Automatic Data Refinement

Peter Lammich

Technische Universität München, lammich@in.tum.de

Abstract. We present the Autoref tool for Isabelle/HOL, which automatically refines algorithms specified over abstract concepts like maps and sets to algorithms over concrete implementations like red-black-trees, and produces a refinement theorem. It is based on ideas borrowed from relational parametricity due to Reynolds and Wadler.
The tool allows for rapid prototyping of verified, executable algorithms. Moreover, it can be configured to fine-tune the result to the user's needs. Our tool is able to automatically instantiate generic algorithms, which greatly simplifies the implementation of executable data structures.
Thanks to its integration with the Isabelle Refinement Framework and the Isabelle Collection Framework, Autoref can be used as a backend to a stepwise refinement based development approach, having access to a rich library of verified data structures. We have evaluated the tool by synthesizing efficiently executable refinements for some complex algorithms, as well as by implementing a library of generic algorithms for maps and sets.

1 Introduction

If one wants to generate efficiently executable code for an algorithm verified in Isabelle/HOL, there are currently two alternatives. The first alternative is to do the formalization with executability in mind, e. g. using lists instead of sets. Then, Isabelle/HOL's code generator [7,9] can extract executable code from the formalization in various functional target languages like ML and Scala. However, being limited to only executable concepts in the formalization has the disadvantage of cluttering the proofs with implementation details. This makes the proofs more complicated, and may even render proofs of medium complex algorithms unmanageable.[1] Moreover, changing the implementation later means essentially redoing the whole formalization.

A well known solution to this problem is refinement [10], in particular refinement calculus [1,2]. Here, an algorithm is formulated and proven correct on an abstract level, and then refined towards an efficient implementation in possibly multiple refinement steps. As each refinement step preserves correctness, the resulting algorithm is correct. Stepwise refinement simplifies the proofs by modularization: The correctness proof of the abstract algorithm focuses on the algorithmic idea, not caring about implementation, while the proof of a refinement

[1] The author had this experience with early versions of algorithm formalizations [17,24].

step shows the correctness of the implementation of particular abstract concepts, not caring about the overall correctness.

In the context of code extraction from Isabelle/HOL, several approaches to data refinement have been explored [15,20]. They focus on the special case of *pure data refinement*, where abstract types (e. g. sets) are replaced by concrete implementations (e. g. red-black trees), but the structure of the algorithm is preserved. Conceptually, this refinement is simple: Rephrase the algorithm using efficient implementations for the abstract concepts and prove that it refines the original algorithm. However, using existing techniques, this still requires much effort. In particular, to produce multiple implementations from the same abstract algorithm, it has to be manually rephrased for each implementation. An alternative is to set up a parameterized version of the algorithm, e. g. by using locales. However, this approach suffers from limited polymorphism in typical HOL theorem provers (cf. [12]).

In this paper, we present the Autoref tool, which performs pure data refinement automatically. Given an algorithm phrased over abstract concepts like sets and maps, it automatically synthesizes a concrete, executable algorithm and the corresponding refinement theorem. It has heuristics that try to choose suitable implementations by default. Moreover, the defaults can easily be overridden by the user. Thus, it can be used for both rapid prototyping and generating the final, fine-tuned version of the code.

Autoref is based on the idea of relational parametricity [25,26], which is used to express data refinement for higher-order types.

To make it applicable for the development of actual algorithms, Autoref is integrated with the Isabelle Refinement Framework [19,18] and the Isabelle Collection Framework [16,15]. The former supports a development approach based on stepwise refinement, and the latter provides a large collection of verified data structures. Both tools have already been used for successful verification of complex algorithms [17,19,5,6]. Using Autoref as a back end greatly simplifies this development process. As a case study, we have applied Autoref to generate executable code for a nested depth-first search algorithm and an algorithm to compute simulation relations on finite state machines.

Another distinguishing feature of Autoref is its support for generic programming [21] in a user-transparent manner. During the synthesis, the concrete implementation of an abstract operation may be synthesized via a generic algorithm. To demonstrate this feature, we have developed a library of generic map and set algorithms.

Moreover, we provide implementations of data structures that overcome some limitations of the implementations provided by the Isabelle Collection Framework. Using Autoref's support for parametricity reasoning, we were able to generalize the existing implementations without redoing their correctness proofs.

The implementation of Autoref and the case studies are available at `https://www21.in.tum.de/~lammich/autoref`.

Related Work We already mentioned the concepts of data refinement [10], refinement calculus [1], and parametricity [25,26] that underly Autoref, as well as various manual approaches to data refinement [15,20].

The transfer package [13] for Isabelle/HOL can automatically transfer theorems over quotient types. It is also based on parametricity, and inspired us to use this technique in Autoref.

Parallel to our work, support for automatic data refinement has been integrated into the Isabelle/HOL code generator by Haftmann et al. [8], and also the verified code generator for HOL4 of Myreen et al. [22] supports automatic data refinement. This work is complementary to ours, and we provide a detailed comparison in Section 4.5.

The remainder of this paper is organized as follows: Section 2 describes the basic ideas that underly our tool. Section 3 describes how to implement a usable tool based on these ideas. Section 4 reports on our case studies, and, finally, Section 5 gives a short conclusion and outlook to future work.

