Automatic Data Refinement

Peter Lammich

TU München

July 2013

Motivation

- Nondeterministic algorithms
  - Quite common on abstract level, e.g. *pick element from set*
  - Automatic refinement via code generator not available
  - Manual refinement is tedious

Motivation

- Nondeterministic algorithms
  - Quite common on abstract level, e.g. *pick element from set*
  - Automatic refinement via code generator not available
  - Manual refinement is tedious
- Here: Autoref - Framework
  - Integrated with Isabelle Refinement Framework [ITP'12]
  - Provides collection framework with generic algorithm library
  - Examples and case studies

Motivation

- Nondeterministic algorithms
  - Quite common on abstract level, e.g. *pick element from set*
  - Automatic refinement via code generator not available
  - Manual refinement is tedious
- Here: Autoref - Framework
  - Integrated with Isabelle Refinement Framework [ITP'12]
  - Provides collection framework with generic algorithm library
  - Examples and case studies
- Features
  - Completely inside the logic
  - Generic algorithms, specialization
  - Compound abstract types (e.g. $\alpha \to (\beta)\,\text{option}$)
  - Refinements with side conditions
  - ...

Outline

1. Identify Abstract Operations
2. Select Implementations
3. Synthesize Implementation and Refinement Theorem
4. Wrap up into usable framework

Outline

1. Identify Abstract Operations
2. Select Implementations
3. Synthesize Implementation and Refinement Theorem
4. Wrap up into usable framework

Operation Identification

- Isabelle already „implements” some abstract concepts
  - e.g. Map: $\alpha \to (\beta)\,\text{option}$, update: function update
  - Makes sense for proving, but problem for refinement
  - Ambiguity: Implement as function or as map?

User writes: $[\text{True} \mapsto 1, \text{False} \mapsto 0]$

Isabelle sees: $(\lambda x.\ \text{None})(\text{True} := \text{Some}\ 1)(\text{False} := \text{Some}\ 0)$

Operation Identification

- Isabelle already „implements” some abstract concepts
  - e.g. Map: $\alpha \to (\beta)$ option, update: function update
  - Makes sense for proving, but problem for refinement
  - Ambiguity: Implement as function or as map?
- Goal: Make abstract concepts visible
  - Define constants for operations, e.g.

$$\text{map-upd}\ m\ k\ v = m(k := \text{Some}\ v)$$
$$\text{map-empty} = \lambda x.\ \text{None}$$

  - Rewrite term to use these constants

User writes: $[\text{True} \mapsto 1, \text{False} \mapsto 0]$

Isabelle sees: $(\lambda x.\ \text{None})(\text{True} := \text{Some}\ 1)(\text{False} := \text{Some}\ 0)$

Rewrite to: $\text{map-upd}\ (\text{map-upd}\ \text{map-empty}\ \text{True}\ 1)\ \text{False}\ 0$

Type-System based Heuristic

- Use type system to ensure that abstract concepts fit together
  - Types represent abstract concepts
    - e.g. map-upd : (α, β) i-map $\rightarrow \alpha \rightarrow \beta \rightarrow (\alpha, \beta)$ i-map
  - Independent from Isabelle's types

Type-System based Heuristic

- Use type system to ensure that abstract concepts fit together
  - Types represent abstract concepts
    - e.g. map-upd : (α, β) i-map $\to \alpha \to \beta \to (\alpha, \beta)$ i-map
  - Independent from Isabelle's types
- Standard typing rules:

$$\frac{c : l}{\Gamma \vdash c : l} \qquad \frac{x : l \in \Gamma}{\Gamma \vdash x : l} \qquad \frac{\Gamma \vdash t : l_1 \quad \Gamma \vdash f : l_1 \to l_2}{\Gamma \vdash f\ t : l_2} \qquad \frac{(x : l_1)\Gamma \vdash t : l_2}{\Gamma \vdash (\lambda x.\ t) : l_1 \to l_2}$$

Type-System based Heuristic

- Use type system to ensure that abstract concepts fit together
  - Types represent abstract concepts
    - e.g. map-upd : (α, β) i-map $\to \alpha \to \beta \to (\alpha, \beta)$ i-map
  - Independent from Isabelle's types
- Standard typing rules:

$$\frac{c : l}{\Gamma \vdash c : l} \qquad \frac{x : l \in \Gamma}{\Gamma \vdash x : l} \qquad \frac{\Gamma \vdash t : l_1 \quad \Gamma \vdash f : l_1 \to l_2}{\Gamma \vdash f\ t : l_2} \qquad \frac{(x : l_1)\Gamma \vdash t : l_2}{\Gamma \vdash (\lambda x.\ t) : l_1 \to l_2}$$

