Isabelle Collections Framework Userguide

By Peter Lammich

March 19, 2025

Contents

1 Isabelle Collections Framework Userguide

1.1 Introduction

This is the Userguide for the (old) Isabelle Collection Framework. It does not cover the Generic Collection Framework, nor the Automatic Refinement Framework.

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 types (e.g. for sets and maps) is established by abstraction functions.

Amongst others, the following interfaces and data-structures are provided by the Isabelle Collections Framework (For a complete list, see the overview section in the implementations chapter of the proof document):

- Set and map implementations based on (associative) lists, red-black trees, hashing and tries.
- An implementation of a FIFO-queue based on two stacks.
- Annotated lists implemented by finger trees.
- Priority queues implemented by binomial heaps, skew binomial heaps, and annotated lists (via finger trees).

The red-black trees are imported from the standard isabelle library. The binomial and skew binomial heaps are imported from the *Binomial-Heaps* entry of the archive of formal proofs. The finger trees are imported from the *Finger-Trees* entry of the archive of formal proofs.

1.1.1 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. More examples are in the examples/ subdirectory of the collection framework. Section 1.2 explains the concepts of the Isabelle Collections Framework in more detail. Section 1.3 provides information on extending the framework along with detailed examples, and Section 1.4 contains a discussion on the design of this framework. There is also a paper [2] on the design of the Isabelle Collections Framework available.

1.1.2 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:

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*:

$rs.invar\ s \Longrightarrow rs.memb\ x\ s = (x \in rs.\alpha\ s)$

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 premise *rs.invar ?s* represents the invariant assumption for the collection data structure. Red-black-trees are invariant-free, so this defaults to *True*. For uniformity reasons, these (unnecessary) invariant assumptions are present in all correctness lemmata.

Many of the correctness lemmas for standard RBT-set-operations are summarized by the lemma *rs.correct*:

$rs.\alpha\ (rs.empty\ ()) = \{\}$
$rs.invar\ (rs.empty\ ())$
$rs.\alpha\ (rs.sng\ x) = \{x\}$
$rs.invar\ (rs.sng\ x)$
$rs.invar\ s \Longrightarrow rs.memb\ x\ s = (x \in rs.\alpha\ s)$
$rs.invar\ s \Longrightarrow rs.\alpha\ (rs.ins\ x\ s) = insert\ x\ (rs.\alpha\ s)$
$rs.invar\ s \Longrightarrow rs.invar\ (rs.ins\ x\ s)$
$[\![rs.invar\ s;\ x \notin rs.\alpha\ s]\!] \Longrightarrow rs.\alpha\ (rs.ins\text{-}dj\ x\ s) = insert\ x\ (rs.\alpha\ s)$
$[\![rs.invar\ s;\ x \notin rs.\alpha\ s]\!] \Longrightarrow rs.invar\ (rs.ins\text{-}dj\ x\ s)$
$rs.invar\ s \Longrightarrow rs.\alpha\ (rs.delete\ x\ s) = rs.\alpha\ s - \{x\}$
$rs.invar\ s \Longrightarrow rs.invar\ (rs.delete\ x\ s)$
$rs.invar\ s \Longrightarrow rs.isEmpty\ s = (rs.\alpha\ s = \{\})$

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

$\mathit{rs.invar}\ s \Longrightarrow \mathit{rs.isSng}\ s = (\exists\, e.\ \mathit{rs}.\alpha\ s = \{e\})$
$\mathit{rs.invar}\ S \Longrightarrow \mathit{rs.ball}\ S\ P = (\forall\, x{\in}\mathit{rs}.\alpha\ S.\ P\ x)$
$\mathit{rs.invar}\ S \Longrightarrow \mathit{rs.bex}\ S\ P = (\exists\, x{\in}\mathit{rs}.\alpha\ S.\ P\ x)$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{rs.size}\ s = \mathit{card}\ (\mathit{rs}.\alpha\ s)$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{rs.size\text{-}abort}\ m\ s = \mathit{min}\ m\ (\mathit{card}\ (\mathit{rs}.\alpha\ s))$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs}.\alpha\ (\mathit{rs.union}\ s1\ s2) = \mathit{rs}.\alpha\ s1 \cup \mathit{rs}.\alpha\ s2$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs.invar}\ (\mathit{rs.union}\ s1\ s2)$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2;\ \mathit{rs}.\alpha\ s1 \cap \mathit{rs}.\alpha\ s2 = \{\}]\!]$
$\Longrightarrow \mathit{rs}.\alpha\ (\mathit{rs.union\text{-}dj}\ s1\ s2) = \mathit{rs}.\alpha\ s1 \cup \mathit{rs}.\alpha\ s2$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2;\ \mathit{rs}.\alpha\ s1 \cap \mathit{rs}.\alpha\ s2 = \{\}]\!]$
$\Longrightarrow \mathit{rs.invar}\ (\mathit{rs.union\text{-}dj}\ s1\ s2)$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs}.\alpha\ (\mathit{rs.diff}\ s1\ s2) = \mathit{rs}.\alpha\ s1 - \mathit{rs}.\alpha\ s2$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs.invar}\ (\mathit{rs.diff}\ s1\ s2)$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{rs}.\alpha\ (\mathit{rs.filter}\ P\ s) = \{e \in \mathit{rs}.\alpha\ s.\ P\ e\}$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{rs.invar}\ (\mathit{rs.filter}\ P\ s)$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs}.\alpha\ (\mathit{rs.inter}\ s1\ s2) = \mathit{rs}.\alpha\ s1 \cap \mathit{rs}.\alpha\ s2$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs.invar}\ (\mathit{rs.inter}\ s1\ s2)$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs.subset}\ s1\ s2 = (\mathit{rs}.\alpha\ s1 \subseteq \mathit{rs}.\alpha\ s2)$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs.equal}\ s1\ s2 = (\mathit{rs}.\alpha\ s1 = \mathit{rs}.\alpha\ s2)$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2]\!] \Longrightarrow \mathit{rs.disjoint}\ s1\ s2 = (\mathit{rs}.\alpha\ s1 \cap \mathit{rs}.\alpha\ s2 = \{\})$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2;\ \mathit{rs.disjoint\text{-}witness}\ s1\ s2 = \mathit{None}]\!]$
$\Longrightarrow \mathit{rs}.\alpha\ s1 \cap \mathit{rs}.\alpha\ s2 = \{\}$
$[\![\mathit{rs.invar}\ s1;\ \mathit{rs.invar}\ s2;\ \mathit{rs.disjoint\text{-}witness}\ s1\ s2 = \mathit{Some}\ a]\!]$
$\Longrightarrow a \in \mathit{rs}.\alpha\ s1 \cap \mathit{rs}.\alpha\ s2$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{set}\ (\mathit{rs.to\text{-}list}\ s) = \mathit{rs}.\alpha\ s$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{distinct}\ (\mathit{rs.to\text{-}list}\ s)$
$\mathit{rs}.\alpha\ (\mathit{rs.from\text{-}list}\ l) = \mathit{set}\ l$
$\mathit{rs.invar}\ (\mathit{rs.from\text{-}list}\ l)$

