Semantics of Programming Languages (6)

Christian Urban

http://www4.in.tum.de/~urbanc/Teaching/semantics08.html
fun
  minus_nat :: "nat \Rightarrow nat \Rightarrow nat" ("_ - _")
where
  "m - 0 = (m::nat)"
| "m - (Suc n) = (case m - n of 0 => 0 | Suc k => k)"

lemma
  fixes m n::"nat"
  shows "m - n + n = m"

Quickcheck should produce a counter example: for example m = 0, n = Suc 0
The Lambda-Calculus

Programming language research considers tiny languages to study features in isolation.

The lambda-calculus has been introduced by Church; only contains variables, function application and function abstraction

\[ t ::= x \]
\[ \mid t_1 t_2 \]
\[ \mid \lambda x.t \]
The Lambda-Calculus

Programming language research considers tiny languages to study features in isolation.

The lambda-calculus has been introduced by Church; only contains variables, function application and function abstraction

$$t ::= x \mid t_1 \, t_2 \mid \lambda x.t$$

Is mathematically already rather interesting....
Some conventions for writing lambda-terms:

- The scope of the dot extends as far as possible to the right: \( \lambda x. t_1 (t_2 t_3) \).
- \( \lambda x y. t \) stands for \( \lambda x. (\lambda y. t) \).
- \( t_1 t_2 t_3 t_4 \) stands for \( ((t_1 t_2) t_3) t_4 \).
What Is the Problem?

The difficulties arise from

bound variables and $\alpha$-equivalence
What Is the Problem?

The difficulties arise from

**bound variables** and **α-equivalence**

**α-equivalence** expresses the fact that names of bound variables are unimportant; all that matters is the binding structure they induce.
What Is the Problem?

The difficulties arise from

 bound variables and $\alpha$-equivalence

in mathematics:

\[
\int x \, dx = \int y \, dy
\]

$\forall x. \, P \, x = \forall y. \, P \, y$

in programming:

\[
\lambda x. \, x = \lambda y. \, y
\]

```c
int sum (int x, int y) {
    return (x+y);
}
```

```c
int sum (int foo, int bar) {
    = return (foo+bar);
}
```
What Is the Problem?

The difficulties arise from bound variables and $\equiv_{\text{AB}}$-equivalence in mathematics:

in programming:

```
int sum (int x, int y) {
  return (x+y);
}
```

This problem might appear minute and trivial, but:

Stephen C. Kleene

Xavier Leroy (INRIA)
Bound Variables

Lambda-terms are not just strings (or trees) of ascii-characters if you want to reason about them.

Assume the dots stand for lambda-terms:

. . . . . . . . . . . . . . . . . .
Bound Variables

Lambda-terms are **not** just strings (or trees) of ascii-characters if you want to reason about them.

Assume the dots stand for lambda-terms:

\[ \alpha \text{-equivalence} \]
Bound Variables

Lambda-terms are **not** just strings (or trees) of ascii-characters if you want to reason about them.

Assume the dots stand for lambda-terms:
Bound Variables

Lambda-terms are not just strings (or trees) of ascii-characters if you want to reason about them.

Assume the dots stand for lambda-terms:

This quotient construction is needed in order to scaffold away from tedious details.
The set of free variables of a lambda-term:

\[
FV(x) \overset{\text{def}}{=} \{x\}
\]

\[
FV(t_1 \ t_2) \overset{\text{def}}{=} FV(t_1) \cup FV(t_2)
\]

\[
FV(\lambda x.t) \overset{\text{def}}{=} FV(t) - \{x\}
\]
Free Variables

The set of free variables of a lambda-term:

\[ FV(x) \overset{\text{def}}{=} \{ x \} \]
\[ FV(t_1 \ t_2) \overset{\text{def}}{=} FV(t_1) \cup FV(t_2) \]
\[ FV(\lambda x.t) \overset{\text{def}}{=} FV(t) - \{ x \} \]

How about the set of bound variables?
Over $\alpha$-equivalence classes the set of bound variables cannot be defined:

\[
\begin{align*}
BV(x) & \overset{\text{def}}{=} \{\} \\
BV(t_1 t_2) & \overset{\text{def}}{=} BV(t_1) \cup BV(t_2) \\
BV(\lambda x.t) & \overset{\text{def}}{=} BV(t) \cup \{x\}
\end{align*}
\]
Over $\alpha$-equivalence classes the set of bound variables cannot be defined:

$$BV(x) \overset{\text{def}}{=} \{\}$$

$$BV(t_1 \ t_2) \overset{\text{def}}{=} BV(t_1) \cup BV(t_2)$$

$$BV(\lambda x.t) \overset{\text{def}}{=} BV(t) \cup \{x\}$$

1. Assume you have distinct $x$ and $y$
Bound Variables

Over $\alpha$-equivalence classes the set of bound variables cannot be defined:

\[
BV(x) \overset{\text{def}}{=} \{ \}
\]
\[
BV(t_1 \ t_2) \overset{\text{def}}{=} BV(t_1) \cup BV(t_2)
\]
\[
BV(\lambda x. t) \overset{\text{def}}{=} BV(t) \cup \{x\}
\]

1. Assume you have distinct $x$ and $y$

2. $\lambda x. x = \lambda y. y$
Over $\alpha$-equivalence classes the set of bound variables cannot be defined:

\[
BV(x) \overset{\text{def}}{=} \{\}
\]

\[
BV(t_1 \; t_2) \overset{\text{def}}{=} BV(t_1) \cup BV(t_2)
\]

\[
BV(\lambda x. \; t) \overset{\text{def}}{=} BV(t) \cup \{x\}
\]

1. Assume you have distinct $x$ and $y$
2. $\lambda x. \; x = \lambda y. \; y$
3. $BV(\lambda x. \; x) = BV(\lambda y. \; y)$  \hspace{1cm} Argh!
Calculating with LTs

Two functions:

\[ \text{term } \lambda x. \ x \]
\[ \text{term } \lambda x \ y. \ x \ z \]

\[ \text{term } (\lambda x. \ x) \ y \] \implies y
\[ \text{term } (\lambda x \ y. \ y \ x) \ z \ w \] \implies w \ z
Calculating with LTs

Two functions:

- term "λx. x"
- term "λx y. x z"

- term "(λx. x) y" \Rightarrow y
- term "(λx y. y x) z w" \Rightarrow w z

This is called \textit{β-reduction} (precise definition later)
atom_decl name

nominal_datatype lam =
    Var "name"
  | App "lam" "lam"
  | Lam "name»lam" ("Lam [_._.]")

term "Var x"
term "App (Var x) (Var y)"
term "Lam [x].(Var y)"
The difference between working with trees/datatypes and $\alpha$-equivalence classes is very subtle.

One often confuses them. (I always find it helpful to only think in terms of $\alpha$-equivalence classes.)