Type Fixpoints: Iteration vs. Recursion

Zdzisław Spławski
Faculty of Informatics and Management,
Wrocław University of Technology,
ul. Wybrzeże Wyspiańskiego 27
50-370 Wrocław, Poland.
(splawski@ci.pwr.wroc.pl)

Paweł Urzyczyn*
Institute of Informatics,
Warsaw University,
ul. Banacha 2,
02-097 Warsaw, Poland.
(urzy@mimuw.edu.pl)

Abstract

Positive recursive (fixpoint) types can be added to the polymorphic (Church-style) lambda calculus $\boldsymbol{\lambda 2}$ (System **F**) in several different ways, depending on the choice of the elimination operator. We compare several such definitions and we show that they fall into two equivalence classes with respect to mutual interpretability by means of beta-eta reductions. Elimination operators for fixpoint types are thus classified as either "iterators" or "recursors". This classification has an interpretation in terms of the Curry-Howard correspondence: types of iterators and recursors can be seen as images of induction axioms under different dependency-erasing maps. Systems with recursors are beta-eta equivalent to a calculus $\boldsymbol{\lambda 2U}$ of recursive types with the operators $\mathbf{Fold} : \sigma[\mu\alpha.\sigma/\alpha] \to \mu\alpha.\sigma$ and $\mathbf{Unfold} : \mu\alpha.\sigma \to \sigma[\mu\alpha.\sigma/\alpha]$, where the composition $\mathbf{Unfold} \circ \mathbf{Fold}$ reduces to identity.

It is known that systems with iterators can be defined within $\boldsymbol{\lambda 2}$, by means of beta reductions. We conjecture that systems with recursors can not. In this paper we show that the system $\boldsymbol{\lambda 2U}$ does not have such a property. For this we study the notion of polymorphic type embeddability (via (beta) left-invertible terms) and we show that if a type σ is embedded into another type τ then τ must be of depth at least equal to the depth of σ.

1 Introduction

We do not attempt to be original and we begin with a very standard example of a recursively defined type. A list of objects of type ρ is either a "`nil`" or has the form $\langle a, \ell \rangle$, where ℓ is a list. Assuming $\mathtt{nil} : \mathbf{1}$, one may then postulate that the type *list* of our lists should satisfy the equation:

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

This is a fixpoint equation for an operator on types that turns a type α into $\mathcal{F}(\alpha) = \mathbf{1} + (\rho \times \alpha)$. An important feature of this definition is that the operator $\mathcal{F}$ is induced by a certain choice of constructors. Thus, it is monotone (because constructors build new objects from the presently available ones), and 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 $\mathcal{F}$. Syntactically, the monotonicity of this construction means that the variable α occurs in $\mathcal{F}(\alpha)$ only positively.

There are calculi (usually Curry-style) of fixpoint types, where such fixpoint equations are taken literally, cf. [6, 27]. The requirement that actual solutions of such equations really exist is a very strong postulate though, and obviously often difficult to satisfy. For instance, there is clearly no solution for the list equation within the polymorphic lambda calculus.

However, what is needed in most cases is a weaker property. It is normally sufficient to know that object of type $\mathbf{1} + (\rho \times \mathbf{list})$ can be properly interpreted as an object of type **list**. Thus we need an introduction operator $\mathbf{in} : \mathbf{1} + (\rho \times \mathbf{list}) \to \mathbf{list}$. In the general setting, we require such an operator

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

for each (positive) recursive type $\mu\alpha.\tau$. Then we need an elimination operator (elimination rule), and a reduction rule. These describe the routine of using objects (proofs) of type $\mu\alpha.\tau$ as inputs (assumptions) in computations. There are various elimination rules occurring in the literature. Our goal is to understand if and how the choice of an elimination rule may influence the properties of a recursive type system.

In particular, we address the question of the interpretability of recursive types within the polymorphic lambda calculus $\boldsymbol{\lambda 2}$, known also as System **F**. It has long been known, see e.g. [4, 12, 29] that various recursively defined data types can be defined within $\boldsymbol{\lambda 2}$ with help of beta reductions. A generic translation (for a particular choice of the eliminator) was probably first given in [10] (see also [26]), but was implicit in [24, 25] and in [16]. That is, for every $\mu\alpha.\tau$ there is a polymorphic type μ with an introduction combinator $\mathbf{in}_{\mu\alpha.\tau} : \tau[\mu/\alpha] \to \tau$, and an eliminator (of an appropriate type), both definable in $\boldsymbol{\lambda 2}$ in such a way that all reductions are preserved. Such a translation applies for instance to the system of recursive types for which strong normalization was proved in Mendler's journal article [21]. (Thus, the strong normalization result of [21] may also be obtained by translation to $\boldsymbol{\lambda 2}$.)

Such interpretation results give the background for the following general claim: *"Recursive types can be implemented in System* **F***"*, which seems to be a folklore-style common knowledge. This common knowledge does not take into account the fact that recursive types can be defined in

*Partly supported by KBN Grant 8 T11C 035 14, NATO Grant HTECH.LG960875, and by the POLONIUM project 981160-746/R98.

To appear in *Proc. 4th ICFP*, Paris, France, September 1999.

various ways.

In this paper we explain why this opinion is highly questionable and most likely not true.

As will be seen in Section 2, the common choices of elimination and reduction rules for recursive types can be classified into two categories: the "iterative" definitions and the "recursive" ones. An example of the latter is the system discussed in [20] — the preliminary version of Mendler's work [21]. The iterative variants can be simulated within $\boldsymbol{\lambda 2}$, but we conjecture that the recursive ones can not.

The names "iterative" and "recursive" were suggested by the difference between *iteration* and *recursion* over integers. This difference was studied by Parigot [22, 23]. Recall that Gödel's System **T** is typically defined with a *recursor*

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

for every type σ. The reduction rules for the recursor are as follows.

$$\mathbf{R}_\sigma MN\underline{0} \Rightarrow N;$$
$$\mathbf{R}_\sigma MN\underline{k+1} \Rightarrow M\underline{k}(\mathbf{R}_\sigma MN\underline{k}).$$

An alternative is to choose the *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 difference is whether a direct access to the recursion argument is permitted or not. For this reason, iteration is also called sometimes "monotonic recursion", see [17].

In a sense, these variants of System **T** are equivalent. In particular, both systems represent exactly the integer functions that are provably recursive in Peano Arithmetic. 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. In fact, Parigot [22] proves that every representation of the predecessor function for Church numerals ("iterative" integers) must be of at least linear time complexity. It follows that **R**-reductions can not be simulated by **It**-reductions in constant time.

An interesting insight into the difference between recursion and iteration can be obtained from the propositions-as-types analogy, known also as the Curry-Howard Isomorphism. The idea is that types of the eliminators (iterator and recursor) are obtained from the induction axiom (seen as a dependent type of $\lambda\mathbf{P}$) by means of a dependency-erasing function (see e.g. [13] or [3] for a formal definition). First note that the induction axiom scheme for integers may be stated in each of the following forms:

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

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

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

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 quantification should be turned into implication, but an unbounded quantification corresponds to a trivial assumption and can be ignored. Thus, formulas (a) and (b) erase respectively to the types of iterator and recursor, while (c) erases to the type of Church numerals. As discussed in [16], also the type of so called Parigot integers (a solution of the fixpoint equation $\mathbf{int} = \forall\beta((\mathbf{int} \to \beta) \to \beta \to \beta)$) can be seen as an erasure of an appropriate form of the induction axiom.[1]

The analogy here is deeper. Namely, the reduction rules for iterators and recursors correspond to proof reduction rules that one should postulate for proofs by induction (an application of induction is represented by a proof constant which type is the induction axiom). We leave a further discussion of this issue to the final version of this paper.