All implementations provided by this library are compatible with the Isabelle/HOL code-generator. Now follow some examples of using the code-generator. Note that the code generator can only generate code for plain constants without arguments, while the operations like *rs.memb* have arguments, that are only hidden by an abbreviation.

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

definition *conv-tests* ≡ (
  rs.from-list [*1*::*int* .. *10*],
  rs.to-list (*rs.from-list* [*1*::*int* .. *10*]),
  rs.to-sorted-list (*rs.from-list* [*1*::*int*,*5*,*6*,*7*,*3*,*4*,*9*,*8*,*2*,*7*,*6*]),
  rs.to-rev-list (*rs.from-list* [*1*::*int*,*5*,*6*,*7*,*3*,*4*,*9*,*8*,*2*,*7*,*6*])
)

ML-val ‹@{*code conv-tests*}›

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*:

$\mathit{rs.invar}\ s \Longrightarrow \mathit{set}\ (\mathit{rs.to\text{-}list}\ s) = \mathit{rs.}\alpha\ s$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{distinct}\ (\mathit{rs.to\text{-}list}\ s)$

Some sets, like red-black-trees, also support conversion to sorted lists, and we have: *rs.to-sorted-list-correct*:

$\mathit{rs.invar}\ s \Longrightarrow \mathit{set}\ (\mathit{rs.to\text{-}sorted\text{-}list}\ s) = \mathit{rs.}\alpha\ s$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{distinct}\ (\mathit{rs.to\text{-}sorted\text{-}list}\ s)$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{sorted}\ (\mathit{rs.to\text{-}sorted\text{-}list}\ s)$

and *rs.to-rev-list-correct*:

$\mathit{rs.invar}\ s \Longrightarrow \mathit{set}\ (\mathit{rs.to\text{-}rev\text{-}list}\ s) = \mathit{rs.}\alpha\ s$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{distinct}\ (\mathit{rs.to\text{-}rev\text{-}list}\ s)$
$\mathit{rs.invar}\ s \Longrightarrow \mathit{sorted}\ (\mathit{rev}\ (\mathit{rs.to\text{-}rev\text{-}list}\ s))$

definition *restrict-list-test* ≡ *rbt-restrict-list* (*rs.from-list* [*1*::*nat*,*2*,*3*,*4*,*5*]) [*1*::*nat*,*9*,*2*,*3*,*4*,*5*,*6*,*5*,*4*,*3*,*6*,*7*,*8*,*9*]

ML-val ‹@{*code restrict-list-test*}›

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

ML-val ‹@{*code big-test*} (@{*code int-of-integer*} *9000*)›

1.1.3 Theories

To make available the whole collections framework to your formalization, import the theory *Collections.Collections* which includes everything. Here is a small selection:

Collections.SetSpec Specification of sets and set functions

Collections.SetGA Generic algorithms for sets

Collections.SetStdImpl Standard set implementations (list, rb-tree, hashing, tries)

Collections.MapSpec Specification of maps

Collections.MapGA Generic algorithms for maps

Collections.MapStdImpl Standard map implementations (list,rb-tree, hashing, tries)

Collections.ListSpec Specification of lists

Collections.Fifo Amortized fifo queue

Collections.DatRef Data refinement for the while combinator

1.1.4 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.iteratei-rule-P*:

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

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: $I\ (\mathit{rs}.\alpha\ S)\ \sigma 0$. Moreover, the invariant has to be preserved by an iteration step:

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

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

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*:

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

Here, interruption of the iteration is handled by the premise

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

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.iteratei s* (λ-. *True*) (#) []

The correctness proof works by establishing the invariant that the list contains all elements that have already been iterated over. Again *hs.invar s* denotes the invariant for hashsets which defaults to *True*.

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=*λit 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 s* (*λσ*. ¬ *σ*) (*λx σ. P x*) *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=*λit 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

1.2 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*.

Operation Records In order to reference an interface with a standard set of operations, those operations are summarized in a record, and there is a locale that fixes this record, and makes available all operations. For example, the locale *SetSpec.StdSet* fixes a record of standard set operations and assumes their correctness. It also defines abbreviations to easily access the members of the record. Internally, all the standard operations, like *hs.memb*, are introduced by interpretation of such an operation locale.

Generic Algorithms A generic algorithm specifies, in a generic way, how to implement a function using other functions. Usually, a generic algorithm lives in a locale that imports the necessary operation locales. For example, the locale *cart-loc* defines a generic algorithm for the cartesian product between two sets.

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 [4] 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 $rs.\alpha$ and invariant *rs.invar*; and the constant *rs.ins* implements the insert-function, as can be verified by *set-ins* $rs.\alpha$ *rs.invar rs.ins*. 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 cartesian-product algorithm can be instantiated to use red-black-trees for both arguments, and output a list, as will be illustrated below in Section 1.2.1. 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 by the compiler.

