The Isabelle Collections Framework

Peter Lammich[1] and Andreas Lochbihler[2]

[1] Universität Münster, peter.lammich@uni-muenster.de
[2] Karlsruher Institut für Technologie, andreas.lochbihler@kit.edu

Abstract. The Isabelle Collections Framework (ICF) provides a unified framework for using verified collection data structures in Isabelle/HOL formalizations and generating efficient functional code in ML, Haskell, and OCaml. Thanks to its modularity, it is easily extensible and supports switching to different data structures any time. For good integration with applications, a data refinement approach separates the correctness proofs from implementation details. The generated code based on the ICF lies in better complexity classes than the one that uses Isabelle's default setup (logarithmic vs. linear time). In a case study with tree automata, we demonstrate that the ICF is easy to use and efficient: An ICF based, verified tree automata library outperforms the unverified Timbuk/Taml library by a factor of 14.

1 Introduction

Isabelle/HOL [15] is an interactive theorem prover for higher order logic. Its code generator [7] extracts (verified) executable code in various functional languages from formalizations. However, the generated code often suffers from being prohibitively slow. Finite sets and maps are represented by chains of pointwise function updates, whose memory usage and run time are unacceptable for larger collections in practice. For example, to obtain an operative implementation, de Dios and Peña manually edited their generated code such that it used a balanced-tree data structure from the Haskell library [5, Sec. 5]. Not only are such manual changes cumbersome and error-prone as they must be redone each time the code is generated, they in fact undermine the trust obtained via formal verification.

There are some Isabelle/HOL formalizations of efficient collection data structures such as red-black trees (RBT), AVL trees [16], and unbalanced binary-search trees [11], each providing its own proprietary interface. This forces the user to chose the data structures at the start of formalization, and severely hinders switching to another data structure later. Moreover, wherever efficient data structures replace the standard types for sets and maps, one runs the risk of cluttering proofs with details from the data structure implementation, which obfuscates the real point of the proof. Furthermore, ad-hoc implementations of efficient data structures are scattered across other projects, thus limiting code reuse. For example, Berghofer and Reiter implemented tries for binary strings (called BDDs there), within a solver for Presburger arithmetic [2].

This paper presents the Isabelle Collections Framework (ICF) that addresses the above problems. The main contribution is a unified framework (Sec. 2) to

define and use verified collection data structures in Isabelle/HOL and extract verified and efficient code. As it works completely inside the logic, it neither increases the trusted code base, nor does it require editing the extracted code. The ICF integrates both existing (red-black trees, associative lists) and new (hashing, tries, array lists) formalizations of collection data structures. It provides a unified abstract interface that is sufficient for defining and verifying algorithms – independently of any concrete data structure implementation. This permits to change the actual data structure at any point without affecting the correctness proofs. To easily integrate existing data structures, the ICF contains a library of generic algorithms that implement most operations from a few basic operations. The ICF uses a data refinement approach that transfers the correctness statements from the abstract specification level to the concrete data structure implementation; Sec. 3 contains a small, but non-trivial example. With this approach, the ICF integrates well in existing formalizations.

Another contribution is our evaluation of the ICF (Sec. 4): (i) To benchmark its performance, we compared the ICF to the standard code generator setup and to library data structures of Haskell, OCaml, and Java. (ii) To demonstrate its usability in a case study, we implemented a formally verified tree-automata library [13] based on the ICF, using the data refinement approach. The ICF based tree-automata library outperforms the OCaml-based Timbuk/Taml library [6] by a factor of 14 and is competitive with the Java library LETHAL [14].

The ICF is published electronically in the *Archive of Formal Proofs* [12]. As the AFP is only updated with new Isabelle releases, a more recent version may be available at http://cs.uni-muenster.de/sev/projects/icf/.

1.1 Related Work

Most interactive theorem provers provide some efficient data structures in their libraries. The Coq standard library [4] features a modular specification for maps and sets that mirrors OCaml's library except for iterators. There are implementations based on strictly ordered (association) lists and on AVL trees for both sets and maps. There is also a trie implementation for maps with binary strings as keys. Coq's type system and code extraction facility allow the inclusion of data structure invariants (orderedness for lists and the search tree property for AVL trees) in the type definition without losing the capability to generate code. At present, Isabelle does not support this, i.e., the data structure invariants must be carried through all theorems explicitly. For ACL2 [10], there is a set implementation based on ordered lists, too.

For Haskell, Peyton Jones [17] proposes an elegant collections framework that uses type constructor classes and multi-parameter constructor classes. Unfortunately, Isabelle's type system supports neither of them.

The C++ Standard Template Library (STL) [18] provides the abstract concepts for the ICF: concepts (= ADTs), container classes (= implementations), algorithms (= generic algorithm), and iterators. In the STL, iterators are first-class objects that represent the state of an iteration. In ICF, iterators are realized as combinators. While the first approach is more general (e.g., it allows one to

