Comp 411
Principles of Programming Languages
Lecture 14
Eliminating Lambda Using Combinators

Corky Cartwright

February 13, 2023

How to Eliminate `lambda` (`map` in Jam)

Goal: devise a few combinators (functions expressed as λ-abstractions with no free variables) that enable us to express all **λ**-expressions without explicitly using **λ**.

Core Idea: let $\lambda^{*}x.M$ denote an occurrence of $\lambda x.M$ that will be *converted* to an equivalent syntactic form eliminating λ^{*}. Then

$\lambda^{*}x.x \rightarrow I$ (where $I = \lambda x.x$)

$\lambda^{*}x.y \rightarrow Ky$ (where $K = \lambda y.\lambda x.y$)

$\lambda^{*}x.(M\ N) \rightarrow S(\lambda^{*}x.M)(\lambda^{*}x.N)$

(where $S = \lambda x.\lambda y.\lambda z.((x\ z)(y\ z))$)

Note that the second and third rules are sound if we add constants to the language and treat constants and free variables uniformly. In the second rule y can be a constant and in the third rule, M and N can contain constants. Of course, in the first rule, the body x must exactly match the abstracting variable x.

How to Eliminate `lambda` (`map` in Jam) cont.

Question: Where did `S` come from?

- Intuition: it falls out when we formulate the translation to combinatory form *using structural recursion* on the abstract syntax of **λ**-expressions.
- The first two cases on the preceding slide do not involve recursion.
- In the third case, the form of the "magic" `S` combinator is determined by structural recursion! It is simply the pure **λ**-abstraction that works when plugged in for λ^*.

How Can We Systematically Eliminate All **λ**s?

Strategy:

- Since the three rewrite rules on the preceding slide generalize to lambda-expressions with free variables and constants, we can eliminate any **λ**-abstraction that does not contain **λ** in its body.
- Algorithm: eliminate **λ**-abstractions from inside-out, one-at-a-time. This process terminates because it strictly reduces a recursively defined weighted **λ**-*depth* measure, which is the sum of the weights of all embedded **λ**-abstractions. The details of this definition are delicate (but not very interesting). (Since this algorithm use general recursion, we must provide a termination argument.)
- Warning: this transformation can (and usually does) cause exponential blow-up in the *expanded* (replacing **`S`**, **`K`**, and **`I`** by their definitions as **λ**-abstractions because the third rule replaces a **λ**-abstraction by a **λ**-abstraction (in **`S`**) with two references to its parameter (**`z`**). Note that the *depth** function grows exponentially with tree depth because the definition of *depth** adds the *depth**s of both subtrees of an application. In essence, *depth** grows as the number of nodes in the tree grows which is exponentially larger than the depth of the original tree.

Final Observations

- Checking the **`App`** case

$$\mathtt{S}\ (\lambda x.M)\ (\lambda x.N)$$

$$= (\lambda x.\lambda y.\lambda z.(x\ z)(y\ z))\ (\lambda x.M)\ (\lambda x.N)$$

$$= (\lambda y.\lambda z.((\lambda x.M)\ z)(y\ z))\ (\lambda x.N)$$

$$= (\lambda z.((\lambda x.M)\ z)((\lambda x.N)\ z))$$

$$= (\lambda z.(M_{x\leftarrow z})\ ((\lambda x.N)\ z))$$

$$= (\lambda z.(M_{x\leftarrow z})\ (N_{x\leftarrow z})) = \lambda x.(M\ N)$$ (by α-conversion)

Note: the variable names `x y z` are fresh and arbitrary, distinct from any free names in `λx.M λx.N`