A Formalisation of Finite Automata in Isabelle / HOL

Thomas Tuerk

29th February 2012

Computer **A**ided **V**erification of **A**utomata (CAVA)

- CAVA is a DFG funded project
- goals:
  - formalise algorithmic parts of automata theory
  - formalise applications in model checking
  - high level of abstraction
  - generation of reference implementations
- members
  - Javier Esparza / René Neumann
  - Tobias Nipkow / Thomas Tuerk / (Peter Lammich)
  - Jan-Georg Smaus / Alexander Schimpf
- in this talk: finite automata on finite words

Outline

Nondeterministic Finite Automata

- **nondeterministic finite automata** (NFAs) are quintuples $(\mathcal{Q}, \Sigma, \Delta, \mathcal{I}, \mathcal{F})$, where
  - $\mathcal{Q}$ :: *state set* — set of states
  - Σ :: *label set* — set of labels
  - Δ :: *(state* $\times$ *label* $\times$ *state) set* — transition relation
  - $\mathcal{I}$ :: *state set* — set of initial states
  - $\mathcal{F}$ :: *state set* — set of accepting states
- additionally **well-formedness** conditions
  - finite $\mathcal{Q}$
  - finite Σ
  - $\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$
  - $\mathcal{I} \subseteq \mathcal{Q}$
  - $\mathcal{F} \subseteq \mathcal{Q}$

Deterministic Finite Automata

- **deterministic finite automata** (DFAs) are NFAs with
  - $|\mathcal{I}| = 1$
  - $\forall q \in \mathcal{Q},\ a \in \Sigma.\ \exists! q' \in \mathcal{Q}.\ (q, a, q') \in \Delta$
- Notice: in this formalisation all DFAs are **complete**
- Δ can be represented as a function
  $\delta :: \mathit{states} \times \mathit{labels} \to \mathit{states}$

Isabelle/HOL Representation

- high-level datastructure for NFAs
  - sets and relations
  - record structure
  - well-formedness predicates
  - locales for NFAs and DFAs
- low-level, executable representation
  - *Isabelle Collection Framework* (ICF) by Peter Lammich
  - set and finite map implementations
  - e. g. red-black trees, arrays, lists

What is there?

- Operations
  - acceptance / language of automaton
  - Boolean operations
  - determinisation
  - minimisation
  - ...
- Concepts
  - isomorphy
  - initially connected automata
  - efficient constructions
  - reversal / left and right languages
  - ...
- Infrastructure
  - construction from lists / destruction
  - interface in SML and OCaml
  - connection with Graphviz
  - ...

Example: Product Automata

Product Automata

$$
\begin{aligned}
&\mathit{product\_NFA}\ \mathcal{A}_1\ \mathcal{A}_2 = (|\\
&\quad \mathcal{Q} = \mathcal{Q}(\mathcal{A}_1) \times \mathcal{Q}(\mathcal{A}_2),\\
&\quad \Sigma = \Sigma(\mathcal{A}_1) \cap \Sigma(\mathcal{A}_2),\\
&\quad \Delta = \{((q_1, q_2), a, (q_1', q_2')) \mid \ (q_1, a, q_1') \in \Delta(\mathcal{A}_1) \wedge\\
&\qquad\qquad\qquad\qquad\quad (q_2, a, q_2') \in \Delta(\mathcal{A}_2)\},\\
&\quad \mathcal{I} = \mathcal{I}(\mathcal{A}_1) \times \mathcal{I}(\mathcal{A}_2),\\
&\quad \mathcal{F} = \mathcal{F}(\mathcal{A}_1) \times \mathcal{F}(\mathcal{A}_2)\\
&|)
\end{aligned}
$$

- clear definition / simple proofs
- **Problem: type of states changes, composition tricky**

  $\mathit{product\_NFA} :: ('q_1, 'a)\mathit{NFA} \Rightarrow ('q_2, 'a)\mathit{NFA} \Rightarrow ('q_1 \times 'q_2, 'a)\mathit{NFA}$
- **Problem: too many states**

Renaming States / Isomorphic Automata

- solving the state-type problem requires renaming
- renaming leads to equivalent automaton, provided only equivalent states are identified
- two corner cases
  - injective functions lead to **isomorphic** automaton
  - identifying all equivalent states useful for minimisation
- isomorphy is a key concept of the library
- implementations operate modulo isomorphy
- state set is finite
- type-class **NFA_states** guarantees infinite universe
- therefore, safe renaming into **NFA_states** possible

