Implementierung und Verifikation von Prozessen mittels Komusterabgleich

Andreas Abel

Department of Computer Science and Engineering
Chalmers and Universität Göteborg

Ein Institut für Informatik einer deutschen Universität
Herbst 2015

Inhalt

Formale Verifikation

- Formale Verifikation ist der Ausschluß von Fehlern mittels rechnergestützter mathematischer Beweisführung.
- Angewendet auf: Hardware, Software, mathematische Theoreme.
- Breites Spektrum zwischen
  - vollautomatischer Verifikation entscheidbarer Eigenschaften und
  - manueller Verifikation; Rechner prüft nur Korrektheit.
- Wo Fehler sehr kostspielig sind, ist Verifikation rentabel:
  - CPUs (Pentium-Bug!)
  - Betriebssysteme (Sicherheit!)
  - Cyber-physical systems (Automotive)

Softwareverifikation

- Ansätze in der Softwareverifikation:
  - *post mortem*: Separate Verifikation fertiger Software.
    (Wie behandelt man Änderungen?)
  - *integriert*: Verifikation ist Teil der Softwareentwicklung
    (*Correct-by-construction*).
    Verifikationsbemühungen prägen Struktur der Software.
- Auch Teilverifikation nützlich, z.B. Verifikation von Datenstrukturinvarianten.
  Z.B. *Baum ist Suchbaum*, *Baum ist balanciert.*

Verifikation durch Typsysteme

- Traditionelle Typsysteme können gewisse Fehler ausschließen:
  - Illegale Operationen wie `5/"bla"`.
  - Dereferenzierung eines Null-Zeigers (Scala/Haskell).
- . . . und gewisse Invarianten ausdrücken.
  - Referenz zeigt auf einen Binärbaum.
- *Abhängige* Typen können beliebige Invarianten ausdrücken.
  - Referenz zeigt auf balancierten Suchbaum.
  - Wert ist nicht Null (Ausschluß von Division durch Null).
- Erfordern vom Benutzer Typ-Annotationen und Beweise.

Große Verifikationsprojekte

- Micro-Kernel in HOL (Klein, NICTA, Australien)
- Teil-C-Kompiler in Coq (Leroy, INRIA, Frankreich)
  (Tony Hoare Grand Challenge)
- Vierfarbensatz in Coq (Gonthier, INRIA)

Entwicklung korrekter Programme mit Agda

- Agda ist eine abhängig-getypte funktionale Programmiersprache.
- Prototyp, open-source Entwicklung.
- Wachsende Benutzergemeinde (> 100 Forschungsartikel verw. Agda)
- Basiert auf der Typentheorie nach Per Martin-Löf.
- Implementiert den Curry-Howard-Isomorphismus.

  Aussagen sind Typen
  Beweise sind Programme
- *Eine* Sprache für Spezifikationen, Programme, und Beweise.
- Programmierung durch Rekursion und Musterabgleich mit benutzerdefinierten Datentypen und Prädikaten.

Induktive Typen und Musterabgleich

- Bool ist die kleinste Menge, die true und false enthält.

```
data Bool : Set where
  true  : Bool
  false : Bool
```

- Um eine Funktion über Bool zu definieren, müssen genau die Fälle true und false behandelt werden:

```
if_then_else_ : ∀{A : Set} (cond? : Bool) (then else : A) → A
if true  then then else else = then
if false then then else else = else
```

Induktive Listen

- Listen bilden die kleinste Menge abgeschlossen unter [] und _::_.
  - [] ist eine Liste über A.
  - Ist $a \in A$ und as eine Liste über A, so ist $(a :: as)$ eine Liste über A.

```
data List A : Set where
  []    : List A
  _::_  : (a : A) (as : List A) → List A
```

- Listenfunktion defininiert durch die Muster [] und $(a :: as)$.
  Rekursiver Aufruf mit as.

```
filter : ∀{A} (p : A → Bool) (as : List A) → List A
filter p []  =  []
filter p (a :: as) with p a
... | true  =  a :: filter p as
... | false =  filter p as
```

