The Isabelle Collections Framework

Peter Lammich
WWU Münster

Andreas Lochbihler
Karlsruhe Institute of Technology

14 July 2010

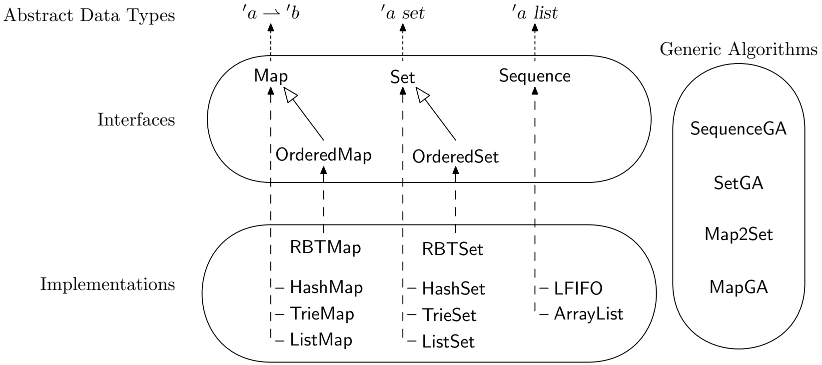
Motivation

- Generation of efficient, verified code from Isabelle/HOL
  - Requires efficient (purely functional) data structures
- Options
  - Some data structures spread around Isabelle library and AFP
    - Different interfaces, different sets of implemented operations
  - Ad-hoc implementations of efficient DS within larger projects
  - Manual editing of generated code
  - Using lists for everything
    - Also common in unverified functional programming
  - Automatic translation from ADT to CDT (Since 2009-2)
    - Problems with underspecified functions
      e.g. select an element from a set

The Isabelle Collection Framework

- Unified interface to collection data structures
- Easy to use
  - Little effort to generate executable code
  - Suited for larger developments
- Extensible
  - Easy to add new interfaces, algorithms, data structures
- Efficient
  - Vastly outperforms default code generator

Overview


Abstract Data Types
$'a \rightharpoonup 'b$
$'a\ set$
$'a\ list$
Interfaces
Map
Set
Sequence
OrderedMap
OrderedSet
Implementations
RBTMap
RBTSet
HashMap
TrieMap
ListMap
HashSet
TrieSet
ListSet
LFIFO
ArrayList
Generic Algorithms
SequenceGA
SetGA
Map2Set
MapGA

Data Refinement

ADT [e.g. $'a\ set$] Abstract Algorithm [f]

$\uparrow \alpha$

CDT [e.g. $'a\ RBTree$] Concrete Algorithm [$\bar{f}$]

- Show
  1. $P\ x \Longrightarrow Q\ (f\ x)$ (Correctness of abstract algorithm)
  2. $invar\ \bar{x} \Longrightarrow invar\ (\bar{f}\ \bar{x}) \wedge \alpha\ (\bar{f}\ \bar{x}) = f\ (\alpha\ \bar{x})$ (Correctness of Implementation)
- Step 1: Independent from ICF.
  - Good setup of automatic methods for Isabelle's standard types
- Step 2: Usually discharged automatically by simplifier.
- Underspecification (e.g. $SOME\ x.\ x \in S$)
  - Nondeterministic abstract algorithm: $\alpha\ (\bar{f}\ \bar{x}) \in f\ (\alpha\ \bar{x})$
  - Parameterize abstract algorithm over underspecified operation

Interfaces

ADT	Abstract Algorithm
$\uparrow \alpha$	$\downarrow$ implementation
Interface	Generic Algorithm
$\uparrow$ interpret	$\downarrow$ instantiation
CDT	Concrete Algorithm

- Interface of CDT specified as locale
- Implementation proof done wrt. locale
- Locale interpreted with CDT