Unreachable states

- **unreachable** states are states that cannot be reached from an initial state
- removing unreachable states leads to an equivalent automaton
- such an automaton is called **initially connected**
- not considering unreachable states is essential for performance

Reconsider Product Automata

- *product_NFA* $\mathcal{A}_1$ $\mathcal{A}_2$
  - original definition
  - intuitive definition / straightforward proofs
- *efficient_product_NFA* $\mathcal{A}_1$ $\mathcal{A}_2$:= *NFA_remove_unreachable_states*(*product_NFA* $\mathcal{A}_1$ $\mathcal{A}_2$)
  - consider only reachable states
  - essential for efficiency
- *NFA_product* $\mathcal{A}_1$ $\mathcal{A}_2$:= *NFA_normalise_states*(*efficient_product_NFA* $\mathcal{A}_1$ $\mathcal{A}_2$)
  - allows composing
- locales provide implementation interface for *NFA_product* and *efficient_product_NFA* modulo isomorphy

Efficient construction

NFA_construct_reachable (I, Σ, FP, Δ) =
  let rm be an injective function on I;
  initialise $\mathcal{A}$ with ($\emptyset$, Σ, $\emptyset$, image rm I, $\emptyset$) and ws with I;
  while $ws \neq \emptyset$ **do**
    choose $q \in ws$;
    if $rm\ q \notin \mathcal{Q}(\mathcal{A})$ **then**
      $N := \{q' \mid \exists a \in \Sigma.\ (q, a, q') \in \Delta\}$;
      extend rm to be still injective and defined on N;
      $\mathcal{A} :=$ ($\{rm\ q\} \cup \mathcal{Q}(\mathcal{A})$, Σ,
        $\Delta(\mathcal{A}) \cup \{(rm\ q,\ a,\ rm\ q') \mid a \in \Sigma\ \wedge (q, a, q') \in \Delta\}$,
        $\mathcal{I}(\mathcal{A})$, if $FP\ q$ then $\{rm\ q\} \cup \mathcal{F}(\mathcal{A})$ else $\mathcal{F}(\mathcal{A})$);
      $ws := N \cup ws$;
    end
  end
  return $\mathcal{A}$

- this constructs only reachable states and renames them
- it provides a simple interface

  well-formed $\mathcal{A} \wedge (\forall q \in \mathcal{Q}(\mathcal{A}).\ FP\ q \Leftrightarrow q \in \mathcal{F}(\mathcal{A})) \Longrightarrow$
  NFA_construct_reachable($I(\mathcal{A}), \Sigma(\mathcal{A}), FP, \Delta(\mathcal{A})$) is isomorphic to
  NFA_remove_unreachable_states($\mathcal{A}$)

Conclusion

Summary

- I implemented a automaton library in Isabelle/HOL
- Boolean operations, determinisation, minimisation, ...
- abstract definitions and lemmata
- efficient implementations and simple interface available

Future Work

- automata with ϵ-transitions
- connection to regular expressions

Questions?

Presburger Arithmetic Library

- there is a Presburger Arithmetic Library by Stefan Berghofer and Markus Reiter
- available in the *Archive of Formal Proofs*
- presented at TPHOLs 2009
- they implement an automata based decision procedure
- they formalised automata on bitstrings
  - efficiently executable
  - tailored towards the application
  - little abstraction

Case Study

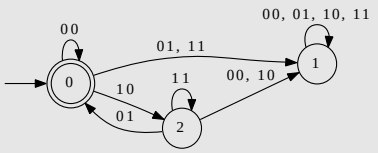
Implement automata using my library.

Required Automata Constructs

- **automata for diophantine (in)equations**
- Boolean combinations
- **existential quantification on automata**
  - projection
  - determinisation
  - (minimisation)
  - **right quotient**

Diophantine (In)equantions

Automaton for $2x - y = 0$



Example

x = 3 = 110000...
y = 6 = 011000...

