Syntactic Considerations on Recursive Types

Martín Abadi
ma@pa.dec.com
Systems Research Center
Digital Equipment Corporation

Marcelo P. Fiore*
mf@dcs.ed.ac.uk
LFCS
University of Edinburgh

Abstract

We study recursive types from a syntactic perspective. In particular, we compare the formulations of recursive types that are used in programming languages and formal systems. Our main tool is a new syntactic explanation of type expressions as functors. We also introduce a simple logic for programs with recursive types in which we carry out our proofs.

1 Introduction

Recursive types are common in both programming languages and formal systems. By now, there is a deep and well-developed semantic theory of recursive types. The syntactic aspects of recursive types are also well understood in some special cases. In particular, there is an important body of knowledge about covariant recursive types, which include datatypes like natural numbers, lists, and trees.

Beyond the covariant case, however, the syntactic understanding of recursive types becomes rather spotty. Consequently, the relations between various alternative formulations of recursive types are generally unclear. Furthermore, the syntactic counterparts to some of the most basic semantic results are unknown.

In this paper we study some syntactic properties of recursive types. Our main purpose is to understand the common rules for recursive types, rather than to invent new ones. We examine the formulations of recursive types as they appear in programming languages and in formal systems; there are two main approaches:

- The first approach relies on a notion of *type equality* (e.g., see [16, 3]). A recursive type is equal to its expansions, and if a term has a type then it has all equal types. For example, one may define a type D equal to $D \to D$, and hence equal also to $(D \to D) \to (D \to D)$; any term that has one of these types has all the other types.
- The second approach relies on an explicit *isomorphism* between a recursive type and its expansions (e.g., see [12, 13]). For example, one may introduce a type D isomorphic to $D \to D$; the isomorphism between these types is represented explicitly by two terms, one of type $D \to (D \to D)$ and the other of type $(D \to D) \to D$. A term of type D does not have type $D \to D$, but can be mapped to a term of that type via the isomorphism.

We relate these two formulations in the setting of a first-order λ-calculus. Specifically, we show that any typing that can be achieved with one formulation can also be achieved with the other, possibly by inserting some explicit coercion functions. This result could seem obvious, but it turns out to be fairly difficult and its details depend on a number of sound but significant assumptions.

The proof of this result requires the development of a theory of type expressions as functors. In semantics, it is by now common to view functions on types as functors; the substantial difficulties that arise for contravariant types can be surmounted with a variety of techniques (categories of embeddings, involutory categories, etc.) [21, 10, 11]. Our explanation of type expressions as functors, although based on semantic ideas, is purely syntactic. It is perhaps more elementary and compelling than its semantic counterparts.

We also introduce a simple logic for the first-order λ-calculus with recursive types. It is a logic of partial terms [4, 18]; because of partiality, we avoid the inconsistencies connected with recursive types. Alternatively, we could have reasoned in the models of the logic, possibly doing without the logic altogether. We

*Research supported by EPSRC project Frameworks for Programming Language Semantics and Logic (grant RR29300) and by HCM project Typed Lambda-Calculus (grant ERBCHRXCT 92-0046).

have preferred to use the logic because of the syntactic character of our enterprise.

The logic serves as an ambiance for our work. For example, we use the logic for proving that certain coercion functions establish isomorphisms, and for checking the functoriality of our functorial interpretation of type expressions. The logic is too weak for the verification of realistic programs, but it may constitute the core of a useful logic for that purpose.

Our work draws on many previous developments in semantics and in logics of programs. It also has affinities with previous syntactic work, particularly that of Amadio and Cardelli [3]. Amadio and Cardelli have shown how to replace subtypings with coercion functions in the presence of recursive types (see also [6]). Our work is complementary to theirs; roughly, their coercion interpretation reduces a calculus with subtyping to one with type equality, and our coercion interpretation reduces one with type equality to one with isomorphism. Another relevant syntactic investigation is that of Plotkin [19], who developed a second-order intuitionistic linear type theory with a fixed-point operator at the level of terms, showed how to encode recursive types, and derived some reasoning principles for them. While it would be interesting to build on Plotkin's theory, we have preferred to avoid type quantification and linear types. Finally, Urzyczyn has studied typability in several systems with covariant recursive types [22]; some of our results are similar to his.

The next section is mostly a review of the formulations of recursive types; we compare them formally in sections 4 and 6.5. In section 3, we define the functorial interpretation of type expressions. In section 5 we describe our logic of programs, and in section 6 we show some of its applications.

2 Formulations of recursive types

In this section we review the main formulations of recursive types in the context of a first-order λ-calculus. The resulting systems are versions of FPC [13].

2.1 Core FPC

We start by briefly describing the part common to all systems; it is rather standard, so we list its rules in the appendix.

A *type context* is a list of distinct type variables. We write $O \vdash \diamond$ to indicate that O is a well-formed type context. As *types*, we have type variables, product types, function types, sum types, and recursive types. We use the notation $\mu X.F$ for a recursive type (the "solution" of $X = F$, where X may occur free in F). We write $O \vdash A$ to indicate that A is a well-formed type in the type context O.

A *term context* is a list of distinct variables with types. We write $O \vdash E$ to indicate that E is a well-formed term context in the type context O. As *terms*, we have variables, pairs, functions, and sums; in section 2.2, below, we introduce additional terms that correspond to recursive types. We write $O; E \vdash e : A$ to indicate that term e has type A in the type context O and the term context E.

Convention. We let O range over type contexts; X, Y, Z over type variables; A, B, C, F, G over types; E over term contexts; x, y, z, f, g, h, u, v over term variables; and e, s, t over terms. We write $A[X]$ to indicate that X may occur free in A, and then write $A[B/X]$ or simply $A[B]$ to represent the result of replacing X with B (avoiding capture of bound variables). Similarly, we write $e[x]$, $e[t/x]$, and $e[t]$.

2.2 FPC with isomorphism

One standard way to complete the definition of FPC is to introduce an explicit isomorphism between a recursive type and its (one-step) expansion. The (one-step) expansion of the recursive type $\mu X.F$ is $F[\mu X.F/X]$. The constructs `fold` and `unfold` represent that isomorphism; they have the rules:

$$\frac{O, X \vdash F[X] \qquad O; E \vdash e : F[\mu X.F]}{O; E \vdash \mathtt{fold}_{\mu X.F}(e) : \mu X.F}$$

$$\frac{O; E \vdash e : \mu X.F[X]}{O; E \vdash \mathtt{unfold}(e) : F[\mu X.F]}$$