$\Rightarrow$ Separation of implementation proof and data structure

Example

Interface for sets:

$$\textbf{locale}\ \mathit{StdSet} = \mathit{StdSetDefs\ ops} +$$
$$\textbf{assumes}\ \mathit{empty\text{-}correct} : \alpha\ \mathit{empty} = \{\}\quad \textbf{and}\quad \mathit{invar\ empty}$$
$$\textbf{assumes}\ \mathit{ins\text{-}correct} : \mathit{invar}\ s \Longrightarrow \alpha\ (\mathit{ins}\ x\ s) = \{x\} \cup \alpha\ s$$
$$\mathit{invar}\ s \Longrightarrow \mathit{invar}\ (\mathit{ins}\ x\ s)$$
$$\ldots$$

Implementation of interface:

$$\textbf{definition}\ \mathit{hs\text{-}ops}\ \equiv\ \ldots$$
$$\textbf{interpretation}\ \mathit{hs}! :\ \mathit{StdSet\ hs\text{-}ops}$$
$$\textbf{proof}$$
$$\ldots$$

Example

Abstract Algorithm:

fun *set-a* **where**
  $set\text{-}a\ [] = \{\}$
  $set\text{-}a\ (a\#l) = (insert\ a\ (set\text{-}a\ l))$

Generic Algorithm:

context *StdSetDefs* **begin**
  fun *set-g* **where**
    $set\text{-}g\ [] = empty$
    $set\text{-}g\ (a\#l) = (ins\ a\ (set\text{-}g\ l))$

Correct implementation:

lemma (**in** *StdSet*) *set-g-correct* :
  $invar\ (set\text{-}g\ l) \wedge \alpha\ (set\text{-}g\ l) = set\text{-}a\ l$
by (*induct l*) (*auto simp add* : *correct*)

Instantiation: Now available: *hs.set-g*, *rs.set-g*, . . .

Extending the ICF

- New interfaces
- New data structures
  - Supported by library of generic algorithms
  - Implement all operations from a few basic operations
  - Adapt one interface to another (e.g. set-by-map)
- New generic algorithms
  - Naming conventions make instantiation canonical
  - Currently: Ad-hoc script for automatic instantiation

ICF for Larger Developments

- Operations used with different types by GA must be specified separately

locale *MyContextDefs* =
  StdSetDefs ops **for** *ops* :: $(nat, 's)$ *set-ops*+
  fixes *iterate* :: $('s, nat, nat \times nat)$ *iterator*
  fixes *iterate'* :: $('s, nat, 's)$ *iterator*
begin
  definition *avg-aux* :: $'s \Rightarrow nat \times nat$ **where**
    $avg\text{-}aux\ s == iterate\ (\lambda\ x\ (c, sum).\ (c + 1, sum + x))\ s\ (0, 0)$

  definition $avg\ s ==$ **let** $(c, sum) = avg\text{-}aux\ s$ **in** $sum\ div\ c$

  definition $filter\text{-}le\text{-}avg\ s ==$ **let** $a = avg\ s$ **in**
    $iterate'\ (\lambda\ x\ s.$ **if** $x \leq a$ **then** $ins\ x\ s$ **else** $s)\ s\ empty$
end

ICF for Larger Developments

- Operations used with different types by GA must be specified separately
  - Alternative: Work with fixed CDT
    - Switching to other CDT still easy due to uniform naming scheme. (E.g. replace *hs-xxx* by *rs-xxx*)

ICF for Larger Developments

- Operations used with different types by GA must be specified separately
  - Alternative: Work with fixed CDT
    - Switching to other CDT still easy due to uniform naming scheme. (E.g. replace *hs-xxx* by *rs-xxx*)
  - Or use abbreviations at top of theory
    - Only local changes required to switch CDT

types $'a$ *my-set* $=$ $'a$ *hs*
abbreviation *my-*α $=$ *hs-*α
abbreviation *my-invar* $=$ *hs-invar*
abbreviation *my-empty* $=$ *hs-empty*
...
lemmas my-correct = hs-correct

ICF for Larger Developments

- Operations used with different types by GA must be specified separately
  - Alternative: Work with fixed CDT
    - Switching to other CDT still easy due to uniform naming scheme. (E.g. replace *hs-xxx* by *rs-xxx*)
  - Or use abbreviations at top of theory
    - Only local changes required to switch CDT
- Invariants explicitly visible
  - Cumbersome with nested data structures, e.g. map from keys to sets of values.