- Rewrite rule:

$$\frac{t \equiv t' \quad \Gamma \vdash t' : l}{\Gamma \vdash t : l}$$

  - Detects operations, e.g. $m(k := \text{Some}\ \ v) \equiv \text{map-upd}\ \ m\ k\ v$

Type-System based Heuristic

- Use type system to ensure that abstract concepts fit together
  - Types represent abstract concepts
    - e.g. map-upd : (α, β) i-map $\to \alpha \to \beta \to (\alpha, \beta)$ i-map
  - Independent from Isabelle's types
- Standard typing rules:

$$\frac{c : l}{\Gamma \vdash c : l} \qquad \frac{x : l \in \Gamma}{\Gamma \vdash x : l} \qquad \frac{\Gamma \vdash t : l_1 \quad \Gamma \vdash f : l_1 \to l_2}{\Gamma \vdash f\ t : l_2} \qquad \frac{(x : l_1)\Gamma \vdash t : l_2}{\Gamma \vdash (\lambda x.\ t) : l_1 \to l_2}$$

- Rewrite rule:

$$\frac{t \equiv t' \quad \Gamma \vdash t' : l}{\Gamma \vdash t : l}$$

  - Detects operations, e.g. $m(k := \text{Some}\ \ v) \equiv \text{map-upd}\ \ m\ k\ v$
- Search for type by backtracking over rewrite rules
  - Order of rewrite rules influences inferred typing

Type-System based Heuristic

- Use type system to ensure that abstract concepts fit together
  - Types represent abstract concepts
    - e.g. map-upd : (α, β) i-map $\to \alpha \to \beta \to (\alpha, \beta)$ i-map
  - Independent from Isabelle's types
- Standard typing rules:

$$\frac{c : l}{\Gamma \vdash c : l} \qquad \frac{x : l \in \Gamma}{\Gamma \vdash x : l} \qquad \frac{\Gamma \vdash t : l_1 \quad \Gamma \vdash f : l_1 \to l_2}{\Gamma \vdash f\ t : l_2} \qquad \frac{(x : l_1)\Gamma \vdash t : l_2}{\Gamma \vdash (\lambda x.\ t) : l_1 \to l_2}$$

- Rewrite rule:

$$\frac{t \equiv t' \quad \Gamma \vdash t' : l}{\Gamma \vdash t : l}$$

  - Detects operations, e.g. $m(k := \text{Some}\ \ v) \equiv \text{map-upd}\ \ m\ k\ v$
- Search for type by backtracking over rewrite rules
  - Order of rewrite rules influences inferred typing
- Works well in practice

Outline

1. Identify Abstract Operations
2. Select Implementations
3. Synthesize Implementation and Refinement Theorem
4. Wrap up into usable framework

Selection of Implementation

- Goal: Annotate operations by implementation types

```
(map-upd
  (map-upd map-empty True 1)
  False 0
)
```

Selection of Implementation

- Goal: Annotate operations by implementation types
- E.g. implement Booleans by integers, maps by association lists

$$
\begin{array}{l}
(\text{map-upd} \\
\quad (\text{map-upd}\ \ \text{map-empty}_{(\text{int} \times \text{int})\ \text{list}}\ \ \text{True}_{\text{int}}\ \ 1_{\text{int}})_{(\text{int} \times \text{int})\ \text{list}} \\
\quad \text{False}_{\text{int}}\ \ 0_{\text{int}} \\
)_{(\text{int} \times \text{int})\ \text{list}}
\end{array}
$$

Selection of Implementation

- Goal: Annotate operations by implementation types
- E.g. implement Booleans by integers, maps by association lists
- Finding suitable implementations automatically is difficult
  - But heuristics do a good job (rapid prototyping)
  - Configurable by user (production version)

$$
\begin{array}{l}
(\text{map-upd} \\
\quad (\text{map-upd}\ \ \text{map-empty}_{(\text{int} \times \text{int})\ \text{list}}\ \ \text{True}_{\text{int}}\ \ 1_{\text{int}})_{(\text{int} \times \text{int})\ \text{list}} \\
\quad \text{False}_{\text{int}}\ \ 0_{\text{int}} \\
)_{(\text{int} \times \text{int})\ \text{list}}
\end{array}
$$

Heuristics to Find Suitable Implementations

1. User annotations: Force specific implementation
   - $[\text{True} \mapsto 1, \text{False} \mapsto 0]_{(\text{int} \times \text{int})\ \text{rbt}}$

