟹ *sati Fi* ⟷ *satisfiable* (*cnf_α Fi*)
lemma *sati_autoref_rl* [*autoref_rules*]*:* (*sati, satisfiable*) ∈ *cnfi_rel* → *bool_rel*

7.3 Combining Autoref and Containers

In the previous section, we have intentionally not detailed the implementation of the implication graph (*gri*). Given the formula as list of pairs of literals, we have to create the successor function of the implication graph. This is best done by iterating over the list once, creating a map from literals to successor lists. This map is then used to efficiently look up the successors of a literal.

In this section, we use the Containers framework to implement the tabulation for the graphs, which the Autoref implementation for DFS will use (Figure 12). We split the tabulation into three functions: *ins_edge u v m* inserts an edge (*u, v*) into the graph represented by the map *m* (lines 1–4), *ins_edges* (l_1, l_2) *m* inserts the edges induced by the clause (l_1, l_2) (line 5), and *tabulate cnfi* folds over the formula to create the implication graph (line 6). The map is represented as the abstract type (*'k, 'v*) *mapping*, which is the conceptual map type of the Containers framework with operations *Mapping.empty*, *Mapping.lookup*, and *Mapping.update* for the empty map, map lookup, and map update. We could have used the isomorphic HOL type *'k* ⇒ *'v option* for maps, but this would require a few declarations for performing the operation identification part. Directly using the conceptual type avoids this boilerplate. We also define a function *αsl* to convert a map into a successor function and show that the converted tabulated graph implements the abstract implication graph (lines 7–10). This is a straightforward proving exercise, with the only challenge to

```
1  definition ins_edge :: lit ⇒ lit ⇒ (lit, lit list) mapping ⇒ (lit, lit list) mapping where
2    ins_edge u v m = (case Mapping.lookup m u of
3      None ⇒ Mapping.update u [v] m
4    | Some l ⇒ Mapping.update u (remdups (v≠l)) m)
5  definition ins_edges (l1, l2) = ins_edge (negate l1) l2 ∘ ins_edge (negate l2) l1
6  definition tabulate cnfi = fold ins_edges cnfi Mapping.empty

7  definition αsl :: (lit, lit list) mapping ⇒ lit ⇒ lit list where
8    αsl m l = (case Mapping.lookup m l of None ⇒ [] | Some ls ⇒ ls)

9  lemma [autoref_rules]:
10   (λcnfi. αsl (tabulate cnfi), imp_graph) ∈ cnfi_rel → ⟨Id⟩succg_rel

11 derive ccompare lit          derive (rbt) mapping_impl lit

12 export_code sati in SML
```

Fig. 12: Tabulation of the implication graph's successor function using Containers

find a good generalisation for the induction over the *fold*-function. After having configured Containers to use red-black trees for the tabulated successor map (line 11), we can generate the 2SAT checker (line 12).

Note that we have formalised the tabulation algorithm directly on the implementation type. In principle, we could have first defined an abstract version of this algorithm that works on sets of unordered pairs (formulas) and sets of edges (graphs). Then, we could use stepwise refinement to refine the graph representation to successor functions and further to maps, as well as the formula representation to ordered pairs and further to lists. Conceptually, these are three different abstraction levels for graphs, and another three for formulas. Autoref and the Refinement Framework support such stepwise refinements, and such a fine-grained abstraction may make sense for complex algorithms. Yet, it would be overkill for such a simple algorithm like tabulation, which can easily be refined and proved correct in a few lines of rather straightforward Isabelle source code.

7.4 Comparison from a User Perspective

Given the two 2SAT checkers, we can now compare the two frameworks from a user perspective.

Many algorithms can be elegantly expressed using non-deterministic operations on abstract data types. In Isabelle, the Refinement Framework allows a natural formalisation of those algorithms, and Autoref can be used to transfer them to efficient implementations. However, using Autoref comes at the cost of some boilerplate code for setting up user-defined conceptual types and operations. The Collections Framework readily provides a variety of verified data structures that implement standard conceptual types like sets, maps, and priority queues.

For algorithms that expose the nondeterminism in their abstract result, like computing a path between two nodes, the Containers approach cannot be used. However, if nondeterminism is only used implicitly, like in our running DFS example, careful engineering can sometimes determinise the algorithm and make it available for refinement via Containers.