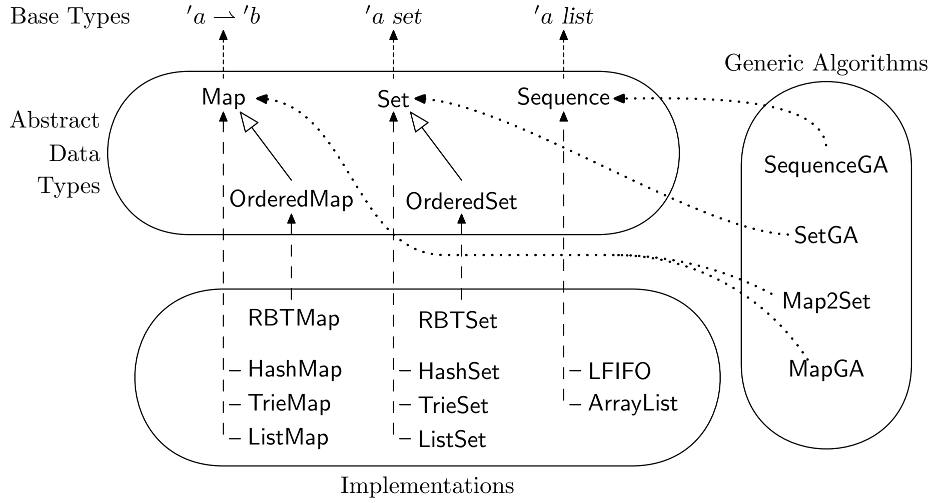

Fig. 1. The structure of the Isabelle Collections Framework

iterate simultaneously over multiple data structures), ours is more convenient for use with a functional language. In C++, the compiler instantiates the templates automatically. This is not possible in Isabelle/HOL. Instead, the ICF contains (automatically generated) explicit instantiations of the generic algorithms. The user has to select the appropriate instantiation, which is easy due to a uniform naming scheme.

As for Java, the Java Collections Framework [9] provides an object-oriented approach. Interfaces describe ADTs. Concrete data structures implement them in classes. Generic algorithms are provided by means of static methods (in the `java.util.Collections` class) and abstract collection classes, which provide default implementations for most operations based on just a few basic ones (e.g. `java.util.AbstractSet`). Dynamic dispatch takes the role of instantiating the generic algorithms.

2 Overview of the Framework

Figure 1 outlines the structure of the Isabelle Collections Framework (ICF). Its main components are *abstract data types* (Sec. 2.1), *generic algorithms* (Sec. 2.4), and *implementations* (Sec. 2.6). An abstract data type (ADT) specifies a set of operations and their behavior (e.g. a set with empty, member, insert, delete and iteration operations) w.r.t. a base type in Isabelle (Sec. 2.2). ADTs can extend other ADTs, denoted by solid lines. An implementation provides an actual data structure with operations (dashed lines), and proves that they match the specification of the ADT (e.g. HashSet implements the ADT set). A generic algorithm implements some operation via other ADT operations (dotted lines) – indepen-

dently of the concrete implementation. For this, the *iterators* (Sec. 2.3) that each ADT provides are extremely useful.

2.1 Abstract Data Types

The ICF currently supports three different abstract data types:

Maps A map is a partial function from keys to values with finite domain. The operations include the empty map constructor, emptiness check, lookup, update, deletion, composition, iteration, conversion to associative lists, and quantification and choice over the domain. Ordered maps extend maps in that they require a linear order on the keys and provide operations to iterate over the keys in ascending and descending order. Currently, there are map implementations using association lists, red-black trees (RBT), hashing, and tries. The RBT implementation is also an ordered map.

Sets A finite set with empty set constructor, insertion, intersection, union, difference, emptiness, membership and subset checks, cardinality, iteration, image, and choice operations. Like for maps, the ICF specifies ordered sets that require a linear order on the elements and provide ordered iteration. Except for ListSet, the set implementations are derived from the corresponding map implementations via the generic algorithm Map2Set.

Sequences A finite sequence. Unlike sets and maps, the insertion order determines the iteration order. Implementations are a queue LFIFO with amortized constant-time *enqueue* and *dequeue* as well as *push* and *pop* operations, and a resizable array implementation that provides access by index positions.

2.2 Data Refinement

The operations of an ADT are specified by its intended behavior w.r.t. an abstraction mapping α that abstracts from the concrete implementation to the so called *base type* of the ADT. The base type of sets is Isabelle/HOL's type $'a\ set$, maps have the base type $'k \rightarrow 'v\ option$[3], and sequences have the base type $'a\ list$. For example, the *empty*, *memb*, and *ins* operations of the ADT set are specified as follows:[4]

$$\begin{array}{ll} \textsf{empty-correct}: & \alpha\ empty = \{\} \\ \textsf{memb-correct}: & memb\ x\ s \Leftrightarrow x \in (\alpha\ s) \\ \textsf{ins-correct}: & \alpha\ (ins\ x\ s) = \{x\} \cup (\alpha\ s) \end{array}$$

A proposition that involves ADT operations is usually proved in two steps:

1. Transform it into a proposition that only involves operations on the base type. This is straightforward using the specifications of the ADT's operations and usually done automatically by Isabelle's simplifier.

[3] The data type $'a\ option = None \mid Some\ 'a$ corresponds to `Maybe a` in Haskell.

[4] To simplify the presentation, we omitted the data structure invariant *invar*, which guards all specifications for abstract data types (cf. Sec. 2.5).

2. Prove the transformed proposition. Since it only involves base types, this proof enjoys the full support of Isabelle's automated proof methods and is completely independent of the ICF.

For example, the proposition $\mathit{memb}\ y\ (\mathit{ins}\ x\ (\mathit{ins}\ y\ (\mathit{ins}\ z\ \mathit{empty})))$ is first transformed to $y \in \{x, y, z\}$ and then proved automatically by the method auto.

Hence, when using the ICF to implement some algorithm, the algorithm is first formalized and proved correct on the base type – independently of the ICF. In a second (straightforward) step, definitions are transformed to use the ADTs of the ICF, and correct data refinement is shown. This approach also simplifies porting existing formalizations to the ICF, which requires only the second step, while the existing correctness proofs remain untouched. Section 3 presents an example of this approach in detail.

