Inductively Defined Types in the Calculus of Constructions

Frank Pfenning

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213

Internet: fp@cs.cmu.edu

Christine Paulin-Mohring

INRIA and LIENS, URA CNRS 1327
Ecole Normale Superiéure
45 Rue d'Ulm
75005 Paris, France

Internet: mohring@ens.ens.fr

Abstract

We define the notion of an *inductively defined type* in the Calculus of Constructions and show how inductively defined types can be represented by closed types. We show that all primitive recursive functionals over these inductively defined types are also representable. This generalizes work by Böhm & Berarducci on synthesis of functions on term algebras in the second-order polymorphic λ-calculus (F_2). We give several applications of this generalization, including a representation of F_2-programs in F_3, along with a definition of functions `reify`, `reflect`, and `eval` for F_2 in F_3. We also show how to define induction over inductively defined types and sketch some results that show that the extension of the Calculus of Construction by induction principles does not alter the set of functions in its computational fragment, F_ω. This is because a proof by induction can be *realized* by primitive recursion, which is already definable in F_ω.

1 Introduction

The motivation for the this paper comes from two sources: work on the extraction of programs from proofs in the Calculus of Constructions (CoC) [23, 24] and work on the implementation of LEAP [25], an explicitly polymorphic ML-like programming language (here we only consider the pure F_ω fragment of LEAP). The former emphasizes the *logical* aspects of CoC, the latter its *computational* aspects. The basic relationship is simple: an extraction process relates proofs in CoC to programs in F_ω. In other words, in F_ω we can express the computational contents of proofs in CoC. Said yet another way: programs in F_ω realize propositions in CoC.[1]

[1]For the purposes of this paper, we are ignoring the distinction between *Data*, *Prop*, and *Spec* made in [23, 24]. For practical purposes, this distinction is extremely important. Here it is more

Both on the logical and computational level, inductively defined propositions or types play a central rôle in any applications. Their logical aspect, that is, proving properties by induction, and their computational aspect, that is, defining functions by primitive recursion, are very closely related: the computational content of a proof by induction is a function definition by primitive recursion. Said another way: primitive recursion realizes induction. One of our results is that, even though induction principles are not provable in CoC, their computational content is already definable in F_ω. Thus augmenting CoC by induction principles over inductively defined types is in some sense "conservative" over its computational fragment: even though we can prove more specifications, any function which we might be able to extract from such proofs is already definable in pure F_ω—we just would not be able to show in CoC without induction that it satisfies its specification.

Closely related is work by Girard [13, 14], Fortune, Leivant & O'Donnell [12], and Leivant [17, 18] who are concerned with the relationship between higher-order logic and polymorphic λ-calculi.

Mendler [19, 20] studied inductive types in the setting of the second-order polymorphic λ-calculus and the NuPrl type theory. He adds to the system F a new scheme for defining recursive types. The system is extended with new constants for representing the type, its constructor and the primitive recursion operator. The rules of conversion of the system are also extended for each new recursive type. In our presentation the inductive types are internally represented using higher-order quantification and the only reduction rule used is β-reduction. An advantage of our approach is that types that in some sense "are already there" are not also added artificially. On the other hand, a significant drawback of our approach is the relative weakness of our notion of equality induced by this representation, even if one adds η-conversion. For example, let R be the closed term for primitive recursion over the natural numbers, defined using iteration and pairing as in Section 5. Then the equality between $R\,\beta\,h'_z\,h'_s\,(\mathtt{succ}\,n)$ and $h'_s\,(\mathtt{pair}\,n\,(R\,\beta\,h'_z\,h'_s\,n))$ is not an internal equality (as it is in Mendler's system) but is only provable using induction on n. The types given for primitive recursion in Mendler's work and in this paper are slightly different but equivalent. Work along Mendler's lines for the Calculus of Constructions is presented by Coquand and Paulin-Mohring [9] and for Martin-Löf's type theory by Dybjer [11].

On the purely computational level, we generalize Böhm & Berarducci's [4] construction of functions on term algebras in the second-order polymorphic λ-calculus (F_2) to F_ω. One of their results does not generalize in unmodified form beyond algebraic types: not every closed term of the representation type will be $\beta\eta$-convertible to the representation of a term in the inductive type. This does not appear to be computationally relevant. One can consider alternative definitions of inductive types outside F_ω (but still inside CoC) which have the same computational content as our definitions. Another alternative would be to strengthen the notion of equality. We conjecture that one can use Reynolds' condition of *parametricity* [26] to recover uniqueness of representations at least in the F_ω fragment.

---

convenient to simply use $*$ to encompass all of them. We thus use the terms "proposition" and "specification" interchangeably.

A facility to generate the definition of inductively defined types, the constructors, and the primitive recursion operator from specifications like the ones in Examples 3 to 9 has been added to the implementation of the Calculus of Constructions V4.10 developed at INRIA. Work on the efficient implementation of inductively defined types and primitive recursion over such types in F_ω is currently under way in the framework of the Ergo project at Carnegie Mellon University.

2 The Calculus of Constructions

The Calculus of Constructions (CoC) of Coquand & Huet (see [7, 6, 16, 8]) is a very powerful type theory, yet it can be formulated very concisely. It encompasses Girard's system F_ω (see [13, 14]) and the type theory of LF, the Edinburgh Logical Framework (see Harper, Honsell & Plotkin [15]) and may be considered the result of combining these two type theories (see Barendregt [2]). The formulation we present here is a very brief summary of the concrete syntax, notation, and inference system given in [8].

We use $M, N, \ldots$ for terms in general and x, y, z for variables (abstractly, though, they are de Bruijn indices [10], where the occurrences of x in $(\lambda x{:}M)\,N$ and $[x{:}M]\,N$ are binding occurrences). We have

$$M ::= x \mid (\lambda x{:}M)\,N \mid (M\ N) \mid [x{:}M]\,N \mid *$$

Following [8] we call $[x{:}M]\,N$ a *product*. $*$ is the universe of all types, but is itself not a type. *Contexts* (denoted by Γ, Δ) are products over $*$ and thus have the form $[x_1{:}M_1]\ldots[x_n{:}M_n]\,*$, all other terms will be referred to as *objects*. Contexts serve as types, but do not have types themselves. When it is clear that a term is a context, we sometimes omit the trailing $*$.

The inference system defines two judgments: $\Gamma \vdash \Delta$ means that Δ is a valid context in the valid context Γ, and $\Gamma \vdash M : P$ means that M is a well-typed term of type P in the valid context Γ. We use $P, Q, \ldots$ for *types*, that is, terms which can appear in the place of P in the judgments below. The inference system below entails that a type P will either be a context, or have the property that $\Gamma \vdash P : *$. $[N/x]Q$ is the notation for substituting N for x in Q (abstractly defined using the de Bruijn notation, and therefore avoiding the issues of name clashes).