For deterministic abstract algorithms, like our tabulation example, the Containers framework typically requires less effort than Autoref. In the best case, it suffices to import an Isabelle theory for each conceptual type and instantiate its type classes as needed, which is automated by the derive tool to a large extent. Containers currently provides such Isabelle theories for sets and maps; others can be developed in a similar way, e.g., for graphs as in Section 5. If the formalisation does not use the conceptual types (e.g., *('a × 'a) set* instead of *'a graph*), the operation identification must be done first. If no underspecified operations are affected, the transfer tool can automatically transfer the correctness statement from the conceptual type constructors back to the original types. Otherwise, the transfer may be a serious endeavor, as one has to show that underspecified functions are only applied to their specified range. This often requires to replay large parts of the abstract correctness proof on the concrete level. In those cases, using the Refinement Framework with Autoref may be the better option, as assertions can be used to transport the required information from the correctness proof to the refinement proof.

The effort difference extends to using the implementation in a formalisation context that itself does not use the framework. For example, we might want to use the executable 2SAT checker in some larger functional formalisation. Note that each 2SAT checkers has been developed in the same framework as the DFS implementation. Accordingly, Autoref's 2SAT implementation *sati* lives in the same monad as the DFS implementation, whereas Containers' *satisfiable* is the same HOL constant that we have started with in Figure 1. Consequently, the Containers implementation can be used without any further effort in any formalisation context.[17] In contrast, if Autoref users want to use their implementations in a functional context, they must manually escape the monad and transfer the correctness theorem, like in Theorem *dfs_code_correct* in Section 4.1.

8 Related Work

The two Coq tools CoqEAL [9] and Fiat [11] are based on ideas similar to Autoref's. In this section, we first compare our frameworks with them in detail (Sections 8.1 and 8.2, respectively) and then discuss further related work (Section 8.3).

8.1 The Coq Effective Algebra Library

Cohen et al. [9] developed a data refinement tool CoqEAL (The Coq Effective Algebra Library) for algebraic structures in the Coq theorem prover. Using the same ideas from relational parametricity as Autoref (Section 4.2), CoqEAL synthesises the implementation while proving the refinement theorem. On the technical level, CoqEAL differs from Autoref in two main aspects.

First, CoqEAL users must explicitly convert their algorithms into a form where the conceptual types are abstract datatypes. That is, these types must all be type

[17] In general, Containers users must only switch to conceptual type constructors in algorithm's input and output types, which is usually painless. For example, no switching is needed for *satisfiable* (as *set* is a type constructors and none of the other types are refined) and the switch from *('a×'a) set* to *'a graph* for using DFS in the 2SAT checker is easy as shown in Figure 11.

variables and the algorithm uses these type variables only through operations which it takes as parameters. The original formulation on the algebraic structure is merely an instance as witnessed by passing the operations of the algebraic structure. In contrast, Autoref tries to infer the conceptual datatypes itself and directly synthesises the algorithm implementation, without defining the parameterised algorithm.

Second, CoqEAL uses Coq's type class mechanism [44] to maintain the database of data refinements and to synthesise the implementation. In case of multiple solutions to the type class inference problem, the synthesis picks the first it finds unless the user has specified a particular instance using annotations. This design nicely separates concerns and relieves the user from spelling out all the arguments to the algorithms. Yet, experience indicates that better search heuristics are needed when refinements are deeply nested [10]. Isabelle's type class mechanism is less powerful than Coq's and does not support such synthesis. Therefore, Autoref keeps its own collection of refinement theorems and implements its own synthesis strategy with many heuristics, in particular preferences (Section 4.5).

As CoqEAL abstracts over operations using type classes, it is less modular than Autoref and Containers in two respects. First, an operation on a data structure can only be specialised if there is an operational type class for it. This means in practice that users should parameterise their functions over all other functions they use and introduce for each of the parameters a type class and an appropriate notation. In the case of finite sets, e.g., a small set of basic operations does not suffice, such as the empty set, insertion, deletion, membership test, and folding. There should also be type classes for all other set operations like set union $\cup$, difference $-$, and symmetric difference $A \mathbin{\triangle} B = (A - B) \cup (B - A)$, even if all of them could be implemented using the basic operations (as described in Section 3.4). CoqEAL users therefore must also declare instances for every implementation type, which scales poorly with the number of implementations and operations. If this principle is not followed, efficiency and locality can suffer. For example, suppose that the algorithm for $\triangle$ abstracts only over the basic set operations and calls the generic implementations of $\cup$ and $-$ directly, implicitly passing them the basic set operations. Then $\triangle$ will not use optimized algorithms for $\cup$ and $-$ for the selected data structure (cf. Section 3.4). Conversely, if $\triangle$ does abstract over $\cup$ and $-$, but some algorithm f using $\triangle$ does not abstract over $\triangle$, then changes to $\triangle$ also affect f. For example, if $\triangle$'s implementation changes to $A \mathbin{\triangle} B = (A \cup B) - (A \cap B)$, i.e., it now also takes an implicit $\cap$ operation, then f must also change, because it now must also depend on the type class for $\cap$.