It is shown in Section 2 that the truly "recursive" variants of recursive types are beta-eta equivalent to the following simple choice. We just postulate that the introduction operator, denoted by **Fold** in this case, is an embedding, i.e., it is left-invertible. Thus, the elimination operator **Unfold** is a retraction of $\mu\alpha.\sigma$ onto $\sigma[\mu\alpha.\sigma/\alpha]$, and we have the following reduction rule:

$$\mathbf{Unfold}(\mathbf{Fold}M) \Rightarrow M.$$

In this paper we show that this calculus, which we call $\boldsymbol{\lambda 2}\mathbf{U}$, cannot be implemented within $\boldsymbol{\lambda 2}$ by means of beta reductions. More precisely, we show the following: if $\alpha \in FV(\sigma)$, but $\sigma \neq \alpha$, then no type of $\boldsymbol{\lambda 2}$ can be used to simulate $\mu\alpha.\sigma$, because the required operators **Fold** and **Unfold** do not exist. That is, for all τ, the type $\sigma[\tau/\alpha]$ cannot be *embedded* into τ. (Type ρ can be *embedded* into τ, written $\rho \leq \tau$, iff there are two terms $F : \rho \to \tau$ and $G : \tau \to \rho$, such that $G \circ F =_\beta \mathbf{I}$.)

In order to prove this result (Corollary 17), we show a necessary condition on two types ρ and τ in order that ρ be embeddable into τ. This condition is that type ρ must be of depth not exceeding the depth of τ.

This necessary condition is not sufficient, and we leave the question of finding one as an open problem, as well as finding a necessary and sufficient condition for the existence of a beta-eta embedding. Let us note that there exists a characterization of beta-eta *isomorphisms* for polymorphic types, given by Di Cosmo [9].

In case of simple types, we can give the following characterization of embeddability (cf. a sufficient condition in Bruce and Longo [5]):

Proposition 1 *Type ρ can be embedded into τ (by means of beta reductions) if and only if $\tau = \sigma_1 \to \cdots \to \sigma_n \to \rho$, where every σ_i can be intuitionistically derived from τ.*

Proof: Follows from the fact, that if $G : \tau \to \rho$ is right-invertible, then it must be of the form $G \equiv \lambda y.yA_1 \ldots A_n$ (see [8, 18]), with $y : \tau$. ∎

A good example for the simply typed case is $\beta \leq \sigma \to \beta$, where σ stands for the type $((\alpha \to \beta) \to \alpha) \to \alpha$. Indeed, we have $G \circ F =_\beta \mathbf{I}$, for $F \equiv \lambda x^\beta \lambda y^\sigma.x$ and $G \equiv \lambda x^{\sigma\to\beta}.x(\lambda y^{(\alpha\to\beta)\to\alpha}.y(\lambda z^\alpha.x(\lambda u^{(\alpha\to\beta)\to\alpha}.z)))$.

Note however that Proposition 1 does not hold in the polymorphic case. For instance, we have $\beta \leq \tau$, where $\tau = \forall\alpha((\beta \to \alpha) \to \alpha)$. Indeed, take $F^{\beta\to\tau} \equiv \lambda x^\beta \Lambda\alpha\lambda v^{\beta\to\alpha}.vx$ and $G^{\tau\to\beta} \equiv \lambda f^\tau.f\beta\mathbf{I}$.

[1] A by-product of the results of this paper is that Parigot integers cannot be defined within System **F**.

Another example shows that $\rho \leq \tau$ does not imply that ρ is a subexpression of τ. Take $\rho = \varphi \to \psi$ and $\tau = \forall\beta[\forall\alpha((\alpha \to \psi) \to \forall\gamma(\gamma \to \gamma) \to \alpha \to \beta) \to (\varphi \to \beta)]$. Then $\lambda x^{\rho}\Lambda\beta\lambda v^{\forall\alpha(\ldots)}.v\varphi x\mathbf{I} : \rho \to \tau$ and $\lambda f^{\tau}.f\psi(\Lambda\alpha(\lambda z^{\alpha\to\psi}\lambda j^{\forall\gamma(\gamma\to\gamma)}.j(\alpha \to \psi)z)) : \tau \to \rho$ make an embedding-projection pair.

The techniques used in the proof of Corollary 17 are borrowed from Geometry of Interaction (cf. [11, 7]) and the theory of optimal reductions, see [1] and Chapter 6 of [2]. Generally speaking, we represent the reduction of $G(Fx)$ to x^{ρ} by means of a regular path in the term $G(Fx)$. The information about every leaf of type ρ of x is transmitted along this path in such a way that it must leave a trace of itself in the type of Fx. In order to accomodate all these traces, this type must be at least as deep as ρ.

The results of Section 2 are essentially from [26]. Some translations mentioned there were obtained independently by Geuvers [10], and later by Ralph Matthes [19]. Corollary 17 was first reported in [28].

2 Iteration and recursion

We extend the syntax of polymorphic types with the construction $\mu\alpha.\tau$, where α is a type variable and τ is a type such that α occurs in τ only positively. The variable α is considered bound in $\mu\alpha.\tau$. In what follows we describe several ways of extending the syntax of (terms of) $\boldsymbol{\lambda 2}$ to handle types of the above form.

2.1 System $\boldsymbol{\lambda 2I}$

One possible way of defining an elimination operator for $\mu\alpha.\tau$ is based on the following idea: To define a function of type $\mu\alpha.\tau \to \beta$ it should be enough to know how to transform an arbitrary function of type $\alpha \to \beta$ into a function of type $\tau \to \beta$. The system $\boldsymbol{\lambda 2I}$ (taken essentially from [21]) is an extension of $\boldsymbol{\lambda 2}$ with two new term constants:

$$\mathbf{in}_{\mu\alpha.\tau} : \tau[\mu/\alpha] \to \mu\alpha.\tau;$$
$$\mathbb{I}_{\mu\alpha.\tau} : \forall\beta(\forall\alpha((\alpha \to \beta) \to (\tau \to \beta)) \to \mu\alpha.\tau \to \beta),$$

and the following reduction scheme:[2]

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

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

To explain this definition let us informally think of type $\mu\alpha.\tau$ as of a union of types μ^k where μ^0 is empty and μ^{k+1} is the image of $\tau(\mu^k)$ under $\mathbf{in}$. We define a function $\mathbb{I}\sigma A$ for the argument $\mathbf{in}\, x$ using the transformation[3] $A : \forall\alpha((\alpha \to \sigma) \to \tau(\alpha) \to \sigma)$. We proceed by induction. If the argument $\mathbf{in}\, x$ is in μ^{k+1} then we assume that our function has already been defined for arguments in μ^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 $x \in \tau(\mu^k)$.

Note that the transformation A is generic in α. When using it for an argument x in $\tau(\mu^k)$ only the "top level" information about x can be used. Thus, there is no direct access to the recursion argument in very much the same way as in the case for iteration over integers.

We leave it to the reader to observe that in system $\boldsymbol{\lambda 2I}$ one can define iterators over integers, if we take e.g. $\mathbf{int} = \mu\alpha.\mathbf{1} + (\mathbf{1} \times \alpha)$.

2.2 Logical interpretation

We have already observed that types of iterators and recursors over integers can be obtained from the induction axiom schemes by means of dependency-erasing operators. In this subsection we explain (informally) how this phenomenon, first described by Daniel Leivant [15, 16] and also by J.-L. Krivine and M. Parigot [14, 23], can be seen in the general case of recursive types in $\boldsymbol{\lambda 2I}$.