$$\mathit{complex\text{-}invar}\ m = \mathit{hm\text{-}invar}\ m \wedge (\forall s \in \mathit{ran}\ (\mathit{hm\text{-}\alpha}\ m).\ \mathit{hs\text{-}invar}\ s)$$

ICF for Larger Developments

- Operations used with different types by GA must be specified separately
  - Alternative: Work with fixed CDT
    - Switching to other CDT still easy due to uniform naming scheme. (E.g. replace *hs-xxx* by *rs-xxx*)
  - Or use abbreviations at top of theory
    - Only local changes required to switch CDT
- Invariants explicitly visible
  - Cumbersome with nested data structures, e.g. map from keys to sets of values.
  - Real problem with function package
    - Termination may depend on invariant, get equations of the form:

$$invar\ s \Longrightarrow f\ s = \dots$$

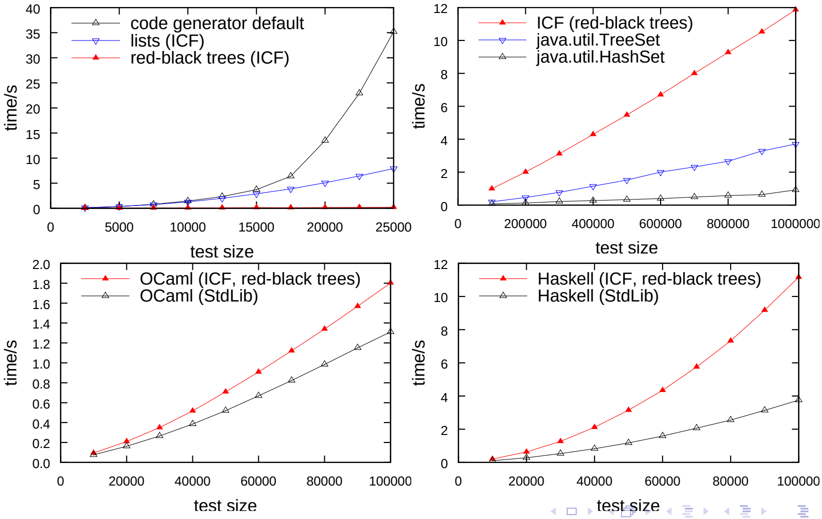
    - Conditional equations cannot be used as code equations

ICF for Larger Developments

- Operations used with different types by GA must be specified separately
  - Alternative: Work with fixed CDT
    - Switching to other CDT still easy due to uniform naming scheme. (E.g. replace *hs-xxx* by *rs-xxx*)
  - Or use abbreviations at top of theory
    - Only local changes required to switch CDT
- Invariants explicitly visible
  - Cumbersome with nested data structures, e.g. map from keys to sets of values.
  - Real problem with function package
  - Isabelle2009-2: Invariants may be hidden in typedefs

Efficiency

- Inserting/deleting/testing random numbers, and iteration to sum up numbers in set



Case Study

- Implemented tree automata library with ICF
  - Fixed CDT: Using red-black trees for maps and sets
- Test: Intersect pairs of random automata and check result for emptiness
  - Haskell is fastest, even comparable with Java implementation
  - Our library vastly outperforms Timbuk/Taml
    - They use lists to implement sets

Language	ICF Haskell	ICF SML	ICF OCaml	ICF OCaml(i)	Taml OCaml(i)	LETHAL Java
complete	1.5s	6.1s	12.5s	121s	1923s	0.46s
reduced	0.07s	0.41s	0.52s	4.98s	71.64s	0.12s

Conclusion

- Collection Framework for Isabelle/HOL
- Efficient
- Easy to use, suitable for larger developments

Conclusion

- Collection Framework for Isabelle/HOL
- Efficient
- Easy to use, suitable for larger developments

Ongoing work

- Add more ADTs (Heaps, Priority Queues, ...)
  - Arrays mapped to persistent arrays in ML/Haskell
- Encapsulate invariants in typedefs

Future work

- Equality of keys/elements
  - Currently, logical equality is used.
  - Not adequate for nested data structures
    e.g., hash-set of tree-sets
- Tune existing implementations for efficiency
- State Monads (Imperative HOL): More efficient, but more effort to create executable code.