Isabelle Collections Framework Userguide

By Peter Lammich

December 13, 2009

Contents

theory *Userguide*
imports
Collections
Efficient-Nat
begin

1 Introduction

The Isabelle Collections Framework defines interfaces of various collection types and provides some standard implementations and generic algorithms.

The relation between the data-structures of the collection framework and standard Isabelle datatypes (e.g. for sets and maps) is established by abstraction functions.

Currently, the Isabelle Collections Framework provides set and map implementations based on associative lists and red-black trees. Moreover, an implementation of a FIFO-queue based on lists is provided.

1.1 Getting Started

To get started with the Isabelle Collections Framework (assuming that you are already familiar with Isabelle/HOL and Isar), you should first read the introduction (this section), that provides many basic examples. Section 2 explains the concepts of the Isabelle Collections Framework in more detail. Section 3 provides information on extending the framework along with detailed examples, and Section 4 contains a discussion on the design of this framework.

1.2 Introductory Example

We introduce the Isabelle Collections Framework by a simple example.

Given a set of elements represented by a red-black tree, and a list, we want to filter out all elements that are not contained in the set. This can be done by Isabelle/HOL's *filter*-function[1]:

definition *rbt-restrict-list* :: *'a::linorder rs* ⇒ *'a list* ⇒ *'a list*
where *rbt-restrict-list s l* == [*x←l. rs-memb x s*]

The type *'a rs* is the type of sets backed by red-black trees. Note that the element type of sets backed by red-black trees must be of sort *linorder*. The function *rs-memb* tests membership on such sets.

Next, we show correctness of our function:

[1] Note that Isabelle/HOL uses the list comprehension syntax [*x←l. P x*] as syntactic sugar for filtering a list.

lemma *rbt-restrict-list-correct*:
assumes [*simp*]: *rs-invar s*
shows *rbt-restrict-list s l* = [*x*←*l*. *x*∈*rs-α s*]
by (*simp add*: *rbt-restrict-list-def rs.memb-correct*)

The lemma *rs.memb-correct*:

rs-invar s ⟹ *rs-memb x s* = (*x* ∈ *rs-α s*)

states correctness of the *rs-memb*-function. The function *rs-α* maps a red-black-tree to the set that it represents. Moreover, we have to explicitely keep track of the invariants of the used data-structure, in this case red-black trees. The predicate *rs-invar* denotes the invariant for red-black trees that represent sets.

Note that many correctness lemmas for standard RBT-set-operations are summarized by the lemma *rs-correct*:

⟦*rs-invar s*; *inj-on f* (*rs-α s* ∩ *dom f*)⟧
⟹ *rs-α* (*rs-inj-image-filter f s*) = {*b*. ∃ *a*∈*rs-α s*. *f a* = *Some b*}
⟦*rs-invar s*; *inj-on f* (*rs-α s* ∩ *dom f*)⟧
⟹ *rs-invar* (*rs-inj-image-filter f s*)
rs-invar s ⟹
rs-α (*rs-image-filter* (λ*x*. *if P x then Some* (*f x*) *else None*) *s*) =
f ‘ {*x* ∈ *rs-α s*. *P x*}
rs-invar s ⟹ *rs-α* (*rs-image-filter f s*) = {*b*. ∃ *a*∈*rs-α s*. *f a* = *Some b*}
rs-invar s ⟹ *rs-invar* (*rs-image-filter f s*)
⟦*rs-invar s*; *inj-on f* (*rs-α s*)⟧ ⟹ *rs-α* (*rs-inj-image f s*) = *f* ‘ *rs-α s*
⟦*rs-invar s*; *inj-on f* (*rs-α s*)⟧ ⟹ *rs-invar* (*rs-inj-image f s*)
⟦*rs-invar s1*; *rs-invar s2*; *rs-α s1* ∩ *rs-α s2* = {}⟧
⟹ *rs-α* (*rs-union-dj s1 s2*) = *rs-α s1* ∪ *rs-α s2*
⟦*rs-invar s1*; *rs-invar s2*; *rs-α s1* ∩ *rs-α s2* = {}⟧
⟹ *rs-invar* (*rs-union-dj s1 s2*)
⟦*rs-invar s1*; *rs-invar s2*⟧ ⟹ *rs-α* (*rs-union s1 s2*) = *rs-α s1* ∪ *rs-α s2*
⟦*rs-invar s1*; *rs-invar s2*⟧ ⟹ *rs-invar* (*rs-union s1 s2*)
⟦*rs-invar s1*; *rs-invar s2*⟧ ⟹ *rs-α* (*rs-inter s1 s2*) = *rs-α s1* ∩ *rs-α s2*
⟦*rs-invar s1*; *rs-invar s2*⟧ ⟹ *rs-invar* (*rs-inter s1 s2*)
rs-invar s ⟹ *rs-α* (*rs-image f s*) = *f* ‘ *rs-α s*
rs-invar s ⟹ *rs-invar* (*rs-image f s*)
⟦*rs-invar s*; *x* ∉ *rs-α s*⟧ ⟹ *rs-α* (*rs-ins-dj x s*) = *insert x* (*rs-α s*)
⟦*rs-invar s*; *x* ∉ *rs-α s*⟧ ⟹ *rs-invar* (*rs-ins-dj x s*)
rs-invar s ⟹ *rs-α* (*rs-delete x s*) = *rs-α s* − {*x*}
rs-invar s ⟹ *rs-invar* (*rs-delete x s*)
rs-invar S ⟹ *rs-ball S P* = (∀ *x*∈*rs-α S*. *P x*)
rs-invar s ⟹ *rs-α* (*rs-ins x s*) = *insert x* (*rs-α s*)
rs-invar s ⟹ *rs-invar* (*rs-ins x s*)
rs-invar s ⟹ *rs-memb x s* = (*x* ∈ *rs-α s*)
rs-invar s ⟹ *set* (*rs-to-list s*) = *rs-α s*
rs-invar s ⟹ *distinct* (*rs-to-list s*)
rs-α (*list-to-rs l*) = *set l*