Let us present the types of $\mathbf{in}$ and $\mathbb{I}$ in the form of the following rules, where μ 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:

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

If we think of τ as of a monotone operator in a complete lattice then the above conditions guarantee that μ is the least fixpoint of τ. 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 relation variable occurring only positively in Φ. 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.

In fact, we can also see the reduction rule as an erasure of a proof reduction rule.

2.3 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 ($\mathbb{I}$):

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

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 $\boldsymbol{\lambda 2J}$, following Leivant [16], 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,

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

[3] The notation $\tau(\alpha)$ is meant to suggest a possible occurrence of α in τ. We also write $\tau(\sigma)$ for $\tau[\sigma/\alpha]$.

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 τ is a monotone operator. Thus, we need a family of combinators

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

for all types φ and ψ. (We may skip sub(super)scripts if clear from the context.) Then we can define the reduction rule as follows.

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

The combinators $\mathcal{M}^{\tau}_{\varphi,\psi}$ correspond to proofs that $\varphi \subseteq \psi$ implies $\tau(\varphi) \subseteq \tau(\psi)$. We assume that $\alpha \notin FV(\varphi) \cup FV(\psi)$. We define these combinators explicitly by induction with respect to a measure $p_\alpha(\tau)$, where the subscript α indicates the variable to be bound by μ.

- If α is not free in τ, then $p_\alpha(\tau) = 0$. Otherwise:
- $p_\alpha(\alpha) = 1$;
- $p_\alpha(\sigma \to \rho) = 1 + \max(p_\alpha(\sigma), p_\alpha(\rho))$;
- $p_\alpha(\forall\beta\sigma(\beta,\alpha)) = 1 + p_\alpha(\sigma(\beta,\alpha))$;
- $p_\alpha(\mu\beta.\sigma(\beta,\alpha)) = 1 + \max(p_\alpha(\sigma(\beta,\alpha)), p_\beta(\sigma(\beta,\alpha)))$[4].

To proceed inductively we have to define also dual combinators $\mathcal{A}^{\tau}_{\varphi,\psi} : (\varphi \to \psi) \to \tau(\psi) \to \tau(\varphi)$, for types τ with only negative occurrences of α.

1) If α does not occur free in τ, then we define
$\mathcal{M}^{\tau}_{\varphi,\psi} \equiv \mathcal{A}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}.\mathbf{I}_\tau$.

2) $\mathcal{M}^{\alpha}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}.z$.

3) Let $\tau = \sigma \to \rho$. We define
$\mathcal{M}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\sigma(\varphi)\to\rho(\varphi)}\, x^{\sigma(\psi)}.\mathcal{M}^{\rho}_{\varphi,\psi} z(y(\mathcal{A}^{\sigma}_{\varphi,\psi} zx))$,
$\mathcal{A}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\sigma(\psi)\to\rho(\psi)}\, x^{\sigma(\varphi)}.\mathcal{A}^{\rho}_{\varphi,\psi} z(y(\mathcal{M}^{\sigma}_{\varphi,\psi} zx))$.

4) For $\tau = \forall\beta.\sigma(\beta,\alpha)$ we define
$\mathcal{M}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\forall\beta.\sigma(\beta,\varphi)}\Lambda\beta.\mathcal{M}^{\sigma(\beta,\alpha)}_{\varphi,\psi} z(y\beta)$, and
$\mathcal{A}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\forall\beta.\sigma(\beta,\psi)}\Lambda\beta.\mathcal{A}^{\sigma(\beta,\alpha)}_{\varphi,\psi} z(y\beta)$.

5) Assume $\tau = \mu\beta.\sigma(\beta,\alpha)$. Let $\mu_1 = \mu\beta.\sigma(\beta,\varphi)$, and $\mu_2 = \mu\beta.\sigma(\beta,\psi)$. We take:
$\mathcal{M}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\mu_1}.\mathbb{J}_{\mu_1}(\mu_2)(\lambda x^{\sigma(\mu_2,\varphi)}\mathbf{in}_{\mu_2}(\mathcal{M}^{\sigma(\mu_2,\alpha)}_{\varphi,\psi} zx))y$,
$\mathcal{A}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\mu_2}.\mathbb{J}_{\mu_2}(\mu_1)(\lambda x^{\sigma(\mu_1,\psi)}.\mathbf{in}_{\mu_1}(\mathcal{A}^{\sigma(\mu_1,\alpha)}_{\varphi,\psi} zx))y$.

Note that in each case the parameter p_α decreases and thus our definition is well-founded.

[4]For **λ2J** this can be simplified to $p_\alpha(\mu\beta.\sigma(\beta,\alpha)) = 1 + p_\alpha(\sigma(\beta,\alpha))$, but the general case is needed for **λ2Rec** below.

2.4 Reductions to λ2

Of course, all the above discussed systems are extensions of **λ2**. However, none of them is essential in that the new types and operators are definable within **λ2**. The polymorphic type representing $\mu\alpha.\tau$ in **λ2J** can be easily guessed from the Curry-Howard interpretation of Section 2.2. 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

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

Proposition 2 *The above intepretation of* **λ2J** *in* **λ2** *is correct, i.e., we have* $\mathbb{J}\,\sigma\, M\,(\mathbf{in}\, N) \twoheadrightarrow_\beta M\,(\mathcal{M}^{\tau}_{\mu,\sigma}\,(\mathbb{J}\,\sigma\, M)\, N)$, *for all* M, N *of appropriate types.*

Proof: An easy calculation. ∎

For system **λ2I**, 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

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

Proposition 3 *The above intepretation of* **λ2I** *in* **λ2** *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.*

Proof: Easy. ∎

The representation in **λ2** of inductive data types like natural numbers or lists can be regarded as an encoding of least fixpoints of monotonic operators. Dual notions of greatest fixpoints, corresponding to types with coiterators, can be similarly defined within **λ2**. Wraith [29] seems to be the first who encoded in **λ2** positive coinductive types with coiterators.

2.5 Fixpoint types with recursors

We classify systems **λ2I** and **λ2J** as systems with "iterators", because of the restricted use of recursion argument. In this section we present two extensions of **λ2** with true "recursors", and we prove that they are beta-eta equivalent to the system **λ2U**, presented in the Introduction.

Mendler [20] introduced the system (which we call **λ2R**), similar to **λ2I**, but with an eliminator $\mathbb{R}_{\mu\alpha.\tau}$ of type

$$\forall\beta(\forall\gamma((\gamma \to \mu) \to (\gamma \to \beta) \to \tau(\gamma) \to \beta) \to \mu \to \beta),$$

and with the following reduction scheme:

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

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

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

Recall that the premise of an analogous rule for $\mathbb{I}$ expresses the closure of β under the operator τ. The premise of the above rule is weaker: β is closed under τ, provided the argument is already in μ. Operationally, the difference is that the operator $A\mu^k : (\mu^k \rightarrow \sigma) \rightarrow \tau(\mu^k) \rightarrow \sigma$ occurring in **λ2I** reductions cannot rely on any specific knowledge about the type μ and the argument $x : \tau(\mu^k)$, because it must be generic in α. 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 $\forall\alpha((\alpha \rightarrow \mu^k) \rightarrow (\alpha \rightarrow \sigma) \rightarrow \tau(\alpha) \rightarrow \sigma)$. 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 σ may even be constructed with no use of the second argument.

