# The Curry-Howard Isomorphism: Remarks on Recursive Types

*Lecture Notes, Edinburgh, April 13, 1999* [1]

Paweł Urzyczyn[2]
Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warszawa, Poland
`urzy@mimuw.edu.pl`

## 1 Iterators and recursors

A routine example of a recursively defined type is a list. A list of integers is either a "`nil`" or has the form $a :: \ell$, where $a$ is an integer and $\ell$ is a list. Thus the type **list** of integer lists can be seen as a union of the sequence $\{\}$, $\mathcal{F}(\{\})$, $\mathcal{F}^2(\{\})$, $\mathcal{F}^3(\{\}), \ldots$, where $\mathcal{F}(\alpha)$ stands for the (type) set $\{a :: \ell \mid a \in \mathbf{int} \text{ and } \ell \in \alpha\}$. Clearly, this union is the least fixpoint of the operator $\mathcal{F}$, that is we should have $\mathbf{list} = \mathcal{F}(\mathbf{list})$.

The above fundamental property of type **list** can be expressed in the form of a (second order) induction axiom

$$\forall R(R(\mathtt{nil}) \to \forall x(R(x) \to \forall a^{\mathbf{int}} R(a :: x)) \to \forall x^{\mathbf{list}} R(x))$$

or a first order induction scheme

$$R(\mathtt{nil}) \to \forall x(R(x) \to \forall a^{\mathbf{int}} R(a :: x)) \to \forall x^{\mathbf{list}} R(x).$$

Of course the best known induction pattern of a similar form is that for the type of integers (constructed by zero and successor).

Proofs by induction applied to particular elements should be reducible to induction-free proofs. For example, a proof of $\varphi(\underline{3})$ can be obtained from $A : \forall x(\varphi(x) \to \varphi(s(x)))$ and $B : \varphi(\underline{0})$ by induction: One derives $\forall x^{\mathbf{int}} \varphi(x)$, and instantiates $x$ to $\underline{3}$. But this proof may be simplified to a direct application of $A$ three times to $B$, that is to $A\underline{2}(A\underline{1}(A\underline{0}B)) : \varphi(\underline{3})$. See also [8], Remark 11.7.8. Let us note however that the induction axiom for integers may be stated in each of the following forms:

a) $\forall R(\forall x(R(x) \to R(s(x))) \to R(\underline{0}) \to \forall x^{\mathbf{int}} R(x))$, or

b) $\forall R(\forall x^{\mathbf{int}}(R(x) \to R(s(x))) \to R(\underline{0}) \to \forall x^{\mathbf{int}} R(x))$, or

c) $\forall R(\forall x(R(x) \to R(s(x))) \to R(\underline{0}) \to \forall x R(x))$.

The difference is whether we specify the type of the quantified variable $x$, which means we assume that it satisfies a predicate **int**, or not. When erasing the first-order contents from a formula, bounded

[1] Available from `http://zls.mimuw.edu.pl/urzy/ftp.html`
[2] This lecture is based on a joint work with Zdzisław Spławski

quantification should be turned into implication, but an unbounded quantification corresponds to a trivial assumption and can be ignored. Thus, our formula (a) erase to the type of *iterator*:

$$\mathbf{It}_\sigma : (\sigma \to \sigma) \to \sigma \to \mathbf{int} \to \sigma,$$

with the reduction rules

$$\mathbf{It}_\sigma MN\underline{0} \Rightarrow N;$$

$$\mathbf{It}_\sigma MN\underline{k+1} \Rightarrow M(\mathbf{It}_\sigma MN\underline{k}).$$

The induction axiom in the form (b) erases to the type of *recursor*, as in Gödel's System **T**:

$$\mathbf{R}_\sigma : (\mathbf{int} \to \sigma \to \sigma) \to \sigma \to \mathbf{int} \to \sigma.$$

The reduction rules for the recursor are as follows.

$$\mathbf{R}_\sigma MN\underline{0} \Rightarrow N;$$

$$\mathbf{R}_\sigma MN\underline{k+1} \Rightarrow M\underline{k}(\mathbf{R}_\sigma MN\underline{k}).$$

In a sense, these variants of System **T** are equivalent. In particular, both system represent exactly the integer functions that are provably recursive in PA. However, while $\mathbf{It}_\sigma$ can be seen as a special case of $\mathbf{R}_\sigma$, to define the latter one needs $\mathbf{It}_{\mathbf{int}\times\sigma}$ rather than $\mathbf{It}_\sigma$. In addition, the translation is not uniform in that it works only for closed terms and a single reduction step is simulated by possibly many steps. Thus, for instance, **R**-reductions can not be simulated by **It**-reductions in constant time (see Parigot [11]).

