3.4 The Pure Lambda Calculus

Dienstag, 21. Juni 2016 14:00

The λ-calculus can be reduced even further:

In the pure λ-calculus, there are no constants C.

Thus, there is no δ-reduction, only β-reduction.

The pure λ-Calculus is still Turing-complete, i.e., one can implement any computable function in the pure λ-calculus.

Since there are no constants (and no δ-rules), the constants have to be represented by suitable pure λ-terms whose β-reductions evaluates them in an appropriate way.

We have to choose a representation of data objects by λ-terms. Then one can implement all computable functions on these objects by λ-terms.

Data structure of natural numbers:

$\underbrace{f^n x}_{f(f \ldots (f\ x)\ldots)}$ ← the n-fold application of a function represents the number n

n times

This representation should not depend on a particular f or x.

⇒ We represent any natural number n

by the λ-term $\overline{n} = \lambda f x . f^n x$

E.g.: $\overline{0} = \lambda f x . x$

$\overline{1} = \lambda f x . f x$

$\overline{2} = \lambda f x . f (f x)$

$\vdots$

Now any computable fct. can be implemented as a λ-term operating on this representation of nat. numbers. E.g.:

$\overline{Succ}$ should be a pure λ-term such that

for any $n \in \mathbb{N}$:

$$\overline{Succ}\ \overline{n} \to_\beta^* \overline{n+1}$$

How does $\overline{Succ}$ look like?

$$\overline{Succ}\ \overline{n} = \overline{Succ}\ (\lambda f x . f^n x) \to^* \underbrace{\lambda f x . f^{n+1} x}_{\overline{n+1}}$$

To find $\overline{Succ}$, it is a good idea to transform the desired result $\overline{n+1}$ such that it contains $\overline{n}$.

$$\overline{n+1} = \lambda f x . f^{n+1} x = \lambda f x . f (\underbrace{f^n x}) \leftarrow_\beta^* \lambda f x . f (\overline{n} f x)$$

$$\underbrace{(\lambda f x . f^n x)}\ f\ x$$

So if $\overline{Succ}$ is applied to $\bar{n}$, the result should be $\lambda f x.\, f\,(\bar{n}\, f\, x)$

Solution $\overline{Succ} = \lambda n f x.\, f\,(n\, f\, x)$

$$\overline{Succ}\ \bar{n} = (\lambda n f x.\, f\,(n f x))\ \underline{(\lambda f x.\, f^n\, x)} \xrightarrow{}_\beta$$

$$\lambda f x.\, f((\underline{\lambda f x.\, f^n\, x})\, f\, x) \xrightarrow{2}_\beta$$

$$\lambda f x.\, f\,(f^n\, x) =$$

$$\lambda f x.\, f^{n+1}\, x = \overline{n+1}$$

In a similar way, one can implement $\overline{plus}$, $\overline{times}$, ...
$\Rightarrow$ any computable fct. on $\mathbb{N}$ can be implemented in this way.

Data structure of Booleans:

We choose the following representation of True and False by λ-terms:

$$\overline{True} = \lambda x y.\, x$$

$$\overline{False} = \lambda x y.\, y$$

Now we can implement any computable fct. on Booleans. E.g.:

$\overline{\text{if}}\ \overline{\text{True}}\ x\ y \to_\beta^* x$

$\overline{\text{if}}\ \overline{\text{False}}\ x\ y \to_\beta^* y$

$\overline{\text{if}}$ should be $\lambda\, a\, x\, y.\ a\, x\, y$

or simpler $\lambda\, a.\, a$

Since pure λ-calculus is a complete prog. language, we can also implement "fix" as a pure λ-term:

$\overline{\text{fix}}$ should satisfy: $\overline{\text{fix}}\ f \to_\beta^* f\ (\overline{\text{fix}}\ f)$

This can be done by Turing's Fixpoint Combinator:

$\overline{\text{fix}}$: $(\lambda x y.\, y\,(x\,x\,y))\ (\lambda x y.\, y\,(x\,x\,y))$

Drawbacks of pure λ-calculus:

- unreadable
- inefficient
- not suitable for type checking: fix can only be implemented by pure λ-terms that are not

well typed ⇝ any recursive Haskell-program would be compiled into a non-well-typed pure λ-term.