The second-order interpretation of this type:

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

can be further simplified to

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

and this gives a hint for a recursive counterpart of **λ2J**, namely the system **λ2Rec** of [26]. The recursor and the reduction scheme for **λ2Rec** are as follows:[5]

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

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

The symbol $\mathcal{M}$ (more adequately $\mathcal{M}^{\tau}_{\mu,\mu\times\sigma}$) stands here again for an appropriately defined "lifting" combinator. The definition of $\mathcal{M}^{\tau}_{\varphi,\psi}$ and of $\mathcal{A}^{\tau}_{\varphi,\psi}$ is similar as in the case of **λ2J**. More precisely, cases (1) – (4) are the same, and case (5) is modified as follows:

5) For $\tau = \mu\beta.\sigma(\beta,\alpha)$, define $\mu_1 = \mu\beta.\sigma(\beta,\varphi)$, and $\mu_2 = \mu\beta.\sigma(\beta,\psi)$. Then

$$\mathcal{M}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\rightarrow\psi}\, y^{\mu_1}.\mathbf{Rec}_{\mu_1}(\mu_2)(\lambda x^{\sigma(\mu_1\times\mu_2,\varphi)}.\mathbf{in}_{\mu_2}(\mathcal{M}^{\sigma(\mu_2,\alpha)}_{\varphi,\psi} z(\mathcal{M}^{\sigma(\beta,\varphi)}_{\mu_1\times\mu_2,\mu_2}\mathsf{snd}\, x)))y,$$

$$\mathcal{A}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\rightarrow\psi}\, y^{\mu_2}.\mathbf{Rec}_{\mu_2}(\mu_1)(\lambda x^{\sigma(\mu_2\times\mu_1,\psi)}.\mathbf{in}_{\mu_1}(\mathcal{A}^{\sigma(\mu_1,\alpha)}_{\varphi,\psi} z(\mathcal{A}^{\sigma(\beta,\psi)}_{\mu_2\times\mu_1,\mu_1}\mathsf{snd}\, x)))y,$$

where snd stands for the second projection. Notice, that in the definition of $\mathcal{M}^{\tau(\alpha)}$ we use combinators $\mathcal{M}^{\sigma}$ defined with respect to α and to β.

We say that a type $\mu\alpha.\tau$ is *non-interleaved* iff the variable α does not occur free in τ in the scope of any other occurrence of the operator μ.

Lemma 4 *Let $\mu\alpha.\tau$ be non-interleaved. Then*

$$(a)\quad \mathcal{M}^{\tau}_{\varphi,\varphi}(\lambda x^{\varphi}.x) \twoheadrightarrow_{\beta\eta} \lambda y^{\tau(\varphi)}.y;$$
$$(b)\quad \mathcal{A}^{\tau}_{\varphi,\varphi}(\lambda x^{\varphi}.x) \twoheadrightarrow_{\beta\eta} \lambda y^{\tau(\varphi)}.y.$$

[5] Remember that product type (definable in **λ2**) corresponds to conjunction.

Proof: The lemma is proved by straightforward induction on the structure of τ. ∎

The difference between **Rec** and $\mathbb{J}$ is similar to that between $\mathbb{R}$ and $\mathbb{I}$ and between the recursor **R** of Gödel's System **T** and the iterator **It**. In each case 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 $\forall x(P(x) \rightarrow R(x))$ or $\forall x(L(x) \rightarrow P(x) \rightarrow R(x))$.)

Now we show that systems **λ2R**, **λ2Rec** and **λ2U** are interdefinable. Propositions 5 and 6 are slightly modified versions of results from [26]. A version of Proposition 6 restricted to non-nested types, and a similar reduction from **λ2Rec** to **λ2R** were already known to Geuvers [10]. Related results were independently obtained by Ralph Matthes, who in his PhD Thesis [19] studies monotone inductive types.

Proposition 5 *Systems **λ2R** and **λ2U** are definable in each other, by means of beta-eta reductions.*

Proof: In order to avoid confusion, we use below the notation μ^{R} and μ^{U} to denote fixpoints in **λ2R** and **λ2U**, respectively. We interprete **λ2U** within **λ2R** by replacing every fixpoint type $\mu = \mu^{U}\alpha.\tau(\alpha)$ by another type $\underline{\mu}$. In this way, every type ρ of **λ2U** is translated to a type $\underline{\rho}$ of **λ2R**.

We define $\underline{\mu}$ as $\mu^{R}\alpha.\sigma(\alpha)$, where

$$\sigma = \forall\beta((\alpha \rightarrow \beta) \rightarrow \underline{\tau}(\beta)).$$

An easy induction shows that in fact all fixpoints occurring in types of the form $\underline{\rho}$ are non-interleaved. Then we represent the operators of **λ2U** as follows:

$$\mathbf{Fold}_{\mu} \equiv \lambda x^{\underline{\tau}(\underline{\mu})}.\mathbf{in}_{\underline{\mu}}(\Lambda\beta\lambda y^{\underline{\mu}\rightarrow\beta}.\mathcal{M}^{\underline{\tau}(\alpha)}_{\underline{\mu},\beta}\, y\, x)$$

$$\mathbf{Unfold}_{\mu} \equiv \mathbb{R}_{\underline{\mu}}(\underline{\tau}(\underline{\mu}))(\Lambda\gamma\lambda x_1^{\gamma\rightarrow\underline{\mu}}\lambda x_2^{\gamma\rightarrow\underline{\tau}(\underline{\mu})}\lambda x_3^{\sigma(\gamma)}.x_3\,\underline{\mu}\,x_1).$$

The meaning of $\mathcal{M}$ is exactly as before (observe that due to non-interleaving we do not need clause (5) of the definition at all.) In the verification that $\mathbf{Unfold}(\mathbf{Fold}(x^{\underline{\tau}(\underline{\mu})})) \twoheadrightarrow_{\mathrm{R}} x$ we use Lemma 4(a). (Note that $\underline{\mu}$ should be considered a fixpoint of $\alpha \mapsto \underline{\tau}(\alpha)$ rather than of $\alpha \mapsto \tau(\alpha)$.)

For the interpretation of **λ2R** in **λ2U**, consider again a fixpoint type $\mu = \mu^{\mathrm{R}}\alpha.\tau$. We define $\underline{\mu}$ as $\mu^{\mathrm{U}}\alpha.\sigma$, where $\sigma = \forall\beta(\forall\gamma((\gamma \rightarrow \alpha) \rightarrow (\gamma \rightarrow \beta) \rightarrow \underline{\tau}(\gamma) \rightarrow \beta) \rightarrow \beta)$. In the following definitions **Fold**, **Unfold** stand for $\mathbf{Fold}_{\mu^{\mathrm{U}}\alpha.\sigma}$ and $\mathbf{Unfold}_{\mu^{\mathrm{U}}\alpha.\sigma}$, respectively. We take:

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

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

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

Proposition 6 ([10, 26]) *Systems **λ2Rec** and **λ2U** are definable in each other, by means of beta-eta reductions.*

Proof: To distinguish between fixpoints in the different systems we again use the notation μ^{U} for the fixpoint operator in **λ2U**, and we write μ^{Rec} whenever we mean the fixpoint of **λ2Rec**. We begin with the translation from **λ2U** to **λ2Rec**.