1.2.1 Instantiation of Generic Algorithms

A generic algorithm is instantiated by interpreting its locale with the wanted implementations. For example, to obtain a cartesian product between two red-black trees, yielding a list, we can do the following:

setup *Locale-Code.open-block*
interpretation *rrl*: *cart-loc rs-ops rs-ops ls-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*
setup ‹*ICF-Tools.revert-abbrevs rrl*›

It is then available under the expected name:

term *rrl.cart*

Note the three lines of boilerplate code, that work around some technical problems of Isabelle/HOL: The *Locale-Code.open-block* and *Locale-Code.close-block* commands set up code generation for any locale that is interpreted in between them. They also have to be specified if an existing locale that already has interpretations is extended by new definitions.

The *ICF-Tools.revert-abbrevs rrl* reverts all abbreviations introduced by the locale, such that the displayed information becomes nicer.

1.2.2 Naming Conventions

The Isabelle Collections Framework follows these general naming conventions. Each implementation has a two-letter (or three-letter) and a one-letter (or two-letter) abbreviation, that are used as prefixes for the related constants, lemmas and instantiations.

The two-letter and three-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 (or two-letter) abbreviations (e.g. *rrl.cart* for the cartesian product between two RBT-Sets, yielding a list-set)

The most important abbreviations are:

lm,l List Map

lmi,li List Map with explicit invariant

rm,r RB-Tree Map

hm,h Hash Map

ahm,a Array-based hash map

tm,t Trie Map

ls,l List Set

lsi,li List Set with explicit invariant

rs,r RB-Tree Set

hs,h Hash Set

ahs,a Array-based hash map

ts,t Trie 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*.

1.3 Extending the Framework

The best way to add new features, i.e., interfaces, functions, generic algorithms, or implementations to the collection framework is to use one of the existing items as example.

1.4 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.

1.4.1 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 the 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.

For a more detailed discussion of the data refinement issue, we refer to the monadic refinement framework, that is available in the AFP (http://isa-afp.org/entries/Refine_Monadic.shtml)

1.4.2 Operation Records

In order to allow convenient access to the most frequently used functions of an interface, we have grouped them together in a record, and defined a locale that only fixes this record. This greatly reduces the boilerplate required to define a new (generic) algorithm, as only the operation locale (instead of every single function) has to be included in the locale for the generic algorithm.

Note however, that parameters of locales are monomorphic inside the locale. Thus, we have to import an own instance for the locale for every element type of a set, or key/value type of a map. For iterators, where this problem was most annoying, we have installed a workaround that allows polymorphic iterators even inside locales.

1.4.3 Locales for Generic Algorithms

A generic algorithm is defined within a locale, that includes the required functions (or operation locales). If many instances of the same interface are required, prefixes are used to distinguish between them. This makes the code for a generic algorithm quite consise and readable.

However, there are some technical issues that one has to consider:

- When fixing parameters in the declaration of the locale, their types will be inferred independently of the definitions later done in the locale context. In order to get the correct types, one has to add explicit type constraints.
- The code generator has problems with generating code from definitions inside a locale. Currently, the *Locale-Code*-package provides a rather convenient workaround for that issue: It requires the user to enclose interpretations and definitions of new constants inside already interpreted locales within two special commands, that set up the code generator appropriately.

1.4.4 Explicit Invariants vs Typedef

The interfaces of this framework use explicit invariants. This provides a more general specification which allows some operations to be implemented more efficiently, cf. *lsi.ins-dj* in *Collections.ListSetImpl-Invar*.

Most implementations, however, hide the invariant in a typedef and setup the code generator appropriately. In that case, the invariant is just λ-. *True*, and removed automatically by the simplifier and classical reasoner. However, it still shows up in some premises and conclusions due to uniformity reasons.

2 Old Monadic Refinement Framework Userguide

2.1 Introduction

This is the old userguide from Refine-Monadic. It contains the manual approach of using the mondaic refinement framework with the Isabelle Collection Framework. An alternative, more simple approach is provided by the Automatic Refinement Framework and the Generic Collection Framework.

The Isabelle/HOL refinement framework is a library that supports program and data refinement.

Programs are specified using a nondeterminism monad: An element of the monad type is either a set of results, or the special element *FAIL*, that indicates a failed assertion.

The bind-operation of the monad applies a function to all elements of the result-set, and joins all possible results.

On the monad type, an ordering $\leq$ is defined, that is lifted subset ordering, where *FAIL* is the greatest element. Intuitively, $S \leq S'$ means that program S refines program S', i.e., all results of S are also results of S', and S may only fail if S' also fails.

2.2 Guided Tour

In this section, we provide a small example program development in our framework. All steps of the development are heavily commented.

2.2.1 Defining Programs

A program is defined using the Haskell-like do-notation, that is provided by the Isabelle/HOL library. We start with a simple example, that iterates over a set of numbers, and computes the maximum value and the sum of all elements.

```
definition sum-max :: nat set ⇒ (nat×nat) nres where
  sum-max V ≡ do {
    (-,s,m) ← WHILE (λ(V,s,m). V≠{}) (λ(V,s,m). do {
      x←SPEC (λx. x∈V);
      let V=V−{x};
      let s=s+x;
      let m=max m x;
      RETURN (V,s,m)
    }) (V,0,0);
    RETURN (s,m)
  }
```

The type of the nondeterminism monad is *'a nres*, where *'a* is the type of the results. Note that this program has only one possible result, however, the order in which we iterate over the elements of the set is unspecified.

This program uses the following statements provided by our framework: While-loops, bindings, return, and specification. We briefly explain the statements here. A complete reference can be found in Section 2.5.1.

A while-loop has the form *WHILE* b f σ_0, where b is the continuation condition, f is the loop body, and σ_0 is the initial state. In our case, the state used for the loop is a triple $(V,\ s,\ m)$, where V is the set of remaining elements, s is the sum of the elements seen so far, and m is the maximum of the elements seen so far. The *WHILE* b f σ_0 construct describes a partially correct loop, i.e., it describes only those results that can be reached by finitely many iterations, and ignores infinite paths of the loop. In order to prove total correctness, the construct WHILE_T b f σ_0 is used. It fails if there exists an infinite execution of the loop.

A binding *do* $\{x \leftarrow (S_1{::}'a\ \mathit{nres});\ S_2\}$ nondeterministically chooses a result of S_1, binds it to variable x, and then continues with S_2. If S_1 is *FAIL*, the bind statement also fails.