Recursive definitions. In the remainder of the paper we often need to give recursive definitions of functions. FPC does not have any primitive facilities for such definitions but, as is well known, one can construct a call-by-value fixed-point operator in FPC. In its definition, which we give next, we write $a \triangleq e : A$ to mean that a equals e by definition and has type A, leaving implicit the context for the typing. Assuming that X does not occur free in A or B, we define $\mathtt{fix}_{A \to B}$ by:

$$\begin{array}{rcl} \mathtt{fix}_{A\to B} & \triangleq & \lambda f{:}\,(A \to B) \to A \to B. \\ & & \quad \mathtt{V}_{A\to B}\ (\mathtt{fold}_{\mu X.X\to A\to B}\ \mathtt{V}_{A\to B}) \\ & & : ((A \to B) \to A \to B) \to A \to B \\ \mathtt{V}_{A\to B} & \triangleq & \lambda x{:}\,\mu X.X \to A \to B.\lambda y{:}\,A. \\ & & \quad f\ (\mathtt{unfold}(x)(x))\ y \\ & & : (\mu X.X \to A \to B) \to A \to B \end{array}$$

Similarly, we can code mutually recursive definitions. For such definitions, we construct a term $\texttt{ffix}_{A\times B}$, à la Bekič:

$$\begin{aligned} \texttt{ffix}_{A\times B} \;\triangleq\; & \lambda\Phi{:}\, A\times B \to A\times B.\\ & \quad (\phi^{\dagger}[\texttt{fix}_B\ \psi], \texttt{fix}_B\ \psi)\\ & : (A\times B \to A\times B) \to A\times B \end{aligned}$$

where

$$\begin{aligned} \phi &\triangleq \lambda z{:}\, A\times B.\, \texttt{fst}(\Phi z)\\ \varphi &\triangleq \lambda z{:}\, A\times B.\, \texttt{snd}(\Phi z)\\ \phi^{\dagger}[x] &\triangleq \texttt{fix}_A(\lambda y{:}\, A.\, \phi(y,x))\\ \psi &\triangleq \lambda x{:}\, B.\, \varphi(\phi^{\dagger}[x], x) \end{aligned}$$

Omitting type information, we write $\lambda(x,y).\, e[x,y]$ for $\lambda z.\, e[\texttt{fst}(z), \texttt{snd}(z)]$ and

$$\texttt{ffix}\left\{\begin{array}{rcl} y &=& \phi[y,x]\\ x &=& \varphi[y,x] \end{array}\right\}$$

for $\texttt{ffix}_{A\times B}(\lambda(y,x){:}\, A\times B.\, (\phi[y,x], \varphi[y,x]))$. This is the pair of solutions to the equations $y = \phi[y,x]$ and $x = \varphi[y,x]$.

2.3 FPC with equality

Another possible way to complete the definition of FPC is to introduce a notion of type equality, and to say that if a term has a type then it has all equal types:

$$\text{(Retyping)}\quad \frac{O;E \vdash e : A \qquad O \vdash A = B}{O;E \vdash e : B}$$

There are at least two possible axiomatisations of type equality, and they induce two variants of FPC with equality. In the first variant, the type $\mu X.F$ equals its expansion $F[\mu X.F]$. In the second, very roughly, the type $\mu X.F$ equals its "infinite" expansion. The first variant includes the following rules for type equality:

$$\text{(Symm)}\quad \frac{O \vdash A = B}{O \vdash B = A}$$

$$\text{(Trans)}\quad \frac{O \vdash A = B \qquad O \vdash B = C}{O \vdash A = C}$$

$$(X)\quad \frac{O \vdash \diamond}{O \vdash X = X}\quad X \in O$$

$$(\star)\quad \frac{\begin{array}{c} O \vdash A = A' \\ O \vdash B = B' \end{array}}{O \vdash A \star B = A' \star B'}\quad \star \in \{\times, \to, +\}$$

$$(\mu)\quad \frac{O, X \vdash F = G}{O \vdash \mu X.F = \mu X.G}$$

$$\text{(Fold/Unfold)}\quad \frac{O, X \vdash F[X]}{O \vdash F[\mu X.F] = \mu X.F}$$

We follow the approach of Amadio and Cardelli for formulating the second variant. We say that a type $F[X]$ is contractive in X (and write $F[X] \succ X$) if every occurrence of X in $F[X]$ is under an operator $\times$, $\to$, or $+$. More precisely, $F[X]$ is contractive in X if X does not occur free in $F[X]$; if $F[X]$ is of the form $F_1[X] \star F_2[X]$ for $\star \in \{\times, \to, +\}$; or if $F[X]$ is of the form $\mu Y.F_1[X]$ where $F_1[X]$ is contractive in X. The second variant extends the first variant with one rule:

$$\text{(Contract)}\quad \frac{\begin{array}{c} O \vdash C[A] = A \\ O \vdash C[B] = B \end{array}}{O \vdash A = B}\quad C[X] \succ X$$

2.4 Isomorphism versus equality

From now on, whenever we mention FPC without qualification, we mean the formulation of FPC with isomorphism; we refer to the first variant of FPC with equality as $\mathrm{FPC}_=$ and to the second as $\mathrm{FPC}_{=+}$. When there could otherwise be ambiguity, we use $\vdash$, $\vdash_=$, and $\vdash_{=+}$ to indicate provability in these systems.

Each variant of FPC has advantages. FPC with isomorphism has a somewhat simpler proof theory and semantics; it can also be argued that it is clearer. On the other hand, the notion of type equality can be appealing for programming, particularly in the presence of a subtyping relation (e.g., as in Modula-3 [16]); hence we may prefer $\mathrm{FPC}_=$ or $\mathrm{FPC}_{=+}$. Of these, $\mathrm{FPC}_=$ is easier to formulate and to understand, while $\mathrm{FPC}_{=+}$ is more liberal. We know advocates of at least two variants of FPC. We do not suggest that one variant of FPC is "right"; rather, we aim to clarify the relations between the variants.

We start with some observations:

- First, $\mathrm{FPC}_=$ is strictly weaker than $\mathrm{FPC}_{=+}$, as discussed by Amadio and Cardelli. For example, we obtain $\emptyset \vdash_{=+} \mu X.\mu Y.(X \to Y) = \mu X.(X \to X)$ but not $\emptyset \vdash_= \mu X.\mu Y.(X \to Y) = \mu X.(X \to X)$.

- FPC can be embedded into $\mathrm{FPC}_=$ by deleting all occurrences of the `fold` and `unfold` constructs. This embedding preserves provability.

