3 Scoping Issues

We motivate the results of §4 with an example illustrating how dynamic scoping arises from a naive implementation of lazy substitution and how capsules and closures remedy the situation.

3.1 The λ-Calculus

The oldest and simplest of all functional languages is the λ-calculus. In this system, a state is a closed λ-term, and computation consists of a sequence of β-reductions

\[(\lambda x.d) e \rightarrow d[x/e],\]

where \(d[x/e]\) denotes the safe substitution of \(e\) for all free occurrences of \(x\) in \(d\). Safe substitution means that bound variables in \(d\) may have to be renamed (α-converted) to avoid capturing free variables of the substituted term \(e\).

For example, consider the closed λ-term \((\lambda y.(\lambda z.y.z 4) \lambda x.y) 3 2\). Evaluating this term in applicative order\(^3\), we get the following sequence of terms leading to the value 3:

\[(\lambda y.(\lambda z.y.z 4) \lambda x.y) 3 2 \rightarrow (\lambda z.y.z 4) (\lambda x.3) 2 \rightarrow (\lambda y.(\lambda x.3) 4) 2 \rightarrow (\lambda x.3) 4 \rightarrow 3\]  (1)

\(^3\)Here applicative order, also known as left-to-right call-by-value order, refers to the order of evaluation in which the leftmost innermost redex is reduced first, except that redexes in the scope of binding operators \(\lambda x\) are ineligible for reduction.
No α-conversion was necessary. In fact, it can be shown that no α-conversion is ever necessary with applicative-order evaluation of closed terms.

However, the λ-calculus is confluent, and we may choose a different order of evaluation; but an alternative order may require α-conversion. For example, the following reduction sequence is also valid:

\[(\lambda y.(\lambda z.\lambda y.z 4) \lambda x.y) 3 2 \rightarrow (\lambda y.\lambda w.(\lambda x.y) 4) 3 2 \rightarrow (\lambda w.(\lambda x.3) 4) 2 \rightarrow (\lambda x.3) 4 \rightarrow 3 \quad (2)\]

A change of bound variable was required in the first step to avoid capturing the free occurrence of \(y\) in \(\lambda x.y\) substituted for \(z\). Failure to do this results in the erroneous value 2:

\[(\lambda y.(\lambda z.\lambda y.z 4) \lambda x.y) 3 2 \rightarrow (\lambda y.\lambda y.(\lambda x.y) 4) 3 2 \rightarrow (\lambda y.(\lambda x.y) 4) 2 \rightarrow (\lambda x.2) 4 \rightarrow 2 \quad (3)\]

### 3.2 Dynamic Scoping

In the early development of functional programming, specifically with the language LISp, it was quickly determined that physical substitution is too inefficient because it requires copying. This led to the introduction of *environments*, used to effect lazy substitution. Instead of doing the actual substitution when performing a β-reduction, one can defer the substitution by saving it in an environment, then look up the value when needed.

An *environment* is a partial function \(\sigma : \text{Var} \rightarrow \text{Irred}\) with finite domain. A *state* is a pair \((e, \sigma)\), where \(e\) is the term to be evaluated and \(\sigma\) is an environment with bindings for the free variables in \(e\). Environments need to be updated, which requires a *rebinding operator*

\[\sigma[x/e](y) = \begin{cases} e, & x = y, \\ \sigma(y), & x \neq y \end{cases}\]

Naively implemented, the rules are

\[((\lambda x.d) e, \sigma) \rightarrow (d, \sigma[x/e]) \quad \langle y, \sigma \rangle \rightarrow \langle \sigma(y), \sigma \rangle\]

where the first rule saves the deferred substitution in the environment and the second looks up the value. This is quite easy to implement. Moreover, it stands to reason that if β-reduction in applicative order does not require any α-conversions, then the lazy approach should not either. After all, the same terms are being substituted, just at a later time.

However, this is not the case. In the example above, we obtain the following sequence of states leading to the value 2:

\[
\begin{align*}
(\lambda y.(\lambda z.\lambda y.z 4) \lambda x.y) 3 2 & \rightarrow \langle 3, \sigma \rangle \\
(\lambda z.\lambda y.z 4) \lambda x.y) 2 & \rightarrow \langle \lambda x.y, \sigma \rangle \\
(\lambda y.z 4) 2 & \rightarrow \langle y = 3, z = \lambda x.y \rangle \\
z 4 & \rightarrow \langle y = 2, z = \lambda x.y \rangle \\
(\lambda x.y) 4 & \rightarrow \langle y = 2, z = \lambda x.y \rangle \\
y & \rightarrow \langle y = 2, z = \lambda x.y, x = 4 \rangle \\
2 & \rightarrow \langle y = 2, z = \lambda x.y, x = 4 \rangle 
\end{align*}
\]
The issue is that the lazy approach fails to observe safe substitution. This example effectively performs the deferred substitutions in the order (3) without the change of bound variable. Nevertheless, this was the strategy adopted by early versions of LISP. It was not considered a bug but a feature and was called dynamic scoping.