rs-invar (*list-to-rs l*)
rs-invar s $\Longrightarrow$ *rs-isEmpty s* = (*rs-α s* = {})
rs-invar s $\Longrightarrow$ *rs-size s* = *card* (*rs-α s*)
rs-α rs-empty = {}
rs-invar rs-empty

All implementations provided by this library are compatible with the Isabelle/HOL code-generator. Now follow some examples of using the code-generator and the related value-command:

There are conversion functions from lists to sets and, vice-versa, from sets to lists:

value *list-to-rs* [*1*::*int*..*10*]
value *rs-to-list* (*list-to-rs* [*1*::*int* .. *10*])
value *rs-to-list* (*list-to-rs* [*1*::*int*,*5*,*6*,*7*,*3*,*4*,*9*,*8*,*2*,*7*,*6*])

Note that sets make no guarantee about ordering, hence the only thing we can prove about conversion from sets to lists is: *rs.to-list-correct*:

rs-invar s $\Longrightarrow$ *set* (*rs-to-list s*) = *rs-α s*
rs-invar s $\Longrightarrow$ *distinct* (*rs-to-list s*)

value *rbt-restrict-list* (*list-to-rs* [*1*::*nat*,*2*,*3*,*4*,*5*]) [*1*::*nat*,*9*,*2*,*3*,*4*,*5*,*6*,*5*,*4*,*3*,*6*,*7*,*8*,*9*]

definition *test n* = (*list-to-rs* [(*1*::*int*)..*n*])

export-code *test* **in** *SML* **module-name** *test*
ML ⟪
  open test;

  test 90000

⟫

1.3 Theories

To make available the whole collections framework to your formalization, import the theory *Collections*.
Other theories in the Isabelle Colelction Framework include:

SetSpec Specification of sets and set functions

SetGA Generic algorithms for sets

SetStdImpl Standard set implementations (list, rb-tree, hash)

MapSpec Specification of maps

MapGA Generic algorithms for maps

MapStdImpl Standard map implementations (list,rb-tree, hash)

