1 Introduction

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

\[ \text{fold}(-) : \texttt{rec}(t.\tau)/t\tau \rightarrow \texttt{rec}(t.\tau) \]
\[ \text{unfold}(-) : \texttt{rec}(t.\tau) \rightarrow \texttt{rec}(t.\tau)/t\tau \]

Recursive types are powerful. They may be used to define self-referential types, including recursive function types, and are sufficient to encode the uni-typed \texttt{\lambda}-calculus.

It is natural to ask whether, when restricted to polynomial (or positive) type operators, recursive types also encompass inductive or coinductive types. The answer depends on whether we are considering the lazy or the eager dynamics of \textbf{FPC}. Briefly, for a polynomial type operator \(t.\tau\), the recursive type \texttt{rec}(t.\tau) represents the \textit{inductive} type \(\mu(t.\tau)\) in the eager case, and represents the \textit{coinductive} type \(\nu(t.\tau)\) in the lazy case.

Thus, it would appear, that the eager formulation lacks coinductive types, and that the lazy formulation lacks inductive types, a symmetric stand-off as regards their respective expressive power. However, the apparent symmetry is broken by the realization that the lazy formulation is \textit{representable} within the eager formulation, given a type of (possibly divergent) computations of each type, but that \textit{no converse representation} is possible. The standoff is resolved in favor of the eager language, which, through the use of computation types, can represent \textit{both} inductive and coinductive types.

Throughout the rest of this supplement the type operator \(t.\tau\) is assumed to be polynomial.

2 Inductive Types in Eager FPC

Under the eager dynamics, in which in particular \texttt{fold}(-) is evaluated eagerly, the recursive type \texttt{rec}(t.\tau) represents the (eager) inductive type \(\mu \cong \mu(t.\tau)\). To see this it suffices to define the introduction and elimination forms for the inductive type in terms of those of the recursive type.
Obviously, \( \text{fold}(\cdot) \) will play the role of the introductory form, with \( \text{fold}(e) \) being a value when and only when \( e \) is a value. The eliminatory form, \( \text{rec}\{t.\tau\}(x.e';e) \), is definable as the application, \( R(e) \), of the recursive function

\[
R \triangleq \text{fun} r(x;\mu) ; \rho \text{is map}\{t.\tau\}(x.\text{ap}(r;x))(\text{unfold}(x)).
\]

Thus, when \( e \) is a value,

\[
R(\text{fold}(e)) \mapsto \ast \text{map}\{t.\tau\}(x.\text{ap}(R;x))(e),
\]

as desired.

It is tempting to consider that the same recursive type can be given a coinductive interpretation as \( \nu \triangleq \nu(t.\tau) \), taking the recursive \( \text{unfold}(\cdot) \) as the eliminatory form, and defining the recursive function \( G \) whose application, \( G(e) \), represents the generator \( \text{gen}\{t.\tau\}(x.e';e) \), defined as follows:

\[
G \triangleq \text{fun} g(x;\sigma) ; \nu \text{is fold}\{t.\tau\}(x.g(x))([e/x]e').
\]

But then, when \( e \) is a value,

\[
\text{unfold}(G(e)) \mapsto \ast \text{map}\{t.\tau\}(x.g(x))([e/x]e').
\]

In cases such as the type operator \( t.\text{unit} + t \), corresponding to the type \( \text{conat} \), this computation diverges, as may be seen by expanding the definition of \( \text{map} \).

### 3 Coinductive Types in Lazy FPC

The situation is dual in the lazy formulation of FPC in that the recursive type \( \text{rec}\{t.\tau\} \) corresponds to the coinductive type \( \nu \triangleq \nu(t.\tau) \). The eliminatory form for \( \nu \) is, of course, the recursive \( \text{unfold}(\cdot) \) unfolding operation. The introductory form is defined as in the preceding section, with \( \text{gen}\{t.\tau\}(x.e';e) \) being \( G(e) \), where

\[
G \triangleq \text{fun} g(x;\sigma) ; \nu \text{is fold}\{t.\tau\}(x.g(x))([e/x]e').
\]

Under lazy evaluation the application \( G(e) \) steps immediately to a value because \( \text{fold}(\cdot) \) is evaluated lazily. And indeed one may check that

\[
\text{unfold}(G(e)) \mapsto \ast \text{map}\{t.\tau\}(x.G(x))([e/x]e'),
\]

as required. Thus coinductive types are definable in the lazy formulation of FPC.

Dually, one might wonder why the definition of the inductive recursor given in the preceding section does not provide the structure of an inductive type. The difficulty is that under the lazy interpretation the principal argument, \( e \), to the recursor could be \( \bot \), an expression of type \( \mu \) that diverges when evaluated. In typical cases, such as that for \( \text{conat} \), the evaluation of \( \text{map} \) case analyzes on \( e \), which provokes divergence, thereby violating the required conditions. Similar behavior arises on values such as \( \text{fold}(\bot) \), which diverge when the folded expression is evaluated, as it would be by the recursor. In essence the lazy interpretation includes unintended values into the (putative) inductive type that ruin its intended behavior, with the result that no recursive type can be construed as an inductive type.\(^1\)

\(^1\)Nor can an \textit{ad hoc} primitive type, such as \( \text{nat} \), be the type of natural numbers, for it will always include, at least, \( \bot \), which is neither zero nor a successor.
References