Type Checking Without Types

Matthias Benkard

9.11.2012

Queries

```
?- prime(X).
X = 2, X = 3, X = 5, X = 7, X = 11, X = 13, ...
```

Queries

```
?- prime(X).
X = 2, X = 3, X = 5, X = 7, X = 11, X = 13, ...
```

Typ von `X`?

Queries

```
?- prime(X).
X = 2, X = 3, X = 5, X = 7, X = 11, X = 13, ...
```

Typ von `X`?

```
let x = exists isPrime in ...
```

Typ von `x`?

Queries

```
let x = exists isPrime in
  if (isEven x) then
    backtrack  --non-odd primes are evil
  else
    ...
```

Typ von x?

Queries

```
let x = exists isPrime in
  if (isEven x) then
    backtrack  --non-odd primes are evil
  else
    ...
```

Typ von `x`? Wünschenswert: `x : int`!

Funktionale Queries

```
exists predicate =
  let yieldFrom n =
    if (predicate n) then
      n ⊕ yieldFrom (n+1)
    else
      yieldFrom (n+1)
  in
    yieldFrom 2
```

Funktionale Queries

```
let x = exists isPrime in
  if (isEven x) then
    backtrack  --non-odd primes are evil
  else
    ...
```

Typ von `x`? Wünschenswert: `x : int`!

Funktionale Queries

```
let x = exists isPrime in
  if (isEven x) then
    backtrack  --non-odd primes are evil
  else
    ...
```

Typ von `x`? Wünschenswert: `x : int`!

```
let x = 2⊕3⊕5⊕7⊕... in
  if (isEven x) then
    backtrack
  else
    ...
```

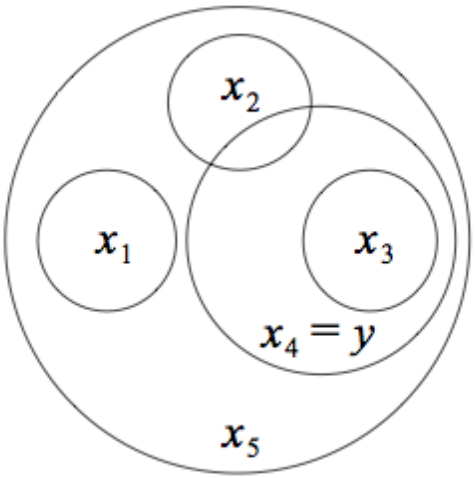
Typ von `x`?

Ausflug: Mereologie

- Geometrische/räumliche Intuition
- Pxy „x ist Teil von y“

Darauf aufbauend:

- $Oxy \iff \exists z\,(Pzx \wedge Pzy)$ „x überlappt y“
- $Uxy \iff \exists z\,(Pxz \wedge Pyz)$ „x unterlappt y“
- $x = y \iff Pxy \wedge Pyx$



	Ux_ny	Ox_ny	PPx_ny	EQx_ny	PEx_ny
$n=1$	+	–	–	–	–
$n=2$	+	+	–	–	–
$n=3$	+	+	+	–	–
$n=4$	+	+	–	+	–
$n=5$	+	+	–	–	+

(Illustration: http://plato.stanford.edu/entries/mereology/)

Fusion

Vereinfachte Fusionsaxiome:

$$\forall x, y\ \exists z\ \forall w.\ Ozw \iff (Oxw \vee Oyw)$$

(Fusion)

Fusion

Vereinfachte Fusionsaxiome:

$$\forall x, y\ \exists z\ \forall w.\ Ozw \iff (Oxw \vee Oyw)$$

(Fusion)

$$\exists z\ \forall w. Ozw \iff \exists v(\phi v \wedge Ovw)$$

(Komprehensionsschema)

Eine nichtdeterministische funktionale Sprache

Idee: Baue eine Sprache zusammen aus:

- Atomen ($\boxed{0}$, $\boxed{1}$, $\boxed{-1}$, $\ldots$; $\boxed{'a}$, $\boxed{'b}$, $\boxed{'succ}$, $\ldots$)
- Paaren $\boxed{(s, t)}$
- Patterns $\boxed{p}$ ($\boxed{x}$, $\boxed{p_1, p_2}$, $\boxed{p : t}$, $\ldots$)
- Funktionen $\boxed{\text{fun } \{\overrightarrow{p \to e}\}}$ mit pattern matching
- Fixpunkten $\boxed{\text{fix } x \to e}$
- Fusionen $\boxed{s \oplus t}$
- Fusionskomprehensionen $\boxed{\{p\}}$