Heuristics to Find Suitable Implementations

1. User annotations: Force specific implementation
   - $[\text{True} \mapsto 1, \text{False} \mapsto 0]_{(\text{int} \times \text{int})\ \text{rbt}}$
2. Type-based rules: Fix implementation for abstract type
   - $(\text{bool}, \text{-})\ \text{i-map} \mapsto (\text{bool} \times \text{-})\ \text{list} \qquad (\text{int}, \text{-})\ \text{i-map} \mapsto (\text{int}, \text{-})\ \text{rbt}$

Heuristics to Find Suitable Implementations

1. User annotations: Force specific implementation
   - $[\text{True} \mapsto 1, \text{False} \mapsto 0]_{(\text{int} \times \text{int})\ \text{rbt}}$
2. Type-based rules: Fix implementation for abstract type
   - $(\text{bool}, \text{-})\ \text{i-map} \mapsto (\text{bool} \times \text{-})\ \text{list}$ $\quad$ $(\text{int}, \text{-})\ \text{i-map} \mapsto (\text{int}, \text{-})\ \text{rbt}$
3. Homogeneity: Prefer same implementation for operands
   - $a \cup b$ — Prefer same implementation for a, b, and result

Heuristics to Find Suitable Implementations

1. User annotations: Force specific implementation
   - $[\text{True} \mapsto 1, \text{False} \mapsto 0]_{(\text{int} \times \text{int})\ \text{rbt}}$
2. Type-based rules: Fix implementation for abstract type
   - $(\text{bool}, \text{-})\ \text{i-map} \mapsto (\text{bool} \times \text{-})\ \text{list}$ $\quad$ $(\text{int}, \text{-})\ \text{i-map} \mapsto (\text{int}, \text{-})\ \text{rbt}$
3. Homogeneity: Prefer same implementation for operands
   - $a \cup b$ — Prefer same implementation for a, b, and result
4. Priorities for data structures and operations
   - hash $\geq$ rbt $\geq$ list
   - list-upd-dj $\geq$ list-upd

Outline

1. Identify Abstract Operations
2. Select Implementations
3. Synthesize Implementation and Refinement Theorem
4. Wrap up into usable framework

Relational Parametricity [Reynolds '83, Wadler '89]

- Relators
  - Construct relations between concrete and abstract values

$$\text{ib-rel} = \{(i, b) \mid b \Leftrightarrow i \neq 0\}$$
$$R_1 \to R_2 = \{(f, g) \mid \forall (x, y) \in R_1.\ (f\ x, g\ y) \in R_2\}$$
$$\langle R \rangle\ \text{list-rel} = \{([c_1, \ldots, c_n], [a_1, \ldots, a_n]) \mid \forall i.\ (c_i, a_i) \in R\}$$

Relational Parametricity [Reynolds '83, Wadler '89]

- Relators
  - Construct relations between concrete and abstract values

$$\text{ib-rel} = \{(i, b) \mid b \Leftrightarrow i \neq 0\}$$
$$R_1 \to R_2 = \{(f, g) \mid \forall (x, y) \in R_1.\ (f\ x, g\ y) \in R_2\}$$
$$\langle R \rangle\ \text{list-rel} = \{([c_1, \ldots, c_n], [a_1, \ldots, a_n]) \mid \forall i.\ (c_i, a_i) \in R\}$$

- Transfer Rules
  - Describe refinement between concrete and abstract operations

$$(\&, \wedge) \in \text{ib-rel} \to \text{ib-rel} \to \text{ib-rel}$$

$$(\text{Nil}, \text{Nil}) \in \langle R \rangle\ \text{list-rel}$$

$$(\text{Cons}, \text{Cons}) \in R \to \langle R \rangle\ \text{list-rel} \to \langle R \rangle\ \text{list-rel}$$

Relational Parametricity [Reynolds '83, Wadler '89]

- Relators
  - Construct relations between concrete and abstract values

$$\text{ib-rel} = \{(i, b) \mid b \Leftrightarrow i \neq 0\}$$
$$R_1 \to R_2 = \{(f, g) \mid \forall (x, y) \in R_1.\ (f\ x, g\ y) \in R_2\}$$
$$\langle R \rangle \text{ list-rel} = \{([c_1, \ldots, c_n], [a_1, \ldots, a_n]) \mid \forall i.\ (c_i, a_i) \in R\}$$

- Transfer Rules
  - Describe refinement between concrete and abstract operations

$$(\&, \wedge) \in \text{ib-rel} \to \text{ib-rel} \to \text{ib-rel}$$
$$\& : \mathit{int} \to \mathit{int} \to \mathit{int},\ \wedge : \text{bool} \to \text{bool} \to \text{bool}$$
$$(\text{Nil}, \text{Nil}) \in \langle R \rangle \text{ list-rel}$$
$$\text{Nil} : \alpha \text{ list}$$
$$(\text{Cons}, \text{Cons}) \in R \to \langle R \rangle \text{ list-rel} \to \langle R \rangle \text{ list-rel}$$
$$\text{Cons} : \alpha \to \alpha \text{ list} \to \alpha \text{ list}$$

