Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm

Peter Lammich, Thomas Tuerk

ITP 2012, 13th August 2012

Background

Peter Lammich (lammich@in.tum.de)

- Isabelle Collection Framework (ICF)
  - verified collection datastructures in Isabelle/HOL
  - generation of efficient functional code
- Refinement for Monadic Programs

Thomas Tuerk (tuerk@in.tum.de)

- **C**omputer **A**ided **V**erification of **A**utomata (http://cava.in.tum.de)
- Hopcroft's minimisation algorithm
- largest case study of Refinement Framework so far

Structure

1. Monadic Refinement Framework Background
2. Explanation using Hopcroft's Minimisation Algorithm
3. Conclusion

Stepwise Program Refinement

- trade-off: **abstract** vs. **concrete** formalisations of programs
  - abstract programs: easy to verify
  - concrete programs: efficiently executable
- standard solution: **stepwise program refinement**
  - Spec $\geq P_1 \geq \ldots \geq P_n \geq$ Impl
  - each step provably preserves correctness
  - steps concentrate on single issues
  - more modular and manageable proofs
- refinement framework supports **relational specifications**
- framework based on shallow embedding using **monads**

Refinement Framework Foundations

- Set/Exception monad

$$\begin{aligned}
\textbf{datatype } r \text{ nres} &::= \textbf{res } (r \text{ set}) \mid \textbf{fail} \\
\textbf{return } x &:= \textbf{res } \{x\} \\
\textbf{bind } m\ f &:= \begin{cases} \textbf{fail} & \text{if } m = \textbf{fail} \\ \bigsqcup_{x \in X} f\ x & \text{if } m = \textbf{res } X \end{cases}
\end{aligned}$$

- refinement: $f\ x \sqsubseteq f'\ x$
  - intuition: possible results of $f\ x$ are also results of $f'\ x$
- programs are HOL-functions $f : a \to r$ nres
- least, greatest fixed point for recursion
- usual programming constructs available

Hopcroft's Algorithm

- Hopcroft's algorithm minimises initially connected DFAs
- efficient and widely-used
- computes the Myhill-Nerode equivalence relation $\{\{q' \mid q' \text{ accepts same language as } q\} \mid q \in \mathcal{Q}\}$
- details unimportant, let's focus on refinement

Abstract Algorithm

Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L) =$
  spec $(\mathcal{P}', L')$. $\mathcal{P}' = \text{Split}_{\mathcal{A}}(\mathcal{P}, (a, C_s)) \ \wedge \ \text{splitter\_P}_{\mathcal{A}}(\mathcal{P}, (a, C_s), L, L')$;

Hopcroft_abstract$(\mathcal{A}) =$
  if $(\mathcal{Q} = \emptyset)$ **then return** $\emptyset$ **else if** $(\mathcal{F} = \emptyset)$ **then return** $\{\mathcal{Q}\}$ **else**
  while$_{\mathbf{T}}^{\text{Hopcroft\_abstract\_invar}}(\lambda(\mathcal{P}, L).\ L \neq \emptyset)\ (\lambda(\mathcal{P}, L).$ **do** $\{$
    $(a, C_s) \leftarrow$ **spec** $x.\ x \in L$;
    $(\mathcal{P}', L') \leftarrow$ Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L)$;
    return $(\mathcal{P}', L')$
  $\})\ (\text{part}_{\mathcal{F}}, \{(a, \mathcal{F}) \mid a \in \Sigma\})$

Hopcroft's Algorithm

- Hopcroft's algorithm minimises initially connected DFAs
- efficient and widely-used
- computes the Myhill-Nerode equivalence relation $\{\{q' \mid q' \text{ accepts same language as } q\} \mid q \in \mathcal{Q}\}$
- do-notation of monads allows readable programs

Abstract Algorithm

Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L) =$
  spec $(\mathcal{P}', L')$. $\mathcal{P}' = \text{Split}_{\mathcal{A}}(\mathcal{P}, (a, C_s)) \ \wedge \ \text{splitter\_P}_{\mathcal{A}}(\mathcal{P}, (a, C_s), L, L')$;

Hopcroft_abstract$(\mathcal{A}) =$
  if $(\mathcal{Q} = \emptyset)$ **then return** $\emptyset$ **else if** $(\mathcal{F} = \emptyset)$ **then return** $\{\mathcal{Q}\}$ **else**
  $\textbf{while}_{\mathbf{T}}^{\text{Hopcroft\_abstract\_invar}}(\lambda(\mathcal{P}, L).\ L \neq \emptyset)\ (\lambda(\mathcal{P}, L).$ **do** {
    $(a, C_s) \leftarrow$ **spec** $x.\ x \in L$;
    $(\mathcal{P}', L') \leftarrow$ Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L)$;
    return $(\mathcal{P}', L')$
  }) $(\text{part}_{\mathcal{F}}, \{(a, \mathcal{F}) \mid a \in \Sigma\})$

Hopcroft's Algorithm

- Hopcroft's algorithm minimises initially connected DFAs
- efficient and widely-used
- computes the Myhill-Nerode equivalence relation $\{\{q' \mid q' \text{ accepts same language as } q\} \mid q \in \mathcal{Q}\}$
- nondeterminism easy to use