2.3 Iterators

Iterators are one of the ICF's key concepts. An iterator over a finite set or map is a generalized *fold* combinator: It applies a state-transforming function to all elements of a set (entries of a map, resp.), starting with an initial state and returning the final state. Additionally, the iteration is interrupted if a continuation condition on the state no longer holds. The iteration order is unspecified.

The Isabelle/HOL standard library provides an uninterruptible *fold* combinator for finite sets that requires the state-transformer to be left-commutative[5] to ensure that the iteration result does not depend on the iteration order. However, generic algorithms in the ICF typically use state-transformers that are not left-commutative. Consider, e.g., a generic algorithm for coercion between set implementations. It iterates over the source set and inserts each element into the target set which is initially empty. Although inserting is left-commutative on the base type, the shape of the target set's data structure usually depends on the insertion order, i.e. inserting is *not* left-commutative for data structures. Hence, ICF iterators do not require left-commutativity.

An iterator *iterate* on a set data structure of type $'s$ has the type $('\sigma \to \mathit{bool}) \to ('a \to '\sigma \to '\sigma) \to 's \to '\sigma \to '\sigma$. It takes a continuation condition c, a state transformer f, the set s, and the initial state σ. For reasoning, the following rule is used, which requires an iteration invariant I:

$$[\text{iterate-rule}]\ \frac{\begin{array}{c} I\ (\alpha\ s)\ \sigma_0 \\ \forall x\ i\ \sigma.\ c\ \sigma \wedge x \in i \wedge i \subseteq \alpha\ s \wedge I\ i\ \sigma \Longrightarrow I\ (i - \{x\})\ (f\ x\ \sigma) \\ \forall \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma \\ \forall \sigma\ i.\ i \subseteq \alpha\ s \wedge i \neq \{\} \wedge \neg c\ \sigma \wedge I\ i\ \sigma \Longrightarrow P\ \sigma \end{array}}{P\ (\mathit{iterate}\ c\ f\ s\ \sigma_0)}$$

The iteration invariant $I :: 'x\ \mathit{set} \to '\sigma \to \mathit{bool}$ takes two parameters: (i) the iteration state i denotes the set of elements that still needs to be iterated over, and (ii) the computed state σ of type $'\sigma$. To establish a property P of the resulting state, it must be shown that:

[5] A function $f :: 'a \to '\sigma \to '\sigma$ is called *left-commutative* iff $\forall x\ y.\ f\ x \circ f\ y = f\ y \circ f\ x$.

1. the iteration invariant I holds for the initial state σ_0 with the whole set $\alpha\ s$ unprocessed,
2. the state transformer f preserves I for any removal of any element from any subset of $\alpha\ s$, and
3. $I\ i\ \sigma$ implies $P\ \sigma$, when the iteration stops either normally ($i = \{\}$) or prematurely ($\neg c\ \sigma$, $i \neq \{\}$).

Example 1. The following algorithm *copy* copies a set from a source implementation (indexed 1) to a target implementation (indexed 2).

$$copy\ s_1 = iterate_1\ (\lambda\sigma.\ True)\ ins_2\ s_1\ empty_2$$

The continuation condition ($\lambda\sigma.\ True$) ensures that iteration does not stop prematurely. The iteration state is the data structure of the target implementation. The initial state is the empty set $empty_2$, and the state transformer function is the insert function ins_2 of the target implementation. To prove *copy* correct – i.e. if $invar_1\ s_1$, then $invar_2\ (copy\ s_1)$ and $\alpha_2\ (copy\ s_1) = \alpha_1\ s_1$ – we use the iteration invariant $I\ i\ s_2 = (\alpha_2\ s_2 = (\alpha_1\ s_1) - i \wedge invar_2\ s_2)$.

Example 2. Bounded existential quantification ($\exists x \in s.\ P\ x$) can be implemented via iteration: $bex\ s\ P = iterate\ (\lambda\sigma.\ \neg\sigma)\ (\lambda x\ \sigma.\ P\ x)\ s\ False$. The state of this iteration is a Boolean that becomes true when the first element satisfying P is found. The iteration stops prematurely when the state becomes true.

For proving *bex* correct – i.e. if $invar\ s$, then $bex\ s\ P = (\exists x \in \alpha\ s.\ P\ x)$ – we use the iteration invariant $I\ i\ \sigma = (\sigma = (\exists x \in (\alpha\ s) - i.\ P\ x))$.

2.4 Generic Algorithms

A generic algorithm implements and proves correct a *target operation* by means of a set of *source operations*, independently from the actual data structure. To obtain an implementation of the target operation together with its correctness statement, the generic algorithm and its correctness statement are instantiated with actual implementations of the source operations.

Generic algorithms reduce redundancy because an algorithm needs to be proved correct only once and is then instantiated for various implementations of the involved ADTs. For example, the *map-to-nat* function computes a bijective map from a finite set into an initial segment of the natural numbers. It is defined and proved correct independently of the actual map and set implementations.

Generic algorithms are also used to reduce the effort of creating a new implementation. The ADTs provide generic algorithms to derive most operations from a small set of basic operations, using iterators as a key concept. For example, all specified map and set operations can be implemented by iterators and four basic operations: the empty map or set constructor, lookup or membership test, insertion, and deletion. In a later development stage, these generic implementations may be replaced by versions optimized for the actual data structure. However, as the generic algorithms are reasonably efficient, this is often not necessary. As a special case, the ICF contains generic algorithms to derive a set implementation from a map implementation by using a map with value type *unit*, whose only element is ().

2.5 Realization within Isabelle/HOL

In this section, technical details and challenges of the ICF's realization within Isabelle/HOL are discussed.