Induktive Prädikate

- Unter dem Curry-Howard-Isomorphismus ist eine Aussage die Menge ihrer Beweise.
- Eine induktive Aussage wie *as ist Teilfolge von bs* wird durch ihre Beweiskonstruktoren beschrieben.

$$
\begin{array}{lll}
\textsf{data}\ \_\subset\_\ \{A\} : (as\ bs : \textsf{List}\ A) \to \textsf{Set}\ \textsf{where} & & \\
\quad \textsf{empty} & : \forall\{bs\} & \to [] \subset bs \\
\quad \textsf{cons} & : \forall\{as\ bs\ a\} \to as \subset bs & \to (a :: as) \subset (a :: bs) \\
\quad \textsf{skip} & : \forall\{as\ bs\ a\} \to as \subset bs & \to as \subset (a :: bs)
\end{array}
$$

- empty: Die leere Liste [] is Teilfolge jeder Liste *bs*.
- cons: Ist *as* Teilfolge von *bs*, so auch $(a :: as)$ von $(a :: bs)$ für jedes *a*.
- skip: Ist *as* Teilfolge von *bs*, so auch von $(a :: bs)$ für jedes *a*.

Induktive Beweise

- Ein Beweis einer induktiven Aussage ist ein terminierendes funktionales Programm, dass eine Herleitung der Aussage konstruiert.
- Z.B. beweisen wir, dass jede Anwendung von filter eine Teilfolge der ursprünglichen Liste erzeugt: filter p $as \subset as$

```
filter-sub : ∀{A} (p : A → Bool) (as : List A) → filter p as ⊂ as
filter-sub p []  =  empty
filter-sub p (a :: as) with p a
... | true  =  cons  (filter-sub p as)
... | false =  skip  (filter-sub p as)
```

- Der induktive Beweis verwendet Musterabgleich und Rekursion.
- Im Fall der leeren Liste $as = []$ ist filter p $as = []$ und Konstruktor empty liefert den (trivialen) Beweis von $[] \subset []$.

Induktive Beweise II

```
filter-sub : ∀{A} (p : A → Bool) (as : List A) → filter p as ⊂ as
filter-sub p []  =  empty
filter-sub p (a :: as) with p a
... | true  =  cons  (filter-sub p as)
... | false =  skip  (filter-sub p as)
```

Im Fall der nicht-leeren Liste $(a :: as)$ zeigen wir filter p $(a :: as) \subset (a :: as)$ durch eine weitere Fallunterscheidung.

- Falls $p\ a =$ true, reduziert filter p $(a :: as)$ zu $(a ::$ filter $p\ as)$ und das Theorem folgt aus der Induktionshypothese filter-sub $p\ as$ mittels cons.
- Falls $p\ a =$ false, reduziert filter p $(a :: as)$ zu filter $p\ as$ und das Theorem folgt aus der Induktionshypothese mittels skip.

Induktion über Herleitungen

- Die Teilfolgenrelation ist transitiv.

$$\begin{aligned}\text{sub-trans} \;:\; & \forall\{A\}\,\{as\ bs\ cs : \text{List}\ A\} \to \\ & as \subset bs \to bs \subset cs \to as \subset cs\end{aligned}$$

- Der Beweis analysiert die Form der Herleitungen ($p : as \subset bs$) und ($q : bs \subset cs$) und konstruiert daraus eine Herleitung von $as \subset cs$.

$$\begin{array}{llll}
\text{sub-trans} & \text{empty} & \text{empty} & = \text{empty} \\
\text{sub-trans} & \text{empty} & (\text{cons}\ q) & = \text{empty} \\
\text{sub-trans} & (\text{cons}\ p) & (\text{cons}\ q) & = \text{cons}\ (\text{sub-trans}\ p\ q) \\
\text{sub-trans} & (\text{skip}\ p) & (\text{cons}\ q) & = \text{skip}\ (\text{sub-trans}\ p\ q) \\
\text{sub-trans} & p & (\text{skip}\ q) & = \text{skip}\ (\text{sub-trans}\ p\ q)
\end{array}$$

Induktion genügt nicht!