The syntactic form *do* $\{$ *let* $x{=}V;\ (S{::}'a \Rightarrow\ 'b\ \mathit{nres})\}$ assigns the value V to variable x, and continues with S.

The return statement *RETURN* x specifies precisely the result x.

The specification statement *SPEC* Φ describes all results that satisfy the predicate Φ. This is the source of nondeterminism in programs, as there may be more than one such result. In our case, we describe any element of set V.

Note that these statement are shallowly embedded into Isabelle/HOL, i.e., they are ordinary Isabelle/HOL constants. The main advantage is, that any other construct and datatype from Isabelle/HOL may be used inside programs. In our case, we use Isabelle/HOL's predefined operations on sets and natural numbers. Another advantage is that extending the framework with new commands becomes fairly easy.

2.2.2 Proving Programs Correct

The next step in the program development is to prove the program correct w.r.t. a specification. In refinement notion, we have to prove that the program S refines a specification Φ if the precondition Ψ holds, i.e., $\Psi \Longrightarrow S \le \mathit{SPEC}\ \Phi$.

For our purposes, we prove that *sum-max* really computes the sum and the maximum.

As usual, we have to think of a loop invariant first. In our case, this is rather straightforward. The main complication is introduced by the partially defined *Max*-operator of the Isabelle/HOL standard library.

definition *sum-max-invar* $V_0 \equiv \lambda(V,s{::}\mathit{nat},m).$

$V \subseteq V_0$
$\wedge$ $s=\sum (V_0 - V)$
$\wedge$ $m=(\mathit{if}\ (V_0 - V)=\{\}\ \mathit{then}\ 0\ \mathit{else}\ \mathit{Max}\ (V_0 - V))$
$\wedge$ $\mathit{finite}\ (V_0 - V)$

We have extracted the most complex verification condition — that the invariant is preserved by the loop body — to an own lemma. For complex proofs, it is always a good idea to do that, as it makes the proof more readable.

lemma *sum-max-invar-step*:
assumes $x \in V$ *sum-max-invar* V_0 (V,s,m)
shows *sum-max-invar* V_0 $(V-\{x\}, s+x, \mathit{max}\ m\ x)$

In our case the proof is rather straightforward, it only requires the lemma *it-step-insert-iff*, that handles the $V_0 - (V - \{x\})$ terms that occur in the invariant.

using *assms* **unfolding** *sum-max-invar-def* **by** (*auto simp*: *it-step-insert-iff*)

The correctness is now proved by first invoking the verification condition generator, and then discharging the verification conditions by *auto*. Note that we have to apply the *sum-max-invar-step* lemma, *before* we unfold the definition of the invariant to discharge the remaining verification conditions.

theorem *sum-max-correct*:
assumes *PRE*: $V \neq \{\}$
shows *sum-max* $V \leq$ *SPEC* $(\lambda(s,m).\ s=\sum V \wedge m=\mathit{Max}\ V)$

The precondition $V \neq \{\}$ is necessary, as the *Max*-operator from Isabelle/HOL's standard library is not defined for empty sets.

using *PRE* **unfolding** *sum-max-def*
apply (*intro WHILE-rule*[**where** *I=sum-max-invar V*] *refine-vcg*) — Invoke vcg

Note that we have explicitely instantiated the rule for the while-loop with the invariant. If this is not done, the verification condition generator will stop at the WHILE-loop.

apply (*auto intro*: *sum-max-invar-step*) — Discharge step
unfolding *sum-max-invar-def* — Unfold invariant definition
apply (*auto*) — Discharge remaining goals
done

In this proof, we specified the invariant explicitely. Alternatively, we may annotate the invariant at the while loop, using the syntax $\mathit{WHILE}^I\ b\ f\ \sigma_0$. Then, the verification condition generator will use the annotated invariant automatically.

Total Correctness Now, we reformulate our program to use a total correct while loop, and annotate the invariant at the loop. The invariant is strengthened by stating that the set of elements is finite.

```
definition $\textit{sum-max'-invar}\ V_0\ \sigma \equiv$
  $\textit{sum-max-invar}\ V_0\ \sigma$
  $\wedge\ (\textit{let}\ (V,\text{-},\text{-})=\sigma\ \textit{in finite}\ (V_0 - V))$

definition $\textit{sum-max'} :: \textit{nat set} \Rightarrow (\textit{nat}\times\textit{nat})\ \textit{nres}$ where
  $\textit{sum-max'}\ V \equiv \textit{do}\ \{$
    $(\text{-},s,m) \leftarrow \textit{WHILE}_T{}^{\textit{sum-max'-invar}\ V}\ (\lambda(V,s,m).\ V\neq\{\})\ (\lambda(V,s,m).\ \textit{do}\ \{$
      $x\leftarrow \textit{SPEC}\ (\lambda x.\ x\in V);$
      $\textit{let}\ V=V-\{x\};$
      $\textit{let}\ s=s+x;$
      $\textit{let}\ m=\textit{max}\ m\ x;$
      $\textit{RETURN}\ (V,s,m)$
    $\})\ (V,0,0);$
    $\textit{RETURN}\ (s,m)$
  $\}$


theorem $\textit{sum-max'-correct}$:
  assumes $\textit{NE}$: $V\neq\{\}$ and $\textit{FIN}$: $\textit{finite}\ V$
  shows $\textit{sum-max'}\ V \leq \textit{SPEC}\ (\lambda(s,m).\ s=\sum V \wedge m=\textit{Max}\ V)$
  using $\textit{NE FIN}$ unfolding $\textit{sum-max'-def}$
  apply ($\textit{intro refine-vcg}$) — Invoke vcg
```

This time, the verification condition generator uses the annotated invariant. Moreover, it leaves us with a variant. We have to specify a well-founded relation, and show that the loop body respects this relation. In our case, the set V decreases in each step, and is initially finite. We use the relation *finite-psubset* and the *inv-image* combinator from the Isabelle/HOL standard library.

```
  apply ($\textit{subgoal-tac wf}\ (\textit{inv-image finite-psubset fst})$,
    $\textit{assumption}$) — Instantiate variant
  apply $\textit{simp}$ — Show variant well-founded

  unfolding $\textit{sum-max'-invar-def}$ — Unfold definition of invariant
  apply ($\textit{auto intro}$: $\textit{sum-max-invar-step}$) — Discharge step

  unfolding $\textit{sum-max-invar-def}$ — Unfold definition of invariant completely
  apply ($\textit{auto intro}$: $\textit{finite-subset}$) — Discharge remaining goals
  done
```

2.2.3 Refinement

The next step in the program development is to refine the initial program towards an executable program. This usually involves both, program refinement and data refinement. Program refinement means changing the structure of the program. Usually, some specification statements are replaced by more concrete implementations. Data refinement means changing the used data types towards implementable data types.