Valid Contexts.

$$\vdash *\qquad \frac{\Gamma \vdash \Delta}{\Gamma[x{:}\Delta] \vdash *}\qquad \frac{\Gamma \vdash P : *}{\Gamma[x{:}P] \vdash *}$$

Product Formation.

$$\frac{\Gamma[x{:}P] \vdash \Delta}{\Gamma \vdash [x{:}P]\Delta}\qquad \frac{\Gamma[x{:}P] \vdash N : *}{\Gamma \vdash [x{:}P]N : *}$$

Variables, Abstraction, and Application.

$$\frac{\Gamma \vdash *}{\Gamma \vdash x : P}\,[x{:}P] \text{ in } \Gamma \qquad \frac{\Gamma[x{:}P] \vdash N : Q}{\Gamma \vdash (\lambda x{:}P)\,N : [x{:}P]\,Q} \qquad \frac{\Gamma \vdash M : [x{:}P]\,Q \qquad \Gamma \vdash N : P}{\Gamma \vdash (M\ N) : [N/x]Q}$$

We will consider β-conversion ($\cong$) in the "full" form (see [8, Page 102]) and have the following rule of type conversion:

$$\frac{\Gamma \vdash M : P \qquad \Gamma \vdash P \cong Q}{\Gamma \vdash M : Q}$$

η-conversion does not play a very important role, but we will have occasion to use it when considering the representation of inductively defined types.

The calculus shares the basic properties of the LF type theory and F_ω, such as strong normalization, decidability of type-checking, and the Church-Rosser property for well-typed terms. We will make use of the properties in the development below. We formulate the basic induction principle over normal forms of types in CoC separately as a lemma, since we will need it frequently. Its proof is immediate from the Lemmas in [8].

Lemma 1 (Normal forms of types) *Given a type R, that is, a term R such that for some Γ and N we have $\Gamma \vdash N : R$. Then the β-normal form of R has the shape $N_0\, N_1 \ldots N_p$, $*$, or $[x{:}R_0]\, R_1$. In particular, the β-normal form of R cannot be an abstraction.*

We say that a type R is *atomic* if it is in normal form and does not begin with a product, that is, is not of the form $[x{:}P]\, Q$.

We will use $P \to Q$ as an abbreviation for any $[x{:}P]\, Q$, if x does not occur free in Q. We will sometimes omit the parentheses surrounding applications in which case application is written simply as juxtaposition and associates to the left. Juxtaposition binds tighter than "$\to$", which associates to the right. Abstraction and product also associate to the right and bind less tightly than "$\to$". The equality in the metalanguage is "$=$". Definitional equality is written as "$\equiv$" and may be thought of as introducing an abbreviation at the level of the Calculus of Construction as available in its implementation at INRIA. We will use this notion of notational definition in examples without formalizing it.

3 Inductively Defined Types

Intuitively, an inductively defined type is given by a complete list of constructors for terms of the type. We reason about the type with an appropriate induction principle, and we write functions over the type using iteration, which is powerful enough to define primitive recursive functionals over elements of the type. This notion encompasses the usual notions of free term algebras with associated induction principles, but it is more general and allows the definition of types such as natural numbers, pairs, lists, ordinal notations, logical quantifiers and connectives, or programs in F_2, a significant fragment of CoC of independent interest.

Below is our concrete syntax for the definition of an inductive type. We refer to α as the *inductively defined type*, and $c_1, \ldots, c_n$ as the *constructors for* α.

indtype $\alpha : [z_1{:}Q_1] \ldots [z_m{:}Q_m] *$ **with**
$c_1 : [x_1{:}P_{11}] \ldots [x_{k_1}{:}P_{1k_1}]\, \alpha M_{11} \ldots M_{1m}$
$\vdots$
$c_n : [x_1{:}P_{n1}] \ldots [x_{k_n}{:}P_{nk_n}]\, \alpha M_{n1} \ldots M_{nm}$
end

In such an inductive definition, α may not occur in Q_j, nor in any M_{ij}. However, α may occur in P_{il}, but only *positively* (see Definition 2). Throughout the paper, we will use the names α, c_i, Q_j, P_{il}, M_{ij} when we need to refer to the components of a given inductive type definition. Annotating a P_{il}^{α} serves only as a reminder that α may be free in P_{il}, and P_{il}^{β} is the result of substituting β for α in P_{il}. We will also use throughout this paper:

$$Q = [z_1{:}Q_1] \ldots [z_m{:}Q_m] *$$
$$P_i^{\alpha} = [x_1{:}P_{i1}^{\alpha}] \ldots [x_{k_i}{:}P_{ik_i}^{\alpha}]\, \alpha M_{i1} \ldots M_{im} \quad \text{for} \quad 1 \leq i \leq n$$

Besides positivity, we make an additional assumption that greatly simplifies the presentation and holds in all examples we are aware of, but is not essential. We require that for any quantifier $[y{:}R_0^{\alpha}]\, R_1^{\alpha}$ appearing in the definition of α, either y does not occur in R_1^{α} or α does not occur in R_0^{α}. For a development without this restriction see Paulin-Mohring [24]. The additional complexity arises primarily in the definition of Φ below (Definition 11)—all theorems remain valid when appropriately modified.

We define by simultaneous induction when a variable occurs only positively and only negatively in a type R, where R is in β-normal form. Since R is a type and assumed to be in normal form the (omitted) case $R = (\lambda z{:}R_0)\, R_1$ cannot arise (see Lemma 1).

Definition 2 (Positive and negative occurrences of variables) *We define by simultaneous induction: a variable x occurs only positively in the β-normal type R if*

Case *$R = x\, N_1 \ldots N_m$ and x does not occur in $N_1, \ldots N_m$,*

Case *R is atomic and x does not occur in R,*

Case *$R = [z{:}R_0]\, R_1$ and x occurs only negatively in R_0 and only positively in R_1.*

and a variable x occurs only negatively in the β-normal type R if

Case *R is atomic and x does not occur in R,*

Case *$R = [z{:}R_0]\, R_1$ and x occurs only positively in R_0 and only negatively in R_1.*

We begin with some examples for inductively defined types. The first one is algebraic (as in [4]).

Example 3 (Natural Numbers) *This is the canonical example for an inductively defined type.*

indtype nat : $*$ **with**
zero : nat
succ : nat $\rightarrow$ nat
end

Pairs and lists, the next two examples, are parameterized types which are hereditarily algebraic: once instantiated with algebraic types, the result will be algebraic. The representation of the parameterized type itself, however, is beyond the framework of [4].

Example 4 (Pairs) *Pairs are definable in this calculus. They will be used in Section 5 in order to define primitive recursion from iteration.*

indtype prod : $* \rightarrow * \rightarrow *$ **with**
pair : $[A{:}*]\,[B{:}*]\,A \rightarrow B \rightarrow \mathsf{prod}\,A\,B$
end

We will have occasion to use a generalized notion of pair in the metalanguage that applies to parameterized types. Given R and S of type $[z_1{:}Q_1]\ldots[z_m{:}Q_m]*$. We define $R \times S = [z_1{:}Q_1]\ldots[z_m{:}Q_m]\,\mathsf{prod}\,(R\,z_1\ldots z_m)\,(S\,z_1\ldots z_m)$.

Example 5 (Lists) *This is a simple example for a parameterized type that involves a non-trivial induction. As we will see later in Example 21 the representation of this parameterized type in our framework is somewhat different from the representation, for example, given by Reynolds [27].*

indtype list : $* \rightarrow *$ **with**
nil : $[A{:}*]\,\mathsf{list}\,A$
cons : $[A{:}*]\,A \rightarrow \mathsf{list}\,A \rightarrow \mathsf{list}\,A$
end

Ordinal notations, the next example, are not algebraic for a different reason: the argument to one of the constructors ranges over sequences (which are naturally represented as functions).

Example 6 (Ordinal Notations) *This example is due to Coquand [6] and generalized by Huet [16, Section 10.3.5]. The limit constructor* olim *is applied to a sequence of ordinals which is represented as a function from natural numbers to ordinals.*

indtype ord : $*$ **with**
ozero : ord
osucc : ord $\rightarrow$ ord
olim : $[A{:}*](A \rightarrow \mathsf{ord}) \rightarrow \mathsf{ord}$
end

The next example is a representation of programs in the polymorphic λ-calculus (F_2). This type is clearly not hereditarily algebraic.