Algos Various generic algorithms

SetIndex Generic algorithm for building indices of sets

Fifo Amortized fifo queue

DatRef Data refinement for the while combinator

1.4 Iterators

An important concept when using collections are iterators. An iterator is a kind of generalized fold-functional. Like the fold-functional, it applies a function to all elements of a set and modifies a state. There are no guarantees about the iteration order. But, unlike the fold functional, you can prove useful properties of iterations even if the function is not left-commutative. Proofs about iterations are done in invariant style, establishing an invariant over the iteration.
The iterator combinator for red-black tree sets is *rs-iterate*, and the proof-rule that is usually used is: *rs.iterate-rule-P*:

$$\begin{array}{l}[\![\mathit{rs\text{-}invar}\ S;\ I\ (\mathit{rs\text{-}}\alpha\ S)\ \sigma 0;\\ \bigwedge x\ it\ \sigma.\ [\![x \in it;\ it \subseteq \mathit{rs\text{-}}\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma);\\ \bigwedge \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma]\!]\\ \Longrightarrow P\ (\mathit{rs\text{-}iterate}\ f\ S\ \sigma 0)\end{array}$$

The invariant I is parameterized with the set of remaining elements that have not yet been iterated over and the current state. The invariant has to hold for all elements remaining and the initial state: $I\ (\mathit{rs}\text{-}\alpha\ S)\ \sigma 0$. Moreover, the invariant has to be preserved by an iteration step:

$$\bigwedge x\ it\ \sigma.\ [\![x \in it;\ it \subseteq \mathit{rs\text{-}}\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$$

And the proposition to be shown for the final state must be a consequence of the invarant for no elements remaining: $\bigwedge \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$.

thm *rs.iteratei-rule-P*[*no-vars*]

A generalization of iterators are *interruptible iterators* where iteration is only continues while some condition on the state holds. Reasoning over interruptible iterators is also done by invariants: *rs.iteratei-rule-P*:

$$\begin{array}{l}[\![\mathit{rs\text{-}invar}\ S;\ I\ (\mathit{rs\text{-}}\alpha\ S)\ \sigma 0;\\ \bigwedge x\ it\ \sigma.\ [\![c\ \sigma;\ x \in it;\ it \subseteq \mathit{rs\text{-}}\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma);\\ \bigwedge \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma;\ \bigwedge \sigma\ it.\ [\![it \subseteq \mathit{rs\text{-}}\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma]\!]\\ \Longrightarrow P\ (\mathit{rs\text{-}iteratei}\ c\ f\ S\ \sigma 0)\end{array}$$

Here, interruption of the iteration is handled by the premise

$\bigwedge\sigma\ it.\ [\![it \subseteq rs\text{-}\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma$

that shows the proposition from the invariant for any intermediate state of the iteration where the continuation condition does not hold (and thus the iteration is interrupted).

As an example of reasoning about results of iterators, we implement a function that converts a hashset to a list that contains precisely the elements of the set.

definition *hs-to-list′ s* == *hs-iterate* (*op #*) *s* []

The correctness proof works by establishing the invariant that the list contains all elements that have already been iterated over.

lemma *hs-to-list′-correct*:
  assumes *INV*: *hs-invar s*
  shows *set* (*hs-to-list′ s*) = *hs-α s*
  apply (*unfold hs-to-list′-def*)
  apply (*rule-tac*
    I=λ*it σ*. *set σ* = *hs-α s* − *it*
    in *hs.iterate-rule-P*[*OF INV*])

The resulting proof obligations are easily discharged using auto:

  apply *auto*
  done

As an example for an interruptible iterator, we define a bounded existential-quantification over the list elements. As soon as the first element is found that fulfills the predicate, the iteration is interrupted. The state of the iteration is simply a boolean, indicating the (current) result of the quantification:

definition *hs-bex s P* == *hs-iteratei* (λ*σ*. ¬ *σ*) (λ*x σ*. *P x*) *s False*

lemma *hs-bex-correct*:
  hs-invar s ⟹ *hs-bex s P* ⟷ (∃ *x*∈*hs-α s*. *P x*)
  apply (*unfold hs-bex-def*)

The invariant states that the current result matches the result of the quantification over the elements already iterated over:

  apply (*rule-tac*
    I=λ*it σ*. *σ* ⟷ (∃ *x*∈*hs-α s* − *it*. *P x*)
    in *hs.iteratei-rule-P*)

The resulting proof obligations are easily discharged by auto:

  apply *auto*
  done

2 Structure of the Framework

The concepts of the framework are roughly based on the object-oriented concepts of interfaces, implementations and generic algorithms.

