DTU Compute Lyngby 17-11-20
Michael R. Hansen

Ongoing exercise: An interpreter for Lambda Calculus

In this exercise you shall make a simple interpreter for the untyped lambda calculus, as introduced the lecture on November 20, 2020. The slide set for this lecture supports the reading of this exercise.

Lambda Calculus was introduced by Alonzo Church in the 1930s in his research within the foundations of mathematics. Lambda Calculus comprises a universal model of computation and constitute the foundation of functional programming languages.

A goal of this exercise is to present few basic concepts and indicate that they suffice to express computations, including those of recursive programs. Another goal is to study step-by-step semantics that, in this case, is expressed using substitution.

This exercise can be solve by completing the program skeleton `lambdaIntpSkeleton.fsx` available on FileSharing.

Untyped Lambda Calculus

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*

The focus of the first part is to cover semantics in the form of *normal-order reduction* of terms, including notions of free and bound variables, substitution and recursion using a fixpoint combinator.

To focus on that we allow special constants to deal with numbers, truth values, operations on integers and an if-then-else construction. These special constants will be replaced by terms in Part 2 hinting at the expressive power of the above three constructs of "pure" lambda calculus.

Part 1: A language having special constants

In this part we introduce *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.
- `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 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>` to extract the set of free variables from a 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*, that is, a situation where a free variable of t' becomes bound by an abstraction in t. Please consult the slides for examples. The program skeleton contains a function `nextVar` that generates a new, fresh variable name to be used when renaming a bound variable.

Normal-order reduction

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

- The redex: $((\lambda x.t)\, t')$ is reduced to $t[t'/x]$. This is a beta-reduction.
- 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: $(((\texttt{ite}\, \mathrm{true})\, t_1)\, t_2)$ is reduced to t_1.
- The redex: $(((\texttt{ite}\, \mathrm{false})\, t_1)\, t_2)$ is reduced to t_2.

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

1. `A(L("y",A(V "y",B True)),L("x",V "x"))` reduces to `A(L("x",V "x"),B True)`.
2. `A(A(O "=", I 1), I 2)` reduces to `B false`.
3. For `A(A(O "+", I 1), I 2)` reduces to `I 3`.

You should implement the *normal-order reduction* strategy. 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 is reduced to a *normal form*, i.e. a term containing no redeces, then a value is obtained and the normal-order reduction terminates. "Lazy" functional languages like Haskell are based on such a strategy.[1]

- Declare a function `red: Lambda` $\rightarrow$ `Lambda option` that reduces the leftmost, outermost 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` $\rightarrow$ `Lambda` implementing the normal-order reduction strategy.

[1]The "eager" language F# is based on *applicative-order reduction*, where the inner, leftmost redex is reduced first. In this strategy an argument to a function is evaluated just once; but some reductions fail to terminate even though normal forms exist. We will not consider applicative-order reductions here.

Examples

- Test your interpreter on a few simple examples.
- Show that the fixed-point combinator due to Turing:

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

  is indeed a fixed-point combinator under normal-order reduction, that is, show $Y_t\ f = f(Y_t\ f)$.
- Make a declaration for Y_t.
- 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.

Part 2: A pure language

In this part we shall encode natural numbers and Boolean values with a few operations, and a conditional as terms without the use of extra constants.

This will illustrate basic techniques and will hint at the expressive power of lambda calculus.

We stick to a minimum of encoding needed to compute factorials. An entry point for further studies is `https://en.wikipedia.org/wiki/Lambda_calculus`.

The parts of the interpreter you have already completed remain as they are. That is, in Part 2 you extend your solution to Part 1.

Church numerals

Natural numbers are represented by *Church numerals*:

$$\begin{array}{cccccc} \overline{0} & \overline{1} & \overline{2} & \overline{3} & \cdots & \overline{n} \\ \lambda f.\lambda x.x & \lambda f.\lambda x.f\,x & \lambda f.\lambda x.f(f\,x) & \lambda f.\lambda x.f(f(f\,x)) & \ldots & \lambda f.\lambda x.f^n\,x \end{array}$$

where $f^0x = x$, $f^{i+1}x = f^i(fx)$ and $\overline{n}$ denote the Church numeral for the natural number n. The main idea is to use a unary representation of numbers. The Church numeral for 3 has, for example, three applications of f. It is an inefficient representation – but it works.

The following are terms for the successor, predecessor and addition functions:

- $\mathit{succ} = \lambda n.\lambda f.\lambda x.n\,f\,(f\,x)$
- $\mathit{pred} = \lambda n.\lambda f.\lambda x.n\,(\lambda g.\lambda h.h\,(g\,f))\,(\lambda u.x)\,(\lambda u.u)$
- $\mathrm{add} = \lambda m.\lambda n.\lambda f.\lambda x.m\,f\,(n\,f\,x)$

The (incomprehensible) predecessor function should satisfy:

$$\mathrm{pred}\,\overline{n} = \begin{cases} \overline{0} & \text{if } n = 0 \\ \overline{n-1} & \text{if } n > 0 \end{cases}$$

There are already terms for a few Church numerals and for succ, pred and add in the skeleton file.

1. Declare a function `toInt` $\overline{n} = n$ that converts the Church numerals to natural numbers. The functions should raise an exception if the argument is not a Church numeral.

   The declaration may consider only the form of terms; but it may also take into account that terms are equal up to renaming of bound variables. For example, $\lambda f.\lambda x.f\,x$ and $\lambda g.\lambda y.g\,y$ are equivalent terms (for the Church Numeral $\overline{1}$), as one can be obtained from the other by renaming of bound variables (also called α-conversion).

2. Test on a few examples the above property of pred and the properties

$$\mathrm{succ}\,\overline{n} = \overline{n+1}$$
$$\mathrm{add}\,\overline{m}\,\overline{n} = \overline{m+n}$$

3. Declare a term mult so that $\mathrm{mult}\,\overline{m}\,\overline{n} = \overline{m \cdot n}$.

Church Booleans

The following two terms can be used for the Boolean values true and false:

$$\begin{aligned} \text{True} &= \lambda x.\lambda y.x \\ \text{False} &= \lambda x.\lambda y.y \end{aligned}$$

Notice that False is equivalent to the Church numeral $\overline{0}$.

Furthermore, the predicate IsZero $n = n = 0$ is represented by the term:

$$\text{IsZero} = \lambda n.n\,(\lambda x.\text{False})\,\text{True}$$

and a term for if-then-else is given by:

$$\text{ITE} = \lambda p.\lambda x.\lambda y.\,p\,x\,y$$

To see how this machinery works, terms for disjunction and negation, for example, should be introduced and truth tables should be checked using the chosen representation. This part is not included here.

The above terms are already in the skeleton file and you should (just) test that

5. IsZero $\overline{0}$ gives the Church Boolean for true,
6. IsZero $\overline{n}$ gives the Church Boolean for false when $n > 0$,
7. ITE True t_1 t_2 gives t_1 and
8. ITE False t_1 t_2 gives t_2.

Furthermore, compute factorials as follows:

9. Make a new declaration for $F = \lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * f(n-1)$ using the above Church representations.
10. Compute $n!$ on some examples. Observe the inefficiency when having unary representation of natural numbers.