Variant (c) corresponds to a common way in which arithmetic is defined as a first-order theory. One assumes that all individuals are integers, and there is no special predicate symbol for that. The erasure of (c) is the type of Church numerals. The difference between case (c) and cases (a) and (b) is that the latter type is inhabited (i.e., logically valid). Each numeral can be seen as one possible proof of this type. But it also can be seen as an erasure of a proof by induction. This time, however one cannot postulate an induction constant with a reasonable reduction rule. A proof by induction of a statement $R(\underline{k})$ from assumptions $A : R(\underline{0})$ and $B : \forall x(R(x) \to R(s(x)))$ reduces to $B^k(A)$. Its erasure should reduce to $\overline{B}^k(\overline{A})$, but since $k$ has been erased, there would be different reduction schemes for the same expression (one for each $k$). The best one can do is to have different erasures for each $k$. The $k$-time iterator is just the Church numeral itself and we conclude that Church numerals are specific instances of the induction scheme.

For integer lists, variant (c) gives type $\forall\alpha(\alpha \to (\alpha \to \mathbf{int} \to \alpha) \to \alpha)$ inhabited by terms like e.g. $\lambda x\lambda f.f(f(f(fx\underline{k_1})\underline{k_2})\underline{k_3})\underline{k_4}$.

## 2 Fixpoint types

In the previous section, type were treated as predicates or subsets of some generic universe of individual objects. An alternative is to think of types as of separate domains. The constructors build complex objects from simple objects and can change their types. For instance, one can think of $a :: \ell$ as of a pair $\langle a, \ell \rangle$. Adding an integer $a$ at front of a list is thus a polymorphic constructor of type $\forall\alpha(\alpha \to \mathbf{int} \times \alpha)$. Assuming that `nil` : $\mathbf{1}$, where $\mathbf{1}$ is some unit type, it would be desirable if the type **list** of integer lists satisfied the equation:

$$\mathbf{list} = \mathbf{1} + (\mathbf{int} \times \mathbf{list}).$$

This is a fixpoint equation for an operator on types that turns a type $\alpha$ into $\mathcal{F}(\alpha) = \mathbf{1} + (\mathbf{int} \times \alpha)$. An important feature of this, and similar definitions is that the operator $\mathcal{F}$ is induced by a certain choice of constructors. Thus, it is monotone (because constructors built new objects from the objects already available) and that each element of our type is in the union of $\{\}$, $\mathcal{F}(\{\})$, $\mathcal{F}^2(\{\})$, $\mathcal{F}^3(\{\}), \ldots$. So the type **list** is the *least* fixpoint of $\mathcal{F}$. The notion of a positive fixpoint (recursive) type is a generalization of such examples.

There are Curry-style calculi of fixpoint types, where fixpoint equations as the one for lists are taken literally, cf. [2, 14]. The requirement that actual solutions of such equations really exist is a very strong postulate though, and obviously often difficult to satisfy: A type system one has to work with does not have to be sufficiently flexible.

However, what is needed in most cases is a weaker property. First of all, one should be able to properly interprete every object of type $\mathbf{1} + (\mathbf{int} \times \mathbf{list})$ as an object of type **list**. Thus we need an introduction operator $\mathbf{in} : \mathbf{1} + (\mathbf{int} \times \mathbf{list}) \to \mathbf{list}$. Then there must also be an elimination operator and a reduction rule to describe the routine of using lists in computations. Here we discuss various possible choices of these in the context of polymorphism.

# 3 System $\boldsymbol{\lambda}$2I

We extend the syntax of polymorphic types with the construction $\mu\alpha.\tau$, where $\alpha$ is a type variable and $\tau$ is a type such that $\alpha$ occurs in $\tau$ only positively. The variable $\alpha$ is considered bound in $\mu\alpha.\tau$.

The system $\boldsymbol{\lambda}$**2I** (taken essentially from [10]) is an extension of $\boldsymbol{\lambda}$**2** with two new term constants:

$$\mathbf{in}_{\mu\alpha.\tau} : \tau[\mu/\alpha] \to \mu\alpha.\tau;$$

$$\mathbb{I}_{\mu\alpha.\tau} : \forall\beta(\forall\alpha((\alpha \to \beta) \to (\tau(\alpha) \to \beta)) \to \mu\alpha.\tau \to \beta),$$

and the following reduction scheme:[3]

$$\mathbb{I}\sigma A(\mathbf{in}\, x) \Rightarrow A(\mu\alpha.\tau)(\mathbb{I}\sigma A)x,$$

where $A$ is a term of type $\forall\alpha((\alpha \to \sigma) \to \tau(\alpha) \to \sigma)$.

To explain this definition let us return to the list example. Type **list** is the union of all the $\mathcal{F}^n(\{\})$'s. Thus, a function of type $\mathbf{list} \to \beta$ may be defined by induction on $n$. For this, one has to describe how to define a function of type $\mathcal{F}^{n+1}(\{\}) \to \beta$, provided a function of type $\mathcal{F}^n(\{\}) \to \beta$ has already been defined. Typically this is done in a generic way: one defines a function of type $\mathcal{F}(\alpha) \to \beta$ from a function of type $\alpha \to \beta$. The genericity of this construction (with respect to $\alpha$) has the following important consequence: for an argument $x \in \mathcal{F}^{n+1}(\{\})$ only the "top level" information about $x$ can be used.

The type of the constant $\mathbb{I}$ expresses this idea in a general setting. The argument of $\mathbb{I}$ is such a generic transformation that makes a function of type $\tau(\alpha) \to \beta$ from a function of type $\alpha \to \beta$.