- There are typings of $\mathrm{FPC}_=$ that cannot be obtained in FPC. For example, the self-application xx can be typed in $\mathrm{FPC}_=$ (with $x : \mu X.(X \to X)$) but not in FPC.

$$
\begin{array}{rcl}
\mathtt{id}_A & \triangleq & (\lambda x{:}\, A.\, x) : A \to A \\
\circ_{A,B,C} & \triangleq & (\lambda(g,f){:}\, (B \to C) \times (A \to B).\, \lambda x{:}\, A.\, g(f(x))) \\
& & : (B \to C) \times (A \to B) \to A \to C \\
\mathtt{fd}_{\mu X.F} & \triangleq & (\lambda f{:}\, F[\mu X.F/X].\, \mathtt{fold}_{\mu X.F}(f)) : F[\mu X.F/X] \to \mu X.F \\
\mathtt{ud}_{\mu X.F} & \triangleq & (\lambda x{:}\, \mu X.F.\, \mathtt{unfold}(x)) : \mu X.F \to F[\mu X.F/X] \\
\times_{(A_1,A_2),(B_1,B_2)} & \triangleq & \lambda(f,g){:}\, (A_1 \to B_1) \times (A_2 \to B_2).\, \lambda(x,y){:}\, A_1 \times A_2.\, (f(x), g(y)) \\
& & : (A_1 \to B_1) \times (A_2 \to B_2) \to (A_1 \times A_2) \to (B_1 \times B_2) \\
\to_{(A_1,A_2),(B_1,B_2)} & \triangleq & \lambda(f,g){:}\, (B_1 \to A_1) \times (A_2 \to B_2).\, \lambda h{:}\, A_1 \to A_2.\, \lambda x{:}\, B_1.\, g(h(f(x))) \\
& & : (B_1 \to A_1) \times (A_2 \to B_2) \to (A_1 \to A_2) \to (B_1 \to B_2) \\
+_{(A_1,A_2),(B_1,B_2)} & \triangleq & \lambda(f,g){:}\, (A_1 \to B_1) \times (A_2 \to B_2).\, \lambda z{:}\, A_1 + A_2. \\
& & \quad \mathtt{case}\ z\ \mathtt{of}\ \mathtt{l}(x).\mathtt{inl}_{B_1,B_2}(f(x))\ \mathtt{or}\ \mathtt{r}(y).\mathtt{inr}_{B_1,B_2}(g(y)) \\
& & : (A_1 \to B_1) \times (A_2 \to B_2) \to (A_1 + A_2) \to (B_1 + B_2)
\end{array}
$$

Figure 1. Auxiliary term definitions

$$
\begin{array}{rcl}
Y_{\overline{A},\overline{B}}(\overline{t,s}) & \triangleq & \mathtt{id}_Y \text{ for } Y \neq X_1, \ldots, X_n \\
X_{i\,\overline{A},\overline{B}}(\overline{t,s}) & \triangleq & s_i \\
(G \times G')_{\overline{A},\overline{B}}(\overline{t,s}) & \triangleq & G_{\overline{A},\overline{B}}(\overline{t,s}) \times G'_{\overline{A},\overline{B}}(\overline{t,s}) \\
(G \to G')_{\overline{A},\overline{B}}(\overline{t,s}) & \triangleq & G_{\overline{B},\overline{A}}(\overline{s,t}) \to G'_{\overline{A},\overline{B}}(\overline{t,s}) \\
(G + G')_{\overline{A},\overline{B}}(\overline{t,s}) & \triangleq & G_{\overline{A},\overline{B}}(\overline{t,s}) + G'_{\overline{A},\overline{B}}(\overline{t,s}) \\
(\mu X_{n+1}.G)_{\overline{A},\overline{B}}(\overline{t,s}) & \triangleq & \mathtt{snd}(\mathtt{ffix} \left\{ \begin{array}{lcl} v & = & \mathtt{fd}_{A_{n+1}} \circ G_{\overline{B};B_{n+1},\overline{A};A_{n+1}}(\overline{s,t}; u, v) \circ \mathtt{ud}_{B_{n+1}} \\ u & = & \mathtt{fd}_{B_{n+1}} \circ G_{\overline{A};A_{n+1},\overline{B};B_{n+1}}(\overline{t,s}; v, u) \circ \mathtt{ud}_{A_{n+1}} \end{array} \right\}) \\
& & \text{where } A_{n+1} = (\mu X_{n+1}.G)[A_1, \ldots, A_n] \text{ and } B_{n+1} = (\mu X_{n+1}.G)[B_1, \ldots, B_n]
\end{array}
$$

Figure 2. Definition of functors

In section 4, we show that in fact it is possible to embed $\mathrm{FPC}_=$ and $\mathrm{FPC}_{=+}$ into FPC by inserting coercion functions (for example, rewriting xx as $\mathtt{unfold}(x)(x)$).

3 Type expressions as functors

In this section we develop a syntactic interpretation of type expressions as functors. Despite the categorical source of our method, we do not discuss categorical ideas until late in this section, in order to emphasize that our results, though suggested by category theory, may be understood without it.

Definitions. For types $F[X]$, A, and B, and terms $s : A \to B$ and $t : B \to A$, we define a term $F[X]_{A,B}(t,s) : F[A] \to F[B]$. As we show in a later section, $F_{\_,\_}(\_,\_)$ has useful properties (such as functoriality). Here we are not yet concerned with the exact nature of those properties, but we try to give a non-trivial definition. For example, we try to avoid defining $F_{A,B}(t,s)$ so that it is a constant function, and instead try to put all inputs (A, B, s, t) to good use.

In order to be more precise, we introduce some auxiliary notation. We write $\overline{A}$ for $A_1; \ldots; A_n$ and $\overline{B}$ for $B_1; \ldots; B_n$, and $\overline{t,s}$ for $t_1, s_1; \ldots; t_n, s_n$. We also write $\lambda(x,y) : A \times B.\, e[x,y]$ for $\lambda z : A \times B.\, e[\mathtt{fst}(z), \mathtt{snd}(z)]$, and make the definitions given in Figure 1. We use the operators $\circ$, $\times$, $\to$, and $+$ of Figure 1 as infix operators on terms, omitting type subscripts. We write $O; E \vdash s : A \leftrightarrows B : t$ for $O; E \vdash s : A \to B$ and $O; E \vdash t : B \to A$.

Definition 3.1 Let $O, X_1, \ldots, X_n \vdash F[X_1, \ldots, X_n]$, and $O; E \vdash s_i : A_i \leftrightarrows B_i : t_i$ for $i = 1, \ldots, n$. We define $F_{\overline{A},\overline{B}}(\overline{t,s})$ so that:

$$O; E \vdash F_{\overline{A},\overline{B}}(\overline{t,s}) : F[A_1, \ldots, A_n] \to F[B_1, \ldots, B_n]$$

The definition is by induction on the structure of F, and is given in Figure 2. $\square$

The definition of $F_{\overline{A},\overline{B}}(\overline{t,s})$ is the basis for other useful definitions, such as the following definition for *iterators* and *coiterators*, which we need in the rest of the paper:

Definition 3.2 For $O, X \vdash F[X]$ and $O \vdash A$, we define $\mathtt{coit} : (A \to F[A]) \times (F[A] \to A) \to A \to \mu X.F$ and $\mathtt{it} : (A \to F[A]) \times (F[A] \to A) \to \mu X.F \to A$:

$$\mathtt{coit} \;\triangleq\; \lambda(g,f){:}\,(A \to F[A]) \times (F[A] \to A).\; \mathtt{fst}(\mathtt{ffix}\left\{\begin{array}{lcl} v & = & \mathtt{fd} \circ F(u,v) \circ g \\ u & = & f \circ F(v,u) \circ \mathtt{ud} \end{array}\right\})$$

$$\mathtt{it} \;\triangleq\; \lambda(g,f){:}\,(A \to F[A]) \times (F[A] \to A).\; \mathtt{snd}(\mathtt{ffix}\left\{\begin{array}{lcl} v & = & \mathtt{fd} \circ F(u,v) \circ g \\ u & = & f \circ F(v,u) \circ \mathtt{ud} \end{array}\right\})$$

We write $\mathtt{it}\begin{pmatrix} t \\ s \end{pmatrix}$ for $\mathtt{it}(t,s)$ and $\mathtt{coit}\begin{pmatrix} t \\ s \end{pmatrix}$ for $\mathtt{coit}(t,s)$. $\square$

Categorical ideas. The category-theoretic solution of recursive type equations is based on the following, by now standard, ideas: types constitute objects of a category, and type constructors are functors on the category of types. Those ideas are the origin of the definition of types as functors that we have given.