Example 7 (Programs in F_2) *This inductive type is noteworthy for several reasons. Its representation will lie in F_3, the third-order polymorphic λ-calculus. Moreover, one can program an evaluation function for F_2 in F_3 over this representation. For a more detailed account, see [25].*

indtype $\mathsf{prog} : * \to * \;\mathbf{with}$
$\quad \mathtt{rep} : [A{:}*]\, A \to \mathsf{prog}\, A$
$\quad \mathtt{lam} : [A{:}*]\, [B{:}*]\, (A \to \mathsf{prog}\, B) \to \mathsf{prog}\, (A \to B)$
$\quad \mathtt{app} : [A{:}*]\, [B{:}*]\, \mathsf{prog}\, (A \to B) \to \mathsf{prog}\, A \to \mathsf{prog}\, B$
$\quad \mathtt{typlam} : [A{:}* \to *]\, ([B{:}*]\, \mathsf{prog}\, (A\, B)) \to \mathsf{prog}\, ([B{:}*]\, (A\, B))$
$\quad \mathtt{typapp} : [A{:}* \to *]\, \mathsf{prog}\, ([B{:}*](A\, B)) \to [B{:}*]\, \mathsf{prog}\, (A\, B)$
$\quad \mathbf{end}$

All the examples so far lie within the F_ω fragment of CoC. The following examples deal with aspects of dependent types in CoC which can be used to define logical notions.

Example 8 (Leibniz' Equality) *Leibniz' equality and other logical connectives can be defined as inductive types. We express here that equality is the least relation which relates every element to itself.*

indtype $\mathsf{eq} : [A{:}*]\, A \to A \to * \;\mathbf{with}$
$\quad \mathtt{refl} : [A{:}*]\, [x{:}A]\, \mathsf{eq}\, A\, x\, x$
$\quad \mathbf{end}$

Example 9 (Existential Quantification) *We express the usual inference rule for existential quantification and (since the type is inductive) that this is the only way we can establish an existentially quantified proposition.*

indtype $\mathsf{exists} : [A{:}*]\, (A \to *) \to * \;\mathbf{with}$
$\quad \mathtt{exists\text{-}intro} : [A{:}*]\, [P{:}A \to *]\, [x{:}A]\, P\, x \to \mathsf{exists}\, A\, P$
$\quad \mathbf{end}$

Similar to the way we generalized prod *to $\times$ we can generalize dependent pairs. This will be used in the definition of induction in Section 6. Given $R : [z_1{:}Q_1] \ldots [z_m{:}Q_m] *$ and $P : [z_1{:}Q_1] \ldots [z_m{:}Q_m]\, R\, z_1 \ldots z_m \to *$. We define the type*

$$R \otimes P = [z_1{:}Q_1] \ldots [z_m{:}Q_m]\, \mathsf{exists}\, (R\, z_1 \ldots z_m)\, (P\, z_1 \ldots z_m)$$

Counterexample 10 (LF encoding of logical systems) *LF, the Logical Framework, is a very weak subsystem of* CoC *in which one can encode inference systems as signatures. Judgments of the inference system become types or type families, logical connectives and quantifiers and inference rules become typed constants. See Harper, Honsell & Plotkin [15] for a description of LF and Avron, Honsell & Mason [1] for LF representations of a variety of logics. These signatures resemble inductive type definitions, but upon closer inspection the analogy fails. Consider the following two problematic declarations which would be part of an inductive type definition derived from an encoding of first-order arithmetic.*

indtype $\vdash : \mathsf{o} \to *$ **with**
$\quad \ldots$
$\quad \supset\mathrm{I} : [A{:}\mathsf{o}]\,[B{:}\mathsf{o}]\,(\vdash A \to \vdash B) \to \vdash A \supset B$
$\quad \forall\mathrm{I} : [A{:}\mathsf{nat} \to \mathsf{o}]\,([x{:}\mathsf{nat}] \vdash A\,x) \to \vdash \forall A$
end

In the case of $\supset$I*, the first occurrence of* $\vdash A$ *is* negative*, and therefore falls outside of our framework of inductive definitions. This is a simple example of a type that is non-empty, even though it may not have a "base case" when one tries to consider it as an inductively defined type, ignoring the negative occurrence of* $\vdash$*. In the case of* $\forall$I*, the rule may become too powerful and actually formalize a version of the* ω*-rule (and not universal introduction) when we make induction over natural numbers available at the level of LF. This failure of induction is not a defect of LF, since induction is done once and for all when the LF type theory itself is defined inductively. However, it does make it considerably more difficult to extend LF while preserving adequacy of representations of logical systems in LF.*

4 Representing Inductively Defined Types

There are two aspects of inductively defined types that we are interested in. The first one might be called the *computational aspect*, the second the *logical aspect*.

When investigating the computational aspect of an inductive type, we consider F_ω only and assume that we have a new (possibly parameterized) type constant α and new term constructors c_i. Functions over α may be defined using primitive recursion at higher type (see Definition 31). We ask if there is already a type in pure F_ω itself that can be used to represent terms built from the constructors such that the functions that are definable by primitive recursion are also definable. The answer here is "yes", though there will be a delicate point about the exact formulation of the theorem to that effect.

The logical aspect is based on the simple premise that one would like to reason inductively about inductive types. Since the various induction principles themselves are not provable in CoC, they have to be added as primitive constants. What are the properties of such an extension? We do not have a complete answer here, but at least we ascertain one pleasant property: when considering the computational content of proofs of specifications under this extension, it is *conservative*: we have new theorems (and proofs), but no new functions in F_ω.

We begin by giving a method for representing inductively defined types. An important property we would like to preserve is that an inductive type in F_ω will also be represented in F_ω. This fact is used vitally in the implementation of LEAP [25].

Now assume we are given an inductively defined type α in the notation at the beginning of Section 3. In this section we show that there is actually a closed type $\underline{\alpha}$ in CoC such that any well-typed term that can be built with the constructors of α and terms in CoC has a representation of type $\underline{\alpha}$. The converse, namely that every closed term M of type $\underline{\alpha}$ can be expressed in terms of the constructors of α is not true if one takes $\beta\eta$-conversion as the notion of term equality. We conjecture that the converse is

true in models that satisfy Reynolds' condition of *parametricity* [26]. This conjecture is based on the intuition that completeness fails because $\beta\eta$-equality is too weak to identify indistinguishable terms, under some reasonable assumptions about when terms should be indistinguishable (see Mitchell and Meyer [21]). Computationally this failure of completeness is not a problem, and the logical characterization of an inductive type in terms of an induction axiom is satisfactory from the logical point of view (though, of course, also incomplete in another sense).

Of course, there may be many ways an inductively defined type could be represented in CoC. We give here a canonical construction in which the representation of an element of the inductive type is its own iteration function. This representation has some drawbacks which we will return to in Section 5, where we show how to define primitive recursion at all types over an inductively defined type.

Before launching into the description of the representation of inductive types, we need an important technical tool. In its simplest form, we define a map Φ on terms that lifts a function $F : P \to Q$ to a function $\Phi_R : R\,P \to R\,Q$ where $R : * \to *$ and R is positive in its argument (that is, $R = (\lambda x{:}*)\,R'$ and x is only positive in R').

