Recitation 11:
Cost, Parallelism, and Modal Separation

15-312: Foundations of Programming Languages

Jeanne Luning Prak

April 4th, 2018

1 Motivation

When defining a cost semantics, as we did in lecture, we would like to be able to say that evaluating a value has 0 cost. After all, it takes 0 steps to turn a value into a value. However, consider how many steps your code actually takes to evaluate a value in the dynamics you've implemented this semester. For example, if we're evaluating a tuple that is already a value, we must go through the tuple and check that each element of that tuple is a value. This doesn't take 0 steps; this takes as many steps as there are elements in the tuple! Our cost semantics would have to take this into account:

$$\frac{e_1 \Downarrow^n v_1 \quad e_2 \Downarrow^m v_2}{\langle e_1, e_2 \rangle \Downarrow^{2 \oplus (n \otimes m)} \langle v_1, v_2 \rangle}$$

This means that our cost semantics does not actually match the number of steps it takes to step to a value in our transition semantics; there is extra overhead for checking that an expression is actually a value. This becomes an even bigger problem when we have function applications, where we can substitute an expression for a variable in multiple places. Then, in each of those places, we must check that that expression is, in fact, a value.

We would like to be able to avoid this check altogether, so that our evaluational cost semantics match up with our transition semantics. We would like for our cost semantics to say:

$$\frac{e_1 \Downarrow^n v_1 \quad e_2 \Downarrow^m v_2}{\langle e_1, e_2 \rangle \Downarrow^{n \otimes m} \langle v_1, v_2 \rangle}$$

and so if both e_1 and e_2 are already values, then it takes 0 steps to evaluate them, and so evaluating the tuple takes 0 steps.

To do this, we define a new language, Modal Parallel PCF, or **MPPCF**[1], which has a different *sort* for expressions and values. That way, it is not necessary to check if an expression is a value: you already know it isn't! In **MPPCF**, expressions *return* values rather than stepping to values, to preserve the distinction between the two.

This is called a *modal separation* between expressions and values. As we will see in lecture on Thursday, another common use of modal separation is to separate commands, which have

[1] Unlike most languages in this class, **MPPCF** is not in *PFPL*, so you should refer to the Homework 5 handout or to these recitation notes for its definition.

effects, from expressions, which do not. This is the idea behind Modernized Algol, which we will be studying, but also the functional language Haskell, which separates effects using monads.

2 MPPCF

The language **MPPCF** (Modal Parallel PCF) has a modal distinction between expressions, which must incur cost to be evaluated, and values, which are evaluated and incur no cost.

2.1 Syntax

The grammar for MPPCF is given below. This grammar is quite different from other ones we've seen so far, as there is a different sort for values and expressions, whereas usually we have a single sort for terms. This distinction is explained below the chart.

τ	$::=$	$\texttt{nat}$	naturals
	$\mid$	$\tau_1 \to \tau_2$	partial functions
	$\mid$	$\langle \tau_1 \times \ldots \times \tau_n \rangle$	eager products
	$\mid$	$[\,\tau_1 \,\&\, \ldots \,\&\, \tau_n\,]$	lazy products
v	$::=$	x	variables
	$\mid$	n	numeric literals
	$\mid$	$\texttt{fn}\ (x : \tau) \Rightarrow e$	functions
	$\mid$	$\langle v_1, \ldots, v_n \rangle$	eager tuples
	$\mid$	$[\,e_1 \,||\, \ldots \,||\, e_n\,]$	lazy tuples
e	$::=$	$\texttt{ret}(v)$	value
	$\mid$	$v_1(v_2)$	application
	$\mid$	$\texttt{s}(v)$	successor
	$\mid$	$\texttt{ifz}\ v\ \{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\}$	zero test
	$\mid$	$\texttt{fix}\, x : \tau\, \texttt{is}\, e$	fixed point
	$\mid$	$\texttt{split}\ v\ \texttt{as}\ x_1, \ldots, x_n\ \texttt{in}\ e$	eager tuple unpack
	$\mid$	$\texttt{par}\ x = v\ \texttt{in}\ e$	parallel tuple evaluation