We represent fixpoints as follows: $\underline{\mu} \equiv \underline{\mu^U \alpha.\tau(\alpha)} \equiv \mu^{Rec}\alpha.\sigma(\alpha)$, where $\sigma = \forall\beta((\alpha \to \beta) \to \underline{\tau}(\beta))$. Then we define

$$\mathbf{Fold}_{\mu^U\alpha.\tau} \equiv \lambda x^{\underline{\tau}(\underline{\mu})}.\mathbf{in}_{\underline{\mu}}\,\Lambda\beta\lambda y^{\underline{\mu}\to\beta}.\mathcal{M}^{\underline{\tau}(\alpha)}_{\underline{\mu},\beta}\; y\, x$$

$$\mathbf{Unfold}_{\mu^U\alpha.\tau} \equiv \mathbf{Rec}_{\underline{\mu}}\,(\underline{\tau}(\underline{\mu}))\,(\lambda v^{\sigma(\underline{\mu}\times\underline{\tau}(\underline{\mu}))}.v\,\underline{\mu}\,\mathsf{fst}).$$

As in the proof of Lemma 5, the types obtained by the translation may only involve non-interleaved fixpoints. With help of Lemma 4(a) we can now verify the reduction $\mathbf{Unfold}(\mathbf{Fold}\,x) \twoheadrightarrow_{Rec} x$. (For this, observe that $\mathcal{M}^{\sigma(\alpha)}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\lambda y^{\forall\beta((\varphi\to\psi)\to\underline{\tau}(\beta))}\Lambda\beta\lambda x^{\psi\to\beta}.y\,\beta\,(\lambda v^{\varphi}.x\,(z\,v)).$)

For the translation of **λ2Rec** to **λ2U**, we define the type $\underline{\mu} \equiv \underline{\mu^{Rec}\alpha.\tau(\alpha)}$ as $\mu^U\alpha.\sigma(\alpha)$, where

$$\sigma = \forall\beta((\forall\gamma((\gamma \to \alpha \times \beta) \to \underline{\tau}(\gamma) \to \beta) \to \beta).$$

The operators are as follows:

$$\mathbf{Rec}_\mu \equiv \Lambda\beta\lambda y^{\underline{\tau}(\underline{\mu}\times\beta)\to\beta}\lambda z^{\underline{\mu}}.\\ \quad \mathbf{Unfold}_{\underline{\mu}}\,z\,\beta\,\Lambda\gamma\lambda v^{\gamma\to\underline{\mu}\times\beta}\lambda w^{\underline{\tau}(\gamma)}.y\,(\mathcal{M}^{\underline{\tau}(\alpha)}_{\gamma,\underline{\mu}\times\beta}\,v\,w);$$

$$\mathbf{in}_\mu \equiv \lambda x^{\underline{\tau}(\underline{\mu})}.\mathbf{Fold}_{\underline{\mu}}\,(\Lambda\beta\lambda y^{\forall\gamma((\gamma\to\underline{\mu}\times\beta)\to\underline{\tau}(\gamma)\to\beta)}.y\,(\underline{\mu}\times\beta)\\ \quad \mathbf{I}_{\underline{\mu}\times\beta}\,(\mathcal{M}^{\underline{\tau}(\alpha)}_{\underline{\mu},\underline{\mu}\times\beta}(\lambda z^{\underline{\mu}}.\langle z, \underline{\mathbf{Rec}_{\underline{\mu}}}\,\beta\,(y\,(\underline{\mu}\times\beta)\,\mathbf{I}_{\underline{\mu}\times\beta})\,z\rangle)\,x)).$$

Using Lemma 4(a), we check the desired property

$$\mathbf{Rec}_\mu\,\sigma M(\mathbf{in}N) \twoheadrightarrow M(\mathcal{M}^{\tau(\alpha)}_{\mu,\mu\times\sigma}(\lambda z^{\mu}.\langle z, \mathbf{Rec}_\mu\,\sigma M z\rangle)N).$$

∎

3 Paths in terms

The set of all (finite) words over an alphabet A is denoted by A^*, with the empty word denoted by ε. The notation $w \subseteq v$ means that w is a prefix of v.

A *binary tree* is understood here as a set of words over the alphabet $\{p, q\}$, closed under prefixes, where p means "left" and q means "right". For a tree t, the set of all leaves of t is denoted by $leaves(t)$.

Every type τ can be seen as a binary tree where leaves are labeled by type variables, and internal nodes are labeled by arrows and possibly quantifiers. By $\overline{\tau}$ we denote the binary tree corresponding to τ, seen as a set of words over $\{p, q\}$. (Thus, labels are ignored.)

We represent terms of **λ2** as directed graphs in the spirit of [1]. We do not use all the edge labels of [1], but we work with terms of **λ2**, and we must add two more kinds of nodes to account for polymorphism.

Every term M is represented as a graph $G(M)$ with a distinguished *top* node (drawn at the top) and some *variable edges* (drawn at the bottom) connecting free variables of M with the outside world. There is one variable edge for each free variable of M. The graph $G(M)$ has labeled directed edges and unlabeled undirected edges.

Graphs representing different subterms of M (two occurrences of the same subterm are taken as two different subterms) have different top nodes, and we will often identify each subterm with the top node of the corresponding subgraph. The definition of $G(M)$ is by induction with respect to M.

Figure 1: Variable

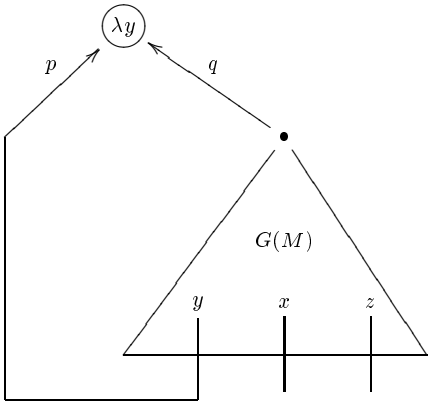

Figure 2: Abstraction $\lambda y.M$

Variable: The graph for a variable y consists of a single node (called a v-*node*) and a single variable edge. Of course, the v-node is the top node of the graph.

Abstraction: The graph $G(\lambda y.M)$, where $y \in FV(M)$ is constructed from $G(M)$ by adding a new top node (called λ-*node*), which becomes the target node for the variable edge of M corresponding to y. This edge is labeled by p. We also add another edge, labeled q, from the top of M to the new λ-node. Both these edges are directed towards λ. Figure 2 is drawn for $FV(M) = \{x, y, z\}$. In case $y \notin FV(M)$, we must add a new initial node for the p-labeled arrow entering λ. This arrow will never be used though.

Application: The graph $G(MN)$ is built up from $G(M)$ and $G(N)$ as follows. We add a new top node (called a c-*node*) and a new internal node (@-*node*) and an arrow labeled q from the c-node to the @-node. Then we add a p-labeled arrow from N to @ and an unlabeled edge from M to @. For each variable z free in both M and N, we add a *fan node* which joins the two variable edges into one.

Type abstraction: $G(\Lambda\alpha.M)$ is obtained from $G(M)$ by adding a new top node (a $\Lambda\alpha$-*node*) and an unlabeled edge between M and $\Lambda\alpha$.

Type application: Also $G(M\tau)$ is obtained in a similar way: by adding a new top node ($@\tau$-*node*) with an unlabeled edge from M.

A *path* in $G(M)$, is a sequence of adjacent edges, such that there is no backtracking, i.e.,

- No edge is taken twice in a row;
- Two edges connecting a fan node with two occurrences of the same variable may not directly follow one another.