Definition 11 (Maps Φ and Ψ) *Given S and T of type $[z_1{:}Q_1]\ldots[z_m{:}Q_m]*$ and a function $F : [z_1{:}Q_1]\ldots[z_m{:}Q_m]\,S\,z_1\ldots z_m \to T\,z_1\ldots z_m$. Furthermore, we are given a type $R = R^x$ with some free occurrences of $x{:}[z_1{:}Q_1]\ldots[z_m{:}Q_m]*$. We define Φ_R for R^x with only positive occurrences of x such that for any term $N : R^S$, $\Phi_R(N) : R^T$, and simultaneously we define Ψ_R for R^x with only negative occurrences of x such that for any term $N : R^T$, $\Psi_R(N) : R^S$.*

Case *$R^x = x\,N_1\ldots N_m$. Then let $\Phi_R(N) = F\,N_1\ldots N_m\,N : R^T$, since x does not occur in $N_1,\ldots,N_m$ by positivity.*

Case *R^x is atomic and x does not occur in R^x. Then $R^S = R^T$ and we let $\Phi_R(N) = N$.*

Case *$R^x = [z{:}R_0^x]\,R_1^x$. Then $\Phi_R(N) = (\lambda z{:}R_0^T)\,\Phi_{R_1}(N\,\Psi_{R_0}(z))$. Note that x will occur only negatively in R_0^x since it occurs only positively in R^x.*

Remember that the case $R^x = (\lambda z{:}R_0^x)\,R_1^x$ cannot arise, since R^x is a type *in normal form (see Lemma 1). Now for R^x with x only occurring only negatively, we define:*

Case *$R^x = x\,N_1\ldots N_m$. This case cannot arise, since x is positive in R^x, but we assumed that x occurs only negatively in R^x.*

Case *R^x is atomic and x does not occur in R^x. Then $R^S = R^T$ and we let $\Psi_R(N) = N$.*

Case *$R^x = [z{:}R_0^x]\,R_1^x$. Then $\Psi_R(N) = (\lambda z{:}R_0^S)\,\Psi_{R_1}(N\,\Phi_{R_0}(z))$.*

The construction of Φ depends on F and its type. If we want to make the dependency explicit, we write Φ^F for the map Φ that is constructed from F.

The term constructed according to this definition will not always be correctly typed. We need an additional restriction that is satisfied in all of our examples and in particular is always satisfied for inductive type in the F_ω fragment of CoC.

Lemma 12 *In the context of Definition 11 and under the assumption that for any quantifier* $[z{:}R_0^x]\,R_1^x$ *in* R^x*, either* z *does not occur in* R_1^x *or* x *does not occur in* R_0^x*,* Φ *and* Ψ *are well-defined and* Φ *satisfies*

$$\Phi_R(N) : R^T \text{ for any } N : R^S$$

The proof is by a simple induction on the structure of R^x. The definition of Φ and Ψ with the same property can be made in full generality, but is quite complex. Details can be found in Paulin-Mohring [24, page 107].

Now we are prepared to state and prove the representation of inductive types.

Definition 13 (Representation $\underline{\alpha}$ of an inductively defined type α) *Given* α*, defined inductively as in Section 3. We will use the notation* P_{il}^α *for* P_{il} *and* P_{il}^β *for the result of substituting* β *for* α *in* P_{il} *and* P_i^β *for the result of substituting* β *for* α *in* P_i*. We let*

$$\underline{\alpha} = (\lambda z_1{:}Q_1)\ldots(\lambda z_m{:}Q_m)\,[\beta{:}Q]\,P_1^\beta \to \cdots \to P_n^\beta \to \beta\, z_1\, \ldots z_m$$

It is easy to see that $\underline{\alpha} : Q$. The definition of the representations of the constructors c_i will make use of the function $()^+$ defined below with the property that if $N : R^{\underline{\alpha}}$ then $N^+ : R^\beta$.

Definition 14 (Representation $\underline{c_i}$ of constructor c_i)

$$\underline{c_i} = (\lambda x_1{:}P_{i1}^{\underline{\alpha}})\ldots(\lambda x_{k_i}{:}P_{ik_i}^{\underline{\alpha}})\,(\lambda\beta{:}Q)\,(\lambda y_1{:}P_1^\beta)\ldots(\lambda y_n{:}P_n^\beta)\,y_i\,x_1^+\ldots x_{k_i}^+$$

Given the property of $()^+$ stated above, it is easy to verify that $\underline{c_i} : P_i^{\underline{\alpha}}$. We now define the map $()^+$ using Φ and its properties.

Definition 15 (Map $()^+$) *Given a context* $[\beta{:}Q]\,[y_1{:}P_1^\beta]\ldots[y_n{:}P_n^\beta]$ *where all occurrences of* β *in the* P_i *are positive. In order to be able to apply* Φ *such that it coerces* $N : R^{\underline{\alpha}}$ *to* $N^+ : R^\beta$ *, we have to define a function* $F : [z_1{:}Q_1]\ldots[z_m{:}Q_m]\,\underline{\alpha}\,z_1\ldots z_m \to \beta\, z_1\ldots z_m$*. But* $\underline{\alpha}\,z_1\ldots z_m = [\beta{:}Q]\,P_1^\beta \to \cdots \to P_n^\beta \to \beta\, z_1\ldots z_m$ *and so we let*

$$F = (\lambda z_1{:}Q_1)\ldots(\lambda z_m{:}Q_m)\,(\lambda g{:}[\beta{:}Q]\,P_1^\beta \to \cdots \to P_n^\beta \to \beta\, z_1\ldots z_m)\,g\,\beta\,y_1\ldots y_n$$

and define N^+ *as* $\Phi_{R^\alpha}^F(N)$*.*

Definition 16 (Γ_α) *Given a type* α *defined inductively as above. Then*

$$\Gamma_\alpha = [\alpha{:}[z_1{:}Q_1]\ldots[z_m{:}Q_m]*]\,[c_1{:}P_1^\alpha]\ldots[c_n{:}P_n^\alpha]*$$

We also extend $\underline{()}$ *homomorphically from* α *and constructors* c_i *to any term* N *that is well-formed in a context* Δ, Γ_α*. We sometimes refer to a term in the context* Γ_α *as a* constructor term*.*

For the adequacy theorem it is convenient to consider η-conversion in addition to β-conversion.

Theorem 17 (Adequacy) *For any inductively defined type α and closed terms $N_1, \ldots, N_m$ such that $\Gamma_\alpha \vdash \alpha\, N_1 \ldots N_m : *$, $\underline{()}$ is a bijection between $\beta\eta$-equivalence classes of terms N such that $\Gamma_\alpha \vdash N : \alpha\, N_1 \ldots N_m$ and equivalence classes of terms M such that $\vdash M : \underline{\alpha}\, N_1 \ldots N_m$.*

Proof sketch: It is easy to verify by calculation as in [4] using Lemma 12 that $\underline{()}$ has the injection properties. The inverse map $\mathcal{F}(M) = M\, \alpha\, c_1 \ldots c_n$ applies the representation M of a term in an inductive type to the constructors of that type to yield the term that it represents. □

It is important to note that the inverse map $\mathcal{F}$ does not need to examine the structure of its argument M to determine what constructor term M represents. This means that even in an implementation where the intensional structure of functions is inaccessible (for example, when functions are compiled into machine code) we can still extract the constructor term that is represented by a function by applying it to the constructor constants.