Abstract Algorithm

Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L) =$
  spec $(\mathcal{P}', L').\ \mathcal{P}' = \text{Split}_{\mathcal{A}}(\mathcal{P}, (a, C_s))\ \wedge\ \text{splitter\_P}_{\mathcal{A}}(\mathcal{P}, (a, C_s), L, L')$;

Hopcroft_abstract$(\mathcal{A}) =$
  if $(\mathcal{Q} = \emptyset)$ **then return** $\emptyset$ **else if** $(\mathcal{F} = \emptyset)$ **then return** $\{\mathcal{Q}\}$ **else**
  while$_{\mathbf{T}}^{\text{Hopcroft\_abstract\_invar}}(\lambda(\mathcal{P}, L).\ L \neq \emptyset)\ (\lambda(\mathcal{P}, L).$ **do** $\{$
    $(a, C_s) \leftarrow$ **spec** $x.\ x \in L$;
    $(\mathcal{P}', L') \leftarrow \text{Hopcroft\_step\_abstract}(\mathcal{A}, a, C_s, \mathcal{P}, L)$;
    return $(\mathcal{P}', L')$
  $\})\ (\text{part}_{\mathcal{F}}, \{(a, \mathcal{F}) \mid a \in \Sigma\})$

Hopcroft's Algorithm

- Hopcroft's algorithm minimises initially connected DFAs
- efficient and widely-used
- computes the Myhill-Nerode equivalence relation $\{\{q' \mid q' \text{ accepts same language as } q\} \mid q \in \mathcal{Q}\}$
- constructs like if-then-else or while available

Abstract Algorithm

Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L) =$
  spec $(\mathcal{P}', L')$. $\mathcal{P}' = \text{Split}_{\mathcal{A}}(\mathcal{P}, (a, C_s)) \ \wedge \ \text{splitter\_P}_{\mathcal{A}}(\mathcal{P}, (a, C_s), L, L')$;

Hopcroft_abstract$(\mathcal{A}) =$
  if $(\mathcal{Q} = \emptyset)$ **then return** $\emptyset$ **else if** $(\mathcal{F} = \emptyset)$ **then return** $\{\mathcal{Q}\}$ **else**
  while$_{\mathbf{T}}^{\text{Hopcroft\_abstract\_invar}}(\lambda(\mathcal{P}, L).\ L \neq \emptyset)\ (\lambda(\mathcal{P}, L).$ **do** {
    $(a, C_s) \leftarrow$ **spec** $x.\ x \in L$;
    $(\mathcal{P}', L') \leftarrow$ Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L)$;
    return $(\mathcal{P}', L')$
  }) $(\text{part}_{\mathcal{F}}, \{(a, \mathcal{F}) \mid a \in \Sigma\})$

Hopcroft's Algorithm

- Hopcroft's algorithm minimises initially connected DFAs
- efficient and widely-used
- computes the Myhill-Nerode equivalence relation $\{\{q' \mid q' \text{ accepts same language as } q\} \mid q \in \mathcal{Q}\}$
- shallow embedding handy

Abstract Algorithm

Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L) =$
  spec $(\mathcal{P}', L')$. $\mathcal{P}' = \text{Split}_{\mathcal{A}}(\mathcal{P}, (a, C_s)) \ \wedge \ \text{splitter\_P}_{\mathcal{A}}(\mathcal{P}, (a, C_s), L, L')$;

Hopcroft_abstract$(\mathcal{A}) =$
  if $(\mathcal{Q} = \emptyset)$ **then return** $\emptyset$ **else if** $(\mathcal{F} = \emptyset)$ **then return** $\{\mathcal{Q}\}$ **else**
  while$_{\mathbf{T}}^{\text{Hopcroft\_abstract\_invar}}(\lambda(\mathcal{P}, L).\ L \neq \emptyset)\ (\lambda(\mathcal{P}, L).$ **do** $\{$
    $(a, C_s) \leftarrow$ **spec** $x.\ x \in L$;
    $(\mathcal{P}', L') \leftarrow$ Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L)$;
    return $(\mathcal{P}', L')$
  $\})\ (\text{part}_{\mathcal{F}}, \{(a, \mathcal{F}) \mid a \in \Sigma\})$

Correctness

- correctness expressed by refinement

$$\forall \mathcal{A}.\ \text{DFA}\ \mathcal{A} \implies \text{Hopcroft\_abstract}(\mathcal{A}) \sqsubseteq \textbf{spec}\ \mathcal{P}.\ \mathcal{P} = \text{Myhill\_Nerode\_partition}\ \mathcal{A}$$

- refinement framework provides syntax driven **verification condition generator**
- proof can focus on algorithm, not refinement or representation
- however, correctness proof non-trivial

Refinement Steps

- step 1: implement **Hopcroft_step_abstract** with foreach loop

  simple preconditions $\Longrightarrow$ Hopcroft_step_set$(\mathcal{A}, a, C_s, \mathcal{P}, L) \sqsubseteq$
  Hopcroft_step_abstract$(\mathcal{A}, a, C_s, \mathcal{P}, L)$