Abstract Data Types. The ICF uses Isabelle/HOL's locale mechanism[6] [1] to specify an ADT. For each ADT, a base locale fixes the data structure invariant and the abstraction function. Each operation is specified by its own locale, which extends the base locale, fixes the operation and specifies its behavior. For example, the following locales specify the ADT set and its delete operation:

locale *set* = **fixes** α :: $'s \rightarrow {'a}\ set$ **and** *invar* :: $'s \rightarrow bool$

locale *set-delete* = *set* +
  fixes *delete* :: $'a \rightarrow {'s} \rightarrow {'s}$
  assumes delete-correct :
    $invar\ s \Longrightarrow \alpha\ (delete\ x\ s) = (\alpha\ s) - \{x\}$
    $invar\ s \Longrightarrow invar\ (delete\ x\ s)$

Note that the data structure invariant *invar* guards all specification equations to allow for ADT implementations that require invariants.

Implementations. An implementation interprets the locales with the operations it provides, thereby showing that they satisfy the ADT's specification. The *HashSet* implementation, e.g., defines the functions *hs-*α, *hs-invar*, and *hs-delete* and proves the lemma hs-delete-impl: *set-delete hs-*α *hs-invar hs-delete* where *set-delete* denotes the assumption predicate of the locale *set-delete*. From this lemma, interpretation produces the lemma hs.delete-correct, which is delete-correct with the parameters instantiated by *hs-*α, *hs-invar*, and *hs-delete*. Note that the dot (instead of a dash) in *hs.delete-correct* has technical reasons.

Naming Conventions. The ICF uses several naming conventions that simplify its usage: The locale specifying an operation *op* for an ADT *adt* is named *adt-op* (e.g., *set-delete*). The correctness assumption is called *op-correct* (e.g., *delete-correct*). Each implementation of an ADT has a short (usually two letters) prefix (e.g., *hs* for HashSet). An implementation with prefix *pp* provides a lemma *pp-op-impl* and interprets the operation's locale with the prefix *pp*, yielding the lemma *pp.op-correct*.

Data Refinement. The proof of the data refinement step is, in many cases, performed automatically by the simplifier. In some complex cases, involving, e.g., recursive definitions or nested ADTs, a small amount of user interaction is necessary. Section 3 contains an example for such a complex case.

[6] Locales provide named local contexts with fixed parameters (**fixes**) and assumptions (**assumes**). They support inheritance (+) and interpretation (i.e., parameter instantiation), which requires to discharge the assumptions. A predicate with the locale's name collects all assumptions of the locale.

Generic Algorithms. A generic algorithm is defined as a function that takes the source operations as arguments. The correctness lemma shows that the target operation meets its specification if the source operations meet theirs.

Example 3. Reconsider the bounded existential quantification from Ex. 2, where an *iterate* operation was used. In a generic algorithm, this operation becomes an additional parameter:

$$\mathit{bex}\ \mathit{iterate}\ s\ P = \mathit{iterate}\ (\lambda\sigma.\ \neg\sigma)\ (\lambda x\ \sigma.\ P\ x)\ s\ \mathit{False}$$

Assume that the locale *set-bex* specifies bounded existential quantification. Then, we prove the correctness lemma

$$\textsf{bex-correct}:\ \mathit{set\text{-}iterate}\ \alpha\ \mathit{invar}\ \mathit{iterate} \Longrightarrow \mathit{set\text{-}bex}\ \alpha\ \mathit{invar}\ (\mathit{bex}\ \mathit{iterate})$$

An instantiation then sets the parameters α, *invar*, and *iterate* to its operations. In Isabelle/HOL, this is easily done with the *OF* and *folded* attributes, as illustrated in the following example, that instantiates the algorithm for HashSets:

definition *hs-bex* = *bex hs-iterate*
lemmas *hs-bex-impl* = *bex-correct*[OF *hs-iterate-impl*, folded *hs-bex-def*]

where hs-iterate-impl: *set-iterate hs-*α *hs-invar hs-iterate*. Hence, we get the lemma hs-bex-impl: *set-bex hs-*α *hs-invar hs-bex*.

Unfortunately, Isabelle cannot generate instantiations of a generic algorithm automatically like Coq with its implicit arguments or C++ with its template mechanism. Instead, the ICF contains (automatically generated) explicit instantiations for each combination of generic algorithm and implementation, using a uniform naming scheme. It remains up to the user to select the appropriate instantiation, which is easy due to the uniform naming scheme. For example, to compute the union of a list-based set with a hash set, yielding a hash set, the user has to pick the function *lhh-union*, where the prefix *lhh* selects the right instantiation of the union-algorithm.

When implementing an algorithm using the ICF, the user must choose between writing a generic algorithm or fixing the data structures in advance. A generic algorithm needs to be parameterized over all used operations. If an algorithm uses only a few operations, the parameterization may be done explicitly, as in Ex. 3. If many different operations are involved, parameterization can be hidden syntactically in locale context, in order not to mess up the definitions with long parameter lists. However, due to restrictions in Isabelle/HOL's polymorphism, every collection with different element type requires its own operation parameters. Similarly, one has to specify one monomorphic instantiation for each iterator with different state. Alternatively, a record can collect all required ADT operations, but the monomorphism issue remains.

When a generic algorithm is not required, one can fix the used data structures beforehand, either by making alias definitions for the concrete operations and lemmas at the beginning of the theory, or by directly using the concrete

operations and lemmas throughout the theory. This avoids the above-mentioned problems with polymorphism. Thanks to the consistent naming conventions used in the ICF, switching to another implementation is as easy as replacing the prefixes of constant and lemma names (e.g., replacing *rs-* by *ts-* to switch from RBTs to Tries).