- Note similarity to types

Synthesis

- Transfer rules for abstraction and application

$$\frac{\forall (x, y) \in R_1 .\ (s, t) \in R_2}{(\lambda x.\ s, \lambda y.\ t) \in R_1 \to R_2} \qquad \frac{(f, g) \in R_1 \to R_2 \quad (x, y) \in R_2}{(f\ x, g\ y) \in R_2}$$

Synthesis

- Transfer rules for abstraction and application

$$\frac{\forall (x, y) \in R_1 .\ (s, t) \in R_2}{(\lambda x.\ s, \lambda y.\ t) \in R_1 \to R_2} \qquad \frac{(f, g) \in R_1 \to R_2 \quad (x, y) \in R_2}{(f\ x, g\ y) \in R_2}$$

- Example: Synthesis for term $\lambda x\ y.\ [x \wedge y]$

Synthesis

- Transfer rules for abstraction and application

$$\frac{\forall (x, y) \in R_1 .\ (s, t) \in R_2}{(\lambda x.\ s, \lambda y.\ t) \in R_1 \to R_2} \qquad \frac{(f, g) \in R_1 \to R_2 \quad (x, y) \in R_2}{(f\ x, g\ y) \in R_2}$$

- Example: Synthesis for term $\lambda x\ y.\ [x \wedge y]$
  1. Start with goal $(?c, \lambda x\ y.\ [x \wedge y]) \in ?R$
     - $?c$ and $?R$ are instantiated during the proof

Synthesis

- Transfer rules for abstraction and application

$$\frac{\forall (x, y) \in R_1 .\ (s, t) \in R_2}{(\lambda x.\ s, \lambda y.\ t) \in R_1 \to R_2} \qquad \frac{(f, g) \in R_1 \to R_2 \quad (x, y) \in R_2}{(f\ x, g\ y) \in R_2}$$

- Example: Synthesis for term $\lambda x\ y.\ [x \wedge y]$
  1. Start with goal $(?c, \lambda x\ y.\ [x \wedge y]) \in ?R$
     - $?c$ and $?R$ are instantiated during the proof
  2. Discharge by resolving with transfer rules

Synthesis

- Transfer rules for abstraction and application

$$\frac{\forall (x, y) \in R_1 .\ (s, t) \in R_2}{(\lambda x.\ s, \lambda y.\ t) \in R_1 \to R_2} \qquad \frac{(f, g) \in R_1 \to R_2 \quad (x, y) \in R_2}{(f\ x, g\ y) \in R_2}$$

- Example: Synthesis for term $\lambda x\ y.\ [x \wedge y]$
  1. Start with goal $(?c, \lambda x\ y.\ [x \wedge y]) \in ?R$
     - $?c$ and $?R$ are instantiated during the proof
  2. Discharge by resolving with transfer rules
  3. Proved theorem

$$(\lambda x\ y.\ [x \& y], \lambda x\ y.\ [x \wedge y]) \in \text{ib-rel} \to \text{ib-rel} \to \langle \text{ib-rel} \rangle \text{ list-rel}$$

Implicit Operations

- Implicit operations on abstract type: e.g. $=$
  - Concrete type must provide corresponding operation

Implicit Operations

- Implicit operations on abstract type: e.g. $=$
  - Concrete type must provide corresponding operation
- Example: Maps by association lists
  - $\langle R, S\rangle$ lm-rel $= \langle R \times S\rangle$ list-rel $\circ \{(l, \text{map-of } l) \mid \text{distinct-keys } l\}$
  - Transfer rule for update:

$$\frac{(eq, =) \in R \to R \to \text{Id}}{(\text{update } eq, \text{map-upd}) \in \langle R, S\rangle \text{ lm-rel} \to R \to S \to \langle R, S\rangle \text{ lm-rel}}$$

Implicit Operations

- Implicit operations on abstract type: e.g. $=$
  - Concrete type must provide corresponding operation
- Example: Maps by association lists
  - $\langle R, S\rangle$ lm-rel $= \langle R \times S\rangle$ list-rel $\circ \{(l, \text{map-of } l) \mid \text{distinct-keys } l\}$
  - Transfer rule for update:

$$\frac{(eq, =) \in R \to R \to \text{Id}}{(\text{update } eq, \text{map-upd}) \in \langle R, S\rangle \text{ lm-rel} \to R \to S \to \langle R, S\rangle \text{ lm-rel}}$$