The concepts used in the framework are the following:

Interfaces An interface describes some concept by providing an abstraction mapping α to a related Isabelle/HOL-concept. The definition is generic in the datatype used to implement the concept (i.e. the concrete data structure). An interface is specified by means of a locale that fixes the abstraction mapping and an invariant. For example, the set-interface contains an abstraction mapping to sets, and is specified by the locale *SetSpec.set*. An interface roughly matches the concept of a (collection) interface in Java, e.g. *java.util.Set*.

Functions A function specifies some functionality involving interfaces. A function is specified by means of a locale. For example, membership query for a set is specified by the locale *SetSpec.set-memb* and equality test between two sets is a function specified by *SetSpec.set-equal*. A function roughly matches a method declared in an interface, e.g. *java.util.Set#contains, java.util.Set#equals*.

Generic Algorithms A generic algorithm specifies, in a generic way, how to implement a function using other functions. For example, the equality test for sets may be implemented using a subset function. It is described by the constant *SetGA.subset-equal* and the corresponding lemma *SetGA.subset-equal-correct*. There is no direct match of generic algorithms in the Java Collections Framework. The most related concept are abstract collection interfaces, that provide some default algorithms, e.g. *java.util.AbstractSet*. The concept of *Algorithm* in the C++ Standard Template Library [1] matches the concept of Generic Algorithm quite well.

Implementation An implementation of an interface provides a data structure for that interface together with an abstraction mapping and an invariant. Moreover, it provides implementations for some (or all) functions of that interface. For example, red-black trees are an implementation of the set-interface, with the abstraction mapping *rs-*α and invariant *rs-invar*; and the constant *rs-ins* implements the insert-function, as stated by the lemma *rs-ins-impl*. An implementation matches a concrete collection interface in Java, e.g. *java.util.TreeSet*, and the methods implemented by such an interface, e.g. *java.util.TreeSet#add*.

Instantiation An instantiation of a generic algorithm provides actual implementations for the used functions. For example, the generic equality-

test algorithm can be instantiated to use red-black-trees for both arguments (resulting in the function *rr-equal* and the lemma *rr-equal-impl*). While some of the functions of an implementation need to be implemented specifically, many functions may be obtained by instantiating generic algorithms. In Java, instantiation of a generic algorithm is matched most closely by inheriting from an abstract collection interface. In the C++ Standard Template Library instantiation of generic algorithms is done implicitely when using them.

2.1 Naming Conventions

There are the following general naming conventions used inside the Isabelle Collections Framework: Each implementation has a two-letter and a one-letter abbreviation, that are used as prefixes for the related constants, lemmas and instantiations.

The two-letter abbreviations should be unique over all interfaces and instantiations, the one-letter abbreviations should be unique over all implementations of the same interface. Names that reference the implementation of only one interface are prefixed with that implementation's two-letter abbreviation (e.g. *hs-ins* for insertion into a HashSet (hs,h)), names that reference more than one implementation are prefixed with the one-letter abbreviations (e.g. *lhh-union* for set union between a ListSet(ls,l) and a HashSet(hs,h), yielding a HashSet)

Currently, there are the following abbreviations:

lm,l List Map

rm,r RB-Tree Map

hm,h Hash Map

ls,l List Set

rs,r RB-Tree Set

hs,h Hash Set

Each function *name* of an interface *interface* is declared in a locale *interface-name*. This locale provides a fact *name-correct*. For example, there is the locale *set-ins* providing the fact *set-ins.ins-correct*. An implementation instantiates the locales of all implemented functions, using its two-letter abbreviation as instantiation prefix. For example, the HashSet-implementation instantiates the locale *set-ins* with the prefix *hs*, yielding the lemma *hs.ins-correct*. Moreover, an implementation with two-letter abbreviation *aa* provides a

lemma *aa-correct* that summarizes the correctness facts for the basic operations. It should only contain those facts that are safe to be used with the simplifier. E.g., the correctness facts for basic operations on hash sets are available via the lemma *hs-correct*.

3 Extending the Framework

This section illustrates, by example, how to add new interfaces, functions, generic algorithms and implementations to the framework:

3.1 Interfaces

An interface provides a new concept, that is usually mapped to a related Isabelle/HOL-concept. An interface is defined by providing a locale that fixes an abstraction mapping and an invariant. For example, consider the definition of an interface for sets:

locale *set′* =
— Abstraction mapping to Isabelle/HOL sets
fixes α :: *′s* ⇒ *′a set*
— Invariant
fixes *invar* :: *′s* ⇒ *bool*

The invariant makes it possible for an implementation to restrict to certain subsets of the type's universal set. Theoretically, this could also be done by a typedef, however, this is not supported by the code generator (as of Isabelle2009).