- **Hopcroft_step_set** very similar to presentation in literature
- step 2: optimise loop by precomputing predecessors

  simple preconditions $\Longrightarrow$ Hopcroft_step_pre$(\mathcal{A}, a, C_s, \mathcal{P}, L) \sqsubseteq$
  Hopcroft_step_set$(\mathcal{A}, a, C_s, \mathcal{P}, L)$

- again, good tool support by the refinement framework
- **assert** very useful

Data Refinement

- implementing partitions as sets of sets is inefficient
- lets use datastructure based on finite maps instead
- **data refinement** needed
- more general: replace **abstract** data type by **concrete** one
  - e. g. implement sets by red-black trees
- abstraction relation $R = \{(c, a) \mid a = \alpha_R\ c \wedge I_R\ c\}$
- **concretisation** function
  - $\Downarrow R : a\ \text{nres} \to c\ \text{nres}$
  - transitive: $m \sqsubseteq \Downarrow R\ m' \wedge m' \sqsubseteq \Downarrow S\ m'' \implies m \sqsubseteq \Downarrow RS\ m''$

Refinement Steps II

- step 3: **implement partitions** by finite maps

$$\forall \mathcal{A}.\ \text{DFA}\ \mathcal{A} \implies \begin{array}{l}\text{Hopcroft\_map}(\mathcal{A}) \sqsubseteq \\ \Downarrow R_1\ \text{Hopcroft\_abstract}(\mathcal{A})\end{array}$$

- step 4: **implement classes** with intervals of natural numbers

$$\forall \mathcal{A}.\ \text{DFA}\ \mathcal{A} \implies \begin{array}{l}\text{Hopcroft\_map2}(\mathcal{A}) \sqsubseteq \\ \Downarrow R_2\ \text{Hopcroft\_map}(\mathcal{A})\end{array}$$

- these refinement steps are non-trivial
- **verification condition generator** for data refinements
- proofs focus on essence of problem

Code Generation

- step 5: use **Isabelle Collection Framework** (ICF)
  - implement finite maps by red-black trees or arrays
  - implement sets by red-black trees or sorted lists
  - ...

$$\forall \mathcal{A}.\ \mathrm{DFA}\ \mathcal{A} \Longrightarrow \mathrm{Hopcroft\_impl}(\mathcal{A}) \sqsubseteq \Downarrow R_3\ \mathrm{Hopcroft\_map2}(\mathcal{A})$$

- step 6: bring into special form for code generation

$$\forall \mathcal{A}.\ \mathrm{DFA}\ \mathcal{A} \Longrightarrow \textbf{return}\ \mathrm{Hopcroft\_code}(\mathcal{A}) \sqsubseteq \mathrm{Hopcroft\_impl}(\mathcal{A})$$

- there is good tool support
- step 6 is nearly fully automatic

Code Generation II

- transitivity leads to

$$\forall \mathcal{A}.\ \text{DFA}\ \mathcal{A} \implies \begin{array}{l} \textbf{return}\ \text{Hopcroft\_code}(\mathcal{A}) \sqsubseteq \Downarrow R_1 R_2 R_3 \\ \textbf{spec}\ \mathcal{P}.\ \mathcal{P} = \text{Myhill\_Nerode\_partition}\ \mathcal{A} \end{array}$$

- Isabelle/HOL allows code generation in functional languages
  - Standard ML
  - OCaml
  - Haskell
  - Scala
  - ...

Experimental Results

No. DFAs	*No. states*	*No. labels*	*Baclet/Pagetti OCaml*	*Lammich/Tuerk OCaml*	*Lammich/Tuerk PolyML*	*Leiß PolyML*
10000	50	2	0.17 s	6.59 s	1.88 s	5.38 s
10000	50	5	0.27 s	12.62 s	3.51 s	19.34 s
10000	100	2	0.31 s	14.31 s	3.97 s	16.41 s
10000	100	5	0.51 s	26.13 s	7.56 s	63.21 s
10000	250	2	0.69 s	41.02 s	11.09 s	83.62 s
1000	1000	2	0.51 s	18.61 s	5.37 s	134.21 s
1000	2500	2	1.44 s	51.35 s	17.88 s	905.82 s

Experimental Results (measured on an Intel Core I7 2720QM)

Conclusion

- refinement framework for monadic programs
  - based on refinement calculus
  - implemented in Isabelle/HOL
  - available in *Archive of Formal Proofs* (`http://afp.sf.net`)
- case study of Hopcroft's algorithm
  - first formalisation
  - efficient version not feasible without refinement
  - available at `http://cava.in.tum.de`

Other Applications

- BFS, DFS graph traversals [Lammich]
- Dijkstra's shortest paths algorithm [Nordhoff, Lammich]
- nested DFS (Büchi automata acceptance) [Neumann]
- Henzinger's Algorithm (simulation preorders for NFAs) [Eberl]
- Gerth's algorithm (LTL to Büchi automata), work in progress [Schimpf]
- saturation algorithm for pre* of PDS/DPN, work in progress [Lammich]

Current / Future Work

- additional automation
- support for complexity proofs
- heap monads with separation logic