2 Basic Ideas

In this section, we describe the basic ideas behind the Autoref tool. After a short introduction to Isabelle/HOL (§2.1), we describe relators (§2.2) and transfer rules (§2.3), and, finally, our treatment of equality and type classes (§2.4).

2.1 Isabelle/HOL

Autoref is implemented in Isabelle/HOL [23], an LCF-style theorem prover for higher order logic. However, the same approach could also be implemented within other HOL theorem provers. We assume the reader has basic knowledge of HOL-style theorem provers. In this subsection, we only describe some aspects of Isabelle/HOL that are essential for this paper.

A type in Isabelle/HOL is either a type variable or a type constructor applied to a list of types. Type variables are written with leading ticks, e. g. *'a*, and application of a type constructor is written in postfix notation, e. g. *'a list* or *('a,'b)prod*. Moreover, there is syntactic sugar for some standard types: the function type $'a \to 'b$, the product type $'a \times 'b$, and the sum type $'a + 'b$.

A term in Isabelle/HOL is either a constant, a variable, a bound variable, function application, or λ-abstraction. Constants and variables are annotated with their type, and λ-abstractions are annotated with the parameter type.

In Isabelle/HOL, there is a further distinction between schematic and free type/term variables. Schematic variables can be instantiated by unification, while fixed variables cannot. Schematic variables are denoted by a leading question mark, e. g. *?'a* or *?a*.

Schematic variables can be used for synthesis: For example, when starting with a proof goal of the form $?a = 1$, $?a$ may be instantiated during the proof. If we resolve the above goal with reflexivity, $?a$ is instantiated to 1, and the theorem that is proved is $1 = 1$. In contrast, free variables cannot be instantiated during

the proof. However, they are converted to schematic variables after the proof has been finished. Thus, reflexivity is stated as the goal $x{=}x$, and later gets converted to $?x{=}?x$. However, by convention, we do not use question marks for variables when referring to a theorem.

Isabelle/HOL has no naming conventions to distinguish free variables from constants. In this paper, the distinction between variables and constants should always be clear from the context.

2.2 Relators

In order to refine an abstract program to an executable, concrete one, all types and operations in the abstract program must be refined to concrete counterparts that are executable. For example, a set in the abstract program may become a red-black tree in the concrete program, and insertion into a set may become insertion into a red-black tree.

We use relators [3] to express the relationship between concrete and abstract types. Let T_A be an $n$ary type constructor, and let T_C be its concrete version. Then, a *relator*[2] R_T between T_C and T_A is an $n$ary function that maps relations between concrete and abstract argument types to a relation between T_C and T_A:

$$R_T :: ('c_1 \times 'a_1)set \rightarrow \ldots \rightarrow ('c_n \times 'a_n)set \rightarrow (('c_1,\ldots,'c_n)T_C \times ('a_1,\ldots,'a_n)T_A)set$$

We use the postfix notation $\langle R_1,\ldots,R_n\rangle R_T$ for relators, to make them similar to the notations $('c_1,\ldots,'c_n)T_C$ and $('a_1,\ldots,'a_n)T_A$ for the corresponding types.

A *natural* relator relates a type constructor to itself, not changing the shape of the values.

Example 1. Consider the list type *'a list ::= Nil | Cons 'a ('a list)*. The natural relator for lists relates two lists element-wise according to a relation on the elements. This relator is defined inductively: For each relation R, $\langle R\rangle list\_rel$ is the smallest relation that satisfies

$(Nil,Nil) \in \langle R\rangle list\_rel$
$[\![(a,a') \in R;\ (l,l') \in \langle R\rangle list\_rel]\!] \Longrightarrow (Cons\ a\ l,\ Cons\ a'\ l') \in \langle R\rangle list\_rel$

Similarly, natural relators can be defined for other algebraic types. The natural relator $\rightarrow$ for functions relates functions that produce related results when applied to related arguments. It is defined as

$$(f, f') \in R_a \rightarrow R_r \longleftrightarrow \forall (x,x') \in R_a.\ (f\ x,\ f'\ x') \in R_r$$

Functions and algebraic types are usually refined to themselves using their natural relators. However, types like maps or sets need to be represented differently. The relator *list_set_rel*, which relates distinct lists to finite sets, is defined as

[2] Relators are typically required to be monotonic, commute with composition and converse, and preserve identity [3]. However, as our technique does not rely on this, we call relator any function with the appropriate type. Actually, most of our relators satisfy these properties, a notable exception being the function relator.

$\langle R\rangle list\_set\_rel = \langle R\rangle list\_rel \circ \{\ (l,s).\ s = set\ l \wedge distinct\ l\ \}$

Here, $\circ$ is relational composition. That is, a list of concrete elements is first related to a list of distinct abstract elements, and this list is then related to a set of abstract elements.