Each node, except @-nodes and fans, is a top node of a subterm, and is associated the type of that subterm. A path in $G(M)$ is identified with a sequence of such typed nodes (i.e.,

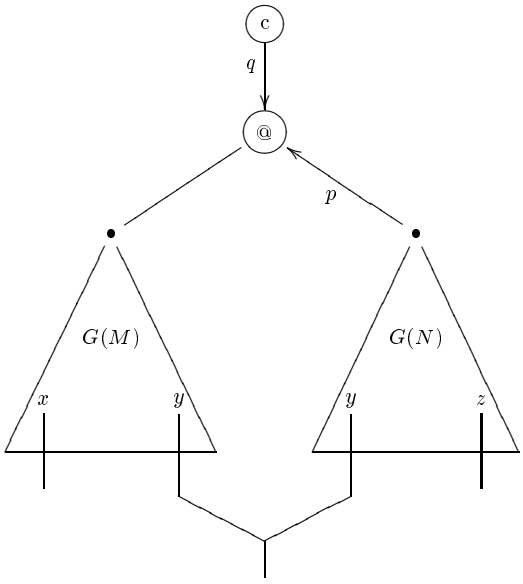

Figure 3: Application MN

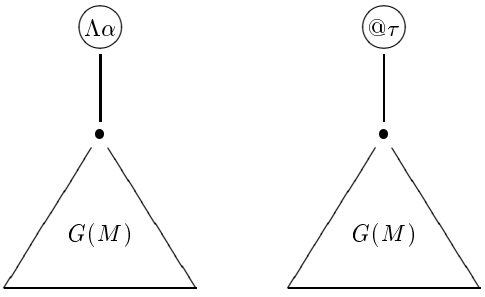

Figure 4: Type abstraction $\Lambda\alpha M$ and type application $M\tau$

we ignore the others). The meaning of the expression "adjacent nodes" is relaxed appropriately, thus e.g., two typed nodes with an @ in between are considered adjacent.

3.1 Global computations

In this section we study the overall ("global") behaviour of paths in terms. Essentially, this is a simplification of the Dynamic Algebra of [11, 7]. Consider an arbitrary fixed term M. A pair (N^τ, π), consisting of a typed node N of $G(M)$, and a word $\pi \in \{p, q\}^*$ (a push-down store), is called a *global configuration*. We think of global configurations as of configurations of a push-down store (pds) automaton. The automaton moves along a path Π in $G(M)$, and each time it traverses a labeled edge forward or backward, it performes a pds operation. Whenever we follow an arrow labeled p, we push p on the pds, and in order to traverse such an arrow backward, we must pop p from the pds. We proceed in an analogous way when we use edges labeled by q, forward or backward. In particular, in order to take a p-arrow backward, the current configuration must be of the form $(M, p \cdot \pi)$, and respectively for q-arrows. Otherwise the automaton cannot move.

Suppose that we have a path Π beginning at a node M and ending at node M'. If our automaton has a correct computation along Π, beginning at the global configuration (M, π) and ending at the global configuration (M', π'), then we write $\Pi(\pi) = \pi'$. Otherwise $\Pi(\pi)$ is undefined.

One easy but important observation about our automaton is as follows.

Proposition 7 *Assume that a term M is simply typed. Let a variable x of type ρ be free in M, and suppose that the automaton has a correct computation along a path Π beginning with the configuration (x, ε). Then every configuration (N^τ, π) reached during this computation has the property that ρ is a subexpression of τ, corresponding to the subtree of $\overline{\tau}$, rooted at π.*

We think of ρ as being passed from one type occurring along Π to another, with the pds storing the relative position of ρ in the currently visited type.

Let us observe at the end how a beta reduction step applied to a term Q may affect a path Π in $G(Q)$. Each of the two kinds of reductions: type reduction and term reduction can be seen as removing certain portions of the path Π and duplicating others. A type reduction

$$(\Lambda\alpha.N)^{\forall\,\alpha\,\tau}\sigma \to_\beta N[\sigma/\alpha]^{\tau[\sigma/\alpha]},$$

results in removing two adjacent nodes, the $@\sigma$-node and the Λ-node. The *deformation* of the path Π amounts to gluing together the cut points.

A term reduction $(\lambda y.A)B \to A[B/y]$ (see Figures 5 and 6) results in removing paths:

- from the c-node via q-arrow to the λ-node and via q-arrow backward to the top of A;
- from a variable node in A via p-arrow to the λ-node and then to top of B via p-arrow backward,

and duplicating portions of the path within B. A path that involves one or more of these pieces can be re-constructed in $G(Q_1)$ again by gluing together the ends of removed segments.

4 The polymorphic case

Let us turn back to the idea of a pds storing the information about a relative position of the initial type within the current type. This idea, expressed by Proposition 7 does not extend to the polymorphic case. The problem is caused by type instantiation. Suppose we are in a configuration of the form $(M\tau : \sigma[\tau/\alpha], \pi)$, and the next node is $M : \forall\alpha.\sigma$. It may happen that the initial type ρ is contained in $\sigma[\tau/\alpha]$, but not in $\forall\alpha.\sigma$, because ρ is completely contained within τ. (This means that a prefix π' of π points in τ at an occurrence of α.) We can say that there is no place in $\forall\alpha.\sigma$ for the relative position of ρ. In fact, we can imagine that this relative position is *below* the leaf π' of our type, at a "virtual" node. In order to represent this virtual node we should use a pair of words (π', π''), where $\pi = \pi' \cdot \pi''$ rather than π alone. One can think of that as of placing a separator (comma) on the push-down store π. We of course must generalize this to permit as many separators as needed, i.e., to allow virtual nodes $(\pi_1, \pi_2, \ldots, \pi_n)$, where π_1 is a leaf of a current type τ.

This is unfortunately not the only complication. We have discussed only the case when the whole type ρ is being introduced by a type instantiation. In general it may happen that various portions of ρ are treated differently. For this reason we are forced to consider a separate "location" for each leaf of $\overline{\rho}$.

4.1 Wires and locations

Sequences of words over an alphabet A are notationally treated as words over the alphabet A^* (not A). For instance, if $s = (\pi_1, \pi_2 \ldots, \pi_n)$ then $head(s) = \pi_1$ and $tail(s) = (\pi_2, \ldots, \pi_n)$. If $s = (\pi_1, \ldots, \pi_n)$ and $s' = (\pi'_1, \ldots, \pi'_m)$ then we write $s \cdot s'$ for the concatenation $(\pi_1, \ldots, \pi_n, \pi'_1, \ldots, \pi'_m)$. The notation $s \circ s'$ stands for a form of concatenation that glues together the last word of s and the first word of s', i.e., $(\pi_1, \ldots, \pi_n) \circ (\pi'_1, \ldots, \pi'_m) = (\pi_1, \ldots, \pi_n \cdot \pi'_1, \ldots, \pi'_m)$. In particular, $(p) \circ s$, for $p \in A$, means adding p at the beginning of the first word in s. For $s = (\pi_1, \ldots, \pi_n)$, we write $cat(s)$ for the word $\pi_1 \cdots \pi_n$ over A.

A technically useful notions are a *quotient* and a *cut* of a sequence of words. If $\pi \in \{p, q\}^*$, and $s = (\pi \cdot \pi_1, \ldots, \pi_k)$, then the quotient s/π is defined as $(\pi_1, \ldots, \pi_k)$ and the cut $s{\downarrow}\pi$ is the sequence $(\pi, \pi_1, \ldots, \pi_n)$. Otherwise, both the cut $s{\downarrow}\pi$ and the quotient s/π are undefined.

In most of the material to follow, we assume that ρ is a certain fixed type. Every leaf w of $\overline{\rho}$ is called a *wire*.

A *location* of wire w is a sequence $s = (\pi_1, \ldots, \pi_n)$, where $\pi_i \in \{p, q\}^*$, and $cat(s) = v \cdot w$, for some v. That is, $s = (\pi_1, \ldots, \pi_k, \ldots, \pi_n)$ with $\pi_k = \pi \cdot \pi'$ and $w = cat(\pi', \pi_{k+1}, \ldots, \pi_n)$. In this case we say that k is the *depth* of s. A location $s = (\pi_1, \ldots, \pi_n)$ of depth k is *correct at type* τ iff π_1 is a leaf of $\overline{\tau}$.