In Autoref, the refinement rules themselves trigger the synthesis of auxiliary operation implementations (Section 4.2.4), so no type class declarations are needed. As implementations are picked according to priorities, optimised algorithms are preferred over generic ones. So users need not worry about ambiguities during type class resolution. In Containers, as the dispatch to the implementation happens at run time, there is no difference between basic and derived operations and users need not worry about their difference when they write their programs. If there is an optimised algorithms for an operation, then this will also be used.

Second, CoqEAL cannot refine the same conceptual type to different implementations in different places of the algorithm, e.g., sets cannot be implemented as lists in one part and as red-black trees in others. This is only possible if the conceptual types are separated, i.e., different type variables and parameters for the operations are used. When users write their algorithms, they must immediately decide which

occurrences of a conceptual type should be refined to the same implementation. This regime enforces the homogeneity rule and users must explicitly use conversions between different implementations, which must also show up in the list of abstracted operations. Autoref and Containers permit violations of the homogeneity rule if no other solution is possible. That is, Autoref and Containers try to fix implementation choice mismatching whereas CoqEAL users get a type error in case of a mismatch. Experience will show which approach causes less problems and surprises in practice.

Moreover, Isabelle's type classes and HOL's type system cannot support CoqEAL's approach in the first place. Isabelle allows only one type variable in type class operations, but ADT operations often need several: For example, set insertion *insert* :: $'a \Rightarrow 'x \Rightarrow 'x$ uses $'a$ for the elements and $'x$ for the ADT of sets. Even without using type classes, it is impossible to abstract over polymorphic operations of ADTs in HOL. For example, the folding operation *fold* :: $\forall 's.\ ('a \Rightarrow 's \Rightarrow 's) \Rightarrow 's \Rightarrow 'x \Rightarrow 's$ for sets should be polymorphic in the state type $'s$ such that algorithms can use *fold* with several different state types. This is not possible in HOL where all parameters must be monomorphic. It is therefore not possible to use locales [4] either, which are closer to Coq's type classes than Isabelle's type classes are.

In the remainder of this section, we analyse whether and how CoqEAL addresses the four challenges we have identified in Section 3. First, CoqEAL does not offer any support to identify the conceptual types and their operations. This is completely left to the user. According to our experience with HOL-based provers, this can substantially increase the effort to generate efficient implementations, because encoding types and operations using existing ones is common in the HOL community. It may be less severe in type theories like Coq's where people seem to use ADTs more often. Second, the selection of data structures, as mentioned before, is delegated to Coq's type class mechanism, which provides only limited control over the search for instances, i.e., implementations. In detail, homogeneity is strictly enforced by the ADT approach, and annotations and preferences can be specified via Coq's type class mechanism. Third, non-determinism is not discussed at all by the CoqEAL authors. As CoqEAL's refinement relations are partial quotients, a single conceptual value may have several logically equivalent representations and the refined implementations may locally exploit the particular representation, but this must lead to observable non-determinism at the conceptual type. This is very similar to the non-determinism that Containers support. Autoref also handles observable non-determinism thanks to its integration with the Monadic Refinement Framework [29]. Fourth, it is not clear how well CoqEAL scales with the number of data structures and operations. In the examples distributed with CoqEAL, there seems to be only one implementation available for each ADT. We conjecture that CoqEAL users will face scalability challenges due to the lack of control over the implementation selection and an abundance of operational type classes.

8.2 Fiat

Delaware et al. [11] present the Fiat tool, which supports refinement of nondeterministic specifications to executable implementations in Coq. The basic idea is very similar to the combination of the Monadic Refinement Framework and Autoref: Both use a nondeterminism monad, a notion of program and data refinement, conceptual

types that provide abstract operations, and both strive for automation of the refinement process.