3.3 Static Scoping with Closures

The semantics of evaluation was brought more in line with the λ-calculus with the introduction of closures, introduced in the language Scheme. Formally, a closure is defined as a pair \( \{ \lambda x.e, \sigma \} \), where the \( \lambda x.e \) is a \( \lambda \)-abstraction and \( \sigma \) is a partial function from variables to values that is used to interpret the free variables of \( \lambda x.e \). When a \( \lambda \)-abstraction is evaluated, it is paired with the environment \( \sigma \) at the point of the evaluation, and the value is the closure \( \{ \lambda x.e, \sigma \} \). Thus we have

\[
\sigma : \text{Var} \rightarrow \text{Val} \quad \text{Val} = \text{Const} + \text{Cl}
\]

where Cl denotes the set of closures. We require that for a closure \( \{ \lambda x.e, \sigma \} \), \( \text{FV}(\lambda x.e) \subseteq \text{dom}\sigma \). Note that the definitions of values and closures are mutually dependent.

The new reduction rules are

\[
\langle \lambda x.d, \sigma \rangle \rightarrow \{ \lambda x.d, \sigma \} \quad \langle \{ \lambda x.d, \sigma \} e, \tau \rangle \rightarrow \langle d, \sigma[x/e] \rangle \quad \langle y, \sigma \rangle \rightarrow \sigma(y).
\]

The second rule says that an application uses the context \( \sigma \) that was in effect when the closure was created, not the context \( \tau \) of the call. Turning to our running example,

\[
\begin{align*}
(\lambda y.(\lambda z.\lambda y.z 4) \lambda x.y) 3 2 & \rightarrow [\ ] \\
(\lambda z.\lambda y.z 4)(\lambda x.y) 2 & \rightarrow [y = 3] \\
(\lambda y.z 4) 2 & \rightarrow [y = 3, z = \{\lambda x.y, [y = 3]\}] \\
z 4 & \rightarrow [y = 2, z = \{\lambda x.y, [y = 3]\}] \\
\{\lambda x.y, [y = 3]\} 4 & \rightarrow [y = 2, z = \{\lambda x.y, [y = 3]\}] \\
(\lambda x.y) 4 & \rightarrow [y = 3] \\
y & \rightarrow [y = 3, x = 4] \\
3 & \rightarrow [y = 3, x = 4]
\end{align*}
\]

3.4 Static Scoping with Capsules

Closures correctly capture the semantics of β-reduction with safe substitution, but at the expense of introducing a rather involved combinatorial notion of state. Capsules allow us to revert to a more algebraic framework without losing the benefits of closures.

Capsules were defined formally in §2.1. The reduction rules for capsules are

\[
\begin{align*}
\langle (\lambda x.e) v, \sigma \rangle & \rightarrow \langle e[x/y], \sigma[y/v] \rangle \quad \text{(y fresh)} \\
\langle y, \sigma \rangle & \rightarrow \langle \sigma(y), \sigma \rangle
\end{align*}
\]
The key difference is the introduction of the fresh variable $y$ in the application rule. This is tantamount to performing an $a$-conversion on the parameter of a function just before applying it. Turning to our running example, we see that this approach gives the correct result.

\[
\begin{align*}
(\lambda y.(\lambda z.\lambda y.z \ 4) \ \lambda x.y) & \; 3 \ 2 & \ [\ ] \\
(\lambda z.\lambda y.z \ 4) \ (\lambda x.y') & \; 2 & \ [y' = 3] \\
(\lambda y.z' \ 4) & \ & \ [y' = 3, \ z' = \lambda x.y'] \\
z' & \ 4 & \ [y' = 3, \ z' = \lambda x.y', \ y'' = 2] \\
(\lambda x.y') & \ 4 & \ [y' = 3, \ z' = \lambda x.y', \ y'' = 2] \\
y' & \ & \ [y' = 3, \ z' = \lambda x.y', \ y'' = 2, \ x' = 4] \\
3 & \ & \ [y' = 3, \ z' = \lambda x.y', \ y'' = 2, \ x' = 4]
\end{align*}
\]

We prove soundness formally in §4.