The reduction rule can be explained as follows. We think of type $\mu\alpha.\tau$ as of a union of types $\mu^k$ where $\mu^k$ is the image of $\tau^k(\bot)$ under **in**. For $x \in \tau^k(\bot)$, we have $\mathbf{in}\, x \in \mu^k$. We define a function $\mathbb{I}\sigma A$ for the argument $\mathbf{in}\, x$ using the transformation $A : \forall\alpha((\alpha \to \sigma) \to \tau(\alpha) \to \sigma)$. If the argument $\mathbf{in}\ x$ is in $\mu^{k+1}$ then we assume that our function has already been defined for arguments in $\mu^k$. Thus we use the transformation $A$ specialized to $A\mu^k : (\mu^k \to \sigma) \to \tau(\mu^k) \to \sigma$ and applied to the $k$-th approximation of $\mathbb{I}\sigma A$. The result of this transformation is applied to $x \in \tau(\mu^k)$.

Another way of looking at this reduction rule is that the left-hand side is an introduction followed by an elimination. The right-hand side removes the redundant introduction of $x : \tau[\mu\alpha.\tau/\alpha]$ into $\mu\alpha.\tau$, and applies directly to $x$. Note that (the normal form of) the term $A$ will typically be of

[3] We skip the indices wherever this does not cause a confusion.

the form $\Lambda\alpha\lambda f^{\alpha\to\sigma}\lambda x^{\tau}.B^{\sigma}$ and that $\alpha$ is not free in $\sigma$. Thus $B$ should typically be an eliminator expression for type $\tau$, its behaviour depending on the input $x$. The action of $B$ on an input $X:\tau$ is thus either to produce an output of type $\sigma$ directly or to apply $f:\alpha\to\sigma$ to an argument obtained by decomposing $X:\tau$. This can be seen in the following example.[4]

**Example:** Let $\tau = \mathbf{1} + (\mathbf{int}\times\alpha)$. Define a combinator $A$ of type $\forall\alpha((\alpha\to\mathbf{int})\to\tau\to\mathbf{int})$, as follows:

$$A \equiv \Lambda\alpha\lambda f^{\alpha\to\mathbf{int}}\lambda x^{\tau}.\mathbf{case}\ x\ \mathbf{of}\ [u]\underline{0},[v]\underline{1}+f(\pi_2(v)).$$

Then the expression $\mathbb{I}\mathbf{int}A(\mathbf{in}\,\ell)$ evaluates to the length of the list $\ell$.

The above example can be generalized as follows: In system $\boldsymbol{\lambda}\mathbf{2I}$, one can simulate list iterators, defined by the following clauses:

$$\mathbf{It}_{\sigma}MN\mathtt{nil} \Rightarrow N;$$

$$\mathbf{It}_{\sigma}MN\langle a,\ell\rangle \Rightarrow Ma(\mathbf{It}_{\sigma}MN\ell).$$

The reader can easily extend this observation to integers defined as $\mu\alpha.\mathbf{1}+(\mathbf{1}\times\alpha)$. (Integers are lists of `nil`s.)

# 4 Second-order interpretation

Let us present the types of **in** and $\mathbb{I}$ in the form of the following rules, where $\mu$ abbreviates $\mu\alpha.\tau$, and $\tau(\mu)$ abbreviates $\tau[\mu\alpha.\tau/\alpha]$. In order to highlight the analogy we are going to point out, let us replace some occurrences of the arrow by the inclusion symbol:

$$(\mathbf{in})\ \tau(\mu)\subseteq\mu \qquad\qquad (\mathbb{I})\ \frac{\forall\alpha(\alpha\subseteq\beta\to\tau(\alpha)\subseteq\beta)}{\mu\subseteq\beta}$$

If we think of $\tau$ as of a monotone operator in a complete lattice then the above conditions guarantee that $\mu$ is the least fixpoint of $\tau$. In addition, the above use of $\subseteq$ is not just *ad hoc* and can be justified as follows.

Consider a formula $\Phi[R](x)$ of second-order logic, where $x$ is an individual variable and $R$ is a first-order (relation) variable occurring only positively in $\Phi$. Then the following second-order axioms assert that $L$ is the least fixpoint of $\Phi[R]$ seen as a monotone operator on $R$:

$$\forall x(\Phi[L](x)\to L(x));$$

$$\forall R(\forall P(P\subseteq R\to\Phi[P]\subseteq R)\to L\subseteq R).$$

Of course, the inclusion $P\subseteq R$ is an abbreviation for $\forall x(P(x)\to R(x))$ and similarly for the other inclusions.

As observed by Daniel Leivant [6, 7] and also by J.-L. Krivine and M. Parigot [5, 12], types of our constants are obtained from the above second-order axioms via a dependency-erasing operator.