2.3 Transfer Rules

In Isabelle/HOL, a program is represented as a term, which contains no schematic variables. Thus, in order to relate a concrete and an abstract term, we have to relate applications, abstractions, constants and free variables. We assume that an abstract constant (or free variable) $f'::'a_1\to\ldots\to'a_n$ is implemented by a concrete constant (or free variable) $f::'c_1\to\ldots\to'c_n$ of the same arity. In order to relate these constants, we have to prove a *transfer rule* of the form $(f,f')\in R_1\to\ldots\to R_n$. For abstraction and application, we use the following transfer rules:

$[\![\ \bigwedge x\ x'.\ (x,x')\in R_a \Longrightarrow (t,t')\in R_r\ ]\!] \Longrightarrow (\lambda x.\ t,\ \lambda x'.\ t')\in R_a\to R_r$
$[\![\ (f,f')\in R_a\to R_r;\ (x,x')\in R_a\ ]\!] \Longrightarrow (f\ x,\ f'\ x')\in R_r$

We now state the *synthesis problem* that our tool has to solve: Given an (abstract) term t' and transfer rules for its constants and free variables, synthesize a (concrete) term t and a relation R, such that $(t,t')\in R$ can be proven by the transfer rules.

Note that the synthesis problem is effective: As the rules decompose the structure of the abstract term, there are only finitely many proof trees for each term t'. In Isabelle/HOL, all solutions to the synthesis problem can be enumerated by solving the goal $(?t,t')\in ?R$ by repeated resolution with the transfer rules, using backtracking to recover from failed attempts or to explore further solutions. However, this approach may produce large search spaces. In Section 3 we describe our actual implementation of the synthesis.

Example 2. The transfer rules for the list constructors (cf. Example 1) are

$(Nil,Nil)\in\langle R\rangle list\_rel$
$(Cons,Cons)\in R\to\langle R\rangle list\_rel\to\langle R\rangle list\_rel$

Now consider the relator $int\_nat\_rel = \{(i,n).\ i = int\ n\}$ that relates integers to natural numbers.[3] The transfer rule for addition is

$(op\ +,\ op\ +)\in int\_nat\_rel\to int\_nat\_rel\to int\_nat\_rel$

Note that in Isabelle/HOL the +-operator is overloaded for both integers and natural numbers.

Moreover, consider the abstract term $t' = \lambda x\ y::nat.\ [x+y]$ that maps two natural numbers to a list, where $[x+y]$ is syntactic sugar for $Cons\ (x+y)\ Nil$. Trying to prove the goal $(?t,t')\in ?R$ by recursive resolution with the transfer rules results in the theorem

$(\lambda x\ y::int.\ [x+y],\ t')\in int\_nat\_rel\to int\_nat\_rel\to\langle int\_nat\_rel\rangle list\_rel$

[3] Implementing natural numbers by integers makes sense, as Isabelle/HOL uses a binary representation for integers, but a unary one for natural numbers.

2.4 Equality and Type Classes

Some refinements also depend on operations that are implicit on the abstract type, like equality or type class operations. In this case, the concrete operation needs to be parameterized by explicit concrete versions of these implicit abstract operations. Then, the transfer rules have the more general form $[\![(c_1, a_1) \in R_1; \ldots; (c_n, a_n) \in R_n]\!] \Longrightarrow (c\ c_1\ \ldots\ c_n, a) \in R$, where the constants a_i are the implicit abstract operations, and c_i are their implementations. Note that the synthesis problem remains effective, as repeated resolution with the transfer rules can only produce finitely many different subgoals. Thus, the finitely many possible proof trees where no subgoal occurs twice on a path can be enumerated.

Example 3. Reconsider the relator *list_set_rel* from Example 1, which implements finite sets by distinct lists. The membership operation $\in$ is implemented by searching the list for an equal element:

primrec *glist_member* :: (*'a*→*'a*→*bool*) → *'a* → *'a list* → *bool* **where**
  glist_member eq x [] ⟷ *False*
| *glist_member eq x* (*y*#*ys*) ⟷ *eq x y* ∨ *glist_member eq x ys*

It is parameterized with an equality operation. Note that we cannot use the default equality operation on the concrete side, as equality of abstract values does not necessarily imply equality of their implementations. For example, the set $\{1, 2\}$ is implemented by both lists, $[1, 2]$ and $[2, 1]$.

The transfer rule for the membership operation is the following:

(*eq,op* =) ∈ *R*→*R*→*Id* ⟹ (*glist_member eq,op* ∈) ∈ *R* → ⟨*R*⟩*list_set_rel* → *Id*

Thus, in order to transfer membership, we need to synthesize an additional equality operation on the element type. Note that we relate Booleans by their natural relator *Id*.

Other examples for implicit operations are hash codes and ordering operations, which are usually defined by type classes on the abstract type. Moreover, transfer rules with premises can be used to automatically instantiate generic algorithms, as described in Section 3.5.

2.5 Summary

In this section, we have described the basic machinery required to synthesize a (concrete) term t and a relation R from an (abstract) term t' such that $(t, t') \in R$ holds. In the next section, we tackle the additional problems that arise when using these ideas to implement a tool for automatic transfer of abstract programs to efficiently executable ones.