We first discuss the three most important differences between Fiat and our frameworks. First, Fiat's automation is focused on domain specific languages: Very powerful refinement methods can be used for very narrow domains, e.g., database queries. In contrast, Autoref and Containers support a single refinement strategy that is meant to be used for all domains, and can be configured by the user to some extent.

Second, refinement in Fiat follows an object-oriented modelling approach: The methods of an abstract class (conceptual type in our parlance) are refined by the corresponding methods of an implementation class. Only the representation of the objects can be refined, but not the arguments or results of the methods. Nested refinement is therefore impossible, as is needed, e.g., for sets of sets.

Third, the interface of a Fiat type has to be defined once and cannot be extended later. This prevents scalable specialisation of generic algorithms, for similar reasons as for CoqEAL: While generic algorithms can be used to provide new operations, they will always be reduced to the operations of the interface, and then to an implementation. This makes it impossible to specialise operations that were not included in the initial interface. At the same time, the interface should be kept small to ease new implementations. Autoref and Containers resolve this tension by delaying the choice of algorithm beyond declaration time. Generic algorithms are resolved in Autoref during synthesis and in Containers at run-time. Therefore, an implementation *needs* provide only a minimal set of basic operations, relying on generic algorithms for the other operations. Additionally, it *can* declare specialisations for some operation, which will then be used instead of the generic algorithms.

Next, we evaluate Fiat according to the four challenges that we have identified in Section 3. First, Fiat does not support identification of conceptual types, which may be less problematic than it is in HOL (cf. Section 8.1). Second, the selection of data structures is done either manually by the user, who directly invokes so-called honing tactics, or automatically by domain specific refinement scripts. The paper [11] mentions no general applicable refinement scripts, although this should be possible in their framework. Third, nondeterminism is built into the Fiat framework, being based on a set monad. However, it seems to lack a notion of recursion. Fourth, it is unclear how well Fiat scales with the number of data structures and operations. As far as we know, only a handful of interfaces and data structures have been implemented in Fiat. However, we expect less scalability problems than for CoqEAL, as implementation selection in Fiat is under complete control of the specialised automatic refinement tactics, each of which typically uses only a limited number of interfaces and implementations.

8.3 Further related work

In the other sections of this paper we have already referenced existing work we build on, in particular the Isabelle packages and tools [14, 16, 18, 45].

Based on the lessons learned from the automatic refinement tool, the first author has developed the Sepref tool [26, 27], which supports automatic refinement to both functional and imperative data structures. The operation identification heuristics of this tool is essentially an advancement of the one used in Autoref. The implementation selection heuristics has been strongly simplified: Only the homogeneity prin-

ciple and user annotations determine the implementation data structures. If several implementations satisfy the constraints, the Sepref tool reports an error instead of automatically selecting an implementation it deems suitable (as Autoref does). While users have to write (slightly) more annotations, they keep full control of the generated data structure, and no magic happens behind the scenes. Currently, Sepref has only very limited support for generic algorithms, but this can be implemented along the lines of Autoref.

The Containers framework relies on the transfer package by Huffman and Kunčar [18] for the identification of conceptual operations. Similar to Autoref, transfer synthesises HOL terms based on relational parametricity, but it is less powerful. For example, it cannot handle side conditions as described in Section 4.2.3: neither the conditional parametricity rules like the one for *hd* nor the stronger rules for control operators like **if**.

Peyton Jones [40] and Chen et al. [8] already had the idea of the heuristics selecting the implementations based on the available type class operations. They showed that Haskell's single-parameter type classes do not suffice if one wants to achieve "bulk-type polymorphism", i.e., users can add new container implementations and new element types without changing any existing code. The Containers approach nevertheless succeeds, even though Isabelle's type classes are even less expressive than Haskell's. The reason is that we do not extend the generated code itself, but the formal definitions that get translated to the code. Hence, after an addition, Isabelle re-generates all of the code and this is when the crucial sort refinement happens.

9 Conclusion and Future Work

In this paper, we have described the Autoref tool and the Containers framework, which both provide automatic data refinement for specifications written in Isabelle/HOL. Both tools have been successfully used in various verification projects.

The most notable project that uses Autoref as its back-end is the CAVA LTL model checker [7,12], a fully featured, verified, and efficient LTL model checker with partial order reduction. Here, Autoref has been used to assemble the overall model checker and many of its components, including the CAVA Automata Library [24] the verified implementations of nested depth first search, Gabow's strongly connected components algorithm [25], and the verified implementation of Gerth's Algorithm [42, 43] to convert LTL formula to Buchi automata. Immler [19] uses Autoref for efficient verified numerical algorithms.