3.2 Functions

A function describes some operation on instances of an interface. It is specified by providing a locale that includes the locale of the interface, fixes a parameter for the operation and makes a correctness assumption. For an interface *interface* and an operation *name*, the function's locale has the name *interface-name*, the fixed parameter has the name *name* and the correctness assumption has the name *name-correct*.

As an example, consider the specifications of the insert function for sets and the empty set:

locale *set′-ins* = *set′* +
— Give reasonable names to types:
constrains α :: *′s* ⇒ *′a set*
— Parameter for function:
fixes *ins* :: *′a* ⇒ *′s* ⇒ *′s*
— Correctness assumption. A correctness assumption usually consists of two parts:

- A description of the operation on the abstract level, assuming that the operands satisfy the invariants.
- The invariant preservation assumptions, i.e. assuming that the result satisfies its invariants if the operands do.

```
  assumes ins-correct:
    invar s ⟹ α (ins x s) = insert x (α s)
    invar s ⟹ invar (ins x s)

locale set'-empty = set' +
  constrains α :: 's ⇒ 'a set
  fixes empty :: 's
  assumes empty-correct:
    α empty = {}
    invar empty
```

In general, more than one interface or more than one instance of the same interface may be involved in a function. Consider, for example, the intersection of two sets. It involves three instances of set interfaces, two for the operands and one for the result:

```
locale set'-inter = set' α1 invar1 + set' α2 invar2 + set' α3 invar3
  for α1 :: 's1 ⇒ 'a set and invar1
  and α2 :: 's2 ⇒ 'a set and invar2
  and α3 :: 's3 ⇒ 'a set and invar3
  +
  fixes inter :: 's1 ⇒ 's2 ⇒ 's3
  assumes inter-correct:
    ⟦invar1 s1; invar2 s2⟧ ⟹ α3 (inter s1 s2) = α1 s1 ∩ α2 s2
    ⟦invar1 s1; invar2 s2⟧ ⟹ invar3 (inter s1 s2)
```

For use in further examples, we also specify a function that converts a list to a set

```
locale set'-list-to-set = set' +
  constrains α :: 's ⇒ 'a set
  fixes list-to-set :: 'a list ⇒ 's
  assumes list-to-set-correct:
    α (list-to-set l) = set l
    invar (list-to-set l)
```

3.3 Generic Algorithm

A generic algorithm describes how to implement a function using implementations of other functions. Thereby, it is parametric in the actual implementations of the functions.

A generic algorithm comes with the definition of a function and a correctness lemma. The function takes the required functions as arguments. The

convention for argument order is that the required functions come first, then the implemented function's arguments.

Consider, for example, the generic algorithm to convert a list to a set[2]. This function requires implementations of the *empty* and *ins* functions[3]:

```
fun list-to-set' :: 's ⇒ ('a ⇒ 's ⇒ 's)
  ⇒ 'a list ⇒ 's where
  list-to-set' empt ins' [] = empt |
  list-to-set' empt ins' (a#ls) = ins' a (list-to-set' empt ins' ls)

lemma list-to-set'-correct:
  fixes empty ins
  — Assumptions about the required function implementations:
  assumes set'-empty α invar empty
  assumes set'-ins α invar ins
  — Provided function:
  shows set'-list-to-set α invar (list-to-set' empty ins)
proof −
  interpret set'-empty α invar empty by fact
  interpret set'-ins α invar ins by fact

  {
    fix l
    have α (list-to-set' empty ins l) = set l
          ∧ invar (list-to-set' empty ins l)
      by (induct l)
         (simp-all add: empty-correct ins-correct)
  }
  thus ?thesis
    by unfold-locales auto
qed
```

Generic Algorithms with ad-hoc function specification The collection framework also contains a few generic algorithms that do not implement a function that is specified via a locale, but the function is specified ad-hoc within the correctness lemma. An example is the generic algorithm *Algos.map-to-nat* that computes an injective map from the elements of a given finite set to an initial segment of the natural numbers. There is no locale specifying such a function, but the function is implicitly specified by the correctness lemma *map-to-nat-correct*:

```
⟦set-iterate α1 invar1 iterate1; map-empty α2 invar2 empty2;
 map-update α2 invar2 update2; invar1 s⟧
⟹ dom (α2 (map-to-nat iterate1 empty2 update2 s)) = α1 s
```

