Interpreters - Part I
Substitutions

COMP2209 - Programming III

Dr Julian Rathke
Writing an interpreter in Haskell

• In this lecture we are going to use Haskell to write an interpreter for the language at the very core of Haskell
• Doing this benefits in the following ways
  • We get to understand the concept of binding and evaluation in Haskell much better
  • We get to see how to use Abstract Syntax Trees to write interpreters.
• We will start by writing an inefficient, but instructive, solution using substitutions of terms but will then refine this in to a more efficient machine like solution.
• The language we will write an interpreter for is a deceptively simple one called the lambda calculus.
The Lambda Calculus

• Lambda Calculus can be considered as the simplest "interesting" programming language.

• In its purest form it contains just two primitives: function abstraction and application

• It is the foundation of all functional programming languages (OCaml, Haskell, Scheme etc)

The grammar of the language is simply:

\[ E :: = x \mid \lambda x \rightarrow E \mid E E \]

We could add some constant values (e.g. bools, ints, floats etc) and operations on them but for this lecture we will not need to.
Lambda Power

• The Lambda Calculus is surprisingly expressive
• It is Turing complete - all computation can be expressed in it
• It can encode arithmetic, logic etc.
• It can encode structured data
• For example look at this encoding of pairs:

\(( \mathit{e}_1, \mathit{e}_2)\) is encoded as \(\lambda \mathit{v} \to \mathit{v} \ \mathit{e}_1 \ \mathit{e}_2\)

\(\mathsf{fst} \ \mathit{e}\) is encoded as \(\mathit{e} \ (\lambda \mathit{x} \to \lambda \mathit{y} \to \mathit{x})\)

\(\mathsf{snd} \ \mathit{e}\) is encoded as \(\mathit{e} \ (\lambda \mathit{x} \to \lambda \mathit{y} \to \mathit{y})\)

Think of a pair structure a bit like a server and the \(\mathsf{fst}/\mathsf{snd}\) functions as clients that follow a protocol to fetch the data
Syntactic Conventions and Terminology

- Applications always associate to the left
  - $e_1 e_2 e_3$ is read as $(e_1 e_2) e_3$ rather than $e_1 (e_2 e_3)$
- Abstraction extends as far to the right as possible
  - $\lambda x \rightarrow \lambda y \rightarrow x y$ is read as $\lambda x \rightarrow (\lambda y \rightarrow x y)$ rather than $\lambda x \rightarrow (\lambda y \rightarrow x) y$
- We say that a variable $x$ is **bound** in the term $\lambda x \rightarrow e$ and that the **scope** of this binding is term expression $e$
- A variable $x$ in a term $e$ that does not appear in the scope of any abstraction $\lambda x \rightarrow$ is referred to as **free** in $e$.
- e.g. $x$, $y$ are bound and $z$ is free in $\lambda x \rightarrow \lambda y \rightarrow x y z$
- A lambda calculus term that has no **free** variables is called **closed**
Interpreters and Evaluation

• We learned about evaluation and evaluation strategies in the previous lecture.

• In order to implement an interpreter for a language we need to follow a particular evaluation strategy.

• We need to identify the redexes in terms of the language and when there are no more redexes available.

• We need to reduce the redexes by simplifying terms according to some reduction rule.

• The only rule we need for the lambda calculus is in fact the beta-reduction rule.
Beta-Reduction Redux

• We introduced the concept of beta-reduction in the previous lecture.

• To reduce the redex \((\lambda x \to e_1) \ e_2\) we must replace every free occurrence of the variable \(x\) inside \(e_1\) with \(e_2\).

• Technically, this is much more complicated than that: let’s have some examples to see why.