Containers has been used in a verified interpreter for Java bytecode in the JinjaThreads project [31,34], the certifier CeTA for the termination analysis of rewrite systems [46], and in the formalisation of executable field extensions [47]. Felgenhauer and Thiemann [13] describe a successful combination of both, Autoref and Containers, in an application to tree automata.

In summary, we conclude that both tools complement each other. The Containers framework offers a very light-weight approach with restrictions on non-deterministic specifications and conceptual types. In contrast, the Autoref framework is more heavy-weight, but fully supports non-deterministic specifications and arbitrary encodings of conceptual types into HOL types. Both can be combined, where the Containers framework is typically used for simple parts of an algorithm and the Autoref tool to assemble the overall algorithm.

Acknowledgements We thank René Thiemann for implementing **derive** for the Container type classes. Andreas Lochbihler is supported by SNSF grant 153217 "Formalising Computational Soundness for Protocol Implementations". The authors are listed in alphabetical order.

References

1. Appel, A.W.: Efficient verified red-black trees (2011). URL `http://www.cs.princeton.edu/~appel/papers/redblack.pdf`
2. Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters **8**(3), 121–123 (1979)
3. Back, R.J.J., Akademi, A., Wright, J.V.: Refinement Calculus: A Systematic Introduction, 1st edn. Springer-Verlag New York, Inc. (1998)
4. Ballarin, C.: Locales: A module system for mathematical theories. J. Automat. Reason. **52**(2), 123–153 (2014). DOI 10.1007/s10817-013-9284-7
5. Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: TPHOLs 2009, *LNCS*, vol. 5674, pp. 147–163. Springer (2009)
6. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: ITP 2014, pp. 93–110 (2014)
7. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. In: NFM 2016, pp. 307–321 (2016)
8. Chen, K., Hudak, P., Odersky, M.: Parametric type classes. In: LFP 1992, pp. 170–181. ACM (1992)
9. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: G. Gonthier, M. Norrish (eds.) CPP 2013, *LNCS*, vol. 8307, pp. 147–162. Springer (2013)
10. Cohen, C., Rouhling, D.: A refinement-based approach to large scale reflection for algebra. In: Journées Francophones des Langages Applicatifs (JFLA 2017) (2017). Tech. Rep. HAL-01414881, `https://hal.inria.fr/hal-01414881`
11. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: Proc. of POPL, pp. 689–700. ACM, New York, NY, USA (2015). DOI 10.1145/2676726.2677006. URL `http://doi.acm.org/10.1145/2676726.2677006`
12. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: N. Sharygina, H. Veith (eds.) CAV 2013, *LNCS*, vol. 8044, pp. 463–478. Springer (2013)
13. Felgenhauer, B., Thiemann, R.: Reachability, confluence, and termination analysis with state-compatible automata. Information and Computation **253**, 467–483 (2017). DOI 10.1016/j.ic.2016.06.011
14. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) ITP 2013, *LNCS*, vol. 7998, pp. 100–115. Springer (2013)
15. Haftmann, F., Lochbihler, A., Schreiner, W.: Towards abstract and executable multivariate polynomials in Isabelle. Isabelle Workshop 2014, `http://www.infsec.ethz.ch/people/andreloc/publications/haftmann14iw.pdf` (2014)
16. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS 2010, *LNCS*, vol. 6009, pp. 103–117. Springer (2010)
17. Hoare, C.: Proof of correctness of data representations. Acta Informatica **1**(4), 271–281 (1972)
18. Huffman, B., Kunčar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: G. Gonthier, M. Norrish (eds.) CPP 2013, *LNCS*, vol. 8307, pp. 131–146. Springer (2013)
19. Immler, F.: Verified reachability analysis of continuous systems. In: TACAS 2015, *LNCS*, vol. 9035, pp. 37–51. Springer (2015)
20. Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: A. Biere, R. Bloem (eds.) CAV 2014, *LNCS*, vol. 8559, pp. 167–183. Springer (2014)
21. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Sys. **28**, 619–695 (2006)
22. Lammich, P.: Tree automata. Archive of Formal Proofs (2009). `http://www.isa-afp.org/entries/Tree-Automata.shtml`, Formal proof development