4.2 Local computations

Consider an arbitrary fixed term Q. A pair (N^τ, s), consisting of a typed node N of $G(Q)$, and a location (of wire w), correct for τ, is called a *local configuration* (or simply a configuration) *for wire* w. The intended meaning is that our wire goes through the type of N at location s. As in the previous section, we should think here in terms of a pds automaton. The difference is that now we also have the commas used as separators on the pds. Formally, we define a transition relation $(N, s) \Rightarrow (N', s')$ between configurations (for a fixed wire w) associated to adjacent nodes, by the following rules:

Rule 1: (Generalization)

$$(M^\sigma, s) \Rightarrow ((\Lambda\alpha M)^{\forall\alpha\,\sigma}, s).$$

Rule 2: (Abstraction)

$$((\lambda y^\sigma.M^\tau)^{\sigma\to\tau}, (p) \circ s) \Rightarrow (y^\sigma, s);$$
$$((\lambda y^\sigma.M^\tau)^{\sigma\to\tau}, (q) \circ s) \Rightarrow (M^\tau, s).$$

The y^σ at the right-hand side stands for an arbitrary occurrence of y in Q. (Of course this can be seen as executing a "pop" instruction on $(p) \circ s$ (resp. on $(q) \circ s$).) If the current location is not of the desired form, the automaton cannot make this move. This happens also when s is not a location for w.

Rule 3: (Application) For the top node of an argument in an application, the conditions are analogous.

$$(M^{\sigma\to\tau}, (q) \circ s) \Rightarrow ((MN)^\tau, s);$$
$$(M^{\sigma\to\tau}, (p) \circ s) \Rightarrow (N^\sigma, s).$$

Rule 4: (Compression) Suppose that our path goes from an $@\tau$-node $N\tau$ (of type $\sigma[\tau/\alpha]$) to the top node of a subterm N of type $\forall\alpha\,\sigma$. Let α occur in σ at a leaf $\pi \subseteq head(s)$. Our relation is defined in this case as follows:

$$((N\tau)^{\sigma[\tau/\alpha]}, s) \Rightarrow (N^{\forall\alpha\,\sigma}, s{\downarrow}\pi).$$

Note that we can write s as $(\pi) \circ tail(s{\downarrow}\pi)$, and $s{\downarrow}\pi$ as $(\pi) \cdot tail(s{\downarrow}\pi)$.

Rule 5: (No Compression) Otherwise (when α does not occur in σ along s) we take

$$((N\tau)^{\sigma[\tau/\alpha]}, s) \Rightarrow (N^{\forall\alpha\,\sigma}, s).$$

Rule 6: If $(N, s) \Rightarrow (N', s')$ then also $(N', s') \Rightarrow (N, s)$. That is, all computation steps are reversible. Note that a reverse operation to that of Rules 2 and 3 corresponds to a "push" action on a pds. A reverse operation to compression is called a *decompression*.

Let now $\Pi = N_1, N_2, \ldots, N_m$ be a path in Q. A *computation* along Π is a sequence of configurations (N_i, s_i) such that $(N_i, s_i) \Rightarrow (N_{i+1}, s_{i+1})$ holds for all $i = 1, \ldots, m-1$.

The following lemma shows that local computations are indeed a refinement of global computations. The part of location above a wire is exactly the "global" part of the push-down store.

Lemma 8 *Consider a local computation for a wire w, along a path Π, beginning at a node $(N, (w))$ and ending at (M, s). Then $cat(s) = \Pi(\varepsilon) \cdot w$.*

Proof: Easy induction. ∎

4.3 A few properties

For a pair (M^τ, s) (which is not necessarily a configuration), we use the notation:

$$(M^\tau, s){\downarrow}\alpha = \begin{cases} (M^\tau, s{\downarrow}\pi), & \text{if } \alpha \text{ occurs free in } \tau \\ & \text{at a leaf } \pi \subseteq head(s); \\ (M^\tau, s), & \text{otherwise.} \end{cases}$$

The following lemmas collect some properties of the relation $\Rightarrow$.

Lemma 9 *Let M_1 and M_2 be adjacent nodes in a graph of a term. Suppose that*

$$(M_1[\sigma/\alpha]^{\tau_1[\sigma/\alpha]}, s_1) \Rightarrow (M_2[\sigma/\alpha]^{\tau_2[\sigma/\alpha]}, s_2),$$

and that this is neither a compression nor a decompression step. Then

$$(M_1^\tau, s_1)\downarrow\alpha \Rightarrow (M_2^\tau, s_2)\downarrow\alpha.$$

Proof: The only trouble may occur when $s_1 = (p) \circ s_2$ or $s_1 = (q) \circ s_2$ and we have $(M_1^\tau, s_1)\downarrow\alpha = (M_1^\tau, (\varepsilon) \cdot s_1)$. But this can only happen when we go from M_1 to M_2 by traversing a p-arrow or a q-arrow backward. Thus, τ must be an arrow type. On the other hand, an empty word at the beginning of $(\varepsilon) \cdot s_1$ means that τ has a free occurence of α at the root — a contradiction. ∎

Lemma 10 *Write $\nu(\sigma_1, \sigma_2)$ for $\nu[\sigma_1/\alpha, \sigma_2/\beta]$. Assume that β is not free in σ and that $M : \forall\beta\,\nu(\alpha, \beta)$. Let*

$$((M[\sigma/\alpha]\zeta[\sigma/\alpha]), s) \Rightarrow (M[\sigma/\alpha], s)\downarrow\beta.$$

Then $((M\zeta)^{\nu(\alpha,\zeta)}, s)\downarrow\alpha \Rightarrow ((M^{\forall\beta\,\nu(\alpha,\beta)}, s)\downarrow\alpha)\downarrow\beta$.

Proof: The only possible difficulty here is when we have an occurrence of β along s, but we cannot make a cut there because it would have to occur after the comma introduced by the operator $\downarrow\alpha$. (Note that the cut at the position of α is necessary to make s correct for type $\nu(\alpha, \zeta)$.) But this would only be possible if β were free in σ. ∎

4.4 Good computations

The definition of compression and decompression steps is such that placing and removing separators (commas) on our push-down store also exhibits a form of push-down store behaviour. Once placed on the pds, a separator can only be removed when it becomes a top separator again. In fact, we need more. We say that a computation is *good* iff every separator is removed exactly at the same node where it was placed. More formally, a good computation along a path Π is one which, after every compression step of the form $((P\xi)^{\nu[\xi/\gamma]}, s) \Rightarrow (P^{\forall\gamma\,\nu}, s\downarrow\pi)$, enters a sequence of configurations, of the form $(P_0, s_0), (P_1, s_1), \ldots, (P_m, s_m)$, with these properties:

- $(P_0, s_0) = (P^{\forall\gamma\,\nu}, s\downarrow\pi)$, that is $s_0 = (\pi) \cdot tail(s_0)$;
- Each s_i is of the form $\overline{s}_i \cdot tail(s_0)$, for some sequence $\overline{s}_i$;
- $(P_m, s_m) = (P, (\pi') \cdot tail(s_0))$, where π' is a leaf of ν, labeled by γ. In other words $(P_m, s_m) = (P, s'\downarrow\pi')$, where $s' = (\pi') \circ tail(s_0)$.

It should be obvious that the next configuration must now be $(P\xi, s')$. Thus, whenever we compress a bunch of wires, we must keep it compressed until we visit the same node again. A computation segment from a compression to a matching decompression, satisfying the above conditions, will be called a $\forall$-*cycle*.

4.5 Good paths

Assume that we have a term $Q : \rho$ with an occurrence of a free variable $x : \rho$, and assume that Q beta-reduces to x. Note that there may be more than one occurrence of x in Q, but one can always choose one that will survive all the reductions. (Thus, for simplicity, one can assume that there is only one occurrence of x.) We are interested in finding a certain path in $G(Q)$ that will begin at the variable edge leading to x and end at the top node of Q. It will be a regular path in the sense of [1], but since we are interested in computations along this path, we must define it from scratch, rather than derive its existence from [1].