The treatment of type constructors as functors is difficult because of contravariance. A solution is to take the category of types to be involutory (i.e., isomorphic to its dual via an involution) [8]. Given a category $\mathcal{C}$ there are various approaches to constructing an involutory category $\mathrm{I}(\mathcal{C})$. In the context of order-enriched categories one can exploit the duality embedding/projection [21]. There are also canonical choices, namely $\breve{\mathcal{C}} = \mathcal{C}^{\mathrm{op}} \times \mathcal{C}$ [11] or its subcategory $\mathcal{C}^D$ [1, 10] consisting of all those objects on the diagonal. In any case type constructors with n free type variables are viewed as symmetric functors $\mathrm{I}(\mathcal{C})^n \to \mathrm{I}(\mathcal{C})$, induced by functors $\mathrm{I}(\mathcal{C})^n \to \mathcal{C}$.

In our definitions, we are constrained by syntax. Interestingly, the syntax seems to steer us away from the approach based on $\breve{\mathcal{C}}$, and towards that based on $\mathcal{C}^D$. Within this latter approach, there are two possible ways to proceed, which correspond respectively to viewing type constructors as symmetric functors $(\mathcal{C}^D)^n \to \mathcal{C}^D$ or as functors $(\mathcal{C}^D)^n \to \mathcal{C}$. For instance, given $F[X]$, to every $s : A \to B$ and $t : B \to A$ we could either associate a term $F^D(t,s) : (F[A] \to F[B]) \times (F[B] \to F[A])$, or a term $F(t,s) : F[A] \to F[B]$. We have essentially taken the second approach, which is more concise.

4 Formulations of recursive types, revisited

We now return to the problem of relating the formulations of FPC. In short, we obtain:

1. For every equality $A = B$ provable in either $\mathrm{FPC}_=$ or $\mathrm{FPC}_{=+}$ there is a corresponding pair of coercion functions $s : A \leftrightarrows B : t$. The definition of these functions is based on similar definitions used for the syntactic treatment of functors in section 3.

$$(\text{Symm})\ \frac{O;\emptyset \vdash s : A \leftrightarrows B : t}{O;\emptyset \vdash t : B \leftrightarrows A : s} \qquad (\text{Trans})\ \frac{O;\emptyset \vdash s : A \leftrightarrows B : t \quad O;\emptyset \vdash s' : B \leftrightarrows C : t'}{O;\emptyset \vdash s' \circ s : A \leftrightarrows C : t \circ t'}$$

$$(X)\ \frac{O \vdash \diamond}{O;\emptyset \vdash \mathtt{id}_X : X \leftrightarrows X : \mathtt{id}_X}\ X \in O \qquad (\star)\ \frac{O;\emptyset \vdash s : A \leftrightarrows A' : t \quad O;\emptyset \vdash s' : B \leftrightarrows B' : t'}{O;\emptyset \vdash s \star s' : A \star B \leftrightarrows A' \star B' : t \star t'}\ \star \in \{\times, +\}$$

$$(\to)\ \frac{O;\emptyset \vdash s : A \leftrightarrows A' : t \quad O;\emptyset \vdash s' : B \leftrightarrows B' : t'}{O;\emptyset \vdash t \to s' : (A \to B) \leftrightarrows (A' \to B') : s \to t'}$$

$$(\mu)\ \frac{O, X;\emptyset \vdash s[X] : F \leftrightarrows G : t[X]}{O;\emptyset \vdash \mathtt{coit}\begin{pmatrix} s[\mu X.F] \circ \mathtt{ud}_{\mu X.F} \\ \mathtt{fd}_{\mu X.F} \circ t[\mu X.F] \end{pmatrix} : \mu X.F \leftrightarrows \mu X.G : \mathtt{it}\begin{pmatrix} s[\mu X.F] \circ \mathtt{ud}_{\mu X.F} \\ \mathtt{fd}_{\mu X.F} \circ t[\mu X.F] \end{pmatrix}}$$

$$(\text{Contract})\ \frac{O;\emptyset \vdash s : C[A] \leftrightarrows A : t \quad O;\emptyset \vdash s' : C[B] \leftrightarrows B : t'}{O;\emptyset \vdash \mathtt{it}(t', s') \circ \mathtt{coit}(t, s) : A \leftrightarrows B : \mathtt{it}(t, s) \circ \mathtt{coit}(t', s')}\ C[X] \succ X$$

Figure 3. Definition of coercion functions

2. This easily implies that for any typing $e : C$ of either $\text{FPC}_=$ or $\text{FPC}_{=+}$ we can construct a corresponding typing $e' : C$ in FPC, where e' is obtained from e by inserting coercion functions.

The construction of the coercion functions is fairly difficult, particularly because of the possibility of nested recursion. For example, the equality

$$\begin{aligned} &\mu X.\mu Y.(Z + (Y \to X)) \\ &\quad = \mu X.(Z + ((\mu Y.(Z + (Y \to X))) \to X)) \end{aligned}$$

is provable in $\text{FPC}_=$ by expanding the inner recursive type; the proof in $\text{FPC}_=$ is straightforward, but the corresponding coercion functions are rather sophisticated.

Formally, we proceed as follows. For every proof of an equality $O \vdash_= A = B$ or $O \vdash_{=+} A = B$ we construct a pair of terms s and t such that $O;\emptyset \vdash s : A \leftrightarrows B : t$. The construction of s and t is by induction on the proof of the equality $A = B$; we give it in Figure 3.

Using the pairs of terms of Figure 3, we deduce that for every derivation of $O;E \vdash_= e : C$ or $O;E \vdash_{=+} e : C$ there exists a corresponding derivation $O;E \vdash e' : C$. This derivation is obtained by replacing applications of the rule (Retyping) (from $O;E \vdash e : A$ and $O \vdash A = B$ to $O;E \vdash e : B$) with applications of $\to$ elimination (from $O;E \vdash e : A$ and $O;E \vdash s : A \to B$ to $O;E \vdash s(e) : B$).