In our example, we implement the set V with a distinct list, and replace the specification statement $SPEC\ (\lambda x.\ x \in V)$ by the head operation on distinct lists. For the lists, we use the list-set data structure provided by the Isabelle Collection Framework [1, 3].

For this example, we write the refined program ourselves. An automation of this task can be achieved with the automatic refinement tool, which is available as a prototype in Refine-Autoref. Usage examples are in ex/Automatic-Refinement.

```
definition sum-max-impl :: nat ls ⇒ (nat×nat) nres where
  sum-max-impl V ≡ do {
    (-,s,m) ← WHILE (λ(V,s,m). ¬ls.isEmpty V) (λ(V,s,m). do {
      x←RETURN (the (ls.sel V (λx. True)));
      let V=ls.delete x V;
      let s=s+x;
      let m=max m x;
      RETURN (V,s,m)
    }) (V,0,0);
    RETURN (s,m)
  }
```

Note that we replaced the operations on sets by the respective operations on lists (with the naming scheme *ls.xxx*). The specification statement was replaced by *the* (*ls.sel* V ($\lambda x.$ *True*)), i.e., selection of an element that satisfies the predicate $\lambda x.$ *True*. As *ls.sel* returns an option datatype, we extract the value with *the*. Moreover, we omitted the loop invariant, as we don't need it any more.

Next, we have to show that our concrete pogram actually refines the abstract one.

```
theorem sum-max-impl-refine:
  assumes (V,V')∈build-rel ls.α ls.invar
  shows sum-max-impl V ≤ ⇓Id (sum-max V')
```

Let R be a *refinement relation*[2], that relates concrete and abstract values.

Then, the function $\Downarrow R$ maps a result-set over abstract values to the greatest result-set over concrete values that is compatible w.r.t. R. The value *FAIL* is mapped to itself.

Thus, the proposition $S \leq \Downarrow R\ S'$ means, that S refines S' w.r.t. R, i.e., every value in the result of S can be abstracted to a value in the result of S'.

Usually, the refinement relation consists of an invariant I and an abstraction function α. In this case, we may use the *br* I α-function to define the refinement relation.

In our example, we assume that the input is in the refinement relation specified by list-sets, and show that the output is in the identity relation. We use the identity here, as we do not change the datatypes of the output.

[2] Also called coupling invariant.

The proof is done automatically by the refinement verification condition generator. Note that the theory *Collection-Bindings* sets up all the necessary lemmas to discharge refinement conditions for the collection framework.

```
  using assms unfolding sum-max-impl-def sum-max-def
  apply (refine-rcg) — Decompose combinators, generate data refinement goals

  apply (refine-dref-type) — Type-based heuristics to instantiate data refinement
     goals
  apply (auto simp add:
    ls.correct refine-hsimp refine-rel-defs) — Discharge proof obligations
  done
```

Refinement is transitive, so it is easy to show that the concrete program meets the specification.

```
theorem sum-max-impl-correct:
  assumes (V, V')∈build-rel ls.α ls.invar and V'≠{}
  shows sum-max-impl V ≤ SPEC (λ(s,m). s=∑ V' ∧ m=Max V')
proof –
  note sum-max-impl-refine
  also note sum-max-correct
  finally show ?thesis using assms .
qed
```

Just for completeness, we also refine the total correct program in the same way.

```
definition sum-max'-impl :: nat ls ⇒ (nat×nat) nres where
  sum-max'-impl V ≡ do {
    (-,s,m) ← WHILE_T (λ(V,s,m). ¬ls.isEmpty V) (λ(V,s,m). do {
      x←RETURN (the (ls.sel V (λx. True)));
      let V=ls.delete x V;
      let s=s+x;
      let m=max m x;
      RETURN (V,s,m)
    }) (V,0,0);
    RETURN (s,m)
  }

theorem sum-max'-impl-refine:
  (V, V')∈build-rel ls.α ls.invar ⟹ sum-max'-impl V ≤ ⇓Id (sum-max' V')
  unfolding sum-max'-impl-def sum-max'-def
  apply refine-rcg
  apply refine-dref-type
  apply (auto simp: refine-hsimp ls.correct refine-rel-defs)
  done

theorem sum-max'-impl-correct:
  assumes (V, V')∈build-rel ls.α ls.invar and V'≠{}
  shows sum-max'-impl V ≤ SPEC (λ(s,m). s=∑ V' ∧ m=Max V')
```

using *ref-two-step*[*OF sum-max′-impl-refine sum-max′-correct*] *assms*

Note that we do not need the finiteness precondition, as list-sets are always finite. However, in order to exploit this, we have to unfold the *build-rel* construct, that relates the list-set on the concrete side to the set on the abstract side.

apply (*auto simp*: *build-rel-def*)
done

2.2.4 Code Generation

In order to generate code from the above definitions, we convert the function defined in our monad to an ordinary, deterministic function, for that the Isabelle/HOL code generator can generate code.

For partial correct algorithms, we can generate code inside a deterministic result monad. The domain of this monad is a flat complete lattice, where top means a failed assertion and bottom means nontermination. (Note that executing a function in this monad will never return bottom, but just diverge). The construct *nres-of x* embeds the deterministic into the nondeterministic monad.