23. Lammich, P.: Automatic data refinement. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) ITP 2013, *LNCS*, vol. 7998, pp. 84–99. Springer (2013)
24. Lammich, P.: The CAVA automata library. Archive of Formal Proofs (2014). `http://www.isa-afp.org/entries/CAVA_Automata.shtml`, Formal proof development
25. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: ITP 2014, *LNCS*, vol. 8558, pp. 325–340. Springer (2014)
26. Lammich, P.: Refinement to Imperative/HOL. In: ITP 2015, *LNCS*, vol. 9236, pp. 253–269. Springer (2015)
27. Lammich, P.: Refinement based verification of imperative data structures. In: CPP 2016, pp. 27–36. ACM (2016)
28. Lammich, P., Lochbihler, A.: The Isabelle collections framework. In: M. Kaufmann, L.C. Paulson (eds.) ITP 2010, *LNCS*, vol. 6172, pp. 339–354. Springer (2010)
29. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: L. Beringer, A. Felty (eds.) ITP 2012, *LNCS*, vol. 7406, pp. 166–182. Springer (2012)
30. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4), 363–446 (2009)
31. Lochbihler, A.: A machine-checked, type-safe model of Java concurrency : Language, virtual machine, memory model, and verified compiler. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)
32. Lochbihler, A.: Light-weight containers. Archive of Formal Proofs (2013). `http://www.isa-afp.org/entries/Containers.shtml`, Formal proof development
33. Lochbihler, A.: Light-weight containers for Isabelle: efficient, extensible, nestable. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) ITP 2013, *LNCS*, vol. 7998, pp. 116–132. Springer (2013)
34. Lochbihler, A., Bulwahn, L.: Animating the formalised semantics of a Java-like language. In: M. van Eekelen, H. Geuvers, J. Schmalz, F. Wiedijk (eds.) ITP 2011, *LNCS*, vol. 6898, pp. 216–232. Springer (2011)
35. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. **411**(50), 4333–4356 (2010)
36. Musser, D.R., Stepanov, A.A.: Generic programming. In: P. Gianni (ed.) ISSAC 1988, *LNCS*, vol. 358, pp. 13–25. Springer (1989)
37. Nipkow, T.: Automatic functional correctness proofs for functional search trees. In: J.C. Blanchette, S. Merz (eds.) ITP 2016, *LNCS*, vol. 9807, pp. 307–322. Springer (2016)
38. Nipkow, T., Paulson, L.C.: Proof pearl: Defining functions over finite sets. In: J. Hurd, T. Melham (eds.) TPHOLs 2005, *LNCS*, vol. 3603, pp. 385–396. Springer (2005)
39. Nordhoff, B., Lammich, P.: Dijkstra's shortest path algorithm. Archive of Formal Proofs (2012). `http://www.isa-afp.org/entries/Dijkstra_Shortest_Path.shtml`, Formal proof development
40. Peyton Jones, S.: Bulk types with class. In: Haskell Workshop 1997 (1997)
41. Plotkin, G.D.: A note on inductive generalization. Machine intelligence **5**(1), 153–163 (1970)
42. Schimpf, A., Lammich, P.: Converting linear-time temporal logic to generalized Büchi automata. Archive of Formal Proofs (2014). `http://www.isa-afp.org/entries/LTL_to_GBA.shtml`, Formal proof development
43. Schimpf, A., Merz, S., Smaus, J.: Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In: TPHOLs 2009, *LNCS*, vol. 5674, pp. 424–439. Springer (2009)
44. Sozeau, M., Oury, N.: First-class type classes. In: O. Ait Mohamed, C. Muñoz, S. Tahar (eds.) TPHOLs 2008, *LNCS*, vol. 5170, pp. 278–293. Springer (2008)
45. Sternagel, C., Thiemann, R.: Deriving comparators and show functions in Isabelle/HOL. In: C. Urban, X. Zhang (eds.) ITP 2015, *LNCS*, vol. 9236, pp. 421–437. Springer (2015)
46. Sternagel, C., Thiemann, R., Winkler, S., Zankl, H.: CeTA – a tool for certified termination analysis. CoRR **abs/1208.1591** (2012). URL `http://arxiv.org/abs/1208.1591`
47. Thiemann, R.: Implementing field extensions of the form Q[sqrt(b)]. Archive of Formal Proofs (2014). `http://www.isa-afp.org/entries/Real_Impl.shtml`, Formal proof development
48. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: TPHOLs 2009, *LNCS*, vol. 5674, pp. 452–468. Springer (2009)
49. Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221–227 (1971)