• First we’ll write the beta-reduction rule using a substitution operator and we’ll write the “reduces to” relation as an arrow

\[
(\lambda x \to e_1) \ e_2 \leftrightarrow e_1 [ \ x := e_2 ]
\]

Here, \(e_1 [ \ x := e_2 ]\) is an operation on Abstract Syntax Trees understood to mean (roughly) “replace all leaf nodes named variable \(x\) with the subtree \(e_2\)

Let’s try to define this operator
Substitution - Take One

- First, let’s write a data type for ASTs for lambda calculus

```haskell
data Expr = Var String | Lam String Expr | App Expr Expr
  deriving (Eq, Show, Read)
```

Here is an incorrect attempt to define substitution: replace the leaf node that matches the variable name and push the substitution through the tree otherwise.

```haskell
subst :: Expr → String → Expr → Expr
subst (Var x) y e | x == y = e
subst (Var x) y e | x /= y = Var x
subst (Lam x e) y e = Lam x (subst e y e)
subst (App e1 e2) y e = App (subst e1 y e) (subst e2 y e)
```

This doesn’t distinguish free and bound variables though. For example, \((\lambda x \to x) [ x := y ]\) should just be \((\lambda x \to x)\) because the \(x\) is bound. The definition above gives \((\lambda x \to y)\)
Substitution - Take Two

• Okay how about this then?

\[
\text{subst} :: \text{Expr} \rightarrow \text{String} \rightarrow \text{Expr} \rightarrow \text{Expr}
\]

\[
\text{subst} (\text{Var } x) \ y \ e \ | \ x == y = e
\]

\[
\text{subst} (\text{Var } x) \ y \ e \ | \ x /= y = \text{Var } x
\]

\[
\text{subst} (\text{Lam } x \ e) \ y \ e \ | \ x /= y = \text{Lam } x \ (\text{subst } e \ y \ e)
\]

\[
\text{subst} (\text{Lam } x \ e) \ y \ e \ | \ x == y = \text{Lam } x \ e
\]

\[
\text{subst} (\text{App } e \ e_2) \ y \ e = \text{App } (\text{subst } e_1 \ y \ e) \ (\text{subst } e_2 \ y \ e)
\]

No, this doesn’t work either. Consider

\[
(\lambda y . x) [ x := y ]
\]

Using the above definition, this would return

\[
(\lambda y . y)
\]

which is not what we want.

This is known as Variable Capture - a free variable of the value being substituted is being bound by the \(\lambda\) term being substituted in to.
Substitution - Take Three

\[
\text{subst} :: \text{Expr} \rightarrow \text{String} \rightarrow \text{Expr} \rightarrow \text{Expr}
\]

\[
\text{subst} (\text{Var} \ x) \ y \ e \ | \ x == y = e
\]

\[
\text{subst} (\text{Var} \ x) \ y \ e \ | \ x /= y = \text{Var} \ x
\]

\[
\text{subst} (\text{Lam} \ x \ e_1) \ y \ e \ | \ x /= y \land \neg \text{free} \ x \ e = \text{Lam} \ x \ (\text{subst} \ e_1 \ y \ e)
\]

\[
\text{subst} (\text{Lam} \ x \ e_1) \ y \ e \ | \ x == y = \text{Lam} \ x \ e_1
\]

\[
\text{subst} (\text{App} \ e_1 \ e_2) \ y \ e = \text{App} \ (\text{subst} \ e_1 \ y \ e) \ (\text{subst} \ e_2 \ y \ e)
\]

where

\[
\text{free} :: \text{String} \rightarrow \text{Expr} \rightarrow \text{Bool}
\]

\[
\text{free} \ x \ (\text{Var} \ y) = x == y
\]

\[
\text{free} \ x \ (\text{Lam} \ y \ e) \ | \ x == y = \text{False}
\]

\[
\text{free} \ x \ (\text{Lam} \ y \ e) \ | \ x /= y = \text{free} \ x \ e
\]

\[
\text{free} \ x \ (\text{App} \ e_1 \ e_2) = (\text{free} \ x \ e_1) \| (\text{free} \ x \ e_2)
\]

This is correct, but unfortunately only a \textit{partial} function
Alpha-Conversion

• Although the substitution on the previous slide is partial we can still use it on all terms.

• Consider \((\lambda y \rightarrow x) \ [\ x := y \ ]\) again

• The definition above is not defined in this case.

• However, if we rewrite this as \((\lambda z \rightarrow x) \ [\ x := y \ ]\) then the substitution is defined (and equals \(\lambda z \rightarrow y\)).

• Are we allowed to do this? Yes, bound names are renamable!