Now that we have defined and used coercion functions, we face the problem of proving that they are reasonable (e.g., that they do not simply diverge). The equational rules β and η do not suffice for this proof, and we cannot use Scott induction directly (since the predicates in which we are interested are not admissible). Instead, we rely on a logic, which we introduce in section 5; we derive properties of the coercion functions in section 6.5.

5 A basic logic for FPC

We introduce a logic for reasoning about FPC programs. The logic is suggested by a call-by-value interpretation of FPC. It is a *logic of definedness* in the sense of Feferman [7] (or a *logic of partial terms* [4, 5]); it enables us to reason about terms which may or may not have a value. The use of a logic of definedness is important for two reasons:

1. In FPC, one can write terms that do not normalize.

2. Our main means for reasoning about recursion is Freyd's *Principle of Versality* (or PV) [11]. This principle is compatible with the theory of the typed partial λ-calculus [15] formalisable within a logic of definedness, but is inconsistent with the equational theory of the simply-typed λ-calculus. This inconsistency arises because $\mu X.X$ becomes a zero object (i.e., both initial and terminal), and then is isomorphic to every type.

Next we define the logic, and in section 6 we show some of its applications.

The language of the logic consists of the many-sorted term algebra given by FPC types and terms, and the positive fragment of propositional calculus extended with equality ($=_A$) and a definedness predicate ($\downarrow_A$). More precisely, the following rules define the well-formed formulas of the logic:

$$\frac{O; E \vdash e : A \qquad O; E \vdash e' : A}{O; E \vdash e =_A e' \text{ wff}}$$

$$\frac{O; E \vdash e : A}{O; E \vdash e \downarrow_A \text{ wff}}$$

$$\frac{O; E \vdash P \text{ wff} \qquad O; E \vdash Q \text{ wff}}{O; E \vdash P \wedge Q \text{ wff}}$$

$$\frac{O; E \vdash P \text{ wff} \qquad O; E \vdash Q \text{ wff}}{O; E \vdash P \supset Q \text{ wff}}$$

Officially, the axioms and rules of the logic deal with judgments of the form $O; E \vdash P$, which means that P is true given O and E. In what follows, we omit O and E; the axioms apply for any O and E such that $O; E \vdash P$ wff. We also omit types for free variables and in subscripts. The axioms and rules are:

I. Natural-deduction rules for $\wedge$ and $\supset$:

$$\frac{P \qquad Q}{P \wedge Q} \qquad \frac{P \wedge Q}{P} \qquad \frac{P \wedge Q}{Q}$$

$$\frac{\begin{matrix}[P]\\ \vdots \\ Q\end{matrix}}{P \supset Q} \qquad \frac{P \qquad P \supset Q}{Q}$$

II. In logics of definedness, free variables are thought of as ranging over terms that exist (or denote a value). Hence, from a general assertion $P[x]$ we can infer the assertion $P[e]$ for a particular term e provided e exists:

$$\frac{e\downarrow \qquad P[x]}{P[e]} \quad x \text{ not free in assumptions above } P[x]$$

III. The assertion of an equality implicitly carries the assertion of the existence of the equated terms:

$s = t \supset s\downarrow \wedge\, t\downarrow$

IV. The axioms for equality are as usual, except for substitutivity which holds only for the weaker notion of equality $\simeq_A$ (Kleene equality) that does not assert the existence of the equated terms. Formally, $s \simeq_A t$ stands for $(s \downarrow_A \supset s =_A t) \wedge (t \downarrow_A \supset s =_A t)$.

$x = x$
$x = y \supset y = x$
$x = y \wedge y = z \supset x = z$
$\bigwedge_{1 \le i \le n}(s_i = t_i) \supset e[s_1, \ldots, s_n] \simeq e[t_1, \ldots, t_n]$

V. For products, we have:

$(s, t)\downarrow \supset s\downarrow \wedge\, t\downarrow$
$\mathtt{fst}(e)\downarrow \supset e\downarrow$
$\mathtt{snd}(e)\downarrow \supset e\downarrow$
$\mathtt{fst}(x, y) = x$
$\mathtt{snd}(x, y) = y$
$(\mathtt{fst}(z), \mathtt{snd}(z)) = z$

VI. For exponentials, the axioms are those of the typed partial λ-calculus [18, 14, 15]:

$(\lambda x.\, e)\downarrow$
$s(t)\downarrow \supset s\downarrow \wedge\, t\downarrow$
$\lambda x.\, e[x] = \lambda y.\, e[y]$
$(\lambda x.\, e[x])(y) \simeq e[y]$
$\lambda x.\, fx = f$

$$\frac{s[x] \simeq t[x]}{\lambda x.\, s[x] = \lambda x.\, t[x]} \quad x \text{ not free in assumptions above } s[x] \simeq t[x]$$

Since the substitutivity axiom applies only to terms that are known to exist, the law for β equality is not equational but *conditionally* equational; we can derive: $t\downarrow \supset (\lambda x.\, e[x])(t) \simeq e[t]$. This is in accordance with the call-by-value interpretations of FPC. (Similar remarks apply to products and sums.)

VII. For sums, we have:

$\mathtt{inl}(e)\downarrow \supset e\downarrow$
$\mathtt{inl}(x)\downarrow$
$\mathtt{inr}(e)\downarrow \supset e\downarrow$
$\mathtt{inr}(y)\downarrow$
$(\mathtt{case}\ e\ \mathtt{of}\ \mathtt{l}(x).s\ \mathtt{or}\ \mathtt{r}(y).t)\downarrow \supset e\downarrow$
$(\mathtt{case}\ z\ \mathtt{of}\ \mathtt{l}(x).s[x]\ \mathtt{or}\ \mathtt{r}(y).t[y])$
$\quad \simeq (\mathtt{case}\ z\ \mathtt{of}\ \mathtt{l}(x').s[x']\ \mathtt{or}\ \mathtt{r}(y').t[y'])$
$(\mathtt{case}\ \mathtt{inl}(z)\ \mathtt{of}\ \mathtt{l}(x).s[x]\ \mathtt{or}\ \mathtt{r}(y).t) \simeq s[z]$
$(\mathtt{case}\ \mathtt{inr}(z)\ \mathtt{of}\ \mathtt{l}(x).s\ \mathtt{or}\ \mathtt{r}(y).t[y]) \simeq t[z]$
$(\mathtt{case}\ z\ \mathtt{of}\ \mathtt{l}(x).e(\mathtt{inl}(x))\ \mathtt{or}\ \mathtt{r}(y).e(\mathtt{inr}(y)))$
$\quad \simeq e(z) \qquad x, y \text{ not free in } e$

$$\frac{s[x] \simeq s'[x] \qquad t[y] \simeq t'[y]}{\begin{matrix}(\mathtt{case}\ z\ \mathtt{of}\ \mathtt{l}(x).s[x]\ \mathtt{or}\ \mathtt{r}(y).t[y]) \\ \simeq (\mathtt{case}\ z\ \mathtt{of}\ \mathtt{l}(x).s'[x]\ \mathtt{or}\ \mathtt{r}(y).t'[y])\end{matrix}} \quad x, y \text{ not free in assumptions above}$$