- about 500 lines in Presburger Library
- mixed with low level definitions
- about 350 (+70) lines on my abstract level
- straightforward mapping

Existential Quantification

- existential quantification combines
  - projection (remove head of label)
  - right quotient
  - determinisation
  - (minimisation)
- implementation right quotient straightforward (about 40 lines)
- with interfaces about 200 lines
- correctness proof tricky (about 270 lines)
- about 200 lines for right quantification in original work

Combination

- straightforward
- but lengthy
- about 100 lines in my and the original work

Experimental Results

Example	*Berghofer/Reiter*	*Tuerk*
stamp	0.01 s	0.01 s
example	0.10 s	0.01 s
example2	0.13 s	0.03 s
harrison1	0.03 s	0.01 s
harrison2	0.15 s	0.02 s
stamp-false	0.01 s	0.01 s
example2-false	0.46 s	0.04 s

speed of generated OCaml code on an Intel Core I7 2720QM

Summary

- straightforward to use my automata library
- about 2 days of work involved
- higher level of abstraction
- more efficient execution
- about 1200 lines in my adaption
- about 4500 lines in original library

Questions?

Motivation

- minimisation is an important operation
- Hopcroft's Algorithm has the best know behaviour in practise
- implemented using Peter Lammich's refinement framework
- good case study for refinement
- here: focus on refinement

General Idea

- Hopcroft's algorithm minimises initially connected DFAs
- it computes the Myhill-Nerode equivalence relation

$$\{\{q' \mid q' \text{ is equivalent to } q\} \mid q \in \mathcal{Q}\}$$

- this equivalence relation is used to rename states

General Idea

initialise $\mathcal{P}$ with $\text{part}_{\mathcal{F}}$;
while *there are* $C \in \mathcal{P}$ *and* $(a, C_s) \in \Sigma \times \mathcal{P}$ *with* $\text{splittable}_{\mathcal{A}}(C, (a, C_s))$ **do**
  choose such C and (a, C_s);
  update $\mathcal{P}$ by removing C and adding the two results of $\text{split}_{\mathcal{A}}(C, (a, C_s))$;
end
return $\mathcal{P}$;

$$\text{split}_{\mathcal{A}}(C, (a, C_s)) := (\{q \mid q \in C \wedge \delta(q, a) \in C_s\},\ \{q \mid q \in C \wedge \delta(q, a) \notin C_s\})$$
$$\text{splittable}_{\mathcal{A}}(C, (a, C_s)) := \text{let } (C_t, C_f) = \text{split}_{\mathcal{A}}(C, (a, C_s)) \text{ in } C_t \neq \emptyset \wedge C_f \neq \emptyset$$
$$\text{part}_{\mathcal{F}} := \{\mathcal{F}, \mathcal{Q} - \mathcal{F}\} - \{\emptyset\}$$

Abstract Algorithm

Idea: maintain explicit list of splitters

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

The proof of correctness needs about 2600 lines.

Set Implementation

Idea: implement the step with a foreach loop

Set Implementation

Hopcroft_step_set$(\mathcal{A}, a, C_s, \mathcal{P}, L) =$
  $\mathcal{P}' \leftarrow$ **spec** $\mathcal{P}'.\ \mathcal{P}' \subseteq \mathcal{P} \ \wedge \ (\forall C \in \mathcal{P}.\ \text{splittable}_{\mathcal{A}}(C, (a, C_s)) \Rightarrow C \in \mathcal{P}')$;
  $(\mathcal{P}', L') \leftarrow \textbf{foreach}^{\text{Hopcroft\_set\_invar}}\ \mathcal{P}'\ (\lambda C\ (\mathcal{P}', L').\ \{$
    let $(C_t, C_f) = \text{split}_{\mathcal{A}}(C, (a, C_s))$;
    if $(C_t = \emptyset \vee C_f = \emptyset)$ **then return** $(\mathcal{P}', L')$ **else** $\{$
      $(C_1, C_2) \leftarrow$ **spec** $x.\ x \in \{(C_f, C_t),\ (C_t, C_f)\}$;
      let $\mathcal{P}' = (\mathcal{P}' - \{C\}) \cup \{C_1, C_2\}$;
      let $L' = (L' - \{(a, C) \mid a \in \Sigma\}) \ \cup \ \{(a, C_1) \mid a \in \Sigma\} \ \cup \ \{(a, C_2) \mid (a, C) \in L'\}$;
      return $(\mathcal{P}', L')$;
    $\}$
  $\})\ (\mathcal{P}, L - \{(a, C_s)\})$;
  return $(\mathcal{P}', L')$;

Refinement needs about 400 lines.
This is the level of abstraction usually used in literature.