- This leads to generic programming by template instantiation

Generic Programming

- Implement operations generically in terms of other operations

Generic Programming

- Implement operations generically in terms of other operations
- Transfer rules of form

$$\frac{(c_1, a_1) \in R_1 \quad \ldots \quad (c_n, a_n) \in R_n}{(c\ c_1\ \ldots\ c_n, a) \in R}$$

Generic Programming

- Implement operations generically in terms of other operations
- Transfer rules of form

$$\frac{(c_1, a_1) \in R_1 \quad \ldots \quad (c_n, a_n) \in R_n}{(c\ c_1\ \ldots\ c_n, a) \in R}$$

- Intuition: To implement a, find implementations for $a_1, \ldots, a_n$

Generic Programming

- Implement operations generically in terms of other operations
- Transfer rules of form

$$\frac{(c_1, a_1) \in R_1 \quad \ldots \quad (c_n, a_n) \in R_n}{(c\ c_1\ \ldots\ c_n, a) \in R}$$

- Intuition: To implement a, find implementations for $a_1, \ldots, a_n$
- Specialization via higher priority
  - e.g. rbt-union $\geq$ gen-union

Generic Programming

- Implement operations generically in terms of other operations
- Transfer rules of form

$$\frac{(c_1, a_1) \in R_1 \quad \ldots \quad (c_n, a_n) \in R_n}{(c\ c_1\ \ldots\ c_n, a) \in R}$$

- Intuition: To implement a, find implementations for $a_1, \ldots, a_n$
- Specialization via higher priority
  - e.g. rbt-union $\geq$ gen-union
- E.g. implement map/set operations using five basic operations
  - empty, lookup, update, delete, iterate

Example: Disjointness Test

- Note: $a \cap b = \{\} \Leftrightarrow \forall x \in a.\ x \notin b$

Example: Disjointness Test

- Note: $a \cap b = \{\} \Leftrightarrow \forall x \in a.\ x \notin b$
- Transfer rule:

$$\frac{\begin{array}{c}(B, \lambda s\ P.\ \forall x \in s.\ P\ x) \in \langle Re \rangle Rs_1 \to (Re \to \mathsf{Id}) \to \mathsf{Id} \\ (M, \in) \in Re \to \langle Re \rangle Rs_2 \to \mathsf{Id}\end{array}}{(\text{gen-disjoint}\ B\ M, \text{set-disjoint}) \in \langle Re \rangle Rs_1 \to \langle Re \rangle Rs_2 \to \mathsf{Id}}$$

where:

$$\text{set-disjoint}\ s_1\ s_2 \equiv s_1 \cap s_2 = \{\}$$
$$\text{gen-disjoint}\ B\ M\ s_1\ s_2 = B\ s_1\ (\lambda x.\ \neg M\ x\ s_2)$$

- B implements bounded quantification, M implements membership

Side Conditions

- Some transfer rules require side conditions
  - E.g. hd : (α) list $\rightarrow \alpha$ only defined on non-empty lists

Side Conditions

- Some transfer rules require side conditions
  - E.g. hd : (α) list $\rightarrow \alpha$ only defined on non-empty lists
- Transfer rule

$$\frac{a \neq [] \quad (c, a) \in \langle R \rangle \text{ list-rel}}{(\text{hd } c, \text{hd } a) \in R}$$

Side Conditions

- Some transfer rules require side conditions
  - E.g. hd : (α) list $\rightarrow \alpha$ only defined on non-empty lists
- Transfer rule

$$\frac{a \neq [] \quad (c, a) \in \langle R \rangle \text{ list-rel}}{(\text{hd } c, \text{hd } a) \in R}$$

- Try to discharge side-conditions during synthesis

Side Conditions

- Some transfer rules require side conditions
  - E.g. hd : $(\alpha)\,\text{list} \to \alpha$ only defined on non-empty lists
- Transfer rule

$$\frac{a \neq [] \quad (c, a) \in \langle R \rangle \text{ list-rel}}{(\text{hd } c, \text{hd } a) \in R}$$

- Try to discharge side-conditions during synthesis
  - Congruence rules to gather required information

$$\frac{(c', c) \in \text{Id} \quad c \implies (t', t) \in R \quad \neg c \implies (e', e) \in R}{(\text{If } c' \; t' \; e', \text{If } c \; t \; e) \in R}$$

Side Conditions

- Some transfer rules require side conditions
  - E.g. hd : $(\alpha)\,\text{list} \to \alpha$ only defined on non-empty lists
- Transfer rule

$$\frac{a \neq [] \quad (c, a) \in \langle R \rangle \text{ list-rel}}{(\text{hd } c, \text{hd } a) \in R}$$