The most prominent feature of **MPPCF** is its modal separation between expressions and values. Variables, functions, numbers, and tuples are values: they cannot contain unevaluated computations inside of them (except for lazy tuples). The rest of the operators are expressions, and cannot be contained in values. However, this brings up a concern: if expressions and values must be separate *how do we evaluate expressions?*. For example, **MPPCF** has a function

$$\texttt{fn}\ (x : \tau) \Rightarrow e$$

whose argument is a value and whose body is an expression. We would like to be able to define evaluation in the same way as always:

$$\overline{(\texttt{fn}\ (x : \tau) \Rightarrow e)(e_1) \longmapsto [e_1/x]e}$$

But we can't, since we can't substitute an expression for a value!

To deal with this, we have three operators that mediate between expressions and values:

- Lazy tuples, $[\, e_1 \,||\, \ldots \,||\, e_n \,]$ which are values that contain expressions.
- Returns, $\texttt{ret}(v)$, which are expressions that contain values.
- Parallel let, $\texttt{par}\ x = v\ \texttt{in}\ e$, which evaluates a lazy tuple in parallel until it contains only returns, unwraps the values from the returns, and puts them into an eager tuple for use in another expression.

If you know about monads, you may notice that `ret` is a return and `par` is a parallel version of bind.

2.2 Statics

Because **MPPCF** has a modal separation of values and expressions, we now have two judgments for typing: $v : \tau$, which states that a value v has type τ and $e \div \tau$, which states that e is an expression, or a computation, of type τ.

2.2.1 Values

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e \div \tau'}{\Gamma \vdash \texttt{lam}\{\tau\}(x.e) : \tau \to \tau'}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{nat}} \qquad \frac{\Gamma \vdash v_1 : \tau_1 \quad \ldots \quad \Gamma \vdash v_n : \tau_n}{\Gamma \vdash \langle v_1, \ldots, v_n \rangle : \langle \tau_1 \times \ldots \times \tau_n \rangle}$$

$$\frac{\Gamma \vdash e_1 \div \tau_1 \quad \ldots \quad \Gamma \vdash e_n \div \tau_n}{\Gamma \vdash [\, e_1 \,||\, \ldots \,||\, e_n \,] : [\, \tau_1 \,\&\, \ldots \,\&\, \tau_n \,]}$$

Here, we have different types for tuples that contain values and tuples that contain expressions.

2.2.2 Expressions

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \texttt{ret}(v) \div \tau} \qquad \frac{\Gamma \vdash v_1 : \tau \to \tau' \quad \Gamma \vdash v_2 : \tau}{\Gamma \vdash \texttt{ap}(v_1; v_2) \div \tau'}$$

$$\frac{\Gamma \vdash v : \texttt{nat}}{\Gamma \vdash \texttt{s}(v) \div \texttt{nat}} \qquad \frac{\Gamma \vdash v : \texttt{nat} \quad \Gamma \vdash e_1 \div \tau \quad \Gamma, x : \texttt{nat} \vdash e_2 \div \tau}{\Gamma \vdash \texttt{ifz}\{e_1; x.e_2\}(v) \div \tau}$$

$$\frac{\Gamma, x : [\, \tau \,] \vdash e \div \tau}{\Gamma \vdash \texttt{fix}\{\tau\}(x.e) \div \tau} \qquad \frac{\Gamma \vdash v : \langle \tau_1 \times \ldots \times \tau_n \rangle \quad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \div \tau}{\Gamma \vdash \texttt{split}[n](v; x_1, \ldots, x_n.e) \div \tau}$$

$$\frac{\Gamma \vdash v : [\, \tau_1 \,\&\, \ldots \,\&\, \tau_n \,] \quad \Gamma, x : \langle \tau_1 \times \ldots \times \tau_n \rangle \vdash e \div \tau}{\Gamma \vdash \texttt{par}(v; x.e) \div \tau}$$