Precomputing Predecessors

Precomputing Predecessors

Hopcroft_step_pre$(\mathcal{A}, a, C_s, \mathcal{P}, L) =$
  let $pre = \{q \mid \delta(q, a) \in C_s\}$;
  $\mathcal{P}' \leftarrow$ **spec** $\mathcal{P}'$. $\mathcal{P}' \subseteq \mathcal{P} \ \wedge (\forall C \in \mathcal{P}'.\ C \cap pre \neq \emptyset) \wedge$
    $(\forall C \in \mathcal{P}.\ C \cap pre \neq \emptyset \wedge \text{splittable}_{\mathcal{A}}(C, (a, C_s)) \Rightarrow C \in \mathcal{P}')$;
  $(\mathcal{P}', L') \leftarrow$ **foreach**$^{\text{Hopcroft\_set\_invar}}$ $\mathcal{P}'$ $(\lambda C\ (\mathcal{P}', L')$. {
    let $(C_t, C_f) = (\{q \mid q \in C \wedge q \in pre\}, \{q \mid q \in C \wedge q \notin pre\})$;
    if $(C_f = \emptyset)$ **then return** $(\mathcal{P}', L')$ **else** {
      let $(C_{\min}, C_{\max}) =$ **if** $(|C_f| < |C_t|)$ **then** (C_f, C_t) **else** (C_t, C_f);
      let $\mathcal{P}' = (\mathcal{P}' - \{C\}) \cup \{C_{\min}, C_{\max}\}$;
      let $L' = \ (L' - \{(a, C) \mid a \in \Sigma\}) \cup$
        $\{(a, C_{\min} \mid a \in \Sigma\} \ \cup \ \{(a, C_{\max} \mid (a, C) \in L'\}$
      return $(\mathcal{P}', L')$;
    }
  }) $(\mathcal{P}, L - \{(a, C_s)\})$;
  return $(\mathcal{P}', L')$;

Refinement needs about 50 lines.

Data Refinement Partitions

- representing partitions as sets of sets is inefficient
- use pair of finite maps instead
  - *im* - finite map from index to classes
  - *sm* - finite map from states to index of class it belongs to
  - *n* - maximal index

Abstraction Function and Invariant

partition_map_invar$(im, sm, n) :=$
  $\text{dom}(im) = \{i \mid 0 \leq i < n\} \ \wedge \ (\forall 0 \leq i < n.\ im(i) \neq \emptyset) \wedge$
  $(\forall q.\ sm(q) = i \Leftrightarrow (0 \leq i < n \wedge q \in im(i)))$

partition_map_$\alpha(im, sm, n) := \{im(i) \mid 0 \leq i < n\}$

- when splitting classes, the old index is reused for $C_{\max}$
- update of L becomes much simpler

Data Refinement Partitions II

Data Refinement Partitions

Hopcroft_step_map$(\mathcal{A}, a, i_s, (im, sm, n), L) =$
  let $pre = \{q \mid \delta(q, a) \in im(i_s)\}$;
  let $\mathcal{I} = \{sm(q) \mid q \in pre\}$;
  $((im', sm', n'), L') \leftarrow$ **foreach**$^{\text{Hopcroft\_map\_invar}}$ $\mathcal{I}$ $(\lambda i\ ((im', sm', n'), L')$. {
    let $(C_t, C_f) = (\{q \mid q \in im(i) \wedge q \in pre\}, \{q \mid q \in im(i) \wedge q \notin pre\})$;
    if $(C_f = \emptyset)$ **then return** $((im', sm', n'), L')$ **else** {
      let $(C_{\min}, C_{\max}) =$ **if** $(|C_f| < |C_t|)$ **then** (C_f, C_t) **else** (C_t, C_f);
      let $(im', sm', n') = (\ im'(i \mapsto C_{\max}, n \mapsto C_{\min}),$
        $\lambda q.\ \textit{if } q \in C_{\min} \textit{ then } n \textit{ else } sm'(q),\ n' + 1)$;
      let $L' = \{(a, n) \mid a \in \Sigma\} \ \cup \ L'$;
      return $((im', sm', n'), L')$;
    }
  }) $((im, sm, n), L - \{(a, i_s)\})$;
  return $((im', sm', n'), L')$;

Refinement needs about 1100 lines.