- Try to discharge side-conditions during synthesis
  - Congruence rules to gather required information

$$\frac{(c', c) \in \text{Id} \quad c \implies (t', t) \in R \quad \neg c \implies (e', e) \in R}{(\text{If } c' \ t' \ e', \text{If } c \ t \ e) \in R}$$

- Also used for optimization

$$\frac{k \notin \text{dom } s \quad (k', k) \in R \quad (v', v) \in S \quad (m', m) \in \langle R, S \rangle \text{ lm-rel}}{((k', v') \# s', \text{map-upd } x \ s) \in \langle R, S \rangle \text{ ls-rel}}$$

Summary (Automatic Refinement)

1. Identify operations

$$[\text{True} \mapsto 1, \text{False} \mapsto 0] \quad = \quad \text{map-upd}\ (\text{map-upd}\ \text{map-empty}\ \text{True}\ 1)\ \text{False}\ 0$$

2. Select implementations

$$\text{map-upd}\ (\text{map-upd}\ \text{map-empty}_{(\text{int},\text{int})\ \text{list}}\ \text{True}_{\text{int}}\ 1_{\text{int}})\ \ldots$$

3. Synthesize concrete term and refinement theorem

$$([(\text{False}, 0), (\text{True}, 1)], [\text{True} \mapsto 1, \text{False} \mapsto 0]) \in \langle \text{ib-rel}, \text{Id} \rangle\ \text{lm-rel}$$

Outline

1. Identify Abstract Operations
2. Select Implementations
3. Synthesize Implementation and Refinement Theorem
4. Wrap up into usable framework

The Autoref-Framework

- Refinement based approach to algorithm verification in Isabelle

The Autoref-Framework

- Refinement based approach to algorithm verification in Isabelle
- Integration with the Isabelle Refinement Framework
  - Provides a refinement calculus for the set-exception monad
  - $\langle R \rangle$ nres-rel $= \{(c, a) \mid c$ refines a w.r.t. data refinement $R\}$

The Autoref-Framework

- Refinement based approach to algorithm verification in Isabelle
- Integration with the Isabelle Refinement Framework
  - Provides a refinement calculus for the set-exception monad
  - $\langle R \rangle$ nres-rel $= \{(c, a) \mid c$ refines a w.r.t. data refinement $R\}$
- Integration with the Isabelle Collection Framework
  - Provides verified collection data structures
  - But nesting (e.g. $((\alpha)$ set$)$ set) not possible

The Autoref-Framework

- Refinement based approach to algorithm verification in Isabelle
- Integration with the Isabelle Refinement Framework
  - Provides a refinement calculus for the set-exception monad
  - $\langle R \rangle$ nres-rel $= \{(c, a) \mid c$ refines a w.r.t. data refinement $R\}$
- Integration with the Isabelle Collection Framework
  - Provides verified collection data structures
  - But nesting (e.g. $((\alpha)$ set$)$ set) not possible
- We implemented a generic collection framework
  - Map and set data structures
  - Library of generic algorithms
  - Arbitrary nesting

Case Studies

- Applied on complex algorithms

  INY Compute Simulation Preorders in NFAs (Ilie, Navarro, Yu)