An alternative approach (similar to Peyton Jones' `XOps` route that automatically selects the data type implementation [17, Sec. 3]) is to hide ADT implementations completely from the ADT inside the logic. To that end, we introduce a new type for each ADT which is isomorphic to its base type. In the generated code, the new type becomes a data type with one constructor for each implementation. The abstract operations then pattern match on the ADT and dispatch to the correct implementation – emulating dynamic dispatch of the Java Collection Framework. Concrete implementations are selected by choice of the constructor, and, thanks to dynamic dispatch, manual instantiation or selection of generic algorithms is no longer necessary. However, this approach currently only works for ADT implementations that do not require invariants. Thus, it is only implemented for tries and array-based hashing. Yet, the Isabelle developers are working on the code generator such that it can handle such invariants.[7]

2.6 Implementations for ADTs

The ICF implementations for the ADTs use four basic data structures: lists, arrays, red-black trees, and tries. Inside Isabelle/HOL, arrays are isomorphic to lists, but for Haskell code, we use the `Data.Array.Diff.DiffArray` implementation from the Haskell library, which supports in-place updates while providing the immutable (functional) interface. To our knowledge, ML's and OCaml's standard libraries do not feature a similar implementation, so we fall back on a list-based implementation. Red-black trees are taken from the Isabelle/HOL standard library and extended with the iterator concept. A trie (prefix tree) is a search tree for strings where the key string identifies the path from the root to the node that stores the value. In contrast to RBTs, tries do not need data structure invariants. Our implementation improves upon and substantially extends the one in [15, Ch. 3.4.4]. The implementations for the ADTs use these data structures to implement maps, sets and sequences (cf. Fig. 1).

Maps. There are four implementations for finite maps: association lists (ListMap), red-black trees (RBTMap), hashing (HashMap, based on either RBTs or arrays), and tries (TrieMap).

Association lists have the data structure invariant that every key is unique in the list. While not being necessary, this allows for a simpler and more efficient implementation of iterators. Association lists work for all key types with executable equality test.

Red-black trees require that the key type is linearly ordered; the invariant ensures that it is a correct RBT, i.e., it has no two consecutive red nodes on a path, balanced height, the root is black, and the entries are ordered by their key.

[7] Personal communication with F. Haftmann.

If a key type does not have a canonical linear order, one can still use red-black trees by prefixing a hash operation *hashcode* that maps keys to integers. Then, the RBT maps an integer (the key's hashcode) to a bucket, which stores the key-value pairs for all keys with that hashcode in an association list. We use Isabelle's type classes to overload the *hashcode* function for different types and provide instantiations for all standard Isabelle type constructors except for functions (because they cannot be tested for equality). The invariant for hashing backed by RBTs is (i) the invariant for the RBT itself and (ii) that the keys in any bucket (i.e. association list) are distinct and have the bucket's hashcode.

The ICF also offers hashing backed by an array, which is currently only sensible with Haskell code (cf. above). This provides access in constant time, but requires to grow the array and rehash all data in the map when the load increases beyond a certain threshold. Our implementation triggers a rehash when the number of keys reaches 75% of the array size, a standard load factor threshold for open hashing.

By definition, keys for tries must be strings. For all other types, we use an encoding function *encode* into strings of integers, which must be injective. For natural numbers, e.g., we compute the 16-adic representation starting with the lowest digit, i.e. $1000 = 3 \cdot 16^2 + 14 \cdot 16 + 8$ is encoded as $[8, 14, 3]$. The type class for this encoding pairs every *encode* function with a left-inverse partial function *decode* that decodes the strings. Since *encode* is one-to-one, only countable types may be used as keys in a trie. Like for hashing, the ICF provides instantiations for all countable types predefined in Isabelle/HOL.

Sets. A map whose value type is the singleton type *unit* is isomorphic to a set, where mapping a key k to () means that the set contains k. The ICF provides generic algorithms (Map2Set) such that an implementation for the ADT map easily yields one for the ADT set. The RBT, hashing, and trie implementation for maps use this setup to define the set implementations.

Sequences. Sequences are typically used in two different styles: array-like and stack- or queue-like. In the array-like style, elements are accessed by index, and new elements are appended at the end. If implemented with a linked-list data type, these operations take linear time. The ICF therefore provides an array-based implementation ArrayList, which provides index and appending operations in amortized constant time (for Haskell), enlarging the array as necessary. However, prepending an element must shift all elements, which takes linear time.

In case a stack or queue is needed, the ICF contains an amortized constant-time queue implementation LFIFO that also provides constant-time stack operations. However, access by index is implemented by iteration, which takes linear time on average.

3 An Example Application

In this section, we demonstrate how to use the ICF in an example application inspired by the Sieve of Eratosthenes. Whereas the traditional Sieve produces a

$$
\begin{aligned}
\mathit{sieve}\ n &\equiv \mathit{sieve}_1\ n\ 2\ (\lambda_.\ \{\}) \\
\mathit{sieve}_1\ n\ i\ M &\equiv \mathsf{if}\ n < i\ \mathsf{then}\ M \\
&\qquad \mathsf{else}\ \mathit{sieve}_1\ n\ (i+1)\ (\mathsf{if}\ M\ i = \{\}\ \mathsf{then}\ \mathit{addp}\ n\ i\ i\ M\ \mathsf{else}\ M) \\
\mathit{addp}\ n\ p\ j\ M &\equiv \mathsf{if}\ j > n\ \mathsf{then}\ M\ \mathsf{else}\ \mathit{addp}\ n\ p\ (j+p)\ (M(j := \{p\} \cup M\ j))
\end{aligned}
$$

Fig. 2. The modified Sieve of Eratosthenes to compute sets of prime divisors.

list of prime numbers less than n, we produce a map from numbers less than n to their set of prime divisors, ignoring their multiplicity. A tail-recursive implementation is shown in Fig. 2, where $\mathit{sieve}\ n$ runs the function sieve_1 for the first n numbers and returns a function M of type $\mathit{nat} \rightarrow \mathit{nat\ set}$ such that for all i within 2 and n, $M\ i$ is the set of all prime divisors of i. $\mathit{sieve}_1\ n\ i\ M$ iterates from i up to n and, whenever it encounters a new prime number i ($M\ i = \{\}$), it adds i to the set $M\ j$ of all multiples j of i up to n via the function *addp*.

