Lifting Lambdas into Supercombinators

Harry R. Schwartz

July 26, 2017

I was recently reading about Edwin Brady's supercombinator compiler, Epic. Epic is the backend powering Idris and Epigram (and, optionally, Agda), so I figured it might be worth a look. A "supercombinator compiler" sure sounded impressive, but I didn't know what it was. Let's figure that out.

First, What's a Combinator? A *combinator* is a piece of code (a *term*) in which all variables are *bound*. A variable is bound in a given term if it's defined in that term. For example, in the

$$\lambda$$

-calculus, the term

$$\lambda x.\lambda y.(x\ y)$$

is a combinator, since all of the variables in the body of the expression (that is,

$$x$$

and

$$y$$

) are bound as arguments. Conversely,

$$\lambda x.(z\ x)$$

isn't a combinator, since

$$z$$

appears as a free variable.

Using

$$\lambda$$

-calculus is traditional, but we can talk about this in terms of a more conventional language, too, like Python. This function is a combinator, since both `x` and `y` are bound:

```python
def combinator(x, y):
  return x + y
```

This function isn't, since `y` is free:

```python
def not_a_combinator(x):
  return x + y
```

Combinators are interesting because they're self-contained, or "closed." They don't rely on any information unless it's passed in as an argument, so they compose well,[1] which makes them easy to reason about.

So, What's a Supercombinator? A *supercombinator* is recursively defined as "a combinator whose every sub-term is also a supercombinator." In other words, a supercombinator is a combinator whose every term, sub-term, sub-sub-term, etc., is also a combinator. Some lambda expressions are combinators, and some combinators are supercombinators.

For example, here's a supercombinator:

$$\lambda x.(\lambda y.y\ y)(x\ x)$$

Note that the inner term

$$\lambda y.y\ y$$

is also a supercombinator.

On the other hand, here's a combinator that *isn't* a supercombinator:

$$\lambda x.(\lambda y.x\ y)$$

The inner term

$$\lambda y.x\ y$$

isn't a combinator because

$$x$$

is free within it, which means that the whole expression isn't a supercombinator.

Generating Supercombinators Every combinator can be transformed into an equivalent[2] supercombinator. For example, in Python, we might have a function like:

[1] They compose *so* well, in fact, that you can build logical systems on top of them with expressiveness equal to the

$$\lambda$$

-calculus. The SKI and BCKW calculi are prominent examples.

[2] By *equivalent,* I specifically mean that two combinators of the same arity will β-reduce to the same expression when given the same arguments. If that doesn't mean anything to you, that's OK; your intuitive definition of "equivalent" is probably correct. :-)

```python
def outside(x):
    def inside(y):
        return x + y
    return inside(5)
```

This is a combinator, but not a supercombinator. `x` is a free variable within the definition of `inside`. However, we could transform this expression by passing in `x` as an additional argument to `inside`, like so:

```python
def outside(x):
    def inside(x, y):
        return x + y
    return inside(x, 5)
```

Now that `inside` doesn't reference a free variable, we can lift it into the global context, like so:

```python
def inside(x, y):
    return x + y

def outside(x):
    return inside(x, 5)
```

We've eliminated the closure and the function nesting, and the original and transformed expressions still do the same thing. Our code now consists of a pair of supercombinators! Neat.

This act of (1) replacing free variables with arguments and (2) extracting the new combinator into the global context is called *lambda lifting.* To phrase it another way, lambda lifting is an algorithm for turning closures (that is, functions with free variables) into pure global functions.

Compiling with Supercombinators Since every term in a supercombinator is independent of its context—that is, it contains no free variables—compiling a program structured as a collection of supercombinators is much simpler than it would be otherwise. Every λ term can be compiled to a global function, with no nesting or closures.

We could imagine designing a compiler for a purely functional language which:

1. Receives some input code which has been structured as a collection of combinators,
2. Applies lambda lifting to transform the input into an equivalent collection of supercombinators, and
3. Compiles them into a target language, with each term corresponding to a top-level function.

I'm sure I'm eliding a lot of complexity here, especially in that last step, but this seems to be the general idea.

So, if we wanted to build a language on top of Epic, we'd first write a compiler from our language to Epic's input language (an extended form of the

$$\lambda$$

-calculus). Epic would take our jumble of expressions, lambda-lift it into a collection of supercombinators, and generate C code based on those pure, global functions.

References There don't seem to be too many references to compiling with supercombinators floating around. The few that I've seen are pretty good, though:

- Simon Peyton Jones, *The Implementation of Functional Programming Languages*, 1987. Specifically, see "§13: Supercombinators and Lambda-Lifting" for a thoroughly relevant elaboration.
- John Hughes, *Super-Combinators: A New Implementation Method for Applicative Languages*, 1982.
- Edwin Brady, *Epic—A Library for Generating Compilers*, 2011.