KeY

Maciej Zielenkiewicz

14 października 2009

Systemy dowodzenia

Chcemy sformalizować dowodzenie dla pewnej logiki.
Wprowadzamy:

- aksjomaty
- reguły

To, że φ *wynika* w naszej logice z Φ oznaczamy $\Phi \models \varphi$.
Jeżeli φ *jest wyprowadzalne* w naszym systemie dowodzenia oznaczamy $\Phi \vdash \varphi$.
Interesują nas takie systemy dowodzenia, w których

$$\Phi \models \varphi \iff \Phi \vdash \varphi$$

(czyli poprawne i pełne).

System Gentzena dla rachunku zdań I

Aksjomat:

$$\frac{}{A \vdash A} \quad (I)$$

Reguły:

$$\frac{\Gamma \vdash \Delta, A \qquad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi} \quad (Cut)$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \quad (\wedge L_1)$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta} \quad (\vee R_1)$$

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \quad (\wedge L_2)$$

$$\frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta} \quad (\vee R_2)$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \vee B \vdash \Delta, \Pi} \quad (\vee L)$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Sigma \vdash B, \Pi}{\Gamma, \Sigma \vdash A \wedge B, \Delta, \Pi} \quad (\wedge R)$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \to B \vdash \Delta, \Pi} \quad (\to L)$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \quad (\to R)$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \quad (\neg L)$$

System Gentzena dla rachunku zdań II

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ (\neg R) \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ (WL)$$

$$\frac{\Gamma, A[t] \vdash \Delta}{\Gamma, \forall x A[x/t] \vdash \Delta}\ (\forall L) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ (WR)$$

$$\frac{\Gamma \vdash A[y], \Delta}{\Gamma \vdash \forall x A[x/y], \Delta}\ (\forall R) \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ (CL)$$

$$\frac{\Gamma, A[y] \vdash \Delta}{\Gamma, \exists x A[x/y] \vdash \Delta}\ (\exists L) \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ (CR)$$

$$\frac{\Gamma \vdash A[t], \Delta}{\Gamma \vdash \exists x A[x/t], \Delta}\ (\exists R) \qquad \frac{\Gamma_1, A, B, \Gamma_2 \vdash \Delta}{\Gamma_1, B, A, \Gamma_2 \vdash \Delta}\ (PL)$$

$$\frac{\Gamma \vdash \Delta_1, A, B, \Delta_2}{\Gamma \vdash \Delta_1, B, A, \Delta_2}\ (PR)$$

Logika modalna I

Logika modalna to rozszerzenie logiki zdaniowej o dodatkowe operacje unarne:

$\Box p$	zawsze p
$\Diamond p$	możliwe, że p

i aksjomat

$$\Box (p \to q) \to (\Box p \to \Box q)$$

Warto dodać kilka aksjomatów:

$$\begin{aligned} \Box p &\to p \\ \Box p &\to \Box\Box p \\ \Box p &\to \Box \Diamond p \\ \Box p &\to \Diamond p \end{aligned}$$

Logika temporalna I

Uzupełniamy logikę Hoare'a tak samo, jak uzupełniliśmy logikę zdaniową.

$\Box p$	zawsze (od teraz) p
$\diamond p$	kiedyś p
$[a]p$	po wykonaniu a zawsze p
$\langle a \rangle p$	po wykonaniu a kiedyś p

z zachowaniem przedtem pokazanych aksjomatów

Ze względów *technicznych* przed częścią formuły może stać dodatkowa konstrukcja syntaktyczna opisująca wywołania metod (dla mapowania *return*), pętli i przypisania (w językach obiektowych nie da się ich zreealizować przez proste podstawienia).

Zazwyczaj korzysta się z bogatszego zestawu operacji, np. `http://en.wikipedia.org/wiki/Temporal_logic`, ale dla nas nie będzie to użyteczne.

System dowodzenia dla logiki temporalnej

- Konstrukcja systemu dowodzenia jest skomplikowana i nie będzie w całości pokazana.
- Problemem jest interpretacja fragmentów programu, odziedziczona z logiki Hoare'a, a nie elementy temporalne.
- System dowodzenia musi być dostosowany do języka, z którego będą programy.

System dowodzenia dla logiki temporalnej

- Konstrukcja systemu dowodzenia jest skomplikowana i nie będzie w całości pokazana.
- Problemem jest interpretacja fragmentów programu, odziedziczona z logiki Hoare'a, a nie elementy temporalne.
- System dowodzenia musi być dostosowany do języka, z którego będą programy.