However, this implementation is not executable, because it contains the test $M\ i = \{\}$ (i.e. a function equality[8]). One solution is to change M's type to $\mathit{nat} \rightarrow \mathit{nat\ set\ option}$ and replace $\{\}$ with *None*, but this is very inefficient because the function M is built from pointwise updates, i.e. a function application $M\ i$ takes time linear in the number of updates. Since the number of prime divisors $\omega(n)$ of n is in $\mathcal{O}(\log(\log(n)))$ [8], there are $\mathcal{O}(n \cdot \log(\log(n)))$ updates and the above application executes $\mathcal{O}(n)$ times. Since sets and maps are coded as functions, insertion and update only add function closures and therefore require constant time. Hence, the overall run time is in $\mathcal{O}(n^2 \cdot \log(\log(n)))$.

We now reformulate the sieve as a generic algorithm by replacing the map M and the sets in the range of M by ICF ADTs (and implementations). Note that the Sieve could be implemented much more efficiently using an array monad and lists instead of sets. Yet, it still is a good non-trivial example to illustrate how to integrate the ICF in one's formalization, because the set ADT is nested in the map ADT. Following the data refinement approach from Sec. 2.2, the Sieve integrates with the ICF in three steps:

1. For code generation, we define new functions that operate on ADTs (Fig. 3).
2. We show that the new functions preserve the data structure invariants.
3. We show transfer equations between original and new functions.

The equations proved in step 3 are then used to transfer correctness theorems from the original functions to the new functions.

The sieve is defined as a generic algorithm, i.e. the definitions are (implicitly) parameterized over the used operations. The implicit parameterization is achieved by combining the locales for maps and sets, thereby prefixing the map and set operations with *m-* and *s-* resp., to avoid name clashes. Fig. 3 shows the new implementation. Note that the algorithm is structurally the same, but the operations on sets and maps have been replaced by the ADT operations, for

[8] In Isabelle/HOL, a set is represented by its characteristic function.

$$
\begin{aligned}
\mathit{sieve}'\ n &\equiv \mathit{sieve}_1'\ n\ 2\ \mathit{m\text{-}empty}\\
\mathit{sieve}_1'\ n\ i\ M &\equiv \mathsf{if}\ n < i\ \mathsf{then}\ M\\
&\quad \mathsf{else}\ \mathit{sieve}_1'\ n\ (i+1)\ (\mathsf{if}\ \mathit{m\text{-}lookup}\ i\ M = \mathit{None}\ \mathsf{then}\ \mathit{addp}'\ n\ i\ i\ M\ \mathsf{else}\ M)\\
\mathit{addp}'\ n\ p\ j\ M &\equiv \mathsf{if}\ n < j\ \mathsf{then}\ M\\
&\quad \mathsf{else}\ \mathit{addp}'\ n\ p\ (j+p)\ (\mathit{m\text{-}update}\ j\ (\mathit{s\text{-}ins}\ p\ (\mathit{opt\text{-}dest}\ (\mathit{m\text{-}lookup}\ j\ M)))\ M)\\
\mathit{opt\text{-}dest}\ \mathit{None} &\equiv \mathit{s\text{-}empty} \qquad \mathit{opt\text{-}dest}\ (\mathit{Some}\ A) \equiv A
\end{aligned}
$$

Fig. 3. Implementation of the modified Sieve with the ICF.

which no syntactic sugar is currently available. The new function *opt-dest* stems from replacing the function M by a map where *None* represents the empty set.

Since the ADTs are nested, their abstraction functions and invariant predicates are combined into new ones:

$$
\begin{aligned}
\alpha\ M &\equiv \mathit{s\text{-}\alpha}\ (\mathit{opt\text{-}dest}\ (\mathit{m\text{-}\alpha}\ M))\\
\mathit{invar}\ M &\equiv \mathit{m\text{-}invar}\ M \wedge (\forall n\ S.\ \mathit{m\text{-}\alpha}\ n = \mathit{Some}\ S \Longrightarrow \mathit{s\text{-}invar}\ S \wedge \mathit{s\text{-}\alpha}\ S \neq \{\})
\end{aligned}
$$

Note that we exclude empty sets being stored in M, because *None* already represents them. Next, we show that *addp'* and *sieve'* preserve the data structure invariant *invar*. This is straightforward because they only use the abstract operations that preserve the invariants by assumption. Finally, proving the following transfer equations is also straightforward under the assumption *invar* M:

$$\alpha\ (\mathit{addp}'\ n\ p\ M) = \mathit{addp}\ n\ p\ (\alpha\ M) \tag{1}$$

$$\alpha\ (\mathit{sieve}_1'\ n\ i\ M) = \mathit{sieve}_1\ n\ i\ (\alpha\ M) \tag{2}$$

$$\alpha\ (\mathit{sieve}'\ n) = \mathit{sieve}\ n \tag{3}$$

Since *invar* $(\mathit{sieve}'\ n)$ holds, *m-lookup* $j\ (\mathit{sieve}'\ n)$ returns the set of j's prime divisors for all $2 \leq j \leq n$.