- Induktive Typen und Prädikate entsprechen Bäumen endlicher Tiefe.
- Rekursive Programme über induktiven Typen müssen terminieren.
- Prozesse sollen nicht terminieren, wie passen sie ins Bild?

Ein simpler Prozess

- Betrachten wir einen Prozess, der einen Strom von Elementen liefert.
- Er akzeptiert zwei Nachrichten:
  - head: Darauf antwortet er mit dem momentan anliegenden Element.
  - tail: Darauf geht er weiter zum nächsten Element.
- In funktionaler Programmierung gibt tail einen neuen Prozess zurück, da alle Objekte unveränderlich sind.
- Die Nachrichten als Funktionen:

$$\begin{array}{lcl} \mathsf{head} & : & \mathsf{Stream\ A} \to \mathsf{A} \\ \mathsf{tail} & : & \mathsf{Stream\ A} \to \mathsf{Stream\ A} \end{array}$$

Ströme in Agda

- In Agda codieren wir Ströme als rekursive Verbundtypen, die als *koinduktiv* deklariert sind.

```
record Stream A : Set where
  coinductive
  field  head  :  A
         tail  :  Stream A
open Stream public using (head; tail)
```

- Das Feld head enthält die Antwort auf die Nachricht head.
- Ebenso für tail.

Ein replizierender Prozess

- repeat a ist der Prozess, der auf head immer a sendet.

$$\begin{array}{l} \text{repeat} : \forall\{A\}\ (a : A) \to \text{Stream}\ A \\ \text{head}\ \ (\text{repeat}\ a)\ =\ a \\ \text{tail}\ \ \ (\text{repeat}\ a)\ =\ \text{repeat}\ a \end{array}$$

- Auf tail gibt er sich selbst zurück.
- Diese rekursive Definition ist *produktiv*, im Gegensatz zu

$$\text{tail}\ \ (\text{repeat}\ a)\ =\ \text{tail}\ (\text{repeat}\ a)$$

- Hinreichendes Kriterium für Produktivität:
  Der rekursive Aufruf erfolgt unter mind. einer Nachricht weniger.

Ein iterierender Prozess

- Variante von repeat, liefert nacheinander a, $f\ a$, $f\ (f\ a)$ ...
- Iteriert die Funktion f, beginnend mit Startwert a.

$$\begin{array}{l} \mathsf{iter} : \forall\{A\}\ (f : A \to A)\ (a : A) \to \mathsf{Stream}\ A \\ \mathsf{head}\ (\mathsf{iter}\ f\ a) = a \\ \mathsf{tail}\ \ (\mathsf{iter}\ f\ a) = \mathsf{iter}\ f\ (f\ a) \end{array}$$

- iter (wie auch repeat) ist definiert durch Komusterabgleich.
- Die Komuster müssen *vollständig* sein, alle möglichen Nachrichten müssen behandelt werden.

Prozess-Transformer

- Der Prozess cons a s liefert zuerst a, dann verhält er sich wie s.

$$\begin{array}{lll} \mathsf{cons} & : & \forall\{A\}\ (a : A)\ (s : \mathsf{Stream}\ A) \to \mathsf{Stream}\ A \\ \mathsf{head} & (\mathsf{cons}\ a\ s) & = \ a \\ \mathsf{tail} & (\mathsf{cons}\ a\ s) & = \ s \end{array}$$

- Der Prozess map f s wendet die Funktion f auf jedes Element a an, das s liefert.

$$\begin{array}{lll} \mathsf{map} & : & \forall\{A\ B\}\ (f : A \to B)\ (s : \mathsf{Stream}\ A) \to \mathsf{Stream}\ B \\ \mathsf{head} & (\mathsf{map}\ f\ s) & = \ f\ (\mathsf{head}\ s) \\ \mathsf{tail} & (\mathsf{map}\ f\ s) & = \ \mathsf{map}\ f\ (\mathsf{tail}\ s) \end{array}$$

- Die Nachricht head gibt map f s an s weiter, und wendet f auf die Antwort an.
- tail wird auch weitergereicht, dann rekursiver Aufruf.

