1 Introduction

The treatment of fork-join parallelism distinguishes two cases, the static and dynamic, according to when the "degree" of parallelism is determined. The static case is formulated using the construct \( \text{par } e_1 = e_2 \text{ and } x_1 = x_2 \text{ in } e \), whose dynamics evaluates \( e_1 \) and \( e_2 \) in parallel, then substitutes their values for the corresponding variables in \( e \). The dynamic case is formulated using a tabulator that creates a sequence of values whose length and content are determined at run-time. All elements are evaluated in parallel, then creating a sequence consisting of their values. The dynamic case can be understood as a cascade of binary parallel splits logarithmic in the length of the sequence.

Although the tabulation mechanism is tied to a type of sequences, the parallel definition mechanism is not associated with a type. It would be more elegant if the definition construct were associated with a type, rather than simply imposed as an ad hoc means to an end. And it would be nice if the dynamic case were a smooth generalization of the static case. Might there be a more systematic solution that encompasses both forms of parallelism?

There is, but to see it requires taking a step back to think about how parallelism arises in the first place. The key is not to somehow impose parallelism in any otherwise sequential dynamics, but rather to expose the parallelism that is naturally present to the maximal extent possible. Functional languages give rise to parallelism without even trying. For example, to evaluate a sum of two expressions it is necessary to obtain the value of both summands, but the only constraint on how these evaluations are to be scheduled is that they both be completed before the addition is performed. This example is prototypical: the essence of parallelism is sequentiality. Get the essential dependencies right and the parallelism will take care of itself.

2 Modal Cost Semantics

First, let us consider how to express sequential dependencies. The best way to do this is to use the modal formulation of by-value PCF given in Harper (2018). The statics is reproduced in Figure 1 for convenience. For the sake of clarity it uses the meta-variable \( v \) for values and \( e \) for computations. The computation \( \text{ret}(v) \) simply returns the given value.

The cost dynamics of PCF-by-value is given in Figure 2, defining the judgment \( e \Downarrow^c v \) is defined for closed computations \( e \), closed values \( v \), and cost graphs \( c \).

The value \( s(v) : \text{nat} \), with \( v : \text{nat} \), is different from the computation \( \text{let}(\text{comp} (\text{ret}(v)); x. \text{ret}(s(x))) \). The latter is a computation that evaluates to the former, incurring a cost to do so.\(^1\)

\(^1\)That is, those with an eager dynamics; the lazy languages are hopeless.

\(^2\)This distinction corrects a mistake in PFPL, which fails to distinguish between a value and an expression that happens to be a value.
\[
\Gamma, x : \tau \vdash x : \tau \quad (1a)
\]
\[
\Gamma \vdash z : \text{nat} \quad (1b)
\]
\[
\Gamma \vdash v : \text{nat} \quad \Gamma \vdash s(v) : \text{nat} \quad (1c)
\]
\[
\Gamma, x : \tau_1 \to \tau_2, y : \tau_1 \vdash e \triangleq \tau_2 \quad (1d)
\]
\[
\Gamma \vdash \text{comp}(e) : \text{comp}(\tau) \quad (1e)
\]
\[
\Gamma \vdash v : \tau \quad \Gamma \vdash \text{ret}(v) \triangleq \tau \quad (1f)
\]
\[
\Gamma \vdash e_0 \triangleq \tau \quad \Gamma, x : \text{nat} \vdash e_1 \triangleq \tau \quad (1g)
\]
\[
\Gamma \vdash \text{fun}\{\tau_1; \tau_2\}(x.y.e) : \tau_1 \to \tau_2 \quad \Gamma \vdash e \triangleq \tau \quad (1h)
\]
\[
\Gamma \vdash \text{ap}(v_1; v_2) \triangleq \tau \quad \Gamma \vdash \text{let}(v_1; x.e_2) \triangleq \tau_2 \quad (1i)
\]

Figure 1: Statics of \textbf{PCF}-by-value

\[
\text{ret}(v) \Downarrow^1 v \quad (2a)
\]
\[
e_0 \Downarrow^c v \quad (2b)
\]
\[
\text{ifz}(\tau)(z; e_0; x; e_1) \Downarrow^{1+c} v \quad (2b)
\]
\[
[e/x]e_1 \Downarrow^c v \quad (2c)
\]
\[
\text{ifz}(\tau)(s(e); e_0; x; e_1) \Downarrow^{1+c} v \quad (2c)
\]
\[
[\text{fun}\{\tau_2; \tau\}(f.x.e), v_2/f, x]e \Downarrow^c v \quad (2d)
\]
\[
\text{ap}(\text{fun}\{\tau_2; \tau\}(f.x.e); v_2) \Downarrow^{1+c} v \quad (2d)
\]
\[
e \Downarrow^c v \quad [v/x]e_2 \Downarrow^2 v_2 \quad (2e)
\]
\[
\text{let}(\text{comp}(e); x; e_2) \Downarrow^{1+c+c_2} v_2 \quad (2e)
\]

Figure 2: Cost Dynamics of \textbf{PCF}-by-value
3 Parallelism, Revisited

The modal framework allows us to manage dependencies, but it is limited to one dependency at a time, precluding parallelism. What is missing is a way to express the simultaneous dependency of one computation on several, perhaps unboundedly many, prior computations whose relative evaluation order is unconstrained. This is provided by a version of the lazy product type in which (a) neither component of a pair is evaluated when the pair is created, but (b) both components are evaluated whenever the pair is needed. A complementary eager product type mediates the dependency. It is eliminated by pattern matching to retrieve both components at once. It is straightforward to generalize the binary forms both products to \(n\)-ary forms, where \(n\) is determined statically. (The dynamic form is considered below.)

The statics for lazy and eager product types is given in Figure 3. The corresponding cost dynamics is given in Figure 4.

Turning to the dynamic case, there are two types corresponding to the lazy and eager product types, but with dynamically determined sizes. The analogue of a lazy pair (or tuple) is a sequence generator, which, when evaluated, determines the width and the components of a finite sequence. The analogue of an eager pair is a sequence whose length, and the value of each element, are determined dynamically. The elimination
form for the generator type creates a new sequence for use within the specified scope. The standard sequence operations given in Harper (2016) are then used to compute with the sequence.

The combined statics and dynamics for generators is given in Figure 5.

References
