Interpreters - Part II
Machines

COMP2209 - Programming III

Dr Julian Rathke
Is that really how you write an interpreter?

- No.
- Using substitutions to interpret λ-calculus is useful as a means of defining a reference semantics as it is straightforward to implement.
- It is horribly inefficient though.
- After repeated substitutions, the abstract syntax trees representing terms can become very large.
- Note that, for any given reduction we only focus on one small subtree of a potentially much larger term - it is inefficient to construct large trees to then have to search them for redexes.
- A smarter approach would be to keep track of the exact term we need to reduce next and keep a data structure that slowly builds up the larger term as we need it.
For example

\( \text{grow} = \lambda \ x \ \rightarrow \ x \ x \ x \)

Although this is a very contrived example to show the idea, notice that at each step the part of the term involved is a very small specific subterm.

We can think of splitting the term up into “the bit that needs evaluating next” and “the rest of the term that needs evaluating later”
Beta-Reduction with lazy substitution

• Consider the beta-reduction rule again:

\[(\lambda x \rightarrow e_1) \ e_2 \mapsto e_1 \ [ \ x := e_2 \] \]

• Suppose we implement this rule by delaying any substitution until it is needed.

• We could keep some sort of record of that substitutions that we need to make and just look them up in the list when needed.

• This is precisely the concept of an environment

• An environment records bindings of variables to expressions.

• It is a partial function from variable names to expressions but we can implement this using an association list
Beta-reduction - using an environment

- Let’s rewrite the beta-reduction rule slightly to allow for this idea of an environment.

- We’ll write $e | E$ to represent an expression $e$ in the environment $E$.

- We have $(\lambda x \rightarrow e_1) e_2 | E \mapsto e_1 | E [ x := e_2 ]$

- Where $E [ X := e ]$ means environment $E$ updated with the new binding of $x$ to the expression $e$.

- Note, however that the expression on the lhs of $|$ is no longer a closed expression.

- $e_1$ may contain free occurrences of $x$

- This means that we need to consider open terms in order to describe reductions too.

- Having a rule that says something like seems necessary

\[
x | E \mapsto \text{“lookup x in } E\text{”} | E
\]
Achieving Focus - Call-by-Value

- The next improvement to the interpreter we suggested is to only focus on the subterm that needs evaluating next.
- **Everything that follows is tailored to a Call-by-Value strategy**
- Consider where redexes live in the $\lambda$ calculus
  - A $\lambda$ term is terminated - it has no redexes
  - So that only leaves applications $e_1 \ e_2$
- Consider the case $(\lambda x \to e_1) \ e_2$ where the function is evaluated
  - In this case, the focus lies in $e_2$ and the “rest” of the expression is $(\lambda x \to e_1) [-]$  
- Consider the case $e_1 \ e_2$ where $e_1$ is not a function
  - In this case, the focus lies in $e_1$ and the “rest” of the expression is $[-] \ e_2$

[-] is a hole into which we can plug an expression.
Frames and Continuations

- This suggests that we could usefully introduce terms that represent the “rest” of the program.

- A grammar for these is

\[
V ::= (\lambda \, x \rightarrow e) \\
F ::= V \, [\cdot] \mid [\cdot] \, e
\]

- We call these terms **frames**.

- As we evaluate a \( \lambda \) term we will identify a number of frames by focussing on smaller and smaller subtrees until we find the next redex.

- We will collect the frames that we identify in a stack.

- A stack of frames is what we call a **continuation**.

- You can think of it very much like a call stack in Java - it’s where the control returns to when the current expression terminates.
Pushing Continuations

• When do we push a new frame on to the stack?
• This depends on the evaluation strategy, were using Call-by-Value
• Let’s write our reductions as follows:

\[
\begin{align*}
  e \mid E \mid K & \rightarrow e' \mid E' \mid K' \\
  e_1 \ e_2 \mid E \mid K & \rightarrow e_1 \mid E \mid ([-] e_2) : : K
\end{align*}
\]

• We see that we need to push a frame when we have an unevaluated expression in function position
• And we need to pop a frames when the focussed expression is terminated. This depends on what is actually at the top of the continuation stack though.
Popping Continuations

• If the focussed expression is terminated, let’s write V for this then there are three possibilities for the continuation stack
  • K is empty.
  Then the evaluation has terminated
  • K has [-] e at the top
  If we are using CBV then we must now focus on e and push a replacement frame

\[
V | E | [-]e :: K \mapsto e | E | V [-] : K
\]

• K has (\lambda x \to e) [-] at the top
  Then we must actually perform the beta-reduction

\[
V | E | (\lambda x \to e) [-] :: K \mapsto e | E [x := V] | K
\]
Collecting this all together

• Let’s write all these ideas together as one set of rules.
• This style of reduction rules using an environment and a continuation is actually called a CEK-Machine.
  • Control, Environment and Continuation.
• They are developed from SECD machines (Landin 1964) by Felleisen and Friedman (1985)

\[
\begin{align*}
x & \mid E \mid K \leftrightarrow \text{“lookup } x \text{ in } E\text{”} \mid E \mid K \\
 e_1 \ e_2 & \mid E \mid K \leftrightarrow e_1 \mid E \mid (\ULL e_2) :: K \\
 V \mid E \mid \ULL e & :: K \leftrightarrow e \mid E \mid V \ [\ULL] :: K \\
 V \mid E \mid (\lambda x \to e) \ [\ULL] & :: K \leftrightarrow e \mid E \ [x := V] \mid K
\end{align*}
\]

Now, I’d like to say, “That’s all folks” but unfortunately there is a problem with the above rules
Nested Functions

- The problem lies in the use of environments instead of substitutions.
- Consider the following λ term and try keep track of what the environment is during evaluation:

\[(\lambda z \rightarrow \lambda x \rightarrow (\lambda y \rightarrow y z \ x \ y) \ (\lambda x \rightarrow z \ x)) \ e_1 \ e_2\]

For the first occurrence of y, x is pointing to e_2 and for the second y x is pointing to e_1

The binding of x should not change in this function

E is \[z := e_1, x := e_2, y := \lambda x \rightarrow z \ x, x := z\]
Closures

• We see that when a function is used as a value, it is important to keep track of what the bindings of its free variables are - at the point of use.