3 Tool Implementation

Given an abstract program specified by a term t', we ideally want to synthesize a term t and a relation R such that 1: (*t,t'*) ∈ *R* holds, 2: t is executable, 3: R

is adequate, and 4: t is optimally efficient. Criterion 1 is a by-product of our synthesis that works by actually proving $(t,t') \in R$. In the Isabelle/HOL setting, Criterion 2 has to be understood w. r. t. the code generator [7,9], which exports a functional fragment of HOL to functional languages like ML or Scala. If the left-hand sides of the transfer rules lay in this functional fragment, the synthesized term is executable. Thus, it is the responsibility of the user to specify transfer rules with executable left-hand sides. Otherwise, exporting the synthesized term will fail. Criterion 3 considers the refinement relations itself. A refinement relation should be able to uniquely represent a sufficiently large subset of the abstract type. Again, it is the responsibility of the user not to use inadequate refinement relations. Otherwise, the synthesized refinement theorem will be too weak to prove useful properties about the synthesized term. Finally, Criterion 4 is the most difficult to achieve, as it depends on many parameters outside the scope of our tool, like the algorithm itself, the expected distribution of input, etc. However, we provide some heuristics for selecting efficient implementations, as well as many configuration options that allow the user to fine-tune the result.

In the remainder of this section, we describe the actual implementation of Autoref and the heuristics.

3.1 Identification of Operations

The first step to solve the synthesis problem is to identify the abstract operations. In the previous section, we optimistically assumed that relators match the structure of the type and operations in the abstract term are expressed by single constants. However, these assumptions are not true in practice. For example, Isabelle/HOL represents maps from *'a* to *'b* by the type $'a \to 'b\ option$. Lookup in a map is expressed by function application, the empty map is $\lambda x.\ None$ and map update is $fun\_upd\ m\ k\ (Some\ v)$[4].

To handle this mismatch, we represent conceptual types like map or set by so called *interfaces*. We write $f :_i I$ to express that operation f has interface I. In order to identify operations that are not represented by a single constant, we define a set of *pattern rewrite rules* of the form $pat \equiv c\ x\_1 \ldots x\_n$.

The actual operation identification is then done by solving a type inference problem according to the following rules:

$$ctxt: \frac{x : I \in \Gamma}{\Gamma \vdash x : I} \quad pat: \frac{t \equiv t' \quad \Gamma \vdash t' : I}{\Gamma \vdash t : I} \quad const: \frac{c :_i I}{\Gamma \vdash c : I}$$

$$app: \frac{\Gamma \vdash t : I_1 \quad \Gamma \vdash f : I_1 \to I_2}{\Gamma \vdash f\ t : I_2} \quad abs: \frac{(x : I_1)\Gamma \vdash t : I_2}{\Gamma \vdash (\lambda x.\ t) : I_1 \to I_2}$$

Here, $I_1 \to I_2$ is the interface for functions. The rules are standard except for the *pat* rule, which replaces the current term according to a pattern rewrite rule. Our type inference algorithm first tries to apply the *pat* rule. Only if this does not

[4] The forms *Map.empty* and $m(k \mapsto v)$ are just syntactic sugar.

lead to a valid typing, it backtracks to use the *const*, *app*, or *abs* rules. If a typing is found, the term is rewritten according to the applied *pat* rules[5]. Moreover, to simplify later processing, we define tagging constants OP and \$ to indicate operations and application of operands: $OP\ c = c$ and $f\$\,x = f\,x$.

Example 4. The interface for maps is $\langle I_k,I_v\rangle i\_map$. Note that we use the same postfix notation for arguments of interfaces as for arguments of relators. The pattern rewrite rule for map lookup is $m\ k \equiv op\_map\_lookup\ m\ k$, and we have $op\_map\_lookup :_i \langle I_k,I_v\rangle i\_map \to I_k \to \langle I_v\rangle i\_option$.

Now consider the term $m\ k$ :: *'a option*. Autoref has to decide whether this is a lookup operation in the map m, or an application of the function m. The type inference first tries the pattern rewrite rule $m\ k \equiv op\_map\_lookup\ m\ k$, thus trying to derive a map interface for m. If this fails, it backtracks and uses the *app*-rule to derive a function interface for m. If m really has a map interface, after rewriting and adding the OP and \$ tags, the term becomes $OP\ op\_map\_lookup\,\$\,m\,\$\,k$. Otherwise, it becomes $m\,\$\,k$.

3.2 Selecting the Implementation Types

After it has identified the operations of the abstract term, Autoref decides what concrete types to use. We separate this decision from the actual synthesis mainly for efficiency reasons. In early versions of Autoref, we had serious efficiency problems due to extensive backtracking in the synthesis phase.

The goal of the next phase is to annotate each operation in the abstract term by the relation that will be used to transfer it to a concrete operation. This annotation is done by another tagging constant ::: that is defined as $f ::: R = f$. In order to influence the result of this phase, the user can manually place :::-annotations in the abstract term. This phase also implements some heuristics that aim at choosing efficient implementations.

Internally, this phase is split into multiple sub-phases, which successively instantiate relation variables to actual relations.

The first sub-phase uses the derived interface types to annotate each operation with a relation that consists of fresh variables and function relators, and also processes :::-annotations. After this sub-phase, every operation is annotated with a relation. Typically, most of these relations still contain fresh variables, and only a few have been specified by the user via explicit annotations.

