PFPL Supplement: Types and Parallelism*

Robert Harper

Fall, 2018

1 Introduction

The treatment of fork-join parallelism distinguishes two forms, the *static* and *dynamic*, according to whether the number of parallel tasks is determined at compile time or at run time. The static form, $\texttt{par}\, e_1{=}e_2 \,\texttt{and}\, x_1{=}x_2 \,\texttt{in}\, e$, evaluates e_1 and e_2 in parallel, then substitutes their values into e, their join point. The dynamic form uses a tabulation mechanism that, given a sequence unevaluated expressions, evaluates all n of them in parallel, then consolidates their values. The dynamic form may be reduced to the static form by a recursive process of binary forks of depth logarithmic in the length of the sequence.

Curiously, the dynamic form of parallelism is naturally associated with the type of sequences, but the binary fork primitive is instead *ad hoc*. In keeping with the principle that a programming language is determined by its type structure it would be preferable for the static form to also arise in this way. To see how to do it, consider the formulation of **PCFv** given in Harper (2019b), which hinges on the type $\texttt{comp}(\tau)$ whose values are unevaluated computations of type τ. The elimination form for this type *sequences* the evaluation of an unevaluated computation by binding its value to a variable within another computation.

What does this have to do with parallelism? To see it requires taking a step back to think about how parallelism arises in the first place. In a functional setting the key is not to *impose* parallelism in an otherwise sequential dynamics, but rather to *expose* the parallelism that is naturally present. For example, to evaluate a sum of two expressions it is necessary to obtain the value of both summands, but there is no restriction on the order in which they themselves are evaluated. Thus, there *is* a sequential dependency of the addition on the summands, but there is *no* sequential dependency between them. Thus, the trick is to express the essential dependencies, and otherwise permit parallelism. Thus, paradoxically, *the essence of parallelism is sequentiality*—get the dependencies right and the parallelism will take care of itself.

Returning to **PCFv**, the key to avoiding over-sequentialization is to generalize the computation type, which encapsulates *one* computation, to a binary form that encapsulates *two* (or any fixed number) of computations. The elimination form evaluates both in parallel, then binds their results for use at the join point. In the case of addition the two computations are the summands, and the join point adds their values.

2 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 the *lazy product type*, which encapsulates *two* unevaluated computations. The elimination form, called *parallel bind*, correspondingly evaluates both computations and binds a variable to the *eager pair* of their values.[1]

*Copyright © Robert Harper. All Rights Reserved.

[1]The binary case may easily be generalized to any $n \geq 0$ suspended computations evaluated in parallel.

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash v_1 \otimes v_2 : \tau_1 \otimes \tau_2} \tag{1a}$$

$$\frac{\Gamma \vdash v : \tau_1 \otimes \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e \mathrel{\dot{\sim}} \tau}{\Gamma \vdash \mathtt{split}(\, v \,; x_1, x_2 \,.\, e \,) \mathrel{\dot{\sim}} \tau} \tag{1b}$$

$$\frac{\Gamma \vdash e_1 \mathrel{\dot{\sim}} \tau_1 \quad \Gamma \vdash e_2 \mathrel{\dot{\sim}} \tau_2}{\Gamma \vdash e_1 \,\&\, e_2 : \tau_1 \,\&\, \tau_2} \tag{2}$$

$$\frac{\Gamma \vdash v : \tau_1 \,\&\, \tau_2 \quad \Gamma, x : \tau_1 \otimes \tau_2 \vdash e \mathrel{\dot{\sim}} \tau}{\Gamma \vdash \mathtt{parbnd}(\, v \,; x \,.\, e \,) \mathrel{\dot{\sim}} \tau} \tag{3}$$

Figure 1: Statics of Eager and Lazy Products

$$\frac{}{e_1 \,\&\, e_2 \ \mathsf{val}} \tag{4a}$$

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad \{v_1 \otimes v_2 / x\} e \Downarrow^{c} v}{\mathtt{parbnd}(\, e_1 \,\&\, e_2 \,; x \,.\, e \,) \Downarrow^{\mathbf{1} \oplus (c_1 \otimes c_2) \oplus c} v} \tag{4b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{e_1 \otimes e_2 \ \mathsf{val}} \tag{4c}$$

$$\frac{\{v_1, v_2 / x_1, x_2\} e \Downarrow^{c} v}{\mathtt{split}(\, v_1 \otimes v_2 \,; x_1, x_2 \,.\, e \,) \Downarrow^{\mathbf{1} \oplus c} v} \tag{4d}$$

Figure 2: Cost Dynamics of Eager and Lazy Products

$$\frac{\Gamma \vdash v : \mathtt{nat} \quad \Gamma, x : \mathtt{nat} \vdash e \overset{\cdot}{\sim} \tau}{\Gamma \vdash \mathtt{seqgen}(\, v \,;\, x \,.\, e \,) : \mathtt{seqgen}(\, \tau \,)} \tag{5a}$$

$$\frac{\Gamma \vdash v_1 : \mathtt{seqgen}(\, \tau \,) \quad \Gamma, x : \mathtt{seq}(\, \tau \,) \vdash e_2 \overset{\cdot}{\sim} \tau_2}{\Gamma \vdash \mathtt{seqbnd}(\, v_1 \,;\, x \,.\, e_2 \,) \overset{\cdot}{\sim} \tau_2} \tag{5b}$$

$$\frac{}{\mathtt{seqgen}(\, \overline{n} \,;\, x \,.\, e_2 \,)\ \mathsf{val}} \tag{5c}$$

$$\frac{\{\overline{0}/x\}e \Downarrow^{c_0} v_0 \quad \ldots \quad \{\overline{n-1}/x\}e \Downarrow^{c_{n-1}} v_{n-1} \quad \{\mathtt{seq}[\, n \,](\, v_0, \ldots, v_{n-1} \,)/y\}e' \Downarrow^{c'} v'}{\mathtt{seqbnd}(\, \mathtt{seqgen}(\, \overline{n} \,;\, x \,.\, e \,) \,;\, y \,.\, e' \,) \Downarrow^{(c_0 \otimes \ldots \otimes c_{n-1}) \oplus c'} v'} \tag{5d}$$

Figure 3: Statics and Cost Dynamics of Sequence Generators

The statics for lazy and eager product types is given in Figure 1. The corresponding cost dynamics is given in Figure 2.

There are two types corresponding to the lazy and eager product types, but with dynamically determined sizes. The analogue of a lazy 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 3.

Exercise 2.1. *Extend the dynamics to account for exceptions using the generalized* `bnd` *construct given in Harper (2019a). Be sure to maintain the same behavior as in the sequential case, so that the leftmost exception is propagated from a parallel computation, and that all parallel computations are completed, regardless of whether any raise an exception. Correspondingly, be sure that the cost graphs are appropriate in all cases!*

References

Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Exceptions: Control and data. Supplement to Harper (2016), Fall 2019a. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/exceptions.pdf`.

Robert Harper. PCF-By-Value. Supplement to Harper (2016), Fall 2019b. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/pcfv.pdf`.