PFPL Supplement: Inductive and Coinductive Types in FPC*

Robert Harper

December 2021

1 Introduction

FPC is an extension of PCF with (unrestricted) recursive types, $\mathtt{rec}(\,t\,.\,\tau\,)$, that represent solutions of type equations "up to isomorphism" mediated by fold and unfold:

$$\mathtt{fold}(\,-\,) : \{\mathtt{rec}(\,t\,.\,\tau\,)/t\}\tau \to \mathtt{rec}(\,t\,.\,\tau\,)$$
$$\mathtt{unfold}(\,-\,) : \mathtt{rec}(\,t\,.\,\tau\,) \to \{\mathtt{rec}(\,t\,.\,\tau\,)/t\}\tau$$

Recursive types are powerful. They may be used to define self-referential types, including recursive function types, and are sufficient to encode the untyped λ-calculus. Divergent computations, and therefore partial functions, are therefore unavoidable in FPC: as soon as recursive types are admitted, divergence is too.

But what is the situation with regard to recursive types for polynomial type constructors $t\,.\,\tau$? The answer, unsurprisingly, sensitive to whether the ambient language is eager or lazy. For example, consider the type operator $t\,.\,\mathtt{unit} + t$. In an eager setting the type $\mathtt{rec}(\,t\,.\,\mathtt{unit} + t\,)$ has as *values* precisely the natural numbers, as in the inductive case. Because the range of significance of variables is confined to values, there is nothing further to say—there are always divergent computations, but these do not infect the values of a type. In a lazy setting the same type has a rather different interpretation: it includes not only the natural numbers extended with a point at infinity (the infinite stack of successors), but also any divergent computation of that type (any infinite loop), and all finite successors of such a computation. Thus it is neither the natural numbers nor the co-natural numbers, though it includes these. Were the successor operation (encoded by a folded injection) deemed strict (evaluates its argument), the type would be cut down to the natural numbers plus any divergent computation of that type. And there is no avoiding it: in a lazy setting divergence inhabits *every* type, because there is no distinction between computations and values.

On the other hand there is, without further provision, no way to represent the conatural numbers as a recursive type either. However, whereas there is no modification of a lazy language to represent inductive types, by adding suspension types to an eager language one may represent all of the types representable in a lazy language using suspension types. For example, the type $\mathtt{rec}(\,t\,.\,\mathtt{susp}(\,\mathtt{unit} + t\,)\,)$ captures the case of the near-as-possible representation of the natural numbers in a lazy language. By interposing suspensions on the summands we fully lazy case can be reproduced as well, as can in general any level of "degrees" of laziness one may seek. Thus, the eager setting *strictly dominates* the expressive power of the lazy setting by providing a much finer grain of control over the use of laziness, if any, allowing a wider range of expressive power than in the lazy case.

2 Inductive Types in Eager **FPC**

As illustrated in the introduction, under the eager dynamics for **FPC** the recursive type $\mathtt{rec}(\, t \,.\, \tau \,)$ is the inductive type $\mu \triangleq \mu(\, t \,.\, \tau \,)$. To see this it suffices to define the introduction and elimination forms for the inductive type in terms of those of the recursive type, and to check that they behave as expected.

The introductory form for the recursive type, $\mathtt{fold}(\, - \,)$, serves as the introductory form for that type viewed as an inductive type. Note that under the eager interpretation $\mathtt{fold}(\, e \,)$ is a value only if e is a value. The eliminatory form $\mathtt{rec}[\, t \,.\, \tau \,](\, e \,;\, x \,.\, e' \,) : \rho$, may be defined as the application, $R(\, e \,)$, of the recursive function

$$R \triangleq \mathtt{fun}\, r(\, x{:}\mu \,){:}\rho\, \mathtt{is}\, \{\mathtt{map}[\, t \,.\, \tau \,](\, x \,.\, \mathtt{ap}(\, r \,;\, x \,)\,)(\, \mathtt{unfold}(\, x \,)\,)/x\}e'.$$