Thus, we have to construct a function *?sum-max-code* such that:

schematic-goal *sum-max-code-aux*: *nres-of ?sum-max-code* $\leq$ *sum-max-impl V*

This is done automatically by the transfer procedure of our framework.

unfolding *sum-max-impl-def*
apply (*refine-transfer*)
done

In order to define the function from the above lemma, we can use the command *concrete-definition*, that is provided by our framework:

concrete-definition *sum-max-code* **for** *V* **uses** *sum-max-code-aux*

This defines a new constant *sum-max-code*:

thm *sum-max-code-def*

And proves the appropriate refinement lemma:

thm *sum-max-code.refine*

Note that the *concrete-definition* command is sensitive to patterns of the form *RETURN* - and *nres-of*, in which case the defined constant will not contain the *RETURN* or *nres-of*. In any other case, the defined constant will just be the left hand side of the refinement statement.

Finally, we can prove a correctness statement that is independent from our refinement framework:

theorem *sum-max-code-correct*:

assumes *ls*.α *V* $\neq$ {}
shows *sum-max-code V* = *dRETURN* (*s*,*m*) $\Longrightarrow$ *s*=$\sum$ (*ls*.α *V*) $\wedge$ *m*=*Max* (*ls*.α *V*)
and *sum-max-code V* $\neq$ *dFAIL*

The proof is done by transitivity, and unfolding some definitions:

using *nres-correctD*[*OF order-trans*[*OF sum-max-code.refine sum-max-impl-correct*, *of V ls*.α *V*]] *assms*
by (*auto simp*: *refine-rel-defs*)

For total correctness, the approach is the same. The only difference is, that we use *RETURN* instead of *nres-of*:

schematic-goal *sum-max′-code-aux*:
RETURN ?sum-max′-code $\leq$ *sum-max′-impl V*
unfolding *sum-max′-impl-def*
apply (*refine-transfer*)
done

concrete-definition *sum-max′-code* **for** *V* **uses** *sum-max′-code-aux*

theorem *sum-max′-code-correct*:
⟦*ls*.α *V* $\neq$ {}⟧ $\Longrightarrow$ *sum-max′-code V* = ($\sum$ (*ls*.α *V*), *Max* (*ls*.α *V*))
using *order-trans*[*OF sum-max′-code.refine sum-max′-impl-correct*, *of V ls*.α *V*]
by (*auto simp*: *refine-rel-defs*)

If we use recursion combinators, a plain function can only be generated, if the recursion combinators can be defined. Alternatively, for total correct programs, we may generate a (plain) function that internally uses the deterministic monad, and then extracts the result.

schematic-goal *sum-max″-code-aux*:
RETURN ?sum-max″-code $\leq$ *sum-max′-impl V*
unfolding *sum-max′-impl-def*
apply (*refine-transfer the-resI*) — Using *the-resI* for internal monad and result extraction
done

concrete-definition *sum-max″-code* **for** *V* **uses** *sum-max″-code-aux*

theorem *sum-max″-code-correct*:
⟦*ls*.α *V* $\neq$ {}⟧ $\Longrightarrow$ *sum-max″-code V* = ($\sum$ (*ls*.α *V*), *Max* (*ls*.α *V*))
using *order-trans*[*OF sum-max″-code.refine sum-max′-impl-correct*, *of V ls*.α *V*]
by (*auto simp*: *refine-rel-defs*)

Now, we can generate verified code with the Isabelle/HOL code generator:

export-code *sum-max-code sum-max′-code sum-max″-code* **checking** *SML*
export-code *sum-max-code sum-max′-code sum-max″-code* **checking** *OCaml?*

export-code *sum-max-code sum-max′-code sum-max″-code* **checking** *Haskell?*
export-code *sum-max-code sum-max′-code sum-max″-code* **checking** *Scala*

2.2.5 Foreach-Loops

In the *sum-max* example above, we used a while-loop to iterate over the elements of a set. As this pattern is used commonly, there is an abbreviation for it in the refinement framework. The construct *FOREACH S f* σ_0 iterates *f*::′*x*⇒′*s*⇒′*s* for each element in *S*::′*x set*, starting with state σ_0::′*s*.

With foreach-loops, we could have written our example as follows:

definition *sum-max-it* :: *nat set* ⇒ (*nat*×*nat*) *nres* **where**
  sum-max-it V ≡ *FOREACH V* (λ*x* (*s*,*m*). *RETURN* (*s*+*x*,*max m x*)) (*0*,*0*)

theorem *sum-max-it-correct*:
  assumes *PRE*: *V*≠{} **and** *FIN*: *finite V*
  shows *sum-max-it V* ≤ *SPEC* (λ(*s*,*m*). *s*=∑ *V* ∧ *m*=*Max V*)
  using *PRE* **unfolding** *sum-max-it-def*
  apply (*intro FOREACH-rule*[**where** *I*=λ*it* σ. *sum-max-invar V* (*it*,σ)] *refine-vcg*)
  apply (*rule FIN*) — Discharge finiteness of iterated set
  apply (*auto intro*: *sum-max-invar-step*) — Discharge step
  unfolding *sum-max-invar-def* — Unfold invariant definition
  apply (*auto*) — Discharge remaining goals
  done

definition *sum-max-it-impl* :: *nat ls* ⇒ (*nat*×*nat*) *nres* **where**
  sum-max-it-impl V ≡ *FOREACH* (*ls*.α *V*) (λ*x* (*s*,*m*). *RETURN* (*s*+*x*,*max m x*)) (*0*,*0*)

Note: The nondeterminism for iterators is currently resolved at transfer phase, where they are replaced by iterators from the ICF.

lemma *sum-max-it-impl-refine*:
  notes [*refine*] = *inj-on-id*
  assumes (*V*, *V*′)∈*build-rel ls*.α *ls.invar*
  shows *sum-max-it-impl V* ≤ ⇓*Id* (*sum-max-it V*′)
  unfolding *sum-max-it-impl-def sum-max-it-def*

Note that we specified *inj-on-id* as additional introduction rule. This is due to the very general iterator refinement rule, that may also change the set over that is iterated.