The next sub-phase tries to restrict the possible instantiations of the relation variables by what we call *homogeneity rules*. The idea is that operations should preserve the implementation if possible. For this purpose, there is a set of homogeneity rules of the form $OP\ f ::: R$, and Autoref tries to unify the annotated operations in the term against the homogeneity rules, using a depth-first strategy. For each operation, a maximal specific homogeneity rule that has a unifier is taken. If there is no such rule, the original relation is not changed. This method propagates the user annotations over the operations, according to the

[5] The actual implementation combines type inference and rewriting.

homogeneity rules. The depth-first strategy ensures that user annotations are propagated upwards in the term, until they conflict with other user annotations.

Example 5. A typical setup provides a generic implementation for the set intersection operation, which iterates over the first set, performs a membership query in the second set, and builds up the result set. It may be instantiated for any combination of implementations of the first, second, and result set. Moreover, consider two set implementations with the relators *rbt_set_rel* and *list_set_rel*.

Assume the user wants to translate the term $a \cap (b ::: \langle R \rangle list\_set\_rel)$. After operator identification and relator annotation, the term becomes:

$$(OP \cap ::: ?R_1 \to \langle R \rangle list\_set\_rel \to ?R_2)\ \$\ a\ \$\ b$$

where $?R_1$ and $?R_2$ are fresh relator variables that need to be instantiated further. Autoref could safely use any relation for $?R_1$ and $?R_2$, as the generic implementation of $\cap$ works for any combination of relations. However, the user probably wanted both a, and the result of the intersection to be implemented via *list_set_rel*. This can be expressed by the homogeneity rule $OP \cap ::: R \to R \to R$. If applied, it instantiates both $?R_1$ and $?R_2$ to $\langle R \rangle list\_set\_rel$.

Now assume the user specified $(a \cap b) ::: \langle R \rangle rbt\_set\_rel$, thus explicitly requesting the result to be implemented by *rbt_set_rel*. Again, the homogeneity rule ensures that both a and b are implemented by *rbt_set_rel*, unless they contain different annotations.

Another useful homogeneity rule is $OP \cap ::: R \to R \to R'$, which tries to at least choose the same representation for both operands.

A homogeneity rule should not be able to render a possible implementable operation unimplementable. For example, if the only implementations of operation f' are $(f_1, f') ::: R_1 \to R_2$ and $(f_2, f') ::: R_2 \to R_1$, the homogeneity rule $OP\ f ::: R \to R$ would make the operation unimplementable.

After application of the homogeneity rules, the term may still contain uninstantiated relation variables. In the final sub-phase, all relation variables are instantiated by means of the available transfer rules. For each operation $OP\ f ::: R$ in the term, we try to find a transfer rule with a conclusion $(\_, f) \in R'$ such that R unifies with R', and instantiate R accordingly. This instantiation is done in a depth-first order, using backtracking until a solution is found. Premises of transfer rules are taken into account only if they have the form $(\_,\_) \in \_$.

In order to influence the solution, the transfer rules are ordered by priorities, such that rules with higher priority are tried first.

The priority of a rule is computed from a direct component, which may be annotated to the rule, and a relator component, which prefers transfer rules involving certain relators. For example, in order to prefer red-black trees over lists, one gives the relator *rbt_set_rel* a higher priority than *list_set_rel*. On the other hand, to prefer an optimized implementation of an operation over an unoptimized one, the transfer rule for the optimized implementation is annotated with a higher direct priority.

Note that the relator annotation phase may render solvable synthesis problems unsolvable. One reason are unsuitable homogeneity rules, as described above.

Another reason is that the last subphase does not consider all side conditions of transfer rules. However, when carefully setting up homogeneity and transfer rules, those effects will not occur. Thus we chose to accept this incompleteness for the advantage of a considerably faster synthesis.

3.3 Side Conditions

Apart from requiring implementations of equality and other type class operations, transfer rules may have other side conditions that need to be solved. For example, the Refinement Framework [19] sometimes requires relations to be single-valued, and functions to be monotonic. It already provides solvers for those properties, which we invoke from our tool.

Another complication arises for transfer rules with preconditions over operands. For example, the *hd*-operation, which returns the first element of a list, can only be transferred if the list is non-empty. Hence, the transfer rule for *hd* cannot be written in the form $(hd,hd) \in \langle R\rangle list\_rel \to R$. We solve this problem by also allowing transfer rules written in first-order form:

$$[\![\ l' \neq []; (l,l') \in \langle R\rangle list\_rel\ ]\!] \Longrightarrow (hd\ l,\ hd\ l') \in R$$

When applying transfer rules in first-order form to operations that do not have enough arguments, the operation is η-expanded. Note that η-expansion is always possible in Isabelle/HOL, as $f = \lambda x.\ f\ x$ is a theorem.