The adequacy theorem is somewhat weaker than Böhm and Berarducci's representation theorem. This is because the mappings $\underline{()}$ and $\mathcal{F}$ do not go between $\beta\eta$-equivalence classes: as the following counterexample shows, non-convertible terms may represent the same constructor term.

Counterexample 18 (Non-uniqueness of representation under $\beta\eta$) *Consider the following inductively defined type with one constructor, where* nat *is defined as in Example 19:*

indtype cex : $*$ **with**
  c : (nat $\to$ nat) $\to$ cex
  end

This type would be represented as

$$\mathsf{cex} \equiv [p{:}*]\,((\mathsf{nat} \to \mathsf{nat}) \to p) \to p$$
$$\mathsf{c} \equiv (\lambda f{:}\mathsf{nat} \to \mathsf{nat})\,(\lambda p{:}*)\,(\lambda y{:}(\mathsf{nat} \to \mathsf{nat}) \to p)\; y\; f$$

The following term is not $\beta\eta$-equivalent to a term $\mathsf{c}\, f$ for any f, even though it has type cex*:*

$$\begin{aligned} M \;=\; & (\lambda p{:}*)\,(\lambda y{:}(\mathsf{nat} \to \mathsf{nat}) \to p) \\ & \quad y\,((\lambda n{:}\mathsf{nat})\, n\,(p \to \mathsf{nat})\,((\lambda x{:}p)\,\mathtt{zero})\,((\lambda x{:}p \to \mathsf{nat})\, x)\,(y\,((\lambda n{:}\mathsf{nat})\, n))) \end{aligned}$$

Using the inverse mapping $\mathcal{F}$ one can calculate what constructor term is represented by M:

$$\mathcal{F}(M) = \mathsf{c}\,((\lambda n{:}\mathsf{nat})\, n\,(\mathsf{cex} \to \mathsf{nat})\,((\lambda x{:}\mathsf{cex})\,\mathtt{zero})\,((\lambda x{:}\mathsf{cex} \to \mathsf{nat})\, x)\,(\mathsf{c}\,((\lambda n{:}\mathsf{nat})\, n)))$$

One can easily see that $\underline{\mathcal{F}(M)}$ and M are not $\beta\eta$-convertible, though they both represent $\mathcal{F}(M)$.

One can recover uniqueness by using dependency: in essence, a term of a constructor type is represented as the proof that it is well-formed. Such a more complex proof term has the same computational contents as our representation (see [24] or [18]). One can also formulate a simple criterion on the types P_i of the constructors that ensures uniqueness of the representation under $\beta\eta$-conversion (see [24, page 125]). Finally, one could claim that the failure of uniqueness is due to incompleteness of $\beta\eta$-conversion in the polymorphic λ-calculus and that they really should be equivalent. We conjecture that Reynolds' condition of *parametricity* [26] can be used to justify this claim, but under parametricity even more terms might be identified than under our notion of equivalence that is induced by the function $\mathcal{F}$. For example, under parametricity, the term M in the counterexample would also be equivalent to $\mathtt{c}\,((\lambda n{:}\mathtt{nat})\,\mathtt{zero})$.

Example 19 (Natural Numbers) *Here we obtain the well-known representation of the natural numbers in the second-order polymorphic λ-calculus.*

$$\mathtt{nat} \equiv [C{:}*]\, C \to (C \to C) \to C$$

Example 20 (Pairs) *Using* $\underline{()}$ *we obtain:*

$$\begin{array}{rcl} \mathtt{prod} & \equiv & (\lambda A{:}*)\,(\lambda B{:}*)\,[C{:}* \to * \to *]\,([A{:}*]\,[B{:}*]\,A \to B \to C\,A\,B) \to C\,A\,B \\ \mathtt{pair} & \equiv & (\lambda A{:}*)\,(\lambda B{:}*)\,(\lambda C{:}* \to * \to *)\,(\lambda f{:}[A{:}*]\,[B{:}*]\,A \to B \to C\,A\,B)\,f\,A\,B\,x\,y \end{array}$$

This is not the encoding given, for example, by Reynolds [27] and is slightly more awkward. The standard definition can be recovered by parameterizing the whole inductive definition by A and B and then abstracting over A and B to obtain global definitions (we refer to this method as uniform parameterization*). Uniform parameterization often leads to simpler equivalent representation of inductively defined parameterized types. Here, we define in the context $A{:}*, B{:}*$ (the superscripts serve only as a reminder of the dependency):*

indtype $\mathtt{prod}^{A,B} : *$ **with**
$\quad \mathtt{pair}^{A,B} : A \to B \to \mathtt{prod}^{A,B}$
$\quad$ **end**

This yields the representation

$$\begin{array}{rcl} \mathtt{prod}^{A,B} & \equiv & [C{:}*]\,(A \to B \to C) \to C \\ \mathtt{pair}^{A,B} & \equiv & (\lambda x{:}A)\,(\lambda y{:}B)\,(\lambda C{:}*)\,(\lambda f{:}A \to B \to C)\,f\,x\,y \end{array}$$

One can then abstract over A and B (discharge them from the context) to obtain the usual, now global definitions of $\mathtt{prod}$ *and* $\mathtt{pair}$*:*

$$\begin{array}{rcl} \mathtt{prod} & \equiv & (\lambda A{:}*)\,(\lambda B{:}*)\,[C{:}*]\,(A \to B \to C) \to C \\ \mathtt{pair} & \equiv & (\lambda A{:}*)\,(\lambda B{:}*)\,(\lambda x{:}A)\,(\lambda y{:}B)\,(\lambda C{:}*)\,(\lambda f{:}A \to B \to C)\,f\,x\,y \end{array}$$

Example 21 (Lists) *The representation of lists obtained this way is also different from, though equivalent to the encoding in F_2 given in [27].*

$$\mathsf{list} \equiv (\lambda B{:}*)\,[C{:}* \to *]\,([A{:}*]\,C\,A) \to ([A{:}*]\,A \to C\,A \to C\,A) \to C\,B$$

As in Example 20, one can obtain the usual definition by uniform parameterization.

Example 22 (Ordinal Notations)

$$\mathsf{ord} \equiv [C{:}*]\,C \to (C \to C) \to ((\mathsf{nat} \to C) \to C) \to C$$

Example 23 (Programs in F_2) *This is an example where uniform parameterization is not possible, since* `prog` *is applied to different arguments at different occurrences in the types of the constructors in Example 7. Thus a representation of this F_2-type will lie in F_3. We conjecture that no F_2 representation is possible such that the normalization function over the representation is definable.*

$$\begin{array}{lll}
\mathtt{prog} \equiv & (\lambda D{:}*)\,[C{:}* \to *] & \\
& ([A{:}*]\,A \to C\,A) & \textit{from } \mathtt{rep} \\
& \to ([A{:}*][B{:}*]\,(A \to C\,B) \to C\,(A \to B)) & \textit{from } \mathtt{lam} \\
& \to ([A{:}*][B{:}*]\,C\,(A \to B) \to C\,A \to C\,B) & \textit{from } \mathtt{app} \\
& \to ([A{:}* \to *]\,([B{:}*]C\,(A\,B)) \to C\,([B{:}*](A\,B))) & \textit{from } \mathtt{typlam} \\
& \to ([A{:}* \to *]\,C\,([B{:}*](A\,B)) \to [B{:}*]C\,(A\,B)) & \textit{from } \mathtt{typapp} \\
& \to C\,D &
\end{array}$$