Note that we can also see the reduction rule as an erasure of a proof reduction rule. Indeed, suppose we have a proof $A$ of a formula of the form $\forall P(P\subseteq R\to\Phi[P]\subseteq R)$ and a proof $B$ of $L(a)$. Then we can derive $R(a)$ using induction (i.e., the second axiom). But if $B$ is a canonical proof of $L(a)$ then it consists of a proof of $\Phi^k[\bot](a)$ followed by $k$ applications of the introduction axiom. This means

[4] In the examples we use various constructs that formally do not occur in $\boldsymbol{\lambda}\mathbf{2I}$, to enhance readability. These constructs can be however simulated in $\boldsymbol{\lambda}\mathbf{2}$.

that our proof of $R(a)$ by induction contains a redundancy and should be replaced by $k$ applications of $A$ to a trivial proof $\varepsilon_\sigma : \bot \to \sigma$. The reduction rule reflects this idea.

## 5 First-order interpretation

To some extent, the above construction reflects a first-order induction scheme. For instance, consider again the induction scheme for integers:

$$\forall x(R(x) \to R(s(x))) \to R(\underline{0}) \to \forall x^{\mathbf{int}} R(x)$$

The above can be written as a property of types as follows:

$$\forall x(x \in \mathbf{1} + (\mathbf{1} \times R) \to x \in R) \to \forall x(x \in \mathbf{int} \to x \in R).$$

One way of replacing individuals by sets (types) is to use singletons, as in the following statement:

$$\forall x(\{x\} \subseteq \mathbf{1} + (\mathbf{1} \times R) \to \{x\} \subseteq R) \to \forall x(\{x\} \subseteq \mathbf{int} \to \{x\} \subseteq R).$$

If we generalize the above from singletons to arbitrary predicates and erase dependencies then we obtain the axiom $\mathbb{I}$.[5]

## 6 System $\boldsymbol{\lambda}$2J

The interpretation of recursive types as fixpoints of monotone maps in a complete lattice suggests the following simpler form of rule ($\mathbb{I}$):

$$(\mathbb{J})\ \frac{\tau(\beta) \subseteq \beta}{\mu \subseteq \beta},$$

where $\tau(\beta)$ abbreviates $\tau[\beta/\alpha]$. This rule directly expresses the fact that the lfp is the infimum of pre-fixpoints, that is, $\mu = \inf\{\beta \mid \tau(\beta) \subseteq \beta\}$. In system $\boldsymbol{\lambda}\mathbf{2J}$, following Leivant [7], we have again

$$\mathbf{in}_{\mu\alpha.\tau} : \tau[\mu/\alpha] \to \mu\alpha.\tau,$$

but we choose an eliminator of a different type:

$$\mathbb{J}_{\mu\alpha.\tau} : \forall\beta((\tau(\beta) \to \beta) \to \mu\alpha.\tau \to \beta).$$

The cost of simplifying the type of the eliminator results in a more complex definition of reduction. Indeed, the meaning of $\mathbb{J}\sigma A(\mathbf{in}\, x)$ with $A : \tau(\sigma) \to \sigma$ should be to run $A$, which is an eliminator for $\tau(\sigma)$, on the "unpacked" input $x$ of type $\tau(\mu)$. For this we must first observe that the "inclusion" $\mu \subseteq \sigma$ given by $\mathbb{J}\sigma A$ can be "lifted" to an inclusion $\tau(\mu) \subseteq \tau(\sigma)$, because $\tau$ is a monotone operator. In fact, one can easily define a family of combinators

$$\mathcal{M}_{\varphi,\psi} : (\varphi \to \psi) \to \tau(\varphi) \to \tau(\psi),$$

for all types $\varphi$ and $\psi$. The reduction rule for $\boldsymbol{\lambda}\mathbf{2J}$ looks as follows:

$$\mathbb{J}\sigma M\,(\mathbf{in}\, N) \quad \Rightarrow \quad M\,(\mathcal{M}_{\mu,\sigma}\,(\mathbb{J}\,\sigma\, M)\, N)$$

[5] A similar correspondence occurs between the induction written as

$$\forall R(\forall x^{\mathbf{int}}(R(x) \to R(s(x))) \to R(\underline{0}) \to \forall x^{\mathbf{int}} R(x))$$

and the axiom $\mathbf{R}$.

Let us try to explain the reduction rule. It describes an inductive definition of a function $\mathbb{J}\sigma A$ with help of a transformation $A : \tau(\sigma) \to \sigma$. We have an argument $\mathbf{in}(x)$ in $\mu^{k+1}$ and we assume that the approximation $(\mathbb{J}\sigma A)^k : \mu^k \to \sigma$ has already been defined. We can get from $\mu^{k+1}$ to $\sigma$ by lifting $(\mathbb{J}\sigma A)^k$ to type $\tau(\mu^k) \to \tau(\sigma)$. We obtain the following commuting diagram:

$$\begin{tikzcd}
\mu^{k+1} \arrow[r, "(\mathbb{J}\sigma A)^{k+1}"] & \sigma \\
\tau(\mu^k) \arrow[u, "\mathbf{in}"] \arrow[r, "\mathcal{M}(\mathbb{J}\sigma A)^k"'] & \tau(\sigma) \arrow[u, "A"']
\end{tikzcd}$$