In order to be able to solve the side conditions, we have to augment some transfer rules to pass on additional information. For example, in order to transfer the term $If\ (l \neq [])\ (hd\ l)\ a$, we have to pass on information about the *If* statement during the transfer. For this purpose, we again use a first-order transfer rule:

$$[\![\ (c,c') \in Id;\ c \Longrightarrow (t,t') \in R;\ \neg c \Longrightarrow (e,e') \in R\ ]\!] \Longrightarrow (If\ c\ t\ e,\ If\ c'\ t'\ e') \in R$$

Thus, when transferring the *hd*-operation, $l \neq []$ is available as an assumption. We use similar rules for other crucial operations like the assertions from the Refinement Framework.

Side conditions may also be used for optimized implementations. Consider, for example, the insert operation for a set represented by a distinct list. If we know that the element is not yet contained in the set, it can be implemented in constant time by prepending the element to the list. Otherwise, we need linear time to check whether the element is already contained in the list. By giving the transfer rule for the optimized operation a higher priority, it is tried first. If it's side condition can be solved, the optimized version is used. Otherwise, the synthesis backtracks and uses the general transfer rule.

3.4 Synthesis

The last phase of Autoref takes the term t', which is completely annotated with relations, and constructs a proof goal of the form $(?t,t') \in ?R$. Here, $?t$ is a schematic variable that will be instantiated during the synthesis process, and R is the relation inferred for t' by the previous phase. Then, it tries to apply the

transfer rules to this goal in order of their priorities. After applying each transfer rule, the process is recursively invoked for the evolving subgoals. If solving one of the subgoals fails, the next matching transfer rule is tried. If the subgoal is not of the form $(t, t') \in R$, it is a side condition and Autoref analyzes its shape to find an adequate solver. As an additional optimization, the premises of a transfer rule are ordered such that side conditions concerning the abstract term or the relator come first. This avoids synthesizing the concrete term when side conditions over the abstract term or relation fail.

3.5 Generic Programming

Many abstract operations can be implemented in terms of other abstract operations. For example, we have $a \cap b = \{\} \longleftrightarrow \forall x \in a.\ x \notin b$, i. e. the disjointness test for sets can be implemented by means of bounded quantification and membership query. Along these lines, most operations on finite sets can be implemented by five basic operations: Empty set, insert, membership, deletion of an element, and iteration over the elements of the set. We extensively exploit this idea already in the Isabelle Collection Framework [16,15]. However, there we have to manually pre-instantiate the generic algorithms for each combination of implementations, which does not scale. Using Autoref, generic algorithms are expressed as transfer rules, and automatically instantiated only on demand. Moreover, the usage of generic algorithms is transparent to the user, who specifies an abstract operation, and lets the tool decide whether it is realized by a direct implementation or a generic algorithm.

Example 6. Reconsider the disjointness test. We define the constant

$$op\_set\_disjoint\ a\ b \longleftrightarrow a \cap b = \{\}$$

and add an appropriate rewrite rule to the operation identification phase. Moreover, we define

$$gen\_disjoint\ ballI\ memI\ a\ b = ballI\ a\ (\lambda x.\ \neg memI\ x\ b)$$

Then, we can easily prove the following transfer rule:

$$[\![\ (b, Ball) \in \langle Re \rangle Rs_1 \to (Re \to Id) \to Id;\ (m,\ op\ \in) \in Re \to \langle Re \rangle Rs_2 \to Id\ ]\!] \\ \Longrightarrow (\ gen\_disjoint\ b\ m,\ OP\ op\_set\_disjoint) \in \langle Re \rangle Rs_1 \to \langle Re \rangle Rs_2 \to Id$$

A low direct priority ensures that it does not override explicit rules for disjointness tests. Thus, whenever Autoref finds no explicit rule for a disjointness test, it tries to find rules for bounded quantification and membership instead, and automatically implements the disjointness test by those operations.

Using such rules, we have to be careful not to follow cycles, trying to implement an operation by means of itself. Checking for such cycles is not yet implemented. Thus, it is the responsibility of the user not to use transfer rule setups with cyclic dependencies. However, even with this restriction, we were able to implement generic algorithm libraries for maps and sets (cf. Section 4.3).

3.6 Summary

In this section we have described how to use the basic idea of synthesis via transfer rules to implement the Autoref tool, which automatically synthesizes efficient implementations of abstractly specified algorithms. The tool has several heuristics that try to automatically produce a suitable implementation. If these heuristics produce a non-adequate result, the user can influence the result by configuration of the heuristics and annotations to the abstract algorithm. In the next section, we present some case studies that prove the practical usefulness of Autoref.

4 Case Studies

In this section, we describe the integration of Autoref with the Isabelle Refinement Framework [18,19] and the Isabelle Collection Framework [16,15]. Moreover, we describe a library of generic map and set algorithms that demonstrates the generic programming capabilities of Autoref. Finally, we report on the automatic refinement of some complex algorithms to efficiently executable code.

4.1 Refinement Framework

In order to be useful for practical algorithms, we have set up Autoref as a back end to the stepwise refinement development process provided by the Isabelle Refinement Framework [18,19].