VIII. For recursion, we have:

$\texttt{fold}(e)\downarrow \supset e\downarrow$

$\texttt{fold}(x)\downarrow$

$\texttt{unfold}(e)\downarrow \supset e\downarrow$

$\texttt{unfold}(x)\downarrow$

$\texttt{unfold}(\texttt{fold}(x)) = x$

$\texttt{fold}(\texttt{unfold}(x)) = x$

$\texttt{it}(\texttt{ud}_{\mu X.F}, \texttt{fd}_{\mu X.F}) = \texttt{id}_{\mu X.F}$

$\texttt{coit}(\texttt{ud}_{\mu X.F}, \texttt{fd}_{\mu X.F}) = \texttt{id}_{\mu X.F}$

PV: for $f : F[A] \leftrightarrows A : g$, $v : A \leftrightarrows \mu X.F[X] : u$,

$$(v = \texttt{fd} \circ F_{A,\mu X.F}(u, v) \circ g) \wedge (u = f \circ F_{\mu X.F,A}(v, u) \circ \texttt{ud}) \supset v = \texttt{coit}(g, f) \wedge u = \texttt{it}(g, f)$$

This logic is essentially an extension of that of Plotkin [18] with new axioms for recursion. Three minor differences are worth noting. First, Plotkin works with a *logic of existence* [7] (or a *logic of partial elements* [20, 9]), where free variables are thought of as ranging over a domain of possibly non-existing elements. Second, he axiomatises approximation ($\sqsubseteq$, $\precsim$) rather than equality. Finally, as here we are interested only in the minimal framework in which our results hold, we have adopted a weaker axiomatisation for sums which allows us to remain within the positive fragment of the propositional calculus.

The standard model of FPC, **pCpo** (the category of cpos and partial continuous functions), gives rise to a model of the logic; equality and existence have their usual meanings (see [18]).

6 Applications of the logic

In this section we develop several applications of our logic. We have two goals at once: (i) to explain the logic, and in particular the Principle of Versality; (ii) to prove that the definitions given in other sections (e.g., that of functors) yield desirable properties. Throughout, we reason rather informally; we often omit types, and when we assert an equality, we mean that it is provable in the logic.

6.1 Basic category theory

The first results are monoidal laws for identities and composition: $e \circ \texttt{id}_A \simeq e$, $\texttt{id}_B \circ e \simeq e$, and $t \circ (s \circ e) \simeq (t \circ s) \circ e$ are provable (for appropriate A and B).

In addition, $\times$, $\rightarrow$, and $+$ are functorial in the following sense. We have $\texttt{id}_A \star \texttt{id}_B = \texttt{id}_{A \star B}$ for $\star \in \{\times, \rightarrow, +\}$, and $(s' \times t') \circ (s \times t) \simeq (s' \circ s) \times (t' \circ t)$ and $(s' + t') \circ (s + t) \simeq (s' \circ s) + (t' \circ t)$, while $(s' \rightarrow t') \circ (s \rightarrow t) \simeq (s \circ s') \rightarrow (t' \circ t)$.

6.2 Fixed-point operators

The operators of section 2.2 are provably fixed-point operators; moreover, coiterators and iterators satisfy the equations used for their recursive definitions:

$$\begin{aligned}
\texttt{fix} &= \lambda f.\, \lambda y.\, f\ (\texttt{fix}\ f)\ y\\
\texttt{ffix} &= \lambda f.\, (\lambda y.\, \texttt{fst}\ (f(\texttt{ffix}\ f))\ y,\\
&\qquad \lambda x.\, \texttt{snd}\ (f(\texttt{ffix}\ f))\ x)\\
\texttt{coit} &= \lambda(g, f).\, \texttt{fd} \circ F(\texttt{it}(g, f), \texttt{coit}(g, f)) \circ g\\
\texttt{it} &= \lambda(g, f).\, f \circ F(\texttt{coit}(g, f), \texttt{it}(g, f)) \circ \texttt{ud}
\end{aligned}$$

6.3 Functoriality

Crucially, our functorial interpretation of type expressions does define functors:

Theorem 6.1 (Functoriality) Let $O, X_1, \ldots, X_n \vdash F[X_1, \ldots, X_n]$.

1. Existence: If $O; E \vdash f_i : A_i \leftrightarrows B_i : g_i$, then $F_{\overline{A},\overline{B}}(\overline{g, f})\downarrow$.
2. Identities are preserved: If $O \vdash A_i$ for $i = 1..n$, then $F_{\overline{A},\overline{A}}(\overline{\texttt{id}_A, \texttt{id}_A}) = \texttt{id}_{F[\overline{A}]}$ where $(\texttt{id}_A)_i \triangleq \texttt{id}_{A_i}$.
3. Composition is preserved: If $O; E \vdash f_i : A_i \leftrightarrows B_i : g_i$ and $O; E \vdash f'_i : B_i \leftrightarrows C_i : g'_i$, for $i = 1..n$, then $F_{\overline{B},\overline{C}}(\overline{g', f'}) \circ F_{\overline{A},\overline{B}}(\overline{g, f}) = F_{\overline{A},\overline{C}}(\overline{g \circ g', f' \circ f})$ where $(g \circ g')_i \triangleq g_i \circ g'_i$ and $(f' \circ f)_i \triangleq f'_i \circ f_i$. $\square$

The proof of this theorem relies on the Principle of Versality.

6.4 Example applications of the Principle of Versality

We show three additional consequences of the Principle of Versality.

Example 6.2 (PV for $\texttt{Nat}$) Let $\texttt{0} \triangleq \mu X.X$, $\texttt{1} \triangleq \texttt{0} \rightarrow \texttt{0}$, $\texttt{Nat} \triangleq \mu X.\texttt{1} + X$, and then let $\texttt{*} \triangleq \lambda x{:}\,\texttt{0}.\, x$, $\texttt{zero} \triangleq \texttt{fold}_{\texttt{Nat}}(\texttt{inl}_{\texttt{1},\texttt{Nat}}(\texttt{*}))$, and $\texttt{succ}(x) \triangleq \texttt{fold}_{\texttt{Nat}}(\texttt{inr}_{\texttt{1},\texttt{Nat}}(x))$. For $f : \texttt{1} + A \leftrightarrows A : g$, we have the coiterator $\texttt{coit}(g, f) : A \rightarrow \texttt{Nat}$ such that

$$\texttt{coit}(g, f)(z) \simeq \texttt{case}\ g(z)\ \texttt{of}\ \texttt{l}(x).\texttt{zero}\ \texttt{or}\ \texttt{r}(y).\texttt{succ}(\texttt{coit}(g, f)(y))$$

so that, roughly, whenever it is defined, $\mathtt{coit}(g,f)(z)$ is the least n for which $g^{n+1}(z) = \mathtt{inl}(*)$. We have also the iterator $\mathtt{it}(g,f) : \mathtt{Nat} \to A$ for which

$$\begin{array}{lcl} \mathtt{it}(g,f)(z) & \simeq & \mathtt{case}\ \mathtt{unfold}(z)\ \mathtt{of}\ \mathtt{l}(x).f(\mathtt{inl}(*)) \\ & & \quad \mathtt{or}\ \mathtt{r}(y).f(\mathtt{inr}(\mathtt{it}(g,f)(y))) \end{array}$$

so $\mathtt{it}(g,f)(\mathtt{zero}) \simeq f(\mathtt{inl}(*))$ and $\mathtt{it}(g,f)(\mathtt{succ}(x)) \simeq f(\mathtt{inr}(\mathtt{it}(g,f)(x)))$. Thus coiterators allow definitions by *minimisation* (in the sense of recursion theory), while iterators provide definitions by *primitive recursion.* □