There are such diagrams for every $k$ and we can join them as follows:

$$\begin{tikzcd}
\mu^{k+1} \arrow[r, "(\mathbb{J}\sigma A)^{k+1}"] & \sigma \\
\tau(\mu^k) \arrow[u, "\mathbf{in}"] \arrow[r, "\mathcal{M}(\mathbb{J}\sigma A)^k"'] & \tau(\sigma) \arrow[u, "A"'] \\
\tau^2(\mu^{k-1}) \arrow[u, "\mathbf{in}"] \arrow[r, "\mathcal{M}^2(\mathbb{J}\sigma A)^{k-1}"'] & \tau^2(\sigma) \arrow[u, "\mathcal{M}A"'] \\
\tau^k(\mu^1) \arrow[u, dotted, no head] \arrow[r, "\mathcal{M}^k(\mathbb{J}\sigma A)^1"'] & \tau^k(\sigma) \arrow[u, dotted, no head] \\
\tau^{k+1}(\perp) \arrow[u, "\mathbf{in}"] \arrow[r, "\mathcal{M}^{k+1}(\mathbb{J}\sigma A)^0"'] & \tau^{k+1}(\sigma) \arrow[u, "\mathcal{M}^k A"']
\end{tikzcd}$$

Remember that all the $(\mathbb{J}\sigma A)^i$ are approximations of $\mathbb{J}\sigma A$, and thus the diagrams above remain correct when $(\mathbb{J}\sigma A)^i$ is replaced by $\mathbb{J}\sigma A$. Clearly, $(\mathbb{J}\sigma A)^0 : \perp \to \sigma$ is just $\varepsilon_\sigma$. The operator $\mathcal{M}^{k+1}(\varepsilon_\sigma)$ is an identity embedding. Thus, the operator $(\mathbb{J}\sigma A)^{k+1} \circ \mathbf{in}^{k+1}$ unfolds to the composition of all the $\mathcal{M}^i A$.

A logical interpretation of the reduction rule of $\boldsymbol{\lambda}\mathbf{2J}$ is as follows: the assumption $A : \tau(\beta) \subseteq \beta$ expresses the fact that $\beta$ is closed under $\tau$. A proof that a particular element of the fixpoint belongs to $\beta$ should only require $\tau^k(\beta) \subseteq \beta$, for a particular $k$, rather than the closure statement in its full generality.

# 7 Translations to $\boldsymbol{\lambda}\mathbf{2}$

Of course, all the above discussed systems are extensions of $\boldsymbol{\lambda}\mathbf{2}$. However, none of them is essential in that the new types and operators are definable within $\boldsymbol{\lambda}\mathbf{2}$. The polymorphic type representing $\mu\alpha.\tau$ in $\boldsymbol{\lambda}\mathbf{2J}$ can be easily guessed from the Curry-Howard interpretation of Section 4. Indeed, the least fixpoint $L$ of $\Phi[R]$ is explicitly definable in second-order logic as follows:

$$L(x) := \forall R((\Phi[R] \subseteq R) \to R)$$

This means exactly that $L$ is the least relation satisfying the inclusion $\Phi[L] \subseteq L$. The dependency-erasure map applied to this formula gives the second-order type $\mu_{\mathrm{J}} = \forall\beta((\tau(\beta) \to \beta) \to \beta)$. One can now define

$$\begin{aligned}\mathbb{J} &= \Lambda\alpha\lambda y^{\tau(\alpha)\to\alpha}\lambda z^{\mu_J}.z\alpha y\\ \mathbf{in} &= \lambda x^{\tau(\mu_J)}\Lambda\beta\lambda y^{\tau(\beta)\to\beta}.y(\mathcal{M}_{\mu_J,\beta}(\mathbb{J}\beta y)x)\end{aligned}$$

**Proposition 1** *(Geuvers, Spławski)*

*The above intepretation of* $\boldsymbol{\lambda}\mathbf{2J}$ *in* $\boldsymbol{\lambda}\mathbf{2}$ *is correct, i.e., we have* $\mathbb{J}\,\sigma\,M\,(\mathbf{in}\,N) \twoheadrightarrow_\beta M\,(\mathcal{M}_{\mu,\sigma}\,(\mathbb{J}\,\sigma\,M)\,N)$, *for all* $M$, $N$ *of appropriate types.*

For system $\boldsymbol{\lambda}\mathbf{2I}$, a more adequate representation of $\mu\alpha.\tau$ is the type

$$\mu_I = \forall\beta(\forall\alpha((\alpha \to \beta) \to \tau(\alpha) \to \beta) \to \beta)$$

with the operators

$$\begin{aligned}\mathbb{I} &= \Lambda\beta\lambda y^{\forall\alpha((\alpha\to\beta)\to\tau(\alpha)\to\beta)}\lambda z^{\mu_I}.z\beta y,\\ \mathbf{in} &= \lambda x^{\tau(\mu_I)}\Lambda\beta\lambda y^{\forall\alpha((\alpha\to\beta)\to\tau(\alpha)\to\beta)}.y\mu(\mathbb{I}\;\beta\;y)x.\end{aligned}$$

**Proposition 2** *(Spławski)*

*The above intepretation of* $\boldsymbol{\lambda}\mathbf{2I}$ *in* $\boldsymbol{\lambda}\mathbf{2}$ *is correct, i.e., we have* $\mathbb{I}\,\sigma\,M\,(\mathbf{in}\,N) \twoheadrightarrow_\beta M\,\mu_I\,(\mathbb{I}\,\sigma\,M)\,N$, *for all* $M$, $N$ *of appropriate types.*

The representation in $\boldsymbol{\lambda}\mathbf{2}$ of inductive data types (least fixpoints of monotonic operators) like natural numbers or lists dates back to [1]. See also [4]. A generic translation (for $\boldsymbol{\lambda}\mathbf{2J}$) was probably first given in [3] (see also [13]), but was implicit in [7].