A detailed description of the Refinement Framework can be found in [19]. Here, we give a very brief overview. The basic concept of the Refinement Framework is a nondeterminism monad, whose inner type is called *result*. A result is either a set of values, describing the possible outcomes of a nondeterministic computation, or it is the special result **fail**, describing that one of the possible outcomes is an exception, i. e. a failed assertion or diverging computation. By lifting the subset ordering, with **fail** being the biggest element, one gets a complete lattice structure on results. The lifted ordering is called *refinement ordering*, where smaller results are more refined. An algorithm is expressed as a function yielding a result. Correctness of an algorithm is expressed by refinement of its specification, e. g. $\Phi \Longrightarrow f\ x \leq \mathbf{spec}\ \Psi$ describes correctness of f w. r. t. precondition Φ and postcondition Ψ. Here, $\mathbf{spec}\ \Psi$ is the result that contains all values satisfying Ψ.

Given a (single-valued) refinement relation R, the concretization function $\Downarrow R$ maps abstract results to concrete results w. r. t. R. Thus, data refinement is expressed by $r \leq \Downarrow R\ r'$, meaning that r refines r' w. r. t. the relation R.

In order to integrate the Refinement Framework with Autoref, we define data refinement as a relator for results: $\langle R\rangle nres\_rel = \{(c,a).\ c \leq \Downarrow R\ a\}$. Then, we provide transfer rules for the combinators of the Refinement Framework. Those transfer rules are already contained in the Refinement Framework, and only have to be rephrased in the format expected by Autoref. Some of the transfer rules have side conditions, for which the Refinement Framework already provides solvers, which could easily be integrated into Autoref.

4.2 Collection Framework

The Isabelle Collection Framework [15,16] provides a rich library of verified collection data structures, and is already based on data refinement. Thus, it is straightforward to set up Autoref to use the data structures provided by the Collection Framework.

However, the Collection Framework only supports refinement relations of the form $\langle Id,\ldots,Id\rangle R$. For example, it is not possible to refine a set of sets of integers to a list of lists of integers. Thus, we implemented a red-black tree based map implementation and a list-based set implementation that do not have this restriction. Using parametricity [26], we were able to reuse the existing theorems about red-black trees and lists, as illustrated in the following example:

Example 7. The existing implementation of sets by distinct lists gives us the following transfer rule:

$$(\mathit{list\_member},\mathit{Set.member}) \in \mathit{Id} \to \{(l,s).\ s = \mathit{set}\ l \wedge \mathit{distinct}\ l\} \to \mathit{Id}$$

Here, *list_member* implicitly uses equality on the elements. It is straightforward to show $\mathit{list\_member} = \mathit{glist\_member}\ (\mathit{op} =)$, where *glist_member* is the one from Example 3. Moreover, Autoref easily shows that *glist_member* is parametric[6]:

$$(\mathit{glist\_member},\ \mathit{glist\_member}) \in (R \to R \to \mathit{Id}) \to R \to \langle R\rangle \mathit{list\_rel} \to \mathit{Id}$$

Combining these theorems, one gets precisely the transfer rule from Example 3.

4.3 Generic Programming

In Section 3.5, we sketched how Autoref can be used for generic programming. In order to demonstrate this feature, we implemented a library of generic map algorithms, which provides a variety of operations based on the five basic operations empty, update, lookup, delete, and iterate. Analogously, we implemented generic set algorithms based on the basic operations empty, insert, member, delete, and iterate. Finally, we implemented the basic set operations by the basic map operations, using a map from elements to unit values to represent a set.

Thus, in order to prototype a new data structure, it is enough to implement the five basic map operations. All other map and set operations become available automatically. Most of the generic algorithms are reasonable efficient, such that they can be kept even for the final version. To specialize a generic algorithm for a particular implementation, it is sufficient to add the specialized transfer rule with a higher priority than the generic rule. For example, we have a generic algorithm for union of finite sets that iterates over one set and inserts its elements into the other set. However, for red-black trees, there is a more efficient algorithm. It is declared as a transfer rule with default priority, thus overriding the lower priority rule for the generic algorithm.

[6] Indeed, this is a theorem that you get for free in the setting of [26]!

4.4 Code Generation for Actual Algorithms

We have tested Autoref on several actual algorithms. The most complex ones are the algorithm by Ilie, Navarro and Yu for the computation of simulation preorders in nondeterministic finite automata [14], and an emptiness check for Buchi automata using a nested depth-first search [11]. For the former algorithm, we adapted an existing formalization [5], where the refinement to executable code was done manually[7]. Here, the size of the Isabelle text required for the refinement to executable code was reduced from more than 500 lines to about 15 lines. In order to use Autoref, we had to insert two additional assertions into the abstract algorithm, which were required to automatically discharge side conditions of transfer rules. In the original formalization, these side conditions were discharged using some non-trivial reasoning during the manual refinement.

For the latter algorithm, the refinement to executable code requires about 10 lines. Moreover, we require about 20 lines for setup of a custom datatype, for which automation is not yet supported. As this algorithm was initially developed using Autoref, we have no data how big a manual refinement would be, but we estimate it to several hundred lines of code.