4 Soundness

In this section we show that capsules correctly capture static scoping under applicative-order evaluation. We first show that capsules correctly model $\beta$-reduction in the $\lambda$-calculus with safe substitution.

4.1 Evaluation Rules for Capsules

Let $d, e, \ldots$ denote $\lambda$-terms and $u, v, \ldots$ irreducible $\lambda$-terms ($\lambda$-abstractions and constants). Variables are denoted $x, y, \ldots$ and constants $c, f$.

The small-step evaluation rules for capsules consist of reduction rules

\[
\begin{align*}
((\lambda x.e) \ v, \sigma) & \rightarrow (\sigma[\lambda y.\ s / y], \sigma[y / \sigma]) \ (y \ \text{fresh}) \tag{4} \\
(f \ c, \sigma) & \rightarrow (f(c), \sigma) \tag{5} \\
(y, \sigma) & \rightarrow (\sigma(y), \sigma) \tag{6}
\end{align*}
\]

and context rules

\[
\begin{align*}
\langle d, \sigma \rangle & \xrightarrow{\Delta} \langle d', \tau \rangle & \langle e, \sigma \rangle & \xrightarrow{\Delta} \langle e', \tau \rangle \\
\langle d \ e, \sigma \rangle & \xrightarrow{\Delta} \langle d' \ e, \tau \rangle & \langle v \ e, \sigma \rangle & \xrightarrow{\Delta} \langle v \ e', \tau \rangle \tag{7}
\end{align*}
\]

The reduction rules (4)-(6) identify three forms of redex: an application $(\lambda x.e) \ v$, an application $f \ c$ where $f$ and $c$ are constants, or a variable $y \in \text{dom} \ \sigma$. The context rules (7) uniquely identify a redex in a well-typed non-irreducible capsule according to an applicative-order reduction strategy.
Jeannin/Kozen example "Computers with Capsules" (2011)

\[
(\lambda y. (\lambda x. 3 y) \; 3) \; 2
\]

\[\text{Step 1:} \quad (\lambda y. (\lambda x. 3 y) \; 4) \; 2\]

\[\text{Step 2:} \quad (\lambda x. 3) \; 4\]

\[\text{Step 3:} \quad 3\]
Jeannin/Kozen example in abstract syntax.

The applicative order is clear using abstract syntax. We are reducing “left-to-right innermost call by value.”

The outer structure is $ap(ap(f;3);2)$.

\[
ap(ap(lambda(y, ap(lambda(z, ap(lambda(y, ap(3;4)))); lambda(x;y))); lambda(x;3)); lambda(x;3); lambda(x;4); 3)
\]

1. Reduce $ap(ap(lambda(y, ap(lambda(z, ap(lambda(y, ap(3;4)))); lambda(x;3))); lambda(x;3); lambda(x;4); 3)

2. Reduce $ap(lambda(y, ap(lambda(z, ap(lambda(y, ap(3;4)))); lambda(x;3))); lambda(x;3); lambda(x;4); 3)

3. Reduce $ap(lambda(y, ap(lambda(z, ap(lambda(y, ap(3;4)))); lambda(x;3))); lambda(x;4))$

4. Reduce $ap(lambda(x;3); lambda(x;4))$

5. Reduce $lambda(x;3)$

6. Reduce $3$
Example of evaluation in an environment

Jensen/Kogon example

\[ \lambda y. (\lambda z. \lambda y. z y) \lambda x. y \quad 3 \; 2 \quad \text{structure is} \]
\[ \text{ap}(\text{ap}(4; 3); 2) \quad f = \lambda (y. \text{ap}(y; 3)) \]
\[ \text{ap}(\text{ap}(\lambda (y. \text{ap}(\lambda (z. \lambda (y. \text{ap}(z; 4)); \lambda (x. y)); 3)); 2); 2) \quad \text{in \ nil} \]
\[ \text{ap}(\text{ap}(\lambda (z. \lambda (y. \text{ap}(y; 3)); \lambda (x. 3)); 2); 2) \quad \text{in} \quad [y = 3] \]
\[ \text{ap}(\lambda (y. \text{ap}(3; y)); 2) \quad \text{in} \quad [y = 3, 3 = \lambda (x. y)] \]
\[ \text{ap}(3; y) \quad \text{in} \quad [y = 3, 3 = \lambda (x. y)] \]
\[ \text{ap}(\lambda (x. y); 4) \quad \text{in} \quad [y = 3, 3 = \lambda (x. y), x = 4] \]
\[ y \]
\[ 2 \]