To obtain an executable implementation with concrete data structures, e.g. RBTs, we simply interpret our locale. The ICF is set up such that all proof obligations are discharged automatically. For the run time complexity of the RBT implementation, the map operations dominate the set operations because the set size is limited by $\mathcal{O}(\log(\log(n)))$. Since M is updated $\mathcal{O}(n \cdot \log(\log(n)))$ times, the overall run time is in $\mathcal{O}(n \cdot \log(n) \cdot \log(\log(n)))$.

4 Evaluation

This section reports on some performance measurements. In Sec. 4.1, we compare the generated code using the ICF with the code generated from the Isabelle/HOL default set representation, and with the tree data structures from the standard libraries of Haskell and OCaml.[9] Then, we briefly describe a tree automata

[9] Unfortunately, the SML standard library contains no tree structure.

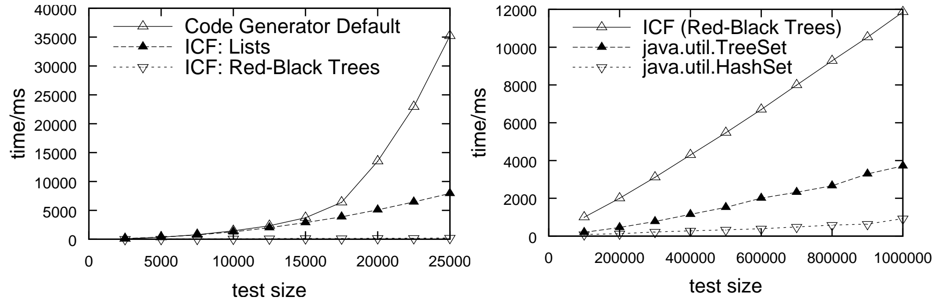

Fig. 4. Comparison of ICF data structures with code generator defaults and with Java

library that is based on the ICF and compare its performance to a well-known OCaml library and a Java library (Sec. 4.2).

All tests where done on a 2.66 GHz x86/64 dual-core machine with 4 GB of memory. We used Poly/ML 5.2, OCaml 3.09.3, GHC 6.10.4, and OpenJDK 1.6.0-b09. The run time values are averages over three test runs.

4.1 Basic Operations

For comparing the performance of basic set operations, we ran a simple program that starts with an empty set, then inserts n times a random number in the range $[0, 2n)$, then removes n times a random number in the range $[0, 2n)$, then tests n times a random number in the range $[0, 2n)$ for membership in the set, and finally iterates over each element in the set. As iteration is not executable in Isabelle/HOL's default code generator setup, we omitted the last phase when comparing with the default setup. This program exercises exactly the basic operations (*empty*, *member*, *insert*, *delete*, *iterate*) from which the other set operations may be (and actually are) derived by generic algorithms.

The left part of Figure 4 shows the runtimes of the code generated by Isabelle from a set-based formalization using the standard code generator setup and of the code that uses ICF data structures. The x axis shows the test size n and the y axis the required time in milliseconds. This test was done on the Poly/ML platform, which was faster than OCaml and GHC for this kind of tests. Clearly, the run time of the code generated from the default setup grows significantly faster (theoretically $\mathcal{O}(n^2)$) than the code using the ICF red-black trees (theoretically $O(n \log n)$). However, even the list-based ICF set implementation (also $O(n^2)$) is significantly faster than the default setup, because the latter's chain of pointwise function updates grows also with every delete operation.

The right part of Fig. 4 compares ICF's red-black trees to Java's `TreeSet` and `HashSet` classes. Java's `HashSet` class is backed by an array, and thus more efficient than the tree implementations, whose overhead per operation is much larger. Moreover, Java uses destructive updates whereas the ICF is purely functional. The ICF test was, again, run on the Poly/ML platform. Java's `TreeSet` is, on average, 3.7 times faster than the RBTs from the ICF – the ratio decreases

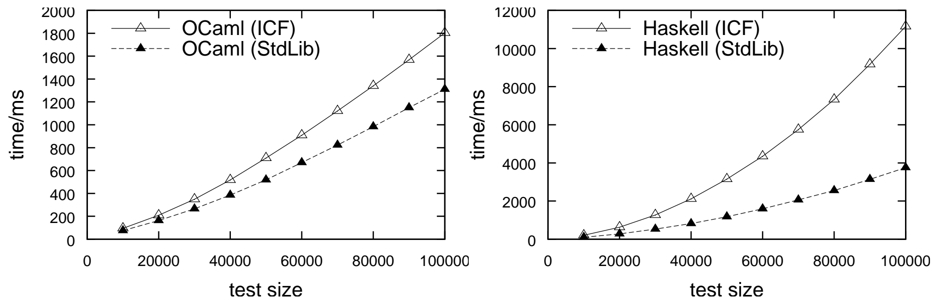

Fig. 5. Comparison of the ICF with the OCaml and Haskell standard library

from 5.0 for $n = 10^5$ to 3.2 for $n = 10^6$. Java's `HashSet` is even 15.2 times faster on average. These results show how much room there is for speed-up, when one is not bound to platform constraints.

In order to capture the potential for improvement on a functional language platform (to which the Isabelle/HOL code generator is restricted), we compare the red-black trees from the ICF with tree data structures from the Haskell and OCaml standard libraries. The results are shown in Fig. 5. For OCaml, the standard library is, on average, 34% faster than the ICF – the ratio increases from 25% for $n = 10^4$ to 38% for $n = 10^5$. For Haskell, the difference is even more significant. Here, the standard library is, on average, about 2.6 times faster – the value increases from 2.0 for $n = 10^4$ to 3.0 for $n = 10^5$. The significant super-linear increase for Haskell also results from lazily evaluated tail-recursive functions.[10]

Currently, the ICF tree data structure uses RBTs from the Isabelle standard library. Our results show that there is still room for improvement on the data structure's efficiency. On the other hand, the ICF data structures are formally verified, whereas those of the Haskell and OCaml standard libraries are not. Moreover, it would also be possible to configure the code generator to use the data structures from the standard library instead of the verified ones.