Example 24 (Leibniz' Equality) *In order to show that Example 8 actually defines Leibniz' equality, we use uniform parameterization (see Example 20) to modify the previous definition. Assume we are in the context $A{:}*, x{:}A$. We would like to define the type of elements equal to x inductively. We define*

indtype $\mathtt{eq}^{A,x} : A \to *$ **with**
  $\mathtt{refl}^{A,x} : \mathtt{eq}^{A,x}\,x$
  end

Our representation function yields

$$\begin{array}{rcl}
\mathtt{eq}^{A,x} & \equiv & (\lambda y{:}A)\,[C{:}A \to *]\,(C\,x \to C\,y) \\
\mathtt{refl}^{A,x} & \equiv & (\lambda C{:}A \to *)\,(\lambda z{:}C\,x)\,z
\end{array}$$

After abstracting over A and x we obtain the usual definition of Leibniz' equality in the setting of CoC *or higher-order logic.*

Example 25 (Existential Quantification) *Here, too, we apply uniform parameterization in order to expose the similarity to the usual definition of existential quantification in* CoC *or higher-order logic. In the context $A{:}*, P{:}A \to *$ we define*

indtype $\mathsf{exists}^{A,P} : *$ **with**
  $\mathtt{exists\text{-}intro}^{A,P} : [x{:}A]\,(P\,x \to \mathsf{exists}^{A,P})$
  end

Our representation function yields

$$\begin{array}{rcl}
\mathsf{exists}^{A,P} & \equiv & [C{:}*]\,([x{:}A]\,(P\,x \to C)) \to C \\
\mathtt{exists\text{-}intro}^{A,P} & \equiv & (\lambda x{:}A)\,(\lambda v{:}P\,x)\,(\lambda C{:}*)\,(\lambda w{:}[x{:}A]\,(P\,x \to C))\,w\,x\,v
\end{array}$$

After discharging A and P from the context, we obtain the usual definitions.

5 Computing with Inductively Defined Types

Enriching CoC by inductively defined types must go along with some method for defining recursive functions over these types. We choose iteration rather than primitive recursion since it is a simpler notion and primitive recursion is definable from iteration. For an implementation of a programming language based on an enriched F_ω one would probably need to choose primitive recursion, since its implementation through iteration is provably inefficient in some cases (see Colson [5] or Parigot [22]).

Definition 26 (Definition by iteration) *Let an α be an inductively defined data type as in Section 3. Given a $\beta : Q$ and functions $h_1{:}P_1^\beta, \ldots, h_n{:}P_n^\beta$. Then the function*

$$f : [z_1{:}Q_1] \ldots [z_m{:}Q_m]\, \alpha\, z_1 \ldots z_m \to \beta\, z_1 \ldots z_m$$

is defined by iteration over α at type β from $h_1, \ldots, h_n$ *if it satisfies the following equations:*

$$\begin{aligned} &f\, M_{11} \ldots M_{1m}\, (c_1\, x_1 \ldots x_{k_1}) = h_1\, \overline{x}_1 \ldots \overline{x}_{k_1} \\ &\vdots \\ &f\, M_{n1} \ldots M_{nm}\, (c_n\, x_1 \ldots x_{k_n}) = h_n\, \overline{x}_1 \ldots \overline{x}_{k_n} \end{aligned}$$

where $\overline{N}$ is defined below.

The idea in the definition of $\overline{N}$ is to replace occurrences of variables whose type has the form $\alpha\, N_1 \ldots N_m$ by recursive calls to f. The map Φ is already of the right form to define $\overline{()}$.

Definition 27 (Map $\overline{()}$) *For $f : [z_1{:}Q_1] \ldots [z_m{:}Q_m]\, \alpha\, z_1 \ldots z_m \to \beta\, z_1 \ldots z_m$ and $N : R^\alpha$ we define $\overline{N}$ such that $\overline{N} : R^\beta$ by $\overline{N} = \Phi_R^f(N)$.*

Given the basic representation $\underline{()}$, how can we define iteration on the representation? A basic insight is that a constructor is implemented as an iterator, thus applying the representation of a constructor term as a function will perform iteration.

Theorem 28 *Given the type β and $h_1, \ldots, h_n$, then*

$$\underline{f} \equiv (\lambda z_1{:}Q_1) \ldots (\lambda z_m{:}Q_m)\, (\lambda x{:}\underline{\alpha}\, z_1 \ldots z_m)\, x\, \beta\, h_1 \ldots h_n$$

is defined from $h_1, \ldots, h_n$ by iteration over type α at type β. Thus we have

$$\underline{f} M_{i1} \ldots M_{im}\, (\underline{c}_i\, x_1 \ldots x_{k_i}) \cong h_i\, \overline{\underline{x}}_1 \ldots \overline{\underline{x}}_{k_i}$$

where $\overline{\underline{x}}_l$ is like $\overline{x_l}$ except that it inserts recursive calls to $\underline{f}$ rather than to f, that is, $\overline{\underline{x}}_l = \Phi_{P_{il}}^{\underline{f}}(x_l)$.

Proof sketch: By simple inductions as in [4]. □

Note that we claim convertibility only for terms in the image of the $\underline{()}$ translation function, not for any term that represents $c_i\, x_1 \ldots x_{k_i}$. We conjecture that under the assumption of parametricity (for the F_ω fragment) a stronger theorem also holds: the equivalence classes of representations from Theorem 17 satisfy the equations for iteration, given the definition of $\underline{f}$ above.

Example 29 (Existential Quantification) *For pairs or dependent pairs, the schema of iteration simply allows access to the components of the pair. We show only the dependent case.*

$$f\, A\, P\, (\texttt{exists-intro}\, A\, P\, x\, w) = h_1\, A\, P\, x\, w$$

with types $f : [A{:}*]\,[P{:}A \to *]\,\textsf{exists}\, A\, P \to \beta\, A\, P$ *and* $h_1 : [A{:}*]\,[P{:}A \to *]\,[x{:}A]\,[w{:}P\, x]\,\beta\, A\, P$. *The first projection function* `fst` *for the usual pairs is easily definable, as is the function* `dfst` *for extracting the first component of a dependent pair shown here. In terms of the notation above we have*

$$\begin{aligned}
&\texttt{dfst}\, A\, P\, (\texttt{exists-intro}\, A\, P\, x\, w) = x\\
&\beta = (\lambda A{:}*)\,(\lambda P{:}A \to *)\, A\\
&h_1 = (\lambda A{:}*)\,(\lambda P{:}A \to *)\,(\lambda x{:}A)\,(\lambda w{:}P\, x)\, x
\end{aligned}$$

Example 30 (Programs in F_2) *We now give definition of* `reify`, `reflect` *and* `eval` *in the form of an iteration. These definitions are in the* F_3 *fragment of* CoC. *The crucial function is* $\texttt{reflect} : [A{:}*]\,\textsf{prog}\, A \to A$. *In terms of the above definition,* $\beta = (\lambda A{:}*)\, A$

$$\begin{array}{lcl}
\texttt{reflect}\, A\, (\texttt{rep}\, A\, x) & = & x\\
\texttt{reflect}\, (A \to B)\, (\texttt{lam}\, A\, B\, x) & = & (\lambda y{:}A)\, \texttt{reflect}\, B\, (x\, y)\\
\texttt{reflect}\, B\, (\texttt{app}\, A\, B\, x\, y) & = & (\texttt{reflect}\, (A \to B)\, x)\, (\texttt{reflect}\, A\, y)\\
\texttt{reflect}\, ([B{:}*](A\, B))\, (\texttt{typlam}\, A\, x) & = & (\lambda B{:}*)\, \texttt{reflect}\, (A\, B)\, (x\, B)\\
\texttt{reflect}\, (A\, B)\, (\texttt{typapp}\, A\, x\, B) & = & \texttt{reflect}\, ([B{:}*]\, A\, B)\, x\, B
\end{array}$$