  using *assms*
  apply *refine-rcg* — This time, we don't need the *refine-dref-type* heuristics, as no schematic refinement relations are generated.
  apply (*auto simp*: *refine-hsimp refine-rel-defs*)
  done

schematic-goal *sum-max-it-code-aux*:
  RETURN ?sum-max-it-code ≤ *sum-max-it-impl V*
  unfolding *sum-max-it-impl-def*

apply (*refine-transfer*)
done

Note that the transfer method has replaced the iterator by an iterator from the Isabelle Collection Framework.

thm *sum-max-it-code-aux*
concrete-definition *sum-max-it-code* **for** *V* **uses** *sum-max-it-code-aux*

theorem *sum-max-it-code-correct*:
  assumes *ls*.α *V* $\neq$ {}
  shows *sum-max-it-code V* = ($\sum$ (*ls*.α *V*), *Max* (*ls*.α *V*))
proof −
  note *sum-max-it-code.refine*[*of V*]
  also note *sum-max-it-impl-refine*[*of V ls*.α *V*]
  also note *sum-max-it-correct*
  finally show *?thesis* **using** *assms* **by** (*auto simp*: *refine-rel-defs*)
qed

export-code *sum-max-it-code* **checking** *SML*
export-code *sum-max-it-code* **checking** *OCaml?*
export-code *sum-max-it-code* **checking** *Haskell?*
export-code *sum-max-it-code* **checking** *Scala*

definition *sum-max-it-list* ≡ *sum-max-it-code o ls.from-list*
ML-val ‹
  @{*code sum-max-it-list*} (*map* @{*code nat-of-integer*} [*1*,*2*,*3*,*4*,*5*])
›

2.3 Pointwise Reasoning

In this section, we describe how to use pointwise reasoning to prove refinement statements and other relations between element of the nondeterminism monad.

Pointwise reasoning is often a powerful tool to show refinement between structurally different program fragments.

The refinement framework defines the predicates *nofail* and *inres*. *nofail S* states that *S* does not fail, and *inres S x* states that one possible result of *S* is *x* (Note that this includes the case that *S* fails).

Equality and refinement can be stated using *nofail* and *inres*:

$$(?S = ?S') = (nofail\ ?S = nofail\ ?S' \wedge (\forall x.\ inres\ ?S\ x = inres\ ?S'\ x))$$

$$(?S \leq ?S') = (nofail\ ?S' \longrightarrow nofail\ ?S \wedge (\forall x.\ inres\ ?S\ x \longrightarrow inres\ ?S'\ x))$$

Useful corollaries of this lemma are *pw-leI*, *pw-eqI*, and *pwD*.

Once a refinement has been expressed via nofail/inres, the simplifier can be used to propagate the nofail and inres predicates inwards over the structure of the program. The relevant lemmas are contained in the named theorem collection *refine-pw-simps*.

As an example, we show refinement of two structurally different programs here, both returning some value in a certain range:

```
lemma do { ASSERT (fst p > 2); SPEC (λx. x≤(2::nat)∗(fst p + snd p)) }
  ≤ do { let (x,y)=p; z←SPEC (λz. z≤x+y);
         a←SPEC (λa. a≤x+y); ASSERT (x>2); RETURN (a+z)}
  apply (rule pw-leI)
  apply (auto simp add: refine-pw-simps split: prod.split)

  apply (rename-tac a b x)
  apply (case-tac x≤a+b)
  apply (rule-tac x=0 in exI)
  apply simp
  apply (rule-tac x=a+b in exI)
  apply (simp)
  apply (rule-tac x=x−(a+b) in exI)
  apply simp
  done
```

2.4 Arbitrary Recursion (TBD)

While-loops are suited to express tail-recursion. In order to express arbitrary recursion, the refinement framework provides the nrec-mode for the *partial-function* command, as well as the fixed point combinators *REC* (partial correctness) and REC_T (total correctness).

Examples for *partial-function* can be found in *ex/Refine-Fold*. Examples for the recursion combinators can be found in *ex/Recursion* and *ex/Nested-DFS*.

2.5 Reference

2.5.1 Statements

SUCCEED The empty set of results. Least element of the refinement ordering.

FAIL Result that indicates a failing assertion. Greatest element of the refinement ordering.

RES X All results from set *X*.

RETURN x Return single result *x*. Defined in terms of *RES*: *RETURN x* = *RES* {*x*}.

EMBED r Embed partial-correctness option type, i.e., succeed if *r=None*, otherwise return value of *r*.

SPEC Φ Specification. All results that satisfy predicate Φ. Defined in terms of *RES*: *SPEC* Φ = *SPEC* Φ

bind M f Binding. Nondeterministically choose a result from M and apply f to it. Note that usually the *do*-notation is used, i.e., *do* $\{x{\leftarrow}M;\ f\,x\}$ or *do* $\{M;f\}$ if the result of M is not important. If M fails, *bind M f* also fails.

ASSERT Φ Assertion. Fails if Φ does not hold, otherwise returns (). Note that the default usage with the do-notation is: *do* $\{$*ASSERT* $\Phi;\ f\}$.

ASSUME Φ Assumption. Succeeds if Φ does not hold, otherwise returns (). Note that the default usage with the do-notation is: *do* $\{$*ASSUME* $\Phi;\ f\}$.

REC body Recursion for partial correctness. May be used to express arbitrary recursion. Returns *SUCCEED* on nontermination.

REC_T *body* Recursion for total correctness. Returns *FAIL* on nontermination.

WHILE $b\ f\ \sigma_0$ Partial correct while-loop. Start with state σ_0, and repeatedly apply f as long as b holds for the current state. Non-terminating paths are ignored, i.e., they do not contribute a result.

$WHILE_T$ $b\ f\ \sigma_0$ Total correct while-loop. If there is a non-terminating path, the result is *FAIL*.

$WHILE^I$ $b\ f\ \sigma_0$, ${WHILE_T}^I$ $b\ f\ \sigma_0$ While-loop with annotated invariant. It is asserted that the invariant holds.

FOREACH $S\ f\ \sigma_0$ Foreach loop. Start with state σ_0, and transform the state with $f\ x$ for each element $x{\in}S$. Asserts that S is finite.

$FOREACH^I$ $S\ f\ \sigma_0$ Foreach-loop with annotated invariant.

Alternative syntax: $FOREACH^I$ $S\ f\ \sigma_0$.