Dual notions of greatest fixpoints, corresponding to types with coiterators, can be similarly defined within $\boldsymbol{\lambda}\mathbf{2}$. Wraith [15] seems to be the first who encoded in $\boldsymbol{\lambda}\mathbf{2}$ positive coinductive types with coiterators.

# 8 Fixpoint types with recursors

As observed by Parigot [11], there is a difference in expressive power between iterators and recursors over integers. For instance, every representation of the predecessor function for Church numerals ("iterative" integers) must be of at least linear time complexity. Thus, the recursor operator over Church numerals can not be interpreted within $\boldsymbol{\lambda 2}$ in such a way that reductions are uniformly preserved.

In this section we present two extensions of $\boldsymbol{\lambda 2}$ with recursors and we prove that they are equivalent to the system $\boldsymbol{\lambda 2}\mathbf{U}$, presented in the Introduction.

Mendler [9] introduced the system (which we call $\boldsymbol{\lambda 2}\mathbf{R}$), similar to $\boldsymbol{\lambda 2}\mathbf{I}$, but with a different eliminator

$$\mathbb{R}_{\mu\alpha\tau} : \forall\beta(\forall\alpha((\alpha \to \mu) \to (\alpha \to \beta) \to \tau(\alpha) \to \beta) \to \mu \to \beta),$$

and the following reduction scheme:

$$\mathbb{R}\sigma A(\mathbf{in}N) \Rightarrow A\mu(\lambda z^{\mu}.z)(\mathbb{R}\sigma A)N.$$

The type of $\mathbb{R}_{\mu\alpha\tau}$ can be represented as the following rule:

$$(\mathbb{R})\ \frac{\forall\gamma(\gamma \subseteq \mu \wedge \gamma \subseteq \beta \to \tau(\gamma) \subseteq \beta)}{\mu \subseteq \beta}$$

Recall that the premise of an analogous rule for $\mathbb{I}$ expresses the closure of $\beta$ under the operator $\tau$. The premise of the above rule is weaker: $\beta$ is closed under $\tau$, provided the argument is already in $\mu$. Operationally, the difference is that the operator $A\mu^k : (\mu^k \to \sigma) \to \tau(\mu^k) \to \sigma$ occurring in $\boldsymbol{\lambda 2}\mathbf{I}$ reductions cannot rely on any specific knowledge about the type $\mu$ and the argument $x : \tau(\mu^k)$, because it must be generic in $\alpha$. Thus, the components of $x$ must be directly passed to $(\mathbb{I}\sigma A)^k$ and that's it. Now, the transformation $A$ has type $\forall\alpha((\alpha \to \mu^k) \to (\alpha \to \sigma) \to \tau(\alpha) \to \sigma)$. It can directly use its third argument (expected to be $x : \tau(\mu^k)$), by passing the components of $x$ to the first argument (expected to be identity). In this way, an output of type $\sigma$ may even be constructed with no use of the second argument.

The second-order interpretation of the type of $\mathbb{R}$, the axiom

$$\forall Q(\forall R((R \subseteq \mu) \to (R \subseteq Q) \to \tau(R) \subseteq Q) \to \mu \to Q)$$

can be further simplified to

$$\forall Q((\mu \cap Q \subseteq Q) \to \mu \subseteq Q),$$

and this gives a hint for a recursive counterpart of $\boldsymbol{\lambda 2}\mathbf{J}$, namely the system $\boldsymbol{\lambda 2}\mathbf{Rec}$ of [13]. The recursor and the reduction scheme for $\boldsymbol{\lambda 2}\mathbf{Rec}$ are as follows:[6]

$$\mathbf{Rec}_{\mu\alpha\tau} : \forall\beta((\tau(\mu \times \beta) \to \beta) \to \mu \to \beta)$$

$$\mathbf{Rec}\,\sigma A(\mathbf{in}x) \Rightarrow A(\mathcal{M}_{\mu,\mu\times\sigma}(\lambda z^{\mu}.\langle z, \mathbf{Rec}\,\sigma A z\rangle)x).$$

The symbol $\mathcal{M}$ stands here again for an appropriately defined "lifting" combinator.