Eine nichtdeterministische funktionale Sprache

Idee: Baue eine Sprache zusammen aus:

- Atomen ($\boxed{0}$, $\boxed{1}$, $\boxed{-1}$, $\ldots$; $\boxed{'a}$, $\boxed{'b}$, $\boxed{'succ}$, $\ldots$)
- Paaren $\boxed{(s, t)}$
- Patterns $\boxed{p}$ ($\boxed{x}$, $\boxed{p_1, p_2}$, $\boxed{p : t}$, $\ldots$)
- Funktionen $\boxed{\text{fun } \{\overrightarrow{p \to e}\}}$ mit pattern matching
- Fixpunkten $\boxed{\text{fix } x \to e}$
- Fusionen $\boxed{s \oplus t}$
- Fusionskomprehensionen $\boxed{\{p\}}$

Und die Typebene?

Up a level

Wir haben Fusionen. Wofür benötigen wir noch Mengen?

Up a level

Wir haben Fusionen. Wofür benötigen wir noch Mengen?

Definiere:

```
int := 0⊕1⊕-1⊕2⊕-2⊕...
```

Up a level

Wir haben Fusionen. Wofür benötigen wir noch Mengen?

Definiere:

```
int := 0⊕1⊕-1⊕2⊕-2⊕...
```

Damit:

- `2⊕3⊕5⊕7⊕11⊕... : int`

Up a level

Wir haben Fusionen. Wofür benötigen wir noch Mengen?

Definiere:

```
int := 0⊕1⊕-1⊕2⊕-2⊕...
```

Damit:

- `2⊕3⊕5⊕7⊕11⊕... : int`
- `0⊕1⊕-1⊕2⊕-2⊕... : int`

Up a level

Wir haben Fusionen. Wofür benötigen wir noch Mengen?

Definiere:

```
int := 0⊕1⊕-1⊕2⊕-2⊕...
```

Damit:

- `2⊕3⊕5⊕7⊕11⊕... : int`
- `0⊕1⊕-1⊕2⊕-2⊕... : int`
- `int : int`

Up a level

Wir haben Fusionen. Wofür benötigen wir noch Mengen?

Definiere:

```
int := 0⊕1⊕-1⊕2⊕-2⊕...
```

Damit:

- `2⊕3⊕5⊕7⊕11⊕... : int`
- `0⊕1⊕-1⊕2⊕-2⊕... : int`
- `int : int`

Das ist nichts anderes als Subtyping.

Alles ist ein Typ!

Beispiele

- `'hi : 'hi`

Beispiele

- `'hi : 'hi`
- `'true : 'true`$\oplus$`'false`

Beispiele

- `'hi : 'hi`
- `'true : 'true⊕'false`
- `(1, 2)⊕(1, 3) : (1, 2⊕3)`

Ein etwas interessanteres Beispiel

```
nat := fix nat → 'zero ⊕ ('succ, nat)
```

Ein etwas interessanteres Beispiel

```
nat := fix nat → 'zero ⊕ ('succ, nat)
```

Damit:

```
nat = 'zero⊕('succ, 'zero)⊕('succ, ('succ, 'zero))⊕...
```

Ein etwas interessanteres Beispiel

```
nat := fix nat → 'zero ⊕ ('succ, nat)
```

Damit:

nat = 'zero⊕('succ, 'zero)⊕('succ, ('succ, 'zero))⊕...

Vergleiche:

```
nat(zero).
nat(succ(X)) :- nat(X).
```

Algebraische Datentypen

```
list := fix list →
  fun {a → 'nil ⊕ ('cons, a, list a)}
```

Algebraische Datentypen

```
list := fix list →
  fun {a → 'nil ⊕ ('cons, a, list a)}
```

Damit:

`list int = 'nil⊕('cons,int,'nil)⊕...`

Abhängige Typen

```
fix vec →
  fun {a →
    fun {'zero       → 'vnil;
         ('succ, n) → ('vcons, a, vec a n)}}
```

Abhängige Typen

```
fix vec →
  fun {a →
    fun {'zero       → 'vnil;
         ('succ, n) → ('vcons, a, vec a n)}}
```

```
id := fun {x → x}
```

Das ist ein abhängiger Typ!