Example 6.3 (PV for 0) Let $\mathtt{0} \triangleq \mu X.X$. By PV, for $g, f : A \to A$, the type $\mathtt{0}$ comes equipped with a unique coiterator and iterator characterised by:

$$\begin{tikzcd} \mathtt{0} \arrow[r, "\mathtt{fd}"] \arrow[d, "\mathtt{it}(g,f)"'] & \mathtt{0} \arrow[d, "\mathtt{it}(g,f)"] \\ A \arrow[r, "f"'] & A \end{tikzcd}$$

$$\begin{tikzcd} \mathtt{0} & \mathtt{0} \arrow[l, "\mathtt{ud}"'] \\ A \arrow[u, "\mathtt{coit}(g,f)"] & A \arrow[l, "g"] \arrow[u, "\mathtt{coit}(g,f)"'] \end{tikzcd}$$

With $A = \mathtt{0}$, $g = \mathtt{ud}$, $f = \mathtt{fd}$, we deduce $\mathtt{fd} = \mathtt{it}(\mathtt{ud},\mathtt{fd}) = \mathtt{id}_{\mathtt{0}} = \mathtt{coit}(\mathtt{ud},\mathtt{fd}) = \mathtt{ud}$. Then, for $v : A \leftrightarrows \mathtt{0} : u$, we obtain $v = \mathtt{fd} \circ v \circ \mathtt{id}_A$ and $u = \mathtt{id}_A \circ u \circ \mathtt{ud}$, and hence $v = \mathtt{coit}(\mathtt{id}_A, \mathtt{id}_A)$ and $u = \mathtt{it}(\mathtt{id}_A, \mathtt{id}_A)$. Thus, $\mathtt{0}$ is a zero object. □

Example 6.4 (PV and retracts) For $h : \mu X.F \to \mu X.F$, by PV we have

$$h = \mathtt{fd} \circ F(h,h) \circ \mathtt{ud} \supset h = \mathtt{it}(\mathtt{ud},\mathtt{fd}) = \mathtt{id}$$

Further, for $f : F[A] \leftrightarrows A : g$ such that $g \circ f = \mathtt{id}_{F[A]}$ (that is, $F[A]$ is provably a retract of A), we have that $\mathtt{coit}(g,f) \circ \mathtt{it}(g,f) = \mathtt{id}_{\mu X.F}$ (that is, $\mu X.F$ is provably a retract of A), since $\mathtt{coit}(g,f) \circ \mathtt{it}(g,f) = \mathtt{fd} \circ F(\mathtt{it}(g,f), \mathtt{coit}(g,f)) \circ v \circ u \circ F(\mathtt{coit}(g,f), \mathtt{it}(g,f)) \circ \mathtt{ud}$ and then $= \mathtt{fd} \circ F(\mathtt{coit}(g,f) \circ \mathtt{it}(g,f), \mathtt{coit}(g,f) \circ \mathtt{it}(g,f)) \circ \mathtt{ud}$. □

6.5 Properties of coercion functions

We return once more to the problem of relating the formulations of FPC. In section 4 we have seen that for every equality $A = B$ provable in either $\mathrm{FPC}_=$ or $\mathrm{FPC}_{=+}$ there is a corresponding pair of coercion functions $s : A \leftrightarrows B : t$; and for every typing $e : C$ there is a corresponding typing $e' : C$. Now we study some properties of s and t, showing that e and e' are equivalent.

In the case where $O \vdash_= A = B$, we can prove that s and t define an isomorphism. This means that $O; \emptyset \vdash t \circ s = \mathtt{id}_A$ and $O; \emptyset \vdash s \circ t = \mathtt{id}_B$ are both provable in the logic of section 5. We write this as $O; \emptyset \vdash s : A \cong B : t$. We further write $O \vdash A \cong B$ when $O; \emptyset \vdash s : A \cong B : t$ for some s and t.

Theorem 6.5 For every derivation of

$$O, X_1, \ldots, X_n \vdash_= F = G$$

with associated terms

$$O, X_1, \ldots, X_n; \emptyset \vdash s : F \leftrightarrows G : t$$

we have that

$$O, X_1, \ldots, X_n; \emptyset \vdash s : F \cong G : t$$

In addition, for $O; E \vdash f_i : A_i \leftrightarrows B_i : g_i$, the following naturality condition holds:

$$\begin{tikzcd} F[\overline{A}] \arrow[r, "s[\overline{A}]"] \arrow[d, "F(\overline{g,f})"'] & G[\overline{A}] \arrow[d, "G(\overline{g,f})"] \\ F[\overline{B}] \arrow[r, "s[\overline{B}]"'] & G[\overline{B}] \end{tikzcd}$$

$$\begin{tikzcd} F[\overline{A}] & G[\overline{A}] \arrow[l, "t[\overline{A}]"'] \\ F[\overline{B}] \arrow[u, "F(\overline{f,g})"] & G[\overline{B}] \arrow[l, "t[\overline{B}]"] \arrow[u, "G(\overline{f,g})"'] \end{tikzcd}$$

□

It follows that it is sound to interpret type equalities in $\mathrm{FPC}_=$ as isomorphisms in FPC:

Corollary 6.6 If $O \vdash_= A = B$ then $O \vdash A \cong B$. □

Furthermore, whenever we transform $O; E \vdash_= e : C$ into $O; E \vdash e' : C$, the terms e and e' are equivalent. In order to make this claim precise, we consider a formalism where both e and e' are legal terms; we extend the logic for FPC to the union of FPC and $\mathrm{FPC}_=$, with the following new axioms and rules:

I. The constructs `fold` and `unfold` are computationally irrelevant:

   $\mathtt{fold}_{\mu X.F}(x) =_{\mu X.F} x$

   $\mathtt{unfold}(x) =_{\mu X.F} x$

II. Equal types can be used interchangeably: whenever $A = B$,
$x =_A y \supset x =_B y$

We believe that the extended logic is sound for a per model, a call-by-value version of existing ones [2, 1].

