02157 Functional programming

Michael R. Hansen
DTU Informatics
November 24, 2011

Exercise: Lambda-calculus interpreter

In this exercise you shall make a simple interpreter for the untyped lambda calculus, as introduced in the third lecture on October 28, 2011.

Solve this exercise by completing the program skeleton `lambdaIntpSkeleton.fs` available on the homepage.

A lambda-calculus based language

The set of λ-*terms* or just *terms* Λ is generated from a set V of variables by the rules:

- if $x \in V$, then $x \in \Lambda$ *atom*
- if $x \in V$ and $t \in \Lambda$, then $(\lambda x.t) \in \Lambda$ *abstraction*
- if $t_1, t_2 \in \Lambda$, then $(t_1 t_2) \in \Lambda$ *application*

In addition to that we shall allow *constants* in order to express values and operations in a convenient manner. We will use constants for numbers: $0, 1, \ldots$, truth values: true and false, and operations such as $+, -, \cdot, =$ and a conditional.

The program skeleton contains a type `lambda` so that the following are values of type `lambda`:

- `V "x"` represents the variable x,
- `O "+"` and `O "ite"` represent the operation '+' and the if-then-else conditional, respectively. The uppercase letter '`O`' is the constructor for operations.
- `I 7` and `B false` represent the integer `7` and the truth value false, respectively.
- `L("x", V "x")` represents the abstraction: $\lambda x.x$, and
- `A(L("x", V "x"), I 7)` represents the application $((\lambda x.x)7)$.

Free and bound variables

An occurrence of a variable x is *bound* in a lambda term t, if x occurs within the scope of an abstraction $\lambda x.M$ in t; otherwise x is *free* in t.

Declare a function `free: lambda -> Set<string>`, which extracts the set of free variables from a lambda term.

Substitution

Declare a function `subst` t x t' for substituting all free occurrences of a variable x in a term t with the term t'. This substitution is written $t[t'/x]$ on the slides.

This function should rename bound variables to *avoid clashes*, i.e. a situation where a free variables of t' becomes bound by an abstraction in t. Please consult the slides for examples. The program skeleton on file sharing contains a function `nextVar` which can generate a new, fresh variable name to be used when renaming a bound variable.

Normal order reduction

By a *redex* we shall understand a term which can be reduced by a beta-reduction or a term which can be reduced by applying an operation to values. We shall consider the following redeces and reductions:

- The redex: $((=, a), b)$ is reduced to the truth value $a = b$, where a and b are integers. Similarly for other relations such as $>$ and $\geq$.
- The redex: $((+, a), b)$ is reduced to $a + b$, where a and b are integers. Similarly for other operations such as $-$ and $\cdot$.
- The redex: $(((\texttt{ite}, \text{true}), t_1), t_2)$ is reduced to t_1.
- The redex: $(((\texttt{ite}, \text{false}), t_1), t_2)$ is reduced to t_2.
- The redex: $((\lambda x.t), t')$ is reduced to $t[t'/x]$. This is a beta-reduction.

Examples of reductions, using the F# representation of terms, are:

1. `A(A(O "=", I 1), I 2)` reduces to `(B false)`.
2. For `A(A(O "+", I 1), I 2)` reduces to `(I 3)`.

You should implement the *normal-order reduction* strategy in the interpreter. In this strategy the leftmost, outermost redex (the same as the textually leftmost redex) is chosen at each reduction step until no further reduction is possible.

If a term can be reduced to a *normal form*, i.e. a term containing no redeces, then the normal-order reduction will terminate. "Lazy" functional languages like Haskell are based on such a strategy.

Remark: The "eager" language F# is based on *applicative-order reduction*, where the inner, leftmost redex is reduced first. In such a strategy an argument to a function is evaluated just once; but reduction may fail to terminate even in cases where a normal form exists. We will not consider applicative-order reductions here.

- Declare a function `red: lambda → lambda option` that reduces the leftmost, outermost redex (the same as the textually leftmost redex) of a term, if such redex exists. This function makes at most **one** reduction. The result is None if the term has no redex.
- Declare a function `reduce: lambda → lambda` for the normal-order reduction strategy.

Examples

- Test your interpreter on a few simple examples.
- Make a declaration of the fixpoint combinator due to Turing:

$$Y_t = (\lambda x.\lambda y.y\,(x\,x\,y))\,(\lambda x.\lambda y.y\,(x\,x\,y))$$

- Make a declaration for the function: $F = \lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * f(n-1)$.
- Compute $n!$ on some examples using the interpreter.