2.3 Dynamics

Notice that here we use an evaluation dynamics annotated with the cost graph for each expression. This allows us to clearly express the main advantage of **MPPCF**: accurate cost semantics.

$$\frac{}{\texttt{ret}(v) \Downarrow^{1} v} \qquad \frac{[v/x]e \Downarrow^{c} v'}{\texttt{ap}((\texttt{lam}\{\tau\}(x.e)); v) \Downarrow^{c \oplus 1} v'}$$

$$\frac{}{\texttt{s}(\texttt{num}[n]) \Downarrow^{1} \texttt{num}[n+1]} \qquad \frac{e_1 \Downarrow^{c} v}{\texttt{ifz}\{e_1; x.e_2\}(\texttt{num}[0]) \Downarrow^{c \oplus 1} v}$$

$$\frac{n \neq 0 \quad [\texttt{num}[n-1]/x]e_2 \Downarrow^{c} v}{\texttt{ifz}\{e_1; x.e_2\}(\texttt{num}[n]) \Downarrow^{c \oplus 1} v} \qquad \frac{[[\,\texttt{fix}\{\tau\}(x.e)\,]/x]e \Downarrow^{c} v}{\texttt{fix}\{\tau\}(x.e) \Downarrow^{c \oplus 1} v}$$

$$\frac{[v_1, \ldots, v_n/x_1, \ldots, x_n]e \Downarrow^{c} v}{\texttt{split}[n](\langle v_1, \ldots, v_n\rangle; x_1, \ldots, x_n.e) \Downarrow^{c \oplus 1} v} \qquad \frac{e_1 \Downarrow^{c_1} v_1 \quad \ldots \quad e_n \Downarrow^{c_n} v_n \quad [\langle v_1, \ldots, v_n\rangle/x]e \Downarrow^{c} v}{\texttt{par}([\,e_1 \,||\, \ldots \,||\, e_n\,]; x.e) \Downarrow^{(c_1 \otimes \ldots \otimes c_n) \oplus c \oplus 1} v}$$

3 Examples

Below are some examples of programs in **PCF** (extended with products), the equivalent program in **MPPCF**, and what the program is meant to do. As an exercise, try covering up the last column and implementing the **MPPCF** functions on your own before looking at the translation.

Meaning	**PCF**	**MPPCF**
Identity Function	$\texttt{fn}\ (x : \texttt{nat}) \Rightarrow x$	$\texttt{fn}\ (x : \texttt{nat}) \Rightarrow \texttt{ret}(x)$
Two	$\texttt{s}(\texttt{s}(\texttt{z}))$	2 or $\texttt{par}\ v = [\texttt{s}(0)]\ \texttt{in}\ \texttt{s}(v)$
The identity applied to s(z)	$(\texttt{fn}\ (x : \texttt{nat}) \Rightarrow x)(\texttt{s}(\texttt{z}))$	$\texttt{par}\ v = [\texttt{s}(0)]\ \texttt{in split}\ v\ \texttt{as}\ x_1\ \texttt{in}$ $(\texttt{fn}\ (x : \texttt{nat}) \Rightarrow \texttt{ret}(x))(x_1)$
First element of a tuple	`e.l`	$\texttt{par}\ v = [e]\ \texttt{in split}\ v\ \texttt{as}\ x_1, x_2\ \texttt{in}$ $\texttt{ret}(x_1)$
If e $>$ 0 then 1 else 0	$\texttt{ifz}\ e\ \{\texttt{z} \Rightarrow z \mid \texttt{s}(x) \Rightarrow s(z)\}$	$\texttt{par}\ v = [e]\ \texttt{in split}\ v\ \texttt{as}\ x_1\ \texttt{in}$ $\texttt{ifz}\ v\ \{\texttt{z} \Rightarrow \texttt{ret}(0) \mid \texttt{s}(x) \Rightarrow \texttt{ret}(1)\}$