The difference between $\mathbf{Rec}$ and $\mathbb{J}$ is similar to the difference between $\mathbb{R}$ and $\mathbb{I}$. With $\mathbb{I}$, the only way from $\mu^{k+1}$ to $\sigma$ was essentially to go around the whole graph (see picture). A graph for $\mathbf{Rec}$ would have shortcuts given by identity functions. To see this, note that $\mathbf{Rec}$-reduction can be written as

$$\mathbf{Rec}\,\sigma A(\mathbf{in}x) \Rightarrow A(\mathcal{M}_{\mu,\mu\times\sigma}(\mathbf{I}_{\mu} \times \mathbf{Rec}\,\sigma A)x).$$

Note that $\mathbf{Rec}$ strengthens the iterator $\mathbb{J}$ in the same way as recursor $\mathbf{R}$ from Gödel's System $\mathbf{T}$ strengthens the iterator $\mathbf{It}$. In both cases this difference can be seen as caused by a different understanding of first-order quantifier in the appropriate second-order axiom. (Read $P \subseteq R$ as either $\forall x(P(x) \to R(x))$ or $\forall x(L(x) \to P(x) \to R(x))$.)

[6]Remember that product type (definable in $\boldsymbol{\lambda 2}$) corresponds to conjunction.

# 9 System $\boldsymbol{\lambda}$2U

This is the simplest calculus of recursive types in Church style. We have two constants **Fold** : $\sigma[\mu\alpha.\sigma/\alpha] \to \mu\alpha.\sigma$ and **Unfold** : $\mu\alpha.\sigma \to \sigma[\mu\alpha.\sigma/\alpha]$ and the reduction rule

$$\mathbf{Unfold} \circ \mathbf{Fold} \Rightarrow \mathbf{I}.$$

**Proposition 3** *(Spławski, Geuvers) Systems* **λ2R**, **λ2Rec** *and* **λ2U** *are interdefinable.*

**Proof:** Here we show only that **λ2R** can be defined in **λ2U** and vice versa.

In order to interprete **λ2U** within **λ2R** we do not need to redefine types $\mu\alpha\tau$. For every $\tau$ we take $\mathbf{Fold}_{\mu\alpha\tau} \equiv \mathbf{in}_{\mu\alpha\tau}$, and we only need to define $\mathbf{Unfold}_{\mu\alpha\tau}$. Abbreviating again $\mu\alpha\tau$ to $\mu$, we define

$$\mathbf{Unfold} \equiv \mathbb{R}(\tau(\mu))(\Lambda\gamma.\lambda x_1^{\gamma\to\mu}\lambda x_2^{\gamma\to\tau(\mu)}\lambda x_3^{\tau(\gamma)}.\mathcal{M}_{\alpha,\mu}x_1x_3).$$

The verification that $\mathbf{Unfold}(\mathbf{Fold}(x^{\tau(\mu)})) \twoheadrightarrow_{\mathrm{R}} x$ is easy, but we must note that $\mathcal{M}_{\mu,\mu}\mathbf{I}^{\mu\to\mu} =_\beta \mathbf{I}^{\tau(\mu)\to\tau(\mu)}$.

The interpretation of **λ2R** in **λ2U** is more complicated because now we cannot use the same type to represent $\mu\alpha\tau$. In order to avoid confusion, below we use the notation $\mu^{\mathrm{R}}$ and $\mu^{\mathrm{U}}$ to denote fixpoints in **λ2R** and **λ2U**, respectively. Let $\sigma = \forall\beta(\forall\gamma((\gamma \to \alpha) \to (\gamma \to \beta) \to \tau(\gamma) \to \beta) \to \beta)$.

We define $\mu^{\mathrm{R}}\alpha.\tau$ as $\mu^{\mathrm{U}}\alpha.\sigma$. In the following definitions $\mu^{\mathrm{R}}\alpha.\tau$ is abbreviated by $\mu$ and **Fold**, **Unfold** stand for $\mathbf{Fold}_{\mu^{\mathrm{U}}\alpha.\sigma}$ and $\mathbf{Unfold}_{\mu^{\mathrm{U}}\alpha.\sigma}$, respectively. We take:

$$\mathbb{R} \equiv \Lambda\beta.\lambda y^{\forall\gamma(\dots)}\lambda z^{\mu}.\mathbf{Unfold}z\beta y.$$

$$\mathbf{in} \equiv \lambda x^{\tau(\mu)}.\mathbf{Fold}(\Lambda\beta.\lambda y^{\forall\gamma(\dots)}.y\mu\mathbf{I}^{\mu\to\mu}(\mathbb{R}\beta y)x).$$

We leave to the reader the details of the reduction $\mathbb{R}\sigma M(\mathbf{in}N) \twoheadrightarrow_{\mathrm{U}} M\mu(\lambda z^{\mu}.z)(\mathbb{R}\sigma M)N$. ∎

# 10 Recursion is stronger than iteration

We say that type $\rho$ can be *embedded* into another type $\tau$, written $\rho \leq \tau$, iff there are two terms $F : \rho \to \tau$ and $G : \tau \to \rho$, such that $G \circ F =_\beta \mathbf{I}$, or equivalently $G(Fx) =_\beta x$, where $x^\rho$ is a fresh variable. For a type $\sigma$, let $|\sigma|$ be the length of $\sigma$.