With this extended logic, we obtain:

Lemma 6.7 If a proof of $O \vdash_= A = B$ yields the coercion functions s and t such that $O; \emptyset \vdash s : A \leftrightarrows B : t$ (according to Figure 3), then $O; \emptyset \vdash s = \mathtt{id}_A$ and $O; \emptyset \vdash t = \mathtt{id}_B$ are provable in the extended logic. □

Theorem 6.8 If $O; E \vdash_= e : C$ and e' is obtained from e by inserting coercion functions so that $O; E \vdash e' : C$ (as described in section 4), then $O; E \vdash e \simeq e'$ is provable in the extended logic. □

We cannot proceed identically for $\mathrm{FPC}_{=+}$, because the rule (Contract) which distinguishes $\mathrm{FPC}_{=+}$ from $\mathrm{FPC}_{=}$ is not even sound when we interpret equality as natural isomorphism. For example, let $C[X] = X \times \mathtt{1}$; clearly $C[X]$ is contractive in X, and any A and B are naturally isomorphic to $C[A]$ and $C[B]$ respectively; the rule (Contract) would tell us that A and B are themselves isomorphic. This observation does not mean that $\mathrm{FPC}_{=+}$ is unsound, only that the isomorphism interpretation of the rule (Contract) is.

Still, whenever we transform $O; E \vdash_{=+} e : C$ into $O; E \vdash e' : C$, the terms e and e' are equivalent. Analogues of Lemma 6.7 and Theorem 6.8 hold for $\mathrm{FPC}_{=+}$.

7 Conclusions

This paper explores some syntactic and logical issues for type systems with recursive types. Some of our results may be usually taken for granted. However, as we have found, they are rather delicate, and not easy reproductions of semantic results.

Much further work is possible. For example, it should be simple to extend our results to type systems with polymorphism. In a more logical direction, it may be interesting to revisit Pitts' relational approach to reasoning with recursive types, and in particular his treatment of nested recursion [17].

Acknowledgments

We thank Roberto Amadio, Luca Cardelli, John Mitchell, Andy Pitts, and Gordon Plotkin for discussions on the subject of this work.

References

[1] Martín Abadi and Gordon Plotkin. A per model of polymorphism and recursive types. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 355–365, 1990.

[2] Roberto M. Amadio. Recursion over realizability structures. *Information and Computation*, 91(1):55–85, 1991.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[4] M. Beeson. Formalizing constructive mathematics: why and how? In *Constructive Mathematics*, volume 873 of *Lecture Notes in Mathematics*, pages 146–190. Springer-Verlag, 1981.

[5] M. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[6] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–222, July 1991.

[7] S. Feferman. Definedness. Manuscript (Lecture for mini-conference on Partial Functions and Programming: Foundational Questions), May 1995.

[8] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. PhD thesis, University of Edinburgh, 1994. To be published by Cambridge University Press in the Distinguished Dissertations Series.

[9] M.P. Fourman. The logic of topoi. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic*. North Holland, 1977.

[10] P.J. Freyd. Recursive types reduced to inductive types. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 498–507, 1990.

[11] P.J. Freyd. Algebraically complete categories. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Category Theory*, volume 1488 of *Lecture Notes in Mathematics*, pages 131–156. Springer-Verlag, 1991.

[12] M.H. Gordon, R.M. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[13] Carl Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Series. The MIT Press, Cambridge, Massachusetts, 1992.

[14] E. Moggi. Categories of partial morphisms and the partial lambda-calculus. In *Proceedings Workshop on Category Theory and Computer Programming, Guildford 1985*, volume 240 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[15] E. Moggi. *The Partial Lambda-Calculus.* PhD thesis, University of Edinburgh, 1987.

[16] Greg Nelson, editor. *Systems Programming in Modula-3.* Prentice Hall, 1991.

[17] A.M. Pitts. Relational properties of recursively defined domains. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 86–97, 1993. An expanded version will appear in *Information and Computation.*

[18] G.D. Plotkin. Denotational semantics with partial functions. Lecture at C.S.L.I. Summer School, 1985.

[19] G.D. Plotkin. Type theory and recursion (extended abstract). In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, page 374, 1993.

[20] D.S. Scott. Identity and existence in intuitionistic logic. In M.P. Fourman, C.J. Mulvey, and D.S. Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics.* Springer-Verlag, 1979.

[21] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[22] P. Urzyczyn. Positive recursive type assignment. In *Mathematical Foundations of Computer Science*, volume 969 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

Appendix: review of core FPC

These are basic rules for FPC. Additional rules for recursive types appear in section 2.

Type contexts. A type context is a list of distinct type variables. We write $O \vdash \diamond$ to indicate that O is a well-formed type context. The rules for this judgment are:

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{O \vdash \diamond}{O, X \vdash \diamond}\ X \notin O$$

Types. As types, we have type variables, product types, function types, sum types, and recursive types. We write $O \vdash A$ to indicate that A is a well-formed type in the type context O:

$$\frac{O \vdash \diamond}{O \vdash X}\ X \in O$$

$$\frac{O \vdash A \qquad O \vdash B}{O \vdash A \star B}\ \star \in \{\times, \rightarrow, +\} \qquad \frac{O, X \vdash F}{O \vdash \mu X.F}$$

Term contexts. A term context is a list of distinct variables with types. We write $O \vdash E$ to indicate that E is a well-formed term context in the type context O:

$$\frac{O \vdash \diamond}{O \vdash \emptyset} \qquad \frac{O \vdash E \qquad O \vdash A}{O \vdash E, x{:}\, A}\ x \notin \mathrm{dom}(E)$$

Terms. As terms, we have variables, products, functions, and sums. We write $O; E \vdash e : A$ to indicate that term e has type A in the type context O and the term context E.

$$\frac{O \vdash E}{O; E \vdash x : A}\ (x{:}\, A) \in E$$

$$\frac{O; E \vdash s : A \qquad O; E \vdash t : B}{O; E \vdash (s, t) : A \times B}$$

$$\frac{O; E \vdash e : A \times B}{O; E \vdash \mathtt{fst}(e) : A}$$

$$\frac{O; E \vdash e : A \times B}{O; E \vdash \mathtt{snd}(e) : B}$$

$$\frac{O; E, x{:}\, A \vdash e : B}{O; E \vdash \lambda x{:}\, A.\, e : A \rightarrow B}$$

$$\frac{O; E \vdash s : A \rightarrow B \qquad O; E \vdash t : A}{O; E \vdash s(t) : B}$$

$$\frac{O; E \vdash e : A \qquad O \vdash B}{O; E \vdash \mathtt{inl}_{A,B}(e) : A + B}$$

$$\frac{O \vdash A \qquad O; E \vdash e : B}{O; E \vdash \mathtt{inr}_{A,B}(e) : A + B}$$

$$\frac{\begin{array}{c} O; E \vdash e : A + B \\ O; E, x{:}\, A \vdash s : C \\ O; E, y{:}\, B \vdash t : C \end{array}}{O; E \vdash \mathtt{case}\ e\ \mathtt{of}\ \mathtt{l}(x).s\ \mathtt{or}\ \mathtt{r}(y).t : C}$$