Rozwiązanie szególne: taklety (*taclets*):

- Uproszczona wersja taktyk, przypominają wyrażenia regularne.
- Przykład:

```
find ( b->c ==> ) if ( b ==> ) replacewith ( c ==> )
                                        heuristics (simplify)
```

- później: dodatkowe informacje o ograniczeniach w stosowaniu.

JML

- JML = Java Modeling Language
- `http://www.eecs.ucf.edu/~leavens/JML/index.shtml`
- realizuje ideę *Design By Contract*

Design By Contract

- Programiści umawiają się, że klasa/metoda będzie wywoływana tylko z parametrami spełniającymi pewne *warunki wstępne*.
- Natomiast wynik działania będzie spełniał *warunki końcowe*.
- Działanie zgodnie z tymi warunkami stanowi *kontrakt* między klasą („implementujących") a jej użytkownikami („klientami").
- JML pozwala osadzać informacje o tych warunkach w kodzie, analogicznie do tego jak Javadoc pozwala osadzać dokumentację w kodzie.
- warunki działają analogicznie do logiki Hoare'a
- za spełnienie warunków początkowych odpowiada klient, a końcowych–implementujący

Podstawowe informacje I

- JML wykorzystuje *adnotacje* zapisane w specjalnych komentarzach:

```
//@ ...

/*@ ... @*/

/*@
  @ ...
  @ ...
  @
  @*/
```

- komentarze zapisujemy bezpośrednio przed metodą, której dotyczą
- warunek początkowy ma postać:

```
//@ requires x > 0
```

Podstawowe informacje II

- warunek końcowy ma postać:

  ```
  //@ ensures y > 3
  ```

- można zdefiniować jakie wyjątki może przekazywać dana metoda, ale jest do tego także gotowy interfejs w Javie, więc lepiej z niego korzystać (**throws**)
- po skompilowaniu za pomocą specjalnego kompilatora (*jmlc*) uzyskujemy kod który oprócz standardowych treści procedur zawiera instrukcje sprawdzające podane warunki
- ale uzyskany w ten sposób kod może być isotnie wolniejszy niż zwykły, więc po skompilowaniu zwykłym kompilatorem możemy mieć wersję bez sprawdzeń
- sprawdzanie warunków nie może mieć efektów ubocznych

Bardziej zaawansowane zastosowania I

- warunki z kwantyfikatorami – np. sprawdzenie czy tablica jest posortowana

```
/*@ requires a != null
  @  && (\forall int i;
  @ 0 < i && i < a.length;
  @ a[i-1] <= a[i]);
  @*/
```

- mamy do wyboru także inne kwantyfikatory:
  `\forall \exists \sum \product \min \max \num_of`
- można też zamiast bezpośrednio podawać zakres wskazać których parametrów dotyczy warunek:

```
  @ (\forall Student s;
  @ juniors.contains(s);
  @ s.age <= 21)
  @*/
```

Bardziej zaawansowane zastosowania II

- sprawdzanie takich warunków może być dużo dłuższe niż sama metoda która ma się wykonać!
- dla klas które po sobie dziedziczą warunki także są dziedziczone
- dostępne są dodatkowe operatory:

`\result`	wynik procedury
`a ==> b`	z a wynika b
`a <== b`	z b wynika a
`a <==> b`	a wtw. b
`a <=!=> b`	$\neg$ a wtw. b
`\old(x)`	wartość `x` przed wywołaniem procedury

- w warunkach można wywoływać tylko metody, które wcześniej oznaczymy modyfikatorem **pure**, które nie mogą zmieniać stanu obiektu

Bardziej zaawansowane zastosowania III

- z komentarzy javadoc i adnotacji JML-owych można wyprodukować za pomocą *jmldoc* dokumentacją podobną do tej generowanej przez *javadoc*, ale uwzględniającą adnotacje
- można także opisywać słownie warunki, ale wtedy nie są one automatycznie sprawdzane:
  `//@requires (* x is positive *);`
- w klasach można deklarować *niezmienniki*, które muszą być prawdziwe zawsze, jeżeli nie jest wykonywana żadna metoda:
  `//@ public invariant !name.equals(‘ ’)`
- specyfikację można umieścić w plikach *.refine-java zamiast w plikach z kodem
- przykładowa klasa:

Bardziej zaawansowane zastosowania IV

```
public class Person {
        private /*@ spec_public non_null @*/
                String name;
        /*@ also
          @ ensures \result != null
          @*/
        public String toString() {
                return "Person(\""+name+"\")";
        }
        /*@ also
          @ requires n != null && !n.equals("");
          @ ensures n.equals(name)
          @*/
        public Person(String n)
        {
                name=n;
        }
}
```