[2]To keep the presentation simple, we use a non-tail-recursive version here

[3]Due to name-clashes with *Map.empty* and *RBT.ins* we have to use slightly different parameter names here

⟦*set-iterate α1 invar1 iterate1*; *map-empty α2 invar2 empty2*;
map-update α2 invar2 update2; *invar1 s*⟧
⟹ *inj-on* (*α2* (*map-to-nat iterate1 empty2 update2 s*)) (*α1 s*)
⟦*set-iterate α1 invar1 iterate1*; *map-empty α2 invar2 empty2*;
map-update α2 invar2 update2; *invar1 s*⟧
⟹ *inatseg* (*ran* (*α2* (*map-to-nat iterate1 empty2 update2 s*)))
⟦*set-iterate α1 invar1 iterate1*; *map-empty α2 invar2 empty2*;
map-update α2 invar2 update2; *invar1 s*⟧
⟹ *invar2* (*map-to-nat iterate1 empty2 update2 s*)

This kind of ad-hoc specification should only be used when it is unlikely that the same function may be implemented differently.

3.4 Implementation

An implementation of an interface defines an actual data structure, an invariant, and implementations of the functions. An implementation has a two-letter abbreviation that should be unique and a one-letter abbreviation that should be unique amongst all implementations of the same interface.

Consider, for example, a set implementation by distinct lists. It has the abbreviations (ls,l). To avoid name clashes with the existing list-set implementation in the framework, we use ticks (') here and there to disambiguate the names.

— The type of the data structure should be available as the two-letter abbreviation:
types *'a ls' = 'a list*
— The abstraction function:
definition *ls'-α == set*
— The invariant: In our case we constrain the lists to be distinct:
definition *ls'-invar == distinct*
— The locale of the interface is interpreted with the two-letter abbreviation as prefix:
interpretation *ls'*: *set' ls'-α ls'-invar* **.**

Next, we implement some functions. The implementation of a function *name* is prefixed by the two-letter prefix:

definition *ls'-empty* == []

Each function implementation has a corresponding lemma that shows the instantiation of the locale. It is named by the function's name suffixed with *-impl*:

lemma *ls'-empty-impl*: *set'-empty ls'-α ls'-invar ls'-empty*
by (*unfold-locales*)
(*auto simp add*: *ls'-empty-def ls'-invar-def ls'-α-def*)

The corresponding function's locale is interpreted with the function implementation and the interface's two-letter abbreviation as prefix:

interpretation *ls'*: *set'-empty ls'-α ls'-invar ls'-empty*
using *ls'-empty-impl* **.**

This generates the lemma *ls'.empty-correct*:

ls'-α ls'-empty = {}
ls'-invar ls'-empty

definition *ls'-ins x l* == *if x∈set l then l else x#l*

Correctness may optionally be established using separate lemmas. These should be suffixed with *_aux* to indicate that they should not be used by other proofs:

lemma *ls'-ins-correct-aux*:
ls'-invar l ⟹ *ls'-α* (*ls'-ins x l*) = *insert x* (*ls'-α l*)
ls'-invar l ⟹ *ls'-invar* (*ls'-ins x l*)
by (*auto simp add*: *ls'-ins-def ls'-invar-def ls'-α-def*)

lemma *ls'-ins-impl*: *set'-ins ls'-α ls'-invar ls'-ins*
by *unfold-locales*
(*simp-all add*: *ls'-ins-correct-aux*)

interpretation *ls'*: *set'-ins ls'-α ls'-invar ls'-ins*
using *ls'-ins-impl* **.**

3.5 Instantiations (Generic Algorithm)

The instantiation of a generic algorithm substitutes actual implementations for the required functions. A generic algorithm is instantiated by providing a definition that fixes the function parameters accordingly. Moreover, an *impl*-lemma and an interpretation of the implemented function's locale is provided. These can usually be constructed canonically from the generic algorithm's correctness lemma:

For example, consider conversion from lists to list-sets by instantiating the *list-to-set'*-algorithm:

definition *ls'-list-to-set* == *list-to-set' ls'-empty ls'-ins*
lemmas *ls'-list-to-set-impl* = *list-to-set'-correct*[*OF ls'-empty-impl ls'-ins-impl*, *folded ls'-list-to-set-def*]
interpretation *ls'*: *set'-list-to-set ls'-α ls'-invar ls'-list-to-set*
using *ls'-list-to-set-impl* **.**

Note that the actual framework slightly deviates from the naming convention here, the corresponding instantiation of *SetGA.gen-list-to-set* is called *list-to-ls*, the *impl*-lemma is called *list-to-ls-impl*.