Abhängige Typen

```
fix vec →
  fun {a →
    fun {'zero       → 'vnil;
         ('succ, n) → ('vcons, a, vec a n)}}
```

```
id := fun {x → x}
```

Das ist ein abhängiger Typ!

- `id 2 : int`

Abhängige Typen

```
fix vec →
  fun {a →
    fun {'zero       → 'vnil;
         ('succ, n) → ('vcons, a, vec a n)}}
```

```
id := fun {x → x}
```

Das ist ein abhängiger Typ!

- `id 2 : int`
- `id 2 : 2`

Polymorphie

```
fix map →
  fun {a →
    fun {(f : (a → a)) →
      fun {'nil          →
             'nil
          ;('cons,(x : a),(xs : list a)) →
             ('cons, f x, map a f xs)}}}
```

Typdisjunktionen

```
2⊕"hi"
```

Typdisjunktionen

$2 \oplus$ `"hi"` : $int \cup string$

Typdisjunktionen

$2 \oplus \texttt{"hi"} : \mathit{int} \cup \mathit{string}$

$2 \oplus \texttt{"hi"} : \texttt{int} \oplus \texttt{string}$

Ausfalten oder aufrunden?

```
map a f xs
```

Ausfalten oder aufrunden?

```
map a f xs
```

Ist a als Typ oder als Laufzeitwert gemeint?

Ausfalten oder aufrunden?

```
map a f xs
```

Ist a als Typ oder als Laufzeitwert gemeint?
f?

Ausfalten oder aufrunden?

```
map a f xs
```

Ist a als Typ oder als Laufzeitwert gemeint?

`f`?

`xs`?

Ausfalten oder aufrunden?

```
map a f xs
```

Ist a als Typ oder als Laufzeitwert gemeint?
`f`?
`xs`?

Benötigen eine Heuristik!

Fehlende Vereinfachung

Kann einfachste Prädikatenlogik momentan nicht darstellen.

```
let T
  : fun {(n : nat) → 'trivial}
  = fun { 0                  → 'trivial
        ; ('s, (x : nat)) → T x       }

let allT
  : fun {(n : nat) → T n}
  = fun { 0                  → 'trivial
        ; ('s, (x : nat)) → allT x    }
```

Problem: FIXME

Fehlende Vereinfachung

Kann einfachste Prädikatenlogik momentan nicht darstellen.

```
let T
  : fun {(n : nat) → 'trivial}
  = fun { 0                  → 'trivial
        ; ('s, (x : nat)) → T x       }

let allT
  : fun {(n : nat) → T n}
  = fun { 0                  → 'trivial
        ; ('s, (x : nat)) → allT x    }
```

Problem: FIXME

Lösbar!

- FP $\cup$ expliziter Nichtdeterminismus
- Queries sind Typen
- Programme sind Queries
- Type checking deutlich unentscheidbar
- Abhängige Typen und Polymorphie *for free*
- Compiler benötigt Heuristik, um sich nicht zu verlaufen

- ~~FP $\cup$ expliziter Nichtdeterminismus~~
- FP $\oplus$ expliziter Nichtdeterminismus
- Queries sind Typen
- Programme sind Queries
- Type checking deutlich unentscheidbar
- Abhängige Typen und Polymorphie *for free*
- Compiler benötigt Heuristik, um sich nicht zu verlaufen

Literatur

- Mereologie: Roberto Casati and Achille C. Varzi. *Parts and Places: The Structures of Spatial Representation.* MIT Press, Cambridge, MA, 1999.
- Programmieren mit abhängigen Typen: Ulf Norell. *Dependently Typed Programming in Agda.* 2008, `http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`.
- Subtyping-basierte Typprüfung: DeLesley S. Hutchins. *Pure Subtype Systems.* POPL 2010.
- Stichworte zum Googlen: *singleton types*, *occurrence typing*, *description logic*