**Theorem 4** *If* $\rho \leq \tau$ *then* $|\rho| \leq |\tau|$.

**Corollary 5** *System* **λ2U** *cannot be defined within* **λ2**.

**Proof:** Suppose there is a type $\mu$ and there are operators **Fold** and **Unfold** implementing $\mu\alpha.\sigma$, where $|\sigma| \neq 1$ then $\sigma[\mu/\alpha] \leq \mu$, contradicting Theorem 4. ∎

## 11 Exercises

**Exercise 1:** Let $\tau$ be a monotone operator in a complete lattice $L$ and let $a \in L$. Show that the following conditions are equivalent:

a) For all $b \in L$, if $b \leq a$ then $\tau(b) \leq a$;

b) $\tau(b) \leq b$.

Conclude that $\mathrm{lfp}(\tau) = \inf\{a \in L \mid \text{for all } b \in L, \text{ if } b \leq a \text{ then } \tau(b) \leq a\}$.

**Exercise 2:** Let $\sigma_1 = \tau(\beta) \to \beta$ and $\sigma_2 = \forall\alpha((\alpha \to \beta) \to \tau(\alpha) \to \beta)$. Find combinators $F : \sigma_1 \to \sigma_2$ and $G : \sigma_2 \to \sigma_1$, such that $G \circ F =_{\beta\eta} \mathbf{I}$.

**Exercise 3:** Let $\tau$ be a monotone operator in a complete lattice $L$. Let $\varphi(x) = \inf\{y : \tau(x \cap y) \subseteq y\}$. Let $a$ be an arbitrary fixpoint of $\varphi$. Then $a = \mathrm{lfp}(\tau)$.

**Exercise 4:** Are the the following conditions equivalent for every complete lattice $L$, every monotone operator $\tau$ in $L$, and every $a \in L$?

a) For all $b \in L$, if $\tau(b) \leq b$ then $a \leq b$;

b) For all $b \in L$, if $\tau(b \cap a) \leq b$ then $a \leq b$.

Are they equivakent under the additional assumption $\tau(a) \leq a$?

**Exercise 5:** Explain the following paradox. Rules of $\boldsymbol{\lambda}\mathbf{2J}$ correspond to the following statement: "Every monotone function has a least fixpoint". Rules of $\boldsymbol{\lambda}\mathbf{2U}$ correspond to a weaker statement: "Every monotone function has a fixpoint" (not necessarily a least one). Yet one can interprete $\boldsymbol{\lambda}\mathbf{2J}$ in $\boldsymbol{\lambda}\mathbf{2U}$ and not the other way round?

## References


[1] C. Böhm and A. Berarducci. Automatic synthesis of typed $\Lambda$-programs on term algebras. *Theoretical Computer Science*, 39(2/3):135–154, 1985.

[2] Cardone, F., Coppo, M., Two extensions of Curry's type inference system, in *Logic and Computer Science* (P. Odifreddi, Ed.), Academic Press, 1990, pp. 19–75.

[3] H. Geuvers. Inductive and coinductive types with iteration and recursion. In *Proceedings of the Workshop on Types for Proofs and Programs*, Båstad, Sweden, 1992, 193–217.

[4] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, Cambridge, 1990. Second edition.

[5] Krivine, J.-L., Parigot, M., Programming with proofs, *J. Inf. Process. Cybern. (EIK)*, **26** (3), 1990, 149–167.

[6] Leivant, D., Reasoning about functional programs and complexity classes associated with type disciplines, *Proc. 24th Symposium on Foundations of Computer Science*, IEEE, 1983, pp. 460-469.

[7] Leivant, D., Contracting proofs to programs, in: *Logic in Computer Science* (P. Odifreddi, ed.), Academic Press, 1990, pp. 279–327.

[8] Sørensen, M.H., Urzyczyn, P., Lectures on the Curry-Howard Isomorphism, Datalogisk Institut, Københavns Universitet, 1998.

[9] Mendler, N.P., Recursive types and type constraints in the second-order lambda calculus, *Proc. 2nd Symposium on Logic in Computer Science*, 1987, pp. 30–36.

[10] Mendler, N.P., Inductive types and type constraints in the second-order lambda calculus, *Annals of Pure and Applied Logic*, **51** (1991), 159–172.

[11] Parigot. M., On the representation of data in lambda-calculus, *Proc. Computer Science Logic'89*, (E. Börger, H. Kleine Büning, M.M. Richter, Eds.), Lecture Notes in Computer Science 440, Springer-Verlag, Berlin, pp. 309–311.

[12] Parigot. M., Recursive programming with proofs *Theoretical Computer Science*, **94**, 1992, 335–356.

[13] Spławski, Z., *Proof-Theoretic Approach to Inductive Definitions in ML-like Programming Language versus Second-Order Lambda Calculus*, PhD Thesis, Wrocław University, 1993.

[14] Urzyczyn, P., Positive recursive type assignment, *Fundamenta Informaticae*, **28**, No. 1-2, 1996, 197–209.

[15] Wraith, G.C., A note on categorical data types, *Category Theory and Computer Science* (D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, A. Poigné, eds.), Lecture Notes in Computer Science 389, Springer-Verlag, Berlin 1989, pp. 118–127.