Bisimulation: Prozesse im Gleichschritt

- Zwei Prozesse (Ströme) s und t verhalten sich gleich, wenn:
  - sie auf die Nachricht head gleiche Elemente liefern, und
  - sie sich auf die Nachricht tail hin weiterhin gleich verhalten.
- Diese Ströme simulieren sich gegenseitig, sind *bisimilar*.
- Unter dem Curry-Howard-Isomorphismus ist Bisimilarität von s und t selbst ein Prozess:
  - auf ~head liefert er einen Beweis, dass head s und tail t identisch sind;
  - auf ~tail einen Beweis, dass tail s und tail t bisimilar sind.

```
record _~_ {A} (s t : Stream A) : Set where
  coinductive
  field ~head : head s ≡ head t
        ~tail : tail s  ~  tail t
open _~_ public using (~head; ~tail)
```

Koinduktion

- Beweis der Bisimilarität ist ein koinduktiver Beweis.
- Wir können den Beweis *beliebig tief* abfragen, durch fortgesetztes Senden von tail.
- Wir haben den Beweis jedoch nie "ganz", das würde unendlich viele Nachrichten erfordern.
- Auch einen Strom können wir nie "ganz" lesen, aber beliebig viel davon.

Ein koinduktiver Beweis

- Wir machen folgende Beobachtung:

iter $f\ a$	a	$f\ a$	$f\ (f\ a)$	$\ldots$
map f (iter $f\ a$)	$f\ a$	$f\ (f\ a)$	$f\ (f\ (f\ a))$	$\ldots$
iter $f\ (f\ a)$	$f\ a$	$f\ (f\ a)$	$f\ (f\ (f\ a))$	$\ldots$

- Wir beweisen map f (iter $f\ a$) ist bisimilar zu iter $f\ (f\ a)$:

~head: head (map f (iter $f\ a$)) $=$ f (head (iter $f\ a$)) $=$ $f\ a$ $=$ head (iter f ($f\ a$))

~tail: tail (map f (iter $f\ a$)) $=$ map f (tail (iter $f\ a$)) $=$ map f (iter f ($f\ a$))
und tail (iter f ($f\ a$)) $=$ (iter f (f ($f\ a$))) sind bisimilar nach Koinduktionshypothese.

```
map-iter : ∀{A} (f : A → A) (a : A) → map f (iter f a) ~ iter f (f a)
~head (map-iter f a) = refl                -- definitionsgleich
~tail (map-iter f a) = map-iter f (f a)    -- Koinduktionshyp.
```

Ein ungültiger Beweis

- Die Koinduktionshypothese hat diesselbe Form wie die ursprüngliche Aussage.
- Können wir durch die Koinduktionshypothese nicht alles beweisen?
- Nein, nicht jede Anwendung der Koinduktionshypothese ist legal!

$$\begin{array}{l} \mathsf{invalid} : \forall\{A\}\ (f\ g : A \to A)\ (a : A) \to \mathsf{iter}\ f\ a \sim \mathsf{iter}\ g\ a \\ \sim\mathsf{head}\ \ (\mathsf{invalid}\ f\ g\ a) \ = \ \mathsf{refl} \\ \sim\mathsf{tail}\ \ \ (\mathsf{invalid}\ f\ g\ a) \ = \ \sim\mathsf{tail}\ (\mathsf{invalid}\ f\ g\ a) \end{array}$$

- invalid ist nicht produktiv, tail (invalid f g a) eine Endlosschleife.
- Zum rekursiven Aufruf wurde hier *keine* Nachricht konsumiert.
- Agda's Terminationsprüfer schlägt Alarm!

Fehlalarm!