• Renaming a bound name throughout a term is known as **alpha-conversion** and can be defined in terms of substitution.

• Any \(\lambda\)-term \(\lambda x \rightarrow e\) is alpha convertible to \(\lambda y \rightarrow e \ [\ x := y \ ]\) where this is defined.

• We can alpha-convert any subterm of a larger term also.
Substitution - Take Four

\[ subst :: Expr \rightarrow String \rightarrow Expr \rightarrow Expr \]

\[ subst (Var x) y e | x == y = e \]

\[ subst (Var x) y e | x /= y = Var x \]

\[ subst (Lam x e_1) y e | x /= y && not (free x e) = Lam x (subst e_1 y e) \]

\[ subst (Lam x e_1) y e | x /= y && (free x e) = let x' = rename x in subst (Lam x' (subst e_1 x (Var x'))) y e \]

\[ subst (Lam x e_1) y e | x == y = Lam x e_1 \]

\[ subst (App e_1 e_2) y e = App (subst e_1 y e) (subst e_2 y e) \]

\[ rename x = x +"""" \]
My first $\lambda$ interpreter

- We are ready to write our single step evaluation function
- We will use a call-by-name evaluation strategy
- This is defined to work for closed $\lambda$ expressions
- If there are no redexes the function simply returns the input expression

```haskell
eval1cbn :: Expr → Expr
eval1cbn (Lam x e) = (Lam x e)
 eval1cbn (App (Lam x e1) e2) = subst e1 x e2
 eval1cbn (App e1 e2) = App (eval1cbn e1) e2
```

no reduction under $\lambda$

beta-reduction

outermost reduction
Multiple steps

- We can iterate this single step evaluation function and record the list of pairs of expressions such that $e \rightarrow e'$

```haskell
reductionscbn :: Expr -> [(Expr, Expr)]
reductionscbn e = [p | p <- zip evals (tail evals)]
  where evals = iterate eval1cbn e
```

- Using this we can then find the first element where evaluating further makes no difference

```haskell
evalcbn = fst . head . dropWhile (uncurry (/=)) . reductionscbn
```

- Or we can create a trace of evaluation steps as a list of expressions that the given term evaluates to before terminating

```haskell
tracecbn = (map fst) . takeWhile (uncurry (/=)) . reductionscbn
```
Call-By-Value - single step evaluation

- Fortunately, it is now very easy to modify the above interpreter to implement a call-by-value evaluation strategy:

\[
\begin{align*}
\text{eval1cbv} &:: \text{Expr} \rightarrow \text{Expr} \\
\text{eval1cbv} \ (\text{Lam} \ x \ e) & = (\text{Lam} \ x \ e) \\
\text{eval1cbv} \ (\text{App} \ (\text{Lam} \ x \ e1) \ e@(\text{Lam} \ y \ e2)) & = \text{subst} \ e1 \ x \ e \\
\text{eval1cbv} \ (\text{App} \ e@(\text{Lam} \ x \ e1) \ e2) & = \text{App} \ e \ (\text{eval1cbv} \ e2) \\
\text{eval1cbv} \ (\text{App} \ e1 \ e2) & = \text{App} \ (\text{eval1cbv} \ e1) \ e2
\end{align*}
\]

Only beta-reduce if both function and argument are evaluated

Otherwise, evaluate the argument to the function
Multistep evaluation - done parametrically

- Of course, we can use a higher-order function to make a parametric version of the multistep evaluation function
- We pass the single step evaluation function in as an argument!

```haskell
reductions :: (Expr → Expr) → (Expr → [(Expr, Expr)])
reductions ssev e = [ p | p <- zip evals (tail evals) ]
  where evals = iterate ssev e
```

```haskell
eval ssev = fst . head . dropWhile (uncurry (/=)) . reductions ssev
```

```haskell
trace ssev = (map fst) . takeWhile (uncurry (/=)) . reductions ssev
```

`eval cbn = eval eval1 cbn`

`trace cbn = trace eval1 cbn`

`eval cbv = eval eval1 cbv`

`trace cbv = trace eval1 cbv`
YOUR QUESTIONS

Next Lecture:
Interpreters Part II - Machines