From this the other definitions follow easily:

$$\begin{array}{lcl}
\texttt{reify} & : & [A{:}*]\, A \to \textsf{prog}\, A\\
\texttt{reify} & \equiv & \texttt{rep}\\
\texttt{eval} & : & [A{:}*]\, \textsf{prog}\, A \to \textsf{prog}\, A\\
\texttt{eval} & \equiv & (\lambda A{:}*)\, (\lambda x{:}\textsf{prog}\, A)\, \texttt{reify}\, A\, (\texttt{reflect}\, A\, x)
\end{array}$$

In [25] we give the expanded definition of `reflect` *in* F_3 *using Theorem 28.*

Primitive recursion at all types is somewhat more difficult, but as shown in various places for the second-order polymorphic λ-calculus (see, for example, Reynolds [27]) it can be reduced to iteration. We briefly state only the form of primitive recursion and the type of the primitive recursive operator pr_α over an inductively defined type α. $\times$ is the generalized product from Definition 4.

Definition 31 (Definition by primitive recursion at arbitrary type) *Let an α be an inductively defined data type as in Section 3. Given a $\beta : Q$ and functions $h'_1, \ldots, h'_n$ where $h'_i : [x'_1{:}P_{i1}^{\alpha\times\beta}] \ldots [x'_{k_i}{:}P_{ik_i}^{\alpha\times\beta}]\, \beta\, M_{i1} \ldots M_{im}$. A function $f : [z_1{:}Q_1] \ldots [z_m{:}Q_m]\ \alpha\, z_1 \ldots z_m \to \beta\, z_1 \ldots z_m$ is* defined by primitive recursion over α at type β from $h'_1, \ldots, h'_n$ *if it satisfies the following equations:*

$$\begin{aligned} &f\, M_{11} \ldots M_{1m}\, (c_1\, x_1 \ldots x_{k_1}) = h'_1\, \hat{x}_1 \ldots \hat{x}_{k_1} \\ &\vdots \\ &f\, M_{n1} \ldots M_{nm}\, (c_n\, x_1 \ldots x_{k_n}) = h'_n\, \hat{x}_1 \ldots \hat{x}_{k_n} \end{aligned}$$

where $\hat{x}_l = \Phi^F_{P_{il}}(x_l)$ for $F = (\lambda z_1{:}Q_1) \ldots (\lambda z_m{:}Q_m)\, (\lambda x{:}\alpha\, z_1 \ldots z_m)\, \mathtt{pair}\, x\, (f\, z_1 \ldots z_m\, x)$ and thus $\hat{(\,)} : R^\alpha \to R^{\alpha\times\beta}$.

Note that the occurrences of M_{ij} are not binding occurrences: they are determined by the type of the constructor c_i. In the simplest case, $\hat{x}$ is merely x (if the type of x does not involve α), or the pair of x and fx (if the type of x is α). In general, the variable pr_α which generates the definition of f given β and functions $h'_1, \ldots, h'_n$ has type

$$\begin{array}{lll} \mathrm{pr}_\alpha \;:\; [\beta{:}[z_1{:}Q_1] \ldots [z_m{:}Q_m]\, *] & & \text{for } \beta \\ \quad ([x'_1{:}P_{11}^{\alpha\times\beta}] \ldots [x'_{k_1}{:}P_{1k_1}^{\alpha\times\beta}]\, \beta\, M_{11} \ldots M_{1m}) & & \text{for } h'_1 \\ \quad \vdots & & \quad \vdots \\ \quad \to ([x'_1{:}P_{n1}^{\alpha\times\beta}] \ldots [x'_{k_n}{:}P_{nk_n}^{\alpha\times\beta}]\, \beta\, M_{n1} \ldots M_{nm}) & & \text{for } h'_n \\ \quad \to [z_1{:}Q_1] \ldots [z_m{:}Q_m]\, \alpha\, z_1 \ldots z_m \to \beta\, z_1 \ldots z_m & & \end{array}$$

Example 32 (Primitive Recursion over Lists) *To illustrate the schema of primitive recursion we use lists as defined in Example 5. Given $\beta : * \to *$, primitive recursion can define a function $f : [A{:}*]\, \mathsf{list}\, A \to \beta\, A$. The schema looks like*

$$\begin{aligned} f\, A\, (\mathtt{nil}\, A) &= h'_1\, A \\ f\, A\, (\mathtt{cons}\, A\, x\, l) &= h'_2\, A\, x\, (\mathtt{pair}\, l\, (f\, A\, l)) \end{aligned}$$

where $h'_1 : [A{:}*]\, \beta\, A$ and $h'_2 : [A{:}*]\, [x{:}A]\, (\mathsf{prod}\, (\mathsf{list}\, A)\, (\beta\, A)) \to \beta\, A$.

As a concrete example consider the function `tl` *which takes a list and a default value and returns the tail of the list or the default value (if the list is empty). We could program this as a primitive recursion with*

$$\begin{aligned} &\mathtt{tl} : [A{:}*]\, \mathsf{list}\, A \to \mathsf{list}\, A \to \mathsf{list}\, A \\ &\mathtt{tl}\, A\, (\mathtt{nil}\, A) = (\lambda d{:}\mathsf{list}\, A)\, d \\ &\mathtt{tl}\, A\, (\mathtt{cons}\, A\, x\, l) = (\lambda d{:}\mathsf{list}\, A)\, l \end{aligned}$$

In the notation above we would have

$$\begin{aligned} &\beta = (\lambda A{:}*)\, \mathsf{list}\, A \to \mathsf{list}\, A \\ &h'_1 = (\lambda A{:}*)\, (\lambda d{:}A)\, d \\ &h'_2 = (\lambda A{:}*)\, (\lambda x{:}A)\, (\lambda p{:}\mathsf{prod}\, (\mathsf{list}\, A)\, (\beta\, A))\, (\lambda d{:}\mathsf{list}\, A)\, \mathtt{fst}\, (\mathsf{list}\, A)\, (\beta\, A) p \end{aligned}$$

6 Reasoning with Induction

One of the motivations behind inductively defined types is that we would like to reason about elements of these types using induction. In particular, we would like to extract provably correct functions from proofs. In this section we state the natural notion of induction over an inductively defined type, and show how induction relates to the notion of primitive recursive functionals.

Induction principles are not definable (that is, provable) in CoC itself, but one could *assume* such induction principles and associated reduction rules (see [8, Section 8] or [24, Section 4.4]). Such an extension of the calculus is in some sense "benign." This can be formalized as saying the computational content of a proof that used induction is already present in pure F_ω. The proof of this fact is surprisingly simple (see Theorem 35). Thus, if one is interested only in the computational content of proofs, the extension of CoC by induction over inductively defined types does not change the set of definable functions. However, with the addition of induction one will in general be able to prove many more specifications. Other conservative extension results for polymorphic λ-calculi have been obtained by Breazu-Tannen & Gallier [3].