When e is a value,

$$R(\, \mathtt{fold}(\, e \,)\,) \longmapsto^* \{\mathtt{map}[\, t \,.\, \tau \,](\, x \,.\, \mathtt{ap}(\, R \,;\, x \,)\,)(\, e \,)/x\}e',$$

which is essentially equivalent to

$$R(\, \mathtt{fold}(\, e \,)\,) \longmapsto^* \{\mathtt{rec}[\, t \,.\, \tau \,](\, e \,;\, x \,.\, e' \,)/x\}e',$$

as specified for inductive types. For example, in the case of the type operator $t \,.\, \mathtt{unit} + t$, the recursor steps to a case analysis distinguishing zero from successor, and performing the recursive call on the predecessor in the latter case.

3 Coinductive Types in Lazy **FPC**

As suggested in the introduction under the lazy interpretation the recursive type $\mathtt{rec}(\, t . \tau \,)$ contains any divergent computation, plus all further elements constructed with it. Indeed, the eliminatory form of the recursive type diverges when applied to a divergent element. The generator is defined by taking $\mathtt{gen}[\, t \,.\, \tau \,](\, e \,;\, x \,.\, e' \,)$ to be the application, $G(\, e \,)$, of the recursive function

$$G \triangleq \mathtt{fun}\, g(\, x{:}\sigma \,){:}\nu\, \mathtt{is}\, \mathtt{fold}(\, \mathtt{map}[\, t \,.\, \tau \,](\, x \,.\, g(\, x \,)\,)(\, e' \,)\,).$$

Under lazy evaluation the application $G(\, e \,)$ steps immediately to a value, because $\mathtt{fold}(\, e \,)$ is always a value. And indeed one may check that

$$\mathtt{unfold}(\, G(\, e \,)\,) \longmapsto^* \mathtt{map}[\, t \,.\, \tau \,](\, x \,.\, G(\, x \,)\,)(\, \{e/x\}e' \,),$$

which is morally equivalent to the transition

$$\mathtt{unfold}(\, G(\, e \,)\,) \longmapsto^* \mathtt{map}[\, t \,.\, \tau \,](\, x \,.\, \mathtt{gen}[\, t \,.\, \tau \,](\, x \,;\, x \,.\, e' \,)\,)(\, \{e/x\}e' \,).$$

4 Not Conversely

Does the definition of the generator give a coinductive interpretation in the eager case? No. Consider the recursive type $\rho \triangleq \mathtt{rec}(\, t \,.\, \mathtt{unit} + t \,)$. Were it coinductive, the function

$$I \triangleq \lambda\,(\, x : \rho \,)\, \mathtt{gen}[\, t \,.\, \mathtt{unit} + t \,](\, x \,;\, y \,.\, \mathtt{unfold}(\, y \,)\,)$$

would be the identity function, which it is not under the by-value dynamics. For example, the application

$$I(\,\texttt{gen}[\,t\,.\,\texttt{unit}+t\,](\,\langle\rangle\,;\,\_\,.\,\texttt{r}\cdot x\,)\,)$$

diverges, rather than computing to a value of the recursive type.

Does the definition of the recursor give an inductive interpretation the lazy case? No, not even if folding and injection are eager. For example, were the recursive type $\rho \triangleq \texttt{rec}(\,t\,.\,\texttt{unit}+t\,)$ inductive, then it would also have elements $\perp$, $\texttt{fold}(\,\perp\,)$, $\texttt{fold}(\,\texttt{r}\cdot\perp\,)$, $\texttt{fold}(\,\texttt{r}\cdot\texttt{fold}(\,\texttt{r}\cdot\perp\,)\,)$, and so on. All of these would be sent to $\perp$ by the function

$$I \triangleq \lambda\,(\,x:\rho\,)\,\texttt{rec}[\,t\,.\,\texttt{unit}+t\,](\,x\,;\,y\,.\,\texttt{fold}(\,y\,)\,),$$

which would be the identity in the inductive case.

References

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