• This means, that we need some way to think of functions as closed entities when they terminate.

• We can do this by simply pairing the function up with the bindings in the current environment.

• This pair of a function and its current bindings is what we call a closure - we’ll write \( \text{cl}(\lambda x \rightarrow e, E) \) for these.

• By passing around closures rather than functions we can rewrite the CEK machine rules to resolve this binding problem.

• This issue of closure is dealt with automatically using substitutions - which is why they are easier to follow.
CEK-Machine Redux

First we rewrite the stack frame grammar etc to use closures:

\[ W ::= \text{cl}(\lambda x \to e, E) \]
\[ F ::= W [-] | [-] e E \]
\[ K ::= [] | F ::= K \]
\[ E ::= \emptyset | E [x:=W] \]

Then we rewrite the CEK rules:

R1: \( x \mid E_1 \mid K \to \lambda x \to e \mid E_2 \mid K \)

where lookup \( x \) in \( E_1 \) is \( \text{cl}(\lambda x \to e, E_2) \)

R2: \( e_1 e_2 \mid E \mid K \to e_1 \mid E \mid ([-] e_2 E) :: K \)

R3: \( \lambda x \to e \mid E \mid K \to \text{cl}(\lambda x \to e, E) \mid E \mid K \)

R4: \( W \mid E_1 \mid [-] e E_2 :: K \to e \mid E_2 \mid W [-] :: K \)

R5: \( W \mid E_1 \mid \text{cl}(\lambda x \to e, E_2) [-] :: K \to e \mid E_2 [x := W] \mid K \)

Push the current environment too

Restore the environment on lookup

Make a closure from a terminated value

Restore the previous environment when popping
Example CEK Reduction Sequence

• Let’s evaluate the term \((\lambda x \to \lambda y \to x)\ e_1\ e_2\) using CBV

• Suppose that \(e_1\), \(e_2\) are terminated values

\[
\begin{align*}
(\lambda x \to \lambda y \to x)\ e_1\ e_2 & | \emptyset | [] \\
R2 \mapsto (\lambda x \to \lambda y \to x)\ e_1 & | \emptyset | ([\ldots]e_2\ \emptyset) :: [] \\
R2 \mapsto (\lambda x \to \lambda y \to x) & | \emptyset | ([\ldots] e_1\emptyset) :: ([\ldots]e_2\ \emptyset) :: [] \\
R3 \mapsto \text{cl}(\lambda x \to \lambda y \to x, \emptyset) & | \emptyset | ([\ldots] e_1\emptyset) :: ([\ldots]e_2\ \emptyset) :: [] \\
R4 \mapsto e_1 & | \emptyset | (\text{cl}(\lambda x \to \lambda y \to x, \emptyset)[\ldots]) :: ([\ldots]e_2\ \emptyset) :: [] \\
R5 \mapsto \lambda y \to x & | [x := e_1] | ([\ldots]e_2\ \emptyset) :: [] \\
R3 \mapsto \text{cl}(\lambda y \to x, [x := e_1]) & | [x := e_1] | ([\ldots]e_2\ \emptyset) :: [] \\
R4 \mapsto e_2 & | \emptyset | (\text{cl}(\lambda y \to x, [x := e_1])[\ldots]) :: [] \\
R5 \mapsto x & | [x := e_1, y := e_2] | [] \\
R1 \mapsto e_1 & | [x := e_1, y := e_2] | []
\end{align*}
\]

Done
Comments on CEK Machines

• There are several advantages of using CEK machines in favour of substitutions when writing a \( \lambda \) interpreter
  • Efficiency is the primary reason
  • It is easy to add extra language features that involve control flow - e.g. exceptions
  • It is easy to get a stack trace during execution
• This here isn’t the most efficient version of a CEK machine - we haven’t implemented graph reduction (sharing) yet.
• Closures (or lack of them) are a common issue in mainstream programming languages - think Java and scoping rules for inner classes, or C# delegates
• The equivalent machine for a Call-by-Name strategy is actually a little simpler and is historically known as a Krivine Machine. (cf. next slide for reference).
For reference: The Krivine Machine

An abstract machine for Call-by-Name reduction in the \( \lambda \) calculus

\[
\begin{align*}
\text{C} &::= <e, E> \\
K &::= [] | C :: K \\
E &::= \emptyset | E [x:=C]
\end{align*}
\]

R1: \( x \mid E_1 \mid K \rightsquigarrow e \mid E_2 \mid K \)

where lookup \( x \) in \( E_1 \) is \( <e, E_2> \)

R2: \( e_1 e_2 \mid E \mid K \rightsquigarrow e_1 \mid E \mid (<e_2, E>) :: K \)

R3: \( \lambda x \rightarrow e \mid E \mid C :: K \rightsquigarrow e \mid E[x:=C] \mid K \)
YOUR QUESTIONS

Next Lecture:
Input/Output in Haskell