  NDFS Nested DFS for emptiness check of Büchi automata

- Drastically reduced code-generation boilerplate

Refinement of INY to Executable Version

Original code:

```
section {* Concretisation of data structures *}

text {* This locale assumes the existence of all the required data
    structures, operations and iterator functions. *}
locale INY_locale =
  NFA 𝒜 +
  qq_set : StdSet qq_set_ops +
  qq_𝒞_set : StdSet qq_𝒞_set_ops +
  q_set : StdSet q_set_ops +
  q_cart_it : set_iteratei q_set.α q_set.invar q_cart_it +
  q_ω𝒩_it : set_iteratei q_set.α q_set.invar q_ω𝒩_it +
  q_ω𝒞_it : set_iteratei q_set.α q_set.invar q_ω𝒞_it +
  qq_qq_it : set_iteratei qq_set.α qq_set.invar qq_qq_it +
  c_set : StdSet c_set_ops +
  N_map : StdMap N_map_ops +
  d_map : StdMap d_map_ops +
  δ𝒻_map : StdMap δ𝒻_map_ops
  for 𝒜 :: "('q,'c,_) NFA_rec_scheme"
  and qq_set_ops :: "('q×'q, 'qq_set, _) set_ops_scheme"
  and qq_𝒞_set_ops :: "('q×'q, 'qq_𝒞_set, _) set_ops_scheme"
  and q_set_ops :: "('q, 'q_set, _) set_ops_scheme"
  and q_cart_it :: "'q_set ⇒ ('q,'qq_set) set_iterator"
  and q_ω𝒩_it :: "'q_set ⇒ ('q,'qq_set×'qq_𝒞_set×'N_map) set_iterator"
  and q_ω𝒞_it :: "'q_set ⇒ ('q,'qq_set×'qq_𝒞_set) set_iterator"
  and qq_qq_it :: "'qq_set ⇒ ('q×'q, 'qq_𝒞_set) set_iterator"
  and c_set_ops :: "('c, 'c_set, _) set_ops_scheme"
  and N_map_ops :: "('c×'q×'q, nat, 'N_map, _) map_ops_scheme"
  and d_map_ops :: "('q×'c, nat, 'd_map, _) map_ops_scheme"
  and δ𝒻_map_ops :: "('q×'c, 'q_set, 'δ𝒻_map, _) map_ops_scheme" +

  fixes q_dδ𝒻_it :: "('q, 'd_map×'δ𝒻_map) set_iterator"
  and q_𝒩_it :: "('q, 'N_map) set_iterator"
  and Σ_𝒩_it :: "('c, 'N_map) set_iterator"
  and q_qq_it :: "('q, 'qq_set) set_iterator"
  and Σ_dδ𝒻_it :: "('c, 'd_map×'δ𝒻_map) set_iterator"
  and Σ_qq_it :: "('c, 'qq_set) set_iterator"
  and Σ_ω𝒩_it :: "('c, 'qq_set×'qq_𝒞_set×'N_map) set_iterator"
  and Δ_dδ𝒻_it :: "('q×'c×'q, 'd_map×'δ𝒻_map) set_iterator"
  and q_impl :: 'q_set
  and 𝒻_impl :: 'q_set
  assumes q_dδ𝒻_it_correct[refine_transfer]: "set_iterator q_dδ𝒻_it (q_ 𝒜)"
  assumes q_𝒩_it_correct[refine_transfer]: "set_iterator q_𝒩_it (q_ 𝒜)"
  assumes Σ_𝒩_it_correct[refine_transfer]: "set_iterator Σ_𝒩_it (Σ 𝒜)"
  assumes q_qq_it_correct[refine_transfer]: "set_iterator q_qq_it (q_ 𝒜)"
  assumes Σ_dδ𝒻_it_correct[refine_transfer]: "set_iterator Σ_dδ𝒻_it (Σ 𝒜)"
  assumes Σ_qq_it_correct[refine_transfer]: "set_iterator Σ_qq_it (Σ 𝒜)"
  assumes Σ_ω𝒩_it_correct[refine_transfer]: "set_iterator Σ_ω𝒩_it (Σ 𝒜)"
  assumes Δ_dδ𝒻_it_correct[refine_transfer]: "set_iterator Δ_dδ𝒻_it (Δ 𝒜)"
  assumes q_impl_correct: "q_set.invar q_impl" "q_set.α q_impl = q_ 𝒜"
  assumes 𝒻_impl_correct: "q_set.invar 𝒻_impl" "q_set.α 𝒻_impl = 𝒻 𝒜"
begin

  text {* Computes the cartesian product of two state sets. *}
  definition "q_cart = cart q_cart_it q_cart_it qq_set.empty qq_set.ins_dj"
```