4.2 Case Study: An ICF-based Tree Automata Library

The first author has implemented a formally verified tree automata library [13]. It uses the ICF to derive efficient code and the data refinement approach to verify algorithms on an implementation independent (and thus simpler) level. We compared the generated code to Timbuk/Taml [6], a tree automata library for OCaml, and to LETHAL [14], a tree automata library for Java that has been developed as a students' project in our group.

The test consisted of intersecting seven pairs of randomly generated tree automata (with a few hundred rules and up to one hundred states each), and

[10] Up to a certain extent, strict evaluation in Haskell can be forced by the *seq*-operator. However, we did not include such platform specific optimizations into the ICF.

Language	ICF Haskell	ICF SML	ICF OCaml	ICF OCaml(i)	Taml OCaml(i)	LETHAL Java
complete	1.5s	6.1s	12.5s	121s	1923s	0.456s
reduced	73ms	407ms	522ms	4983ms	71636ms	120ms

Table 1. Tree Automata Library using the ICF compared to other libraries

then checking the results for emptiness. Table 1 shows the run time for various platforms. All ICF versions used RBT-based hashing. Most notably the ICF library running on Haskell is three orders of magnitude faster than Timbuk/Taml. However, this mainly results from comparing compiled Haskell with interpreted OCaml (marked as OCaml(i) in the table header). Another issue is that the ICF-based library uses a different algorithm for checking emptiness that performs better for automata with non-empty languages. However, even when comparing the ICF-based library and Timbuk/Taml both on interpreted OCaml, with a reduced test set (second row) where the tested automata's languages are all empty, the ICF based library is still about 14 times faster. We conjecture that Timbuk/Taml's use of plain lists for sets and maps – which is common practice in functional programming – explains that difference.

For the complete test set, the Java-based LETHAL library is about three times faster than the ICF-based library running on Haskell. For the reduced test set, the latter is even a bit faster than the Java implementation. These encouraging results demonstrate that it is possible to use the ICF to develop efficient verified algorithms that are competitive with existing unverified ones.

5 Conclusion

The Isabelle Collections Framework is a unified, easy-to-use framework for using verified data structures in Isabelle/HOL formalizations. Abstract data types for common Isabelle types provide the option to generate efficient code for a wider class of operations than the default setup. Data refinement allows one to transfer correctness results from existing formalizations to efficient implementations by means of transfer equations. The ICF implementations vastly outperform the standard code generator setup. The ICF proved its usability and efficiency in a verified tree automata library: The generated code outperforms the well-known (unverified) Timbuk/Taml library by a factor of 14, and is even competitive with the Java-based (also unverified) LETHAL library.

However, a lot remains to be done. The evaluation shows that the data structures are not yet optimally efficient. Some data structures, like heaps and priority queues, are still missing. Concerning usability, Isabelle's code generator currently poses the biggest limitation. When it will support invariants for data types, the ICF will integrate much more smoothly into existing formalizations.

Another approach to make functional code more efficient are state monads that support, e.g., arrays with destructive updates. The Imperative HOL frame-

work [3] adds support for monads to Isabelle/HOL. It remains future work to implement and verify monadic collection data structures.

Acknowledgement. We thank Nicholas Kidd and Alexander Wenner for proof-reading and many helpful comments.

References

1. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In Borwein, J. M. Farmer, W. M. (eds.) MKM 2006. LNAI, vol. 4108, pp. 31–43. Springer (2006)
2. Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: TPHOLs '09, pp. 147–163. Springer-Verlag, Berlin, Heidelberg (2009)
3. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: TPHOLs '08, pp. 134–149. Springer-Verlag, Berlin, Heidelberg (2008)
4. The Coq standard library. http://coq.inria.fr/stdlib/index.html
5. de Dios, J., Peña, R.: Formal certification of a resource-aware language implementation. In: TPHOLs '09, pp. 196–211. Springer-Verlag, Berlin, Heidelberg (2009)
6. Genet, T., Tong, V.V.T.: Timbuk 2.2. http://www.irisa.fr/celtique/genet/timbuk/
7. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Functional and Logic Programming (FLOPS 2010). LNCS. Springer (2010)
8. Hardy, G.H., Ramanujan, S.: The normal number of prime factors of a number. Quart. J. of Math. 48, 76–92 (1917)
9. Java: The collections framework. http://java.sun.com/javase/6/docs/technotes/guides/collections/
10. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common lisp. IEEE Transactions on Software Engineering 23, 203–213 (1997)
11. Kuncak, V.: Binary search trees. In Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. http://afp.sf.net/entries/BinarySearchTree.shtml (2004) Formal proof development.
12. Lammich, P.: Isabelle collection library. In Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. http://afp.sf.net/entries/collections.shtml (2009) Formal proof development.
13. Lammich, P.: Tree automata. In Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. http://afp.sf.net/entries/Tree-Automata.shtml (2009) Formal proof development.
14. LETHAL tree and hedge automata library. http://lethal.sourceforge.net/
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer (2002)
16. Nipkow, T., Pusch, C.: AVL trees. In Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. http://afp.sf.net/entries/AVL-Trees.shtml (2004) Formal proof development.
17. Peyton Jones, S.: Bulk types with class. In: FPW '96. (1996)
18. Stepanov, A., Lee, M.: The standard template library. Technical Report 95-11(R.1), HP Laboratories (1995)