Generating all possible instantiations of generic algorithms based on the available implementations can be done mechanically. Currently, we have not

implemented such an approach on the Isabelle ML-level. However, we used an ad-hoc ruby-script (*scripts/inst.rb*) to generate the thy-file *StdInst.thy* from the file *StdInst.in.thy*.

4 Design Issues

In this section, we motivate some of the design decisions of the Isabelle Collections Framework and report our experience with alternatives. Many of the design decisions are justified by restrictions of Isabelle/HOL and the code generator, so that there may be better options if those restrictions should vanish from future releases of Isabelle/HOL.

The main design goals of this development are:

1. Make available various implementations of collections under a unified interface.
2. It should be easy to extend the framework by new interfaces, functions, algorithms, and implementations.
3. Allow simple and concise reasoning over functions using collections.
4. Allow generic algorithms, that are independent of the actual data structure that is used.
5. Support generation of executable code.
6. Let the user precisely control what data structures are used in the implementation.

4.1 Data Refinement

In order to allow simple reasoning over collections, we use a data refinement approach. Each collection interface has an abstraction function that maps it on a related Isabelle/HOL concept (abstract level). The specification of functions are also relative to the abstraction. This allows most of the correctness reasoning to be done on the abstract level. On this level, the tool support is more elaborated and one is not yet fixed to a concrete implementation. In a next step, the abstract specification is refined to use an actual implementation (concrete level). The correctness properties proven on the abstract level usually transfer easily to the concrete level.

Moreover, the user has precise control how the refinement is done, i.e. what data structures are used. An alternative would be to do refinement completely automatic, as e.g. done in the code generator setup of the Theory *Executable-Set*. This has teh advantage that it induces less writing overhead. The disadvantage is that the user looses a great amount of control

over the refinement. For example, in *Executable-Set*, all sets have to be represented by lists, and there is no possibility to represent one set differently from another.

4.2 Grouping Functions

We have experimented with grouping functions of an interface together via a record. This has the advantage that parameterization of generic algorithms becomes simpler, as multiple function parameters are replaced by a single record parameter. However, on the other hand, a choice of sensible groups of functions is not obvious, and the use of the functions is a bit more writing overhead. As the generic algorithms in this framework only depend on a few function parameters, we have not grouped operations together. However, this may well make sense for more complex generic algorithms, that depend on many functions. A borderline example are the generic indexing algorithms defined in Theory *SetIndex*. The algorithms depend on quite a few functions. While we need to explicitely specify them as parameters, we try to simplify reasoning about them by defining an appropriate locale.

4.3 Locales for Generic Algorithms

Another tempting possibility to define a generic algorithm is to define a locale that includes the locales of all required functions, and do the definition of the generic algorithm inside that locale. This has the advantage that the function parameters are made implicit, thus improving readability. On the other hand, the code generator has problems with generating code from definitions inside a locale. Currently, one has to manually set up the code generator for such definitions. Moreover, when fixing function parameters in the declaration of the locale, their types will be infered independently of the definitions later done in the locale context. In order to get the correct types, one has to add explicit type constraints. These tend to become rather lengthy, especially for iterator states. The approach taken in this framework – passing the required functions as explicit parameters to a generic algorithm – usually needs less type constraints, as type inference usually does most of the job, in particular it infers the correct types of iterator states.

4.4 Explicit Invariants vs Typedef

The interfaces of this framework use explicit invariants. A more elegant alterative would be to use typedefs to make the invariants implicit, e.g. one could define the type of all well-formed RB-trees. However, this approach is not supported by the code generator (as of Isabelle2009), hence we had to chose the explicit invariant approach.

end

References

[1] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, November 1995.