Definition 33 (Induction principle ind_α for inductively defined α) *Let α be an inductively defined type as before. We define* ind_α*, the* induction principle over α *by*

$$\begin{aligned}\mathrm{ind}_\alpha \;:\; & [A{:}[z_1{:}Q_1]\ldots[z_m{:}Q_m]\,\alpha\,z_1\ldots z_m \to *] \\ & ([x'_1{:}P_{11}^{\alpha\otimes A}]\ldots[x'_{k_1}{:}P_{1k_1}^{\alpha\otimes A}]\,A\,M_{11}\ldots M_{1m}\,(c_1\,\check{x}'_1\ldots\check{x}'_{k_1})) \\ & \quad\vdots \\ & \to ([x'_1{:}P_{n1}^{\alpha\otimes A}]\ldots[x'_{k_n}{:}P_{nk_n}^{\alpha\otimes A}]\,A\,M_{n1}\ldots M_{nm}\,(c_n\,\check{x}'_1\ldots\check{x}'_{k_n})) \\ & \to [z_1{:}Q_1]\ldots[z_m{:}Q_m]\,[x{:}\alpha\,z_1\ldots z_m]\,A\,z_1\ldots z_m\,x\end{aligned}$$

where $\check{x}'$ is defined below and $\alpha\otimes A$ is the type of generalized dependent pairs (see Definition 9).

In the simplest case $\check{x}'$ will simply turn out to be x' (if the type of x' does not involve α) or $\mathtt{dfst}\,\alpha\,A\,x'$, extracting the element x from the pair consisting of an x and the proof that x satisfies property A (if x' has type α).

Definition 34 (Map $\check{x}$) *Let F be the generalized first projection function (derived easily from* `dfst`*, see Example 29) on elements of dependent pair type $\alpha\otimes A$. Then*

$$F : [z_1{:}Q_1]\ldots[z_m{:}Q_m]\;\mathsf{exists}\,(\alpha\,z_1\ldots z_m)\,(A\,z_1\ldots z_m)\to\alpha\,z_1\ldots z_m$$

and for R^x and $N : R^{\alpha\otimes A}$ we define $\check{N} = \Phi_R^F(N) : R^\alpha$.

Coquand & Huet define ν, the *stripping map*, which extracts an untyped λ-term as the computational content of a proof in CoC. We use a less drastic erasure in the proof of our conservative extension result below, which maps terms in CoC into terms in F_ω. The partial erasure map $\mathcal{E}$ is defined in detail in [23, 24].

Theorem 35 (Primitive recursion realizes induction) *We use* pind_α *and* ppr_α *as abbreviation for the types of* ind_α *and* pr_α*, respectively. Then* $\mathcal{E}(\mathrm{pind}_\alpha)\cong\mathcal{E}(\mathrm{ppr}_\alpha)$.

Proof sketch: The map $\mathcal{E}$ will erase $\alpha\, z_1 \ldots z_m$ from the type of A, and all corresponding arguments to A at all occurrences of A (notation as in definition 33). The resulting term is a valid type and $\beta\eta$-equivalent to the type of pr_α (see Definition 31). The crucial observation is that $\mathcal{E}(P_{il}^{\alpha\otimes A}) = \mathcal{E}(P_{il}^{\alpha\times\mathcal{E}(A)})$. □

This theorem means that the set of functions that can be extracted from induction proofs over α can already be defined explicitly by primitive recursion at arbitrary types. This corollary generalizes one direction of results obtained by Girard [14], and Fortune, Leivant & O'Donnell [12], and Leivant [17, 18] which may be summarized as "The number-theoretic functions representable in F_n are exactly the functions provably recursive in n^{th}-order arithmetic."

Example 36 (Induction over Lists) *Here we obtain a principle of induction over the construction of lists. Since induction is a logical statement, it best to think of* [] *as universal quantification.*

$$\begin{aligned}\mathrm{ind}_{\mathsf{list}} \;:\; & [P{:}[A{:}*]\,\mathsf{list}\,A \to *] \\ & \quad ([A{:}*]\,P\,A\,(\mathtt{nil}\,A)) \\ & \quad \to ([A{:}*]\,[x{:}A]\,[l'{:}\mathsf{exists}\,(\mathsf{list}\,A)\,(P\,A)]\,P\,A\,(\mathtt{cons}\,A\,x\,(\mathtt{dfst}\,A\,P\,l')) \\ & \quad \to [A{:}*]\,[l{:}\mathsf{list}\,A]\,P\,A\,l\end{aligned}$$

The induction principle will look more familiar after we curry at the argument l' to eliminate the dependent pair and also apply uniform parameterization over the argument A. We then get:

$$\begin{aligned}& [P{:}\mathsf{list}\,A \to *] \\ & \quad (P\,(\mathtt{nil}\,A)) \\ & \quad \to [x{:}A]\,[l{:}\mathsf{list}\,A]\,P\,A\,l \to P\,A\,(\mathtt{cons}\,A\,x\,l) \\ & \quad \to [l{:}\mathsf{list}\,A]\,P\,l\end{aligned}$$

Acknowledgments

We would like to thank Thierry Coquand, Jean Gallier, Bob Harper, Peter Lee, and Dan Leivant for helpful discussions. The first author was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

References

[1] Arnon Avron, Furio A. Honsell, and Ian A. Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1987.

[2] Henk Barendregt. The forest of lambda calculi with types. Talk given at the Workshop on Semantics of Lambda Calculus and Category Theory, Carnegie Mellon University, April 1988.

[3] Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic strong normalization and confluence. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming, Stresa, Italy*, pages 137–150. Springer-Verlag LNCS 372, July 1989.

[4] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] Loïc Colson. About primitive recursive algorithms. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming, Stresa, Italy*, pages 194–206. Springer-Verlag LNCS 372, July 1989.

[6] Thierry Coquand. *Une Théorie des Constructions*. PhD thesis, University Paris VII, January 1985.

[7] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL85*. Springer-Verlag LNCS 203, 1985.

[8] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[9] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. Talk presented at the *Workshop on Programming Logic*, University of Göteborg and Chalmers University of Technology, May 1989.

[10] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[11] Peter Dybjer. An inversion principle for Martin-Löf's type theory. Talk presented at the *Workshop on Programming Logic*, University of Göteborg and Chalmers University of Technology, May 1989.

[12] Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30:151–185, 1983.

[13] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, Amsterdam, London, 1971. North-Holland Publishing Co.

[14] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE, June 1987.

[16] Gérard Huet. Formal structures for computation and deduction. Lecture notes for a graduate course at Carnegie Mellon University, May 1986.

[17] Daniel Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *Proceedings of the Twenty Fourth Annual Symposium on the Foundations of Computer Science*, pages 160–169. IEEE, 1983.

[18] Daniel Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990. To appear.

[19] N. P. Mendler. First- and second-order lambda calculi with recursive types. Technical Report TR 86–764, Department of Computer Science, Cornell University, Ithaca, New York, July 1986.

[20] Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, September 1987.

[21] John C. Mitchell and Albert Meyer. Second-order logical relations. In Rohit Parikh, editor, *Logics of Programs*, pages 225–236. Springer-Verlag LNCS 193, June 1985.

[22] Michel Parigot. On the representation of data in lambda-calculus. Draft, 1988.

[23] Christine Paulin-Mohring. Extracting F_ω programs from proofs in the calculus of constructions. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 89–104. ACM Press, January 1989.

[24] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris VII, January 1989.

[25] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic lambda-calculus. *Theoretical Computer Science*, 1990. To appear. A preliminary version appeared in *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359, Springer-Verlag LNCS 352, March 1989.

[26] John Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier Science Publishers B. V., 1983.

[27] John Reynolds. Three approaches to type structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development*, pages 97–138. Springer-Verlag LNCS 185, March 1985.