

1 Combinators

Environments or closures are complicated and expensive to deal with. Using an environment to bind free variables is an issue that is orthogonal to code generation. Free variables are a hindrance towards compilation since they act as placeholders without giving meaning.

Definition 1 (Combinator) *A combinator is a λ term without free variables.*

If we only have combinators, there is only one primitive operation, application.

Notation

We use capital letters to represent combinators. We use f for bound functions that we can not express in the λ calculus, such as $+$ or $-$ operators. We use \Rightarrow to denote reduction and \rightarrow to denote the application of a transformation rule.

Example

Suppose we have two combinators

$$\begin{aligned}Axy &= x + y \\Mxy &= x \cdot y\end{aligned}$$

If we want to define a combinator F that is equivalent to the function

$$fxyz = (x \cdot z) + (x \cdot z),$$

then we can do this in terms of the A and M combinators using:

$$Fxyz = A(Mxy)(Mxz).$$

In order to remove the “free” variables x , y and z , we define

$$Qxyzuv := x(yzv)(yzv).$$

With this definition, we can thus conclude:

$$\begin{aligned}Fxyz &= QAMxyz \\ \Rightarrow F &= QAM.\end{aligned}$$

1.1 The S and K Combinators

The language of combinators sufficient to express lambda calculus consists of two functions, S and K .

$$\begin{aligned}Sxyz &:= xz(yz) \\ Kxy &:= x\end{aligned}$$

All terms in the λ calculus can be expressed in terms of S and K by binding all free variables to a context created by S and K as shown in the example above. The S and K combinators can thus be used to express to the compiler how to handle the environment. The transformation of a λ term into an equivalent combinator term is thus a step in the compilation that removes environments from the list of problems the compiler must to deal with.

The simplest example

How to implement $\lambda x.x = I$ in terms of combinators?

$$I := SKK \tag{1}$$

We will use the I combinator as a short version of SKK from now on. If we expand I , we get:

$$\begin{aligned} I &\Rightarrow SKK \\ &\Rightarrow \lambda xyz.xz(yz) \\ &\Rightarrow \lambda z.kz(kz) \\ &\Rightarrow \lambda z.z \end{aligned}$$

1.2 Compiling λ calculus to Combinators

We now want to derive an algorithm to compile λ calculus to combinators. We start with the inverse problem, compiling combinators to λ calculus. We define a function M that turns a term written in combinators into λ calculus:

$$\begin{aligned} M[x] &= x \\ M[f] &= f \\ M[I] &= \lambda x.x \\ M[K] &= \lambda x\lambda y.x \\ M[S] &= \lambda x.\lambda y.\lambda z.xz(yz) \\ M[FA] &= M[F]M[A] \end{aligned}$$

To compile λ calculus to combinators, we define the inverse operation $[v]\chi$ which will yield a term equivalent to χ without the free variable v .

Theorem 1

$$M[[x]v] = \lambda x.M[v] \tag{2}$$

We now derive $[x]$ by “inverting” M .

$$[x]y = \begin{cases} I & \text{if } x = y \text{ (variable was free)} \\ y & \text{otherwise (variable was already bound)} \end{cases} \quad (3)$$

$$[x]f = Kf \quad (4)$$

$$[x]I = KI \quad (5)$$

$$[x]K = KK \quad (6)$$

$$[x]S = KS \quad (7)$$

$$[x](FA) = S([x]F)([x]A) \quad (8)$$

1.2.1 Examples

Definition of N , a function to apply the $[x]$ abstraction over lambda terms:

$$\begin{aligned} N[[x]] &= x \\ N[[f]] &= f \\ N[[FA]] &= N[[F]]N[[A]] \\ N[[\lambda x.b]] &= [x].N[[b]] \end{aligned}$$

Example 1:

$$\begin{aligned} &N[[\lambda x. + x 1]] \\ \rightarrow &[x]N[[+ x 1]] \\ \rightarrow &[x](+ x 1) \quad \text{use (8)} \\ \rightarrow &S([x] + x)([x]1) \\ \rightarrow &S((S[x])(+ x)([x]x)([x]1)) \\ \rightarrow &S(S(K+)I)(K1) \quad \text{now expand} \\ \Rightarrow &\lambda z.(S(K+)I)z((K1)z) \\ \Rightarrow &\lambda z.(k + z)(Iz)1 \\ \Rightarrow &\lambda z. + z 1 \end{aligned}$$

Example 2:

$$\begin{aligned} &N[\lambda x.xx] \\ \rightarrow &N[[\lambda x.xx]]N[[\lambda x.xx]] \\ \rightarrow &[x](xx)[x](xx) \\ \rightarrow &S([x]x[x]x)S([x]x[x]x) \\ \rightarrow &SII(SII) \\ \Rightarrow &I(SII)I(SII) \\ \Rightarrow &SII(SII) \end{aligned}$$

So we get the infinite recursion from the lambda term.

Example 3:

$$\begin{aligned} & N[\lambda x.\lambda y.fxy] \\ \longrightarrow & S(SKS)(S(KS)(S(KK)(Kf)))(S(KK)I)(KI) \end{aligned}$$

1.3 How to use this for compilation?

Free variables are hard to compile. Compiler writers can easily handle operators in the mathematical sense. Using combinators, we can remove free variables and compile without these obstacles. Using just the S and K combinators, the code size can grow dramatically (as example 3 demonstrates). Early research has focused on using additional combinators that were more expressive to reduce the code-size. At the end, it all was not really practical anyway.

1.4 References

```
@book(Stansifier95,  
      title = "The Study of Programming Languages",  
      author = "Ryan Stansifier",  
      publisher = "Prentice-Hall",  
      year = 1995)
```

```
@book(Gordon88,  
      title = "Programming Language Theory and Implementation",  
      author = "Michael Gordon",  
      publisher = "Prentice-Hall",  
      year = 1988)
```