4.5 Data Refinement within the Code Generator

The Isabelle/HOL code generator also supports automatic data refinement [8]. However, it has some limitations that render it unsuitable for our purpose, namely code generation for programs defined in the Refinement Framework. For example, the refinement relations are restricted to the form $\alpha\ c = a$. This is essential for integration into the Isabelle/HOL code generator. However, it is not possible to express reduction of nondeterminism, which is required to be used as back end for the Refinement Framework. Moreover, it lacks the operation identification of our tool, thus limiting the refinement to types with their own type constructor. On the other hand, due to the direct integration into the code generator, one gets support of the Isabelle packages for defining recursive functions and algebraic datatypes for free, and tools like evaluate and quickcheck [4] immediately profit from the more efficient code. Here, Autoref currently requires manual setup for each non-primitive recursion scheme and for each algebraic datatype, and automating this task would require quite some effort.

The code generator of Myreen et al. [22] for the HOL4 theorem prover translates terms to the deeply embedded MiniML language, and proves correctness of the translation. It uses a synthesis procedure that is similar to ours, i. e. it keeps track of a relation between the generated code and the original term. While the currently implemented features seems to be limited[8], in theory it should be possible to support the same generality as Autoref does, which yields an interesting topic for future research.

[7] A very early prototype of Autoref was already used for some simple steps.
[8] For example, equality on abstract values is mapped to equality in the target language.

5 Conclusions

We have presented Autoref, a tool for automatic data refinement in Isabelle/HOL. Given an abstract algorithm that uses abstract concepts like maps or sets, it synthesizes a concrete algorithm that uses efficient implementations like red-black trees, and a corresponding refinement theorem. Autoref allows for both rapid prototyping of executable code and fine-tuning the results to get the final version. Compared to previous manual approaches, our tool saves the user from tedious and time consuming writing of boilerplate code. To substantiate the usefulness of Autoref, we have shown how it can be used to refine actual algorithms for simulation preorder computation and for nested depth-first search.

Directions of future work include to add even more automation. For example, transfer rules with natural relators correspond to the "theorems for free" of [26], and could be derived automatically. Moreover, we are currently working on several algorithm verifications using the Refinement Framework. The feedback from those projects will lead to improvements of the tool, and extension of its data structure and generic algorithm libraries. Another interesting topic is to use the heuristics that we developed for Autoref as a front end to the code generator based data refinement [8].

Acknowledgements We thank Andrei Popescu for proofreading and inspiring discussions about parametricity, and the anonymous reviewers for their useful comments.

References

1. Back, R.J.: On the correctness of refinement steps in program development. Ph.D. thesis, Department of Computer Science, University of Helsinki (1978)
2. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
3. Backhouse, R.C., de Bruin, P., Malcolm, G., Voermans, E., van der Woude, J.: Relational catamorphisms. In: Proc. of the IFIP TC2/WG2.1 Working Conference on Constructing Programs. Elsevier Science Publishers BV (1991)
4. Bulwahn, L.: The new quickcheck in Isabelle: Random, exhaustive and symbolic testing under one roof. In: Proc. of CPP. LNCS, vol. 7679, pp. 92–108. Springer (2012)
5. Eberl, M.: Efficient and Verified Computation of Simulation Relations on NFAs. Bachelor's thesis, Technische Universität München (2012)
6. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker (2013), to appear in Proc. of CAV 2013
7. Haftmann, F.: Code Generation from Specifications in Higher Order Logic. Ph.D. thesis, Technische Universität München (2009)
8. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL (2013), to appear in Proc. of ITP 2013
9. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Functional and Logic Programming (FLOPS 2010). LNCS, Springer (2010)

10. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1, 271–281 (1972)
11. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: Proc. of SPIN Workshop. Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–32. American Mathematical Society (1997)
12. Homeier, P.V.: The HOL-Omega logic. In: Proc. of TPHOLs. LNCS, vol. 5674, pp. 244–259. Springer (2009)
13. Huffman, B., Kunar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL (2012), isabelle Users Workshop 2012
14. Ilie, L., Navarro, G., Yu, S.: On NFA reductions. In: Theory Is Forever, LNCS, vol. 3113, pp. 112–124. Springer (2004)
15. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Proc. of ITP. LNCS, vol. 6172, pp. 339–354. Springer (2010)
16. Lammich, P.: Collections framework. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Collections.shtml` (Dec 2009), formal proof development
17. Lammich, P.: Tree automata. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Tree-Automata.shtml` (Dec 2009), formal proof development
18. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Refine_Monadic.shtml` (2012), formal proof development
19. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Proc. of ITP. LNCS, vol. 7406, pp. 166–182. Springer (2012)
20. Lochbihler, A., Bulwahn, L.: Animating the formalised semantics of a Java-like language. In: Proc. of ITP. LNCS, vol. 6898, pp. 216–232. Springer (2011)
21. Musser, D.R., Stepanov, A.A.: Generic programming. In: Proc. of ISSAC. LNCS, vol. 358, pp. 13–25. Springer (1989)
22. Myreen, M.O., Owens, S.: Proof-producing synthesis of ML from higher-order logic. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. pp. 115–126. ICFP '12, ACM (2012)
23. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
24. Nordhoff, B., Lammich, P.: Formalization of Dijkstra's algorithm (2012), formal proof development
25. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
26. Wadler, P.: Theorems for free! In: Proc. of FPCA. pp. 347–359. ACM (1989)