Data Refinement Classes

- represent classes by interval of natural numbers
- use pair of finite maps for mapping between classes and natural numbers
  - *pm* - finite map from states to natural numbers
  - *pim* - inverse map

Abstraction Function and Invariant

class_map_invar Q $(pm, pim) :=$
  $\text{dom}(pm) = Q \ \wedge \ \text{dom}(pim) = \{i \mid i < |Q|\} \wedge$
  $(\forall q\ i.\ pm(q) = i \Leftrightarrow pim(i) = q)$

class_map_$\alpha(pm, pim)(l, u) := \{pim(i) \mid l \leq i \leq u\}$

Data Refinement Classes II

Data Refinement Classes

Hopcroft_step_map2$(\mathcal{A}, a, i_s, (im, sm, n), L, pm, pim) =$
  let $pre = \{q \mid \delta(q, a) \in im(i_s)\}$;
  let $(iM, (pm, pim)) =$
    Hopcroft_step_map2_compute_iM$(\mathcal{A}, (pm, pim), pre, (im, sm, n))$;
  $((im', sm', n'), L') \leftarrow$ **foreach**$^{\text{Hopcroft\_map2\_invar}}$ $\{(i, s) \mid iM(i) = s\}$
  $(\lambda(i, s)\ ((im', sm', n'), L').\ \{$
    let $(l, u) = im(i)$;
    let $(C_t, C_f) = ((l, s-1), (s, u))$;
    if $((u+1) - s = 0)$ **then return** $((im', sm', n'), L')$ **else** $\{$
      let $(C_{\min}, C_{\max}) =$ **if** $((u+1) - s < s - l)$ **then** (C_f, C_t) **else** (C_t, C_f);
      let $(im', sm', n') =$
        $(\ im'(i \mapsto C_{\max}, n \mapsto C_{\min}),$
        $\lambda q.\ \textit{if } q \in \text{class\_map\_}\alpha(pm, pim)(C_{\min})\ \textit{then } n \textit{ else } sm'(q),\ \ n' + 1)$;
      let $L' = \{(a, n) \mid a \in \Sigma\} \cup L'$;
      return $((im', sm', n'), L')$;
    $\}$
  $\})\ ((im, sm, n), L - \{(a, i_s)\})$;
  return $(((im', sm', n'), L'), (pm, pim))$;

Data Refinement Classes III

Data Refinement Classes

Hopcroft_step_map2_compute_iM$(\mathcal{A}, (pm, pim), pre, (im, sm, n)) =$
  foreach pre $(\lambda q\ (iM, (pm, pim)).\ \{$
    let $i =\ im(q)$;
    let $s =$ if defined$(iM(q))$ then $iM(q)$ else fst$(im(i))$;
    let $iq = pm(q)$;
    if $iq = s$ **then**
      return$(iM(i \mapsto s+1), (pm, pim))$;
    else
      let $qs = pim(s)$;
      let $pm = pm(q \mapsto s)(qs \mapsto iq)$;
      let $pim = pim(s \mapsto q)(iq \mapsto qs)$;
      return$(iM(i \mapsto s+1), (pm, pim))$;
    end
  $\}$
$\})\ (\emptyset, (pm, pim))$;

Refinement needs about 1600 lines.

Collection Framework

- the next refinement step introduces executable datastructures
- Peter Lammich's **Collection Framework** used
- red-black-trees and arrays chosen as data-structures
- 800 lines needed
- further 380 lines for connection with automata implementation (computing *pre*)

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	7.87 s	1.91 s	5.38 s
10000	50	5	0.28 s	15.03 s	3.67 s	19.34 s
10000	100	2	0.30 s	16.21 s	4.06 s	16.41 s
10000	100	5	0.53 s	30.79 s	7.86 s	63.21 s
10000	250	2	0.76 s	46.53 s	11.48 s	83.62 s
1000	1000	2	0.44 s	20.95 s	5.67 s	134.21 s
1000	2500	2	1.44 s	58.24 s	17.83 s	905.82 s

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

Conclusion

- refinement worked well for Hopcroft's algorithm
- there is a non-trivial, efficient implementation
- experimental result show decent performance

Questions?

Conclusion

- I implemented a library for finite automata in Isabelle/HOL
- this library provides abstract concepts for reasoning about automata
- there are efficient implementations of non-trivial algorithms
- the library can easily be used by other projects
- generated code in SML and OCaml exists