We say that a path Π in $G(Q)$ is a *good* path if it satisfies the following conditions.

1. The path Π begins at the variable edge associated to (a particular occurrence of) x and ends at the top node.
2. For each wire w, there is a good computation along Π, beginning with the configuration $(x, (w))$, and ending with the configuration $(Q, (w))$.

We want to show that there is a good path in $G(Q)$. The construction of our path is by induction with respect to the number of reduction steps needed to reach x from Q. The base case is obvious: If $Q \equiv x$ then the trivial path from x to itself is good.

4.6 Induction step: type reduction

Assume that Q_1 is obtained from Q by a type reduction step

$$(\Lambda\alpha.A)^{\forall\alpha\,\tau}\sigma \to_\beta A[\sigma/\alpha]^{\tau[\sigma/\alpha]},$$

and that there is a good path Π_1 in $G(Q_1)$. The reduction on the graph consists in eliminating two adjacent nodes, the $@\sigma$-node and the Λ-node. The path Π_1 can visit these nodes several times in both directions. Of course we define Π as Π_1 with these nodes deleted. We need to prove:

Lemma 11 *If the path Π_1 in $G(Q_1)$ is good then so is Π in $G(Q)$.*

Since Π and Π_1 are essentially the same path, we denote both by Π. It is also convenient to identify the corresponding nodes in $G(Q)$ and $G(Q_1)$ with the convention that the node $A[\sigma/\alpha]$ of $G(Q_1)$ corresponds to either of $(\Lambda\alpha.A)\sigma$, $\Lambda\alpha.A$ and A in $G(Q)$.

Note that our push-down automaton is deterministic in that there is at most one possible computation along the path Π. What we have to prove is that this unique computation is successful (all steps along Π are defined) and that it is good.

We begin with a definition. We say that a compression step in a certain computation is *of depth r* iff the separator placed at this step will remain at most $r + 1$-st separator in all configurations to follow before the separator is finally removed or the computation is successfully completed or aborted.

In the following consideration we only take into account configurations and computations for a fixed wire w.

Lemma 12 *Consider a node N in $G(Q)$ and assume that the computation step to be executed in the configuration $(N, (\pi) \cdot s)$ is a compression. Let N_1 be the corresponding*

node in $G(Q_1)$, and suppose that, for some s_1, the configuration $(N_1, (\pi)\cdot s_1)$ occurs in the good computation in $G(Q_1)$ along the path Π. Then there is a $\forall$-cycle in $G(Q)$ from the configuration $(N, (\pi)\cdot s)$ back to N.

Proof: The proof is by induction with respect to the depth of the compression step to be executed from $(N, (\pi)\cdot s)$. Let N have the form $P\xi$ and the compression to be executed be $((P\xi)^{\nu[\xi/\gamma]}, (\pi)\cdot s) \Rightarrow (P^{\forall\gamma\,\nu}, ((\pi)\cdot s)\downarrow\pi')$.

Let us start with a compression of depth zero. That is, the separator placed at this step will remain at the top. We have two cases.

The first case is when $\gamma \neq \alpha$, i.e., the compression step also applies to N_1. Observe that the locations of w in the configurations obtained after the compression in both N and N_1 must begin with the same top word (pointing to the position of γ in the appropriate type). Indeed, this is quite immediate if our compression takes place outside of A, and if it happens within A then we can apply Lemma 10. We should observe that since the compression took place, our wires must remain "below" occurrences of γ in all the types they cross. More precisely, all configurations must be of the form $(R[\xi/\gamma]^{\nu[\xi/\gamma]}, s\downarrow\pi)$, where γ occurs in ν at node π. This guarantees that all steps we encounter are "ordinary", in particular we do not enter an unwanted decompression, and we must remain within the term $P\xi$. (Remember than a single leaf can only be governed by a single quantifier.)

Observe now that a suffix of the location that does not take part in the computation step can be replaced by another suffix with no harm. This observation applied to all the subsequent steps of the $\forall$-cycle from N_1 to N_1 in $G(Q_1)$ implies that an analogous $\forall$-cycle must be obtained also in the graph $G(Q)$.

The second case is when $\gamma = \alpha$, i.e, the compression is caused by type application $(\Lambda\alpha.A)^{\forall\alpha\,\tau}\sigma$. We now use Lemma 9 to check correctness of the "ordinary" steps following this compression. Again, our wires are "below" occurrences of α in all the types they cross. In particular there must be a free α always above. Thus the only way to leave the term without additional compression is through the top node (which completes the $\forall$-cycle) since no free variable of A can contain α free.

But we must eventually leave the term somehow in order to reach the top node of Q.

For the induction step (when the depth r of our compression is greater than zero), the argument is similar to the above. The difference is that we can enter another compression step, in which case, by the induction hypothesis for $r-1$, we must return to where this additional compression took place. We should also consider two cases here. Note that Lemma 10 is used to guarantee the correctness of the next compression in case $\gamma = \alpha$. This next compression must happen within A because, as we noted above, our locations must remain below α and thus cannot leave the term A before next compression. ∎

Proof of Lemma 11: From Lemma 12 we obtain that our path Π behaves in the proper way with respect to the compression-decompression mechanism. It remains to observe that our computation works correctly at the segments of Π outside of all $\forall$-cycles. ∎

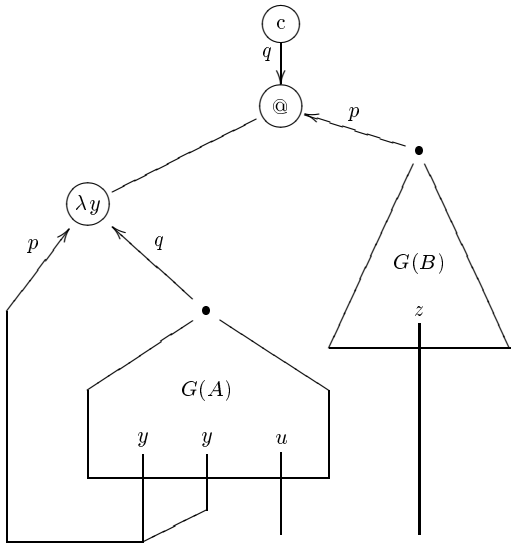

Figure 5: Before reduction: $G((\lambda y.A)B)$

4.7 Induction step: term reduction

Now assume that Q_1 is obtained from Q by an ordinary beta-reduction $(\lambda y.A)B \to A[B/y]$. We consider only the case when y is free in A, the other case is easier. Figure 5 presents the graph of the redex, and Figure 6 shows the graph of its contractum.

Assume the path Π_1 in $G(Q_1)$ is already correctly defined. We define Π in $G(Q)$as follows

- Outside of the redex the path is unchanged.
- The same for portions of the path within $A[B/y]$ but outside of B.
- Each copy of B in $A[B/y]$ is represented in the redex by the single argument B. Every portion of the path Π_1 that goes through any of the multiple B's in $A[B/y]$ is replaced by an identical portion going through B in $(\lambda y.A)B$.
- Whenever Π_1 enters $A[B/y]$ through its top node, the path Π enters $(\lambda y.A)B$ through its top node (a q-arrow), then goes to $(\lambda y.A)$ and takes the q-arrow backward to enter A. (Note that there is no other choice.)
- Whenever Π_1 enters B through its top node the path Π reaches a v-node within A. Then it must go to the argument B, traversing p and then p backward.
- Whenever Π_1 enters or leaves $A[B/y]$ through a variable, the path Π enters or leaves the redex via the corresponding variable. (Note that no