[...] ca. 500 more lines

Refinement of INY to Executable Version

Original code:

```
section {* Concretisation of data structures *}

text {* This locale assumes the existence of all the required data
    structures, operations and iterator functions. *}
locale INY_locale =
  NFA 𝒜 +
  qq_set : StdSet qq_set_ops +
  qq_𝒞_set : StdSet qq_𝒞_set_ops +
  q_set : StdSet q_set_ops +
  q_cart_it : set_iteratei q_set.α q_set.invar q_cart_it +
  q_ω𝒩_it : set_iteratei q_set.α q_set.invar q_ω𝒩_it +
  q_ω𝒞_it : set_iteratei q_set.α q_set.invar q_ω𝒞_it +
  qq_qq_it : set_iteratei qq_set.α qq_set.invar qq_qq_it +
  c_set : StdSet c_set_ops +
  N_map : StdMap N_map_ops +
  d_map : StdMap d_map_ops +
  δ𝒻_map : StdMap δ𝒻_map_ops
  for 𝒜 :: "('q,'c,_) NFA_rec_scheme"
  and qq_set_ops :: "('q×'q, 'qq_set, _) set_ops_scheme"
  and qq_𝒞_set_ops :: "('q×'q, 'qq_𝒞_set, _) set_ops_scheme"
  and q_set_ops :: "('q, 'q_set, _) set_ops_scheme"
  and q_cart_it :: "'q_set ⇒ ('q,'qq_set) set_iterator"
  and q_ω𝒩_it :: "'q_set ⇒ ('q,'qq_set×'qq_𝒞_set×'N_map) set_iterator"
  and q_ω𝒞_it :: "'q_set ⇒ ('q,'qq_set×'qq_𝒞_set) set_iterator"
  and qq_qq_it :: "'qq_set ⇒ ('q×'q, 'qq_𝒞_set) set_iterator"
  and c_set_ops :: "('c, 'c_set, _) set_ops_scheme"
  and N_map_ops :: "('c×'q×'q, nat, 'N_map, _) map_ops_scheme"
  and d_map_ops :: "('q×'c, nat, 'd_map, _) map_ops_scheme"
  and δ𝒻_map_ops :: "('q×'c, 'q_set, 'δ𝒻_map, _) map_ops_scheme" +

  fixes q_dδ𝒻_it :: "('q, 'd_map×'δ𝒻_map) set_iterator"
  and q_N_it :: "('q, 'N_map) set_iterator"
  and Σ_N_it :: "('c, 'N_map) set_iterator"
  and q_qq_it :: "('q, 'qq_set) set_iterator"
  and Σ_dδ𝒻_it :: "('c, 'd_map×'δ𝒻_map) set_iterator"
  and Σ_qq_it :: "('c, 'qq_set) set_iterator"
  and Σ_ω𝒩_it :: "('c, 'qq_set×'qq_𝒞_set×'N_map) set_iterator"
  and Δ_dδ𝒻_it :: "('q×'c×'q, 'd_map×'δ𝒻_map) set_iterator"
  and q_impl :: 'q_set
  and 𝒻_impl :: 'q_set
  assumes q_dδ𝒻_it_correct[refine_transfer]: "set_iterator q_dδ𝒻_it (q_ 𝒜)"
  assumes q_N_it_correct[refine_transfer]: "set_iterator q_N_it (q_ 𝒜)"
  assumes Σ_N_it_correct[refine_transfer]: "set_iterator Σ_N_it (Σ 𝒜)"
  assumes q_qq_it_correct[refine_transfer]: "set_iterator q_qq_it (q_ 𝒜)"
  assumes Σ_dδ𝒻_it_correct[refine_transfer]: "set_iterator Σ_dδ𝒻_it (Σ 𝒜)"
  assumes Σ_qq_it_correct[refine_transfer]: "set_iterator Σ_qq_it (Σ 𝒜)"
  assumes Σ_ω𝒩_it_correct[refine_transfer]: "set_iterator Σ_ω𝒩_it (Σ 𝒜)"
  assumes Δ_dδ𝒻_it_correct[refine_transfer]: "set_iterator Δ_dδ𝒻_it (Δ 𝒜)"
  assumes q_impl_correct: "q_set.invar q_impl" "q_set.α q_impl = q_ 𝒜"
  assumes 𝒻_impl_correct: "q_set.invar 𝒻_impl" "q_set.α 𝒻_impl = 𝒻 𝒜"
begin

  text {* Computes the cartesian product of two state sets. *}
  definition "q_cart = cart q_cart_it q_cart_it qq_set.empty qq_set.ins_dj"
```

[...] ca. 500 more lines

Using Autoref:

```
concrete_definition compute_simrel_ex uses NFA.compute_simrel_unfold_complete

schematic_lemma compute_simrel_impl:
  assumes [autoref_rules]: "(𝒜impl,𝒜)∈⟨nat_rel,nat_rel⟩dflt_NFA_rel"
  shows "(?f,compute_simrel_ex 𝒜)∈(?R::(?'c×_) set)"
  unfolding compute_simrel_ex_def[abs_def]
  by (autoref_monadic)

concrete_definition compute_simrel_impl uses compute_simrel_impl

lemma compute_simrel_impl_refine[autoref_rules]:
  "(λ𝒜impl. RETURN (compute_simrel_impl 𝒜impl),compute_simrel_ex)∈
     ⟨nat_rel, nat_rel⟩dflt_NFA_rel
  → ⟨⟨nat_rel,⟨nat_rel⟩dflt_rs_rel⟩dflt_rm_rel⟩nres_rel"
  by (parametricity add: compute_simrel_impl.refine)

export_code compute_simrel_impl in SML file -
```

Conclusion

- Automatic Data Refinement
  - Operation Identification, Implementation Selection, Synthesis
- Autoref-Framework
  (`http://www21.in.tum.de/~lammich/autoref`)
  - Isabelle Refinement Framework + Collection Framework
- Case studies:
  - Complex algorithms, got rid of code-generation boilerplate

- Current/Future work:
  - Refinement to imperative programs
  - More complex refinements (e.g. nat to int32)
  - Smoother integration with datatype and function package