KeY I

`http://www.key-project.org/`

- Pozwala na częściowo automatyczne dowodzenie poprawności programów w Javie (dokładniej: w Java Card).
- Pozwala na korzystanie ze specyfikacji w JMLu, OCLu i Java Card DL.
- Wykorzystuje taklety.
  - Dodając nieprawdziwe taklety można udowodnić nieprawdziwe rzeczy!
  - Ale taklety można *weryfikować*...
  - Łatwiej pisać taklety niż taktyki, ale możliwości są ograniczone.
- Korzysta przez OCL z informacji z diagramów UML.
- Poprzez *triki* można dowodzić innych własności niż poprawność (np. pewne warianty bezpieczeństwa).
- Może korzystać z zewnętrznych *prooverów*, można też dowodzić zdania bez związku z programem.
- Przy dowodzeniu pętli trzeba podać niezmienniki ręcznie.

Weryfikacje z sukcesami:

- proste aplety Java Card
- parser poleceń dla analizatorów chemicznych Agilent
- (niedokończone) ustalanie maksymalnych prędkości pociągów na torach Deutsche Bahn AG

Weryfikacje z sukcesami:

- proste aplety Java Card
- parser poleceń dla analizatorów chemicznych Agilent
- (niedokończone) ustalanie maksymalnych prędkości pociągów na torach Deutsche Bahn AG

Dalszy rozwój:

- generowanie kontrprzykładów
- odczytywanie zależności czasowych z diagramów UML

Dowodzenie w KeY

- Wewnątrz KeY stosujemy notację Java DL.

Formuły:

Java DL	notacja książkowa	interpretacja
true	1	pradwa
false	0	fałsz
!φ	$\neg\varphi$	negacja
φ&γ	$\varphi \wedge \gamma$	koniunkcja
$\varphi\|\gamma$	$\varphi \vee \gamma$	dysjunkcja
$\varphi->\gamma$	$\varphi \rightarrow \gamma$	implikacja
\if ϕ \ then γ \ else η		formuła warunkowa
\forall $T\,x$; φ	$\forall x : T.\varphi$	kwant. ogólna po typie T
\exists $T\,x$; φ	$\exists x : T.\varphi$	kwant. egzystencjalna
$\mathcal{U}\varphi$	$\mathcal{U}\varphi$	przypisania
\<{p}\>φ	$\langle p\rangle\varphi$	diamond modality
\[{p}\]φ	$[p]\,\varphi$	box modality

Predykaty: $=$, $<$, $<=$, $>$, $>=$, `inReachableState`, `arrayStoreValid(., .)`.

Dowodzenie w KeY I

Termy: oprócz podstawienia termu za zmienną we wcześniej wymienionych formułach, można także aplikować funkcje

$$f(t_1, \ldots, t_n)$$

oraz rzutować (na pewien typ T)

$$(T)t$$

Istnieją predefiniowane funkcje matematyczne:

prefix	infix	opis
$add(\cdot,\cdot)$	$\cdot + \cdot$	dodawnie w $\mathcal{Z}$
$sub(\cdot,\cdot)$	$\cdot - \cdot$	odejmowanie w $\mathcal{Z}$
$mul(\cdot,\cdot)$	$\cdot * \cdot$	mnożenie w $\mathcal{Z}$
$div(\cdot,\cdot)$	$\cdot / \cdot$	dzielenie w $\mathcal{Z}$
$mod(\cdot,\cdot)$	$\cdot \% \cdot$	modulo w $\mathcal{Z}$
$jdiv(\cdot,\cdot)$		dzielenie z Javy ograniczone do $\mathcal{Z}$
$jmod(\cdot,\cdot)$		modulo z Javy ograniczone do $\mathcal{Z}$
$divJint(\cdot,\cdot)$		dzielenie z Javy ograniczone do zakresu int

Dowodzenie w KeY II

i inne:

prefix	opis
null	stała null z Javy typu *Null*
TRUE, FALSE	stałe logiczne z Javy typu *boolean*
$T \ldots instance(o)$	funkcja typu *boolean*, prawda w.t.w. o jest instancją

Aktualizacje:

$$\backslash \mathrm{for}\, T\, x;\ \backslash \mathrm{if}(\varphi) loc := val$$

kilka aktualizacji zapisujemy równolegle:

$$u_1 || u_2$$

W programie zapisujemy meta-operacje:

```
method-frame( result->zmienna,
                        source=nazwa_klasy,
                        this=referencja):{
                        lista\dywiz operacji
}
\end{verbatim}
```