- Inspiriert durch die Bisimilarität von iter f (f a) und map f (iter f a), versuchen wir folgende alternative Definition der Iteration:

$$\begin{array}{lll} \text{iter}' : \forall\{A\}\ (f : A \to A)\ (a : A) \to \text{Stream}\ A \\ \text{head}\ (\text{iter}'\ f\ a) & = & a \\ \text{tail}\ (\text{iter}'\ f\ a) & = & \text{map}\ f\ (\text{iter}'\ f\ a) \end{array}$$

- Der Terminationsprüfer weist die 2. Gleichung zurück, warum?
- Ersetzen wir map durch map-evens vom gleichen Typ!

$$\begin{array}{lll} \text{map-evens} : \forall\{A\ B\}\ (f : A \to B)\ (s : \text{Stream}\ A) \to \text{Stream}\ B \\ \text{head}\ (\text{map-evens}\ f\ s) & = & f\ (\text{head}\ s) \\ \text{tail}\ (\text{map-evens}\ f\ s) & = & \text{map-evens}\ f\ (\text{tail}\ (\text{tail}\ s)) \end{array}$$

- tail (tail (iter′ f a)) = tail (map-evens f (tail (tail (iter′ f a))))
divergiert!

Nachrichtentiefe im Typ deklarieren

- Der Prozess map f s leitet tail-Nachrichten 1:1 an s weiter.
- Aber der Typ Stream $A \to$ Stream A von map f schließt nicht aus, dass zusätzliche Nachrichten abgesetzt werden, wie bei map-evens.
- Auch tail hat Typ Stream $A \to$ Stream A.
- Wir brauchen aussagekräftigere Prozesstypen!
- Idee: Mitführen der Nachrichtentiefe im Typen.
- Stream i A kann maximal i-mal mit tail befragt werden.
- In der Definition eines Stream i A darf der rekursive Aufruf nur Stream j A für $j < i$ erzeugen.

Vorschau: Sized Types

- Verfeinerter Typ für map:

$$\begin{array}{lll} \mathsf{map} : \forall\{i\ A\ B\}\ (f : A \to B)\ (s : \mathsf{Stream}\ i\ A) \to \mathsf{Stream}\ i\ B \\ \mathsf{head}\ (\mathsf{map}\ f\ s) & = & f\ (\mathsf{head}\ s) \\ \mathsf{tail}\ \ (\mathsf{map}\ f\ s) & = & \mathsf{map}\ f\ (\mathsf{tail}\ s) \end{array}$$

- Verfeinerter Typ für iter′:

$$\begin{array}{lll} \mathsf{iter}' : \forall\{i\ A\}\ (f : A \to A)\ (a : A) \to \mathsf{Stream}\ i\ A \\ \mathsf{head}\ (\mathsf{iter}'\ f\ a) & = & a \\ \mathsf{tail}\ \ (\mathsf{iter}'\ f\ a) & = & \mathsf{map}\ f\ \ (\mathsf{iter}'\ f\ a) \end{array}$$

Vorschau: Sized Types

- Definition von Stream i A.

```
open import Size

record Stream i A : Set where
  coinductive
  field  head  :  A
         tail  :  ∀{j : Size< i} → Stream j A
open Stream public using (head; tail)
```

- Mehr in *Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity* (Abel, Pientka, ICFP 2013)

Zusammenfassung

- Komusterabgleich: Natürliches Arbeiten mit Prozessen im "objekt-orientierten" Stil (Nachrichten).
- Basierend auf *Copatterns: Programming Infinite Structures by Observations* (Abel, Pientka, Thibodeau, Setzer, POPL 2013)
- Implementierung in Agda (2012–)
- *Coq* (INRIA) ist ähnliches Beweissystem mit industrieller Anwendung
- Coq hat anderes Modell von Prozessen: unendlich tiefe Bäume
- Nachteil: keine Typerhaltung unter Reduktion

Weiterentwicklung von Agda

- Vollendung der typbasierten Terminationsprüfung mit Sized Types
- Effizienter Typ-Prüfung
- Übersetzung in effiziente Maschinensprache
- Meta-programmierung (Reflektion) und taktische Beweise

Orte der Agda-Entwicklung

- Göteborg (SE): Norell, Danielsson, Vezzosi, Moulin
- Utrecht (NL): Swierstra, Hausmann
- Leuven (BE): Devriese, Cockx
- Sophia-Antipolis (FR): Brunerie
- Strathclyde (Scotland): Chapman, Andjelkovic, Allais
- Copenhagen (DK): Pouillard, Gustafsson
- Kanagawa (JP): Takeyama