The invariant is a predicate of type $I::'a\ set \Rightarrow\ 'b \Rightarrow bool$, where $I\ it\ \sigma$ means, that the invariant holds for the remaining set of elements it and current state σ.

$FOREACH_C$ $S\ c\ f\ \sigma_0$ Foreach-loop with explicit continuation condition.

Alternative syntax: $FOREACH_C$ $S\ c\ f\ \sigma_0$.

If $c::'\sigma{\Rightarrow}bool$ becomes false for the current state, the iteration immediately terminates.

$FOREACH_C{}^I\ S\ c\ f\ \sigma_0$ Foreach-loop with explicit continuation condition and annotated invariant.

Alternative syntax: $FOREACH_C{}^I\ S\ c\ f\ \sigma_0$.

partial-function (*nrec*) Mode of the partial function package for the nondeterminism monad.

2.5.2 Refinement

($\leq$) Refinement ordering. $S \leq S'$ means, that every result in S is also a result in S'. Moreover, S may only fail if S' fails. $\leq$ forms a complete lattice, with least element *SUCCEED* and greatest element *FAIL*.

$\Downarrow R$ Concretization. Takes a refinement relation $R::('c\times'a)\ set$ that relates concrete to abstract values, and returns a concretization function $\Downarrow R$.

$\Uparrow R$ Abstraction. Takes a refinement relation and returns an abstraction function. The functions $\Downarrow R$ and $\Uparrow R$ form a Galois-connection, i.e., we have: $S \leq \Downarrow R\ S' \longleftrightarrow \Uparrow R\ S \leq S'$.

$br\ \alpha\ I$ Builds a refinement relation from an abstraction function and an invariant. Those refinement relations are always single-valued.

nofail S Predicate that states that S does not fail.

inres S x Predicate that states that S includes result x. Note that a failing program includes all results.

2.5.3 Proof Tools

Verification Condition Generator:

Method: *intro refine-vcg*

Attributes: *refine-vcg*

Transforms a subgoal of the form $S \leq SPEC\ \Phi$ into verification conditions by decomposing the structure of S. Invariants for loops without annotation must be specified explicitely by instantiating the respective proof-rule for the loop construct, e.g., *intro WHILE-rule*[*where I*=...] *refine-vcg*.

refine-vcg is a named theorems collection that contains the rules that are used by default.

Refinement Condition Generator:

Method: *refine-rcg* [thms].

Attributes: *refine0*, *refine*, *refine2*.

Flags: *refine-no-prod-split*.

Tries to prove a subgoal of the form $S \leq \Downarrow R\ S'$ by decomposing the structure of S and S'. The rules to be used are contained in the theorem collection *refine*. More rules may be passed as argument to the method. Rules contained in *refine0* are always tried first, and rules in *refine2* are tried last. Usually, rules that decompose both programs equally should be put into *refine*. Rules that may make big steps, without decomposing the program further, should be put into *refine0* (e.g., *Id-refine*). Rules that decompose the programs differently and shall be used as last resort before giving up should be put into *refine2*, e.g., *remove-Let-refine*.

By default, this procedure will invoke the splitter to split product types in the goals. This behaviour can be disabled by setting the flag *refine-no-prod-split*.

Refinement Relation Heuristics:

Method: *refine-dref-type* [(trace)].

Attributes: *refine-dref-RELATES*, *refine-dref-pattern*.

Flags: *refine-dref-tracing*.

Tries to instantiate schematic refinement relations based on their type. By default, this rule is applied to all subgoals. Internally, it uses the rules declared as *refine-dref-pattern* to introduce a goal of the form *RELATES ?R*, that is then solved by exhaustively applying rules declared as *refine-dref-RELATES*.

The flag *refine-dref-tracing* controls tracing of resolving *RELATES*-goals. Tracing may also be enabled by passing (trace) as argument.

Pointwise Reasoning Simplification Rules:

Attributes: *refine-pw-simps*

A theorem collection that contains simplification lemmas to push inwards *nofail* and *inres* predicates into program constructs.

Refinement Simp Rules:

Attributes: *refine-hsimp*

A theorem collection that contains some simplification lemmas that are useful to prove membership in refinement relations.

Transfer:

> **Method:** *refine-transfer* [thms]
>
> **Attribute:** *refine-transfer*
>
> Tries to prove a subgoal of the form $\alpha\ f \leq S$ by decomposing the structure of f and S. This is usually used in connection with a schematic lemma, to generate f from the structure of S.
>
> The theorems declared as *refine-transfer* are used to do the transfer. More theorems may be passed as arguments to the method. Moreover, some simplification for nested abstraction over product types ($\lambda(a,b)$ $(c,d). \ldots$) is done, and the monotonicity prover is used on monotonicity goals.
>
> There is a standard setup for α=*RETURN* (transfer to plain function for total correct code generation), and α=*nres-of* (transfer to deterministic result monad, for partial correct code generation).

Automatic Refinement:

> **Method:** *refine-autoref*
>
> **Attributes:** ...
>
> See automatic refinement package for documentation (TBD)

Concrete Definition:

> **Command:** *concrete-definition name* [*attribs*] *for params uses thm* where *attribs* and the *for*-part are optional.
>
> Declares a new constant from the left-hand side of a refinement lemma. Has special handling for left-hand sides of the forms *RETURN* - and *nres-of*, in which cases those topmost functions are not included in the defined constant.
>
> The refinement lemma is folded with the new constant and registered as *name.refine*.
>
> **Command:** *prepare-code-thms thms* takes a list of definitional theorems and sets up lemmas for the code generator for those definitions. This includes handling of recursion combinators.

2.5.4 Packages

The following parts of the refinement framework are not included by default, but can be imported if necessary:

Collection-Bindings: Sets up refinement rules for the Isabelle Collection Framework. With this theory loaded, the refinement condition generator will discharge most data refinements using the ICF automatically. Moreover, the transfer procedure will replace *FOREACH*-statements by the corresponding ICF-iterators.

end

References

[1] P. Lammich. Collections framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/collections.shtml, Dec. 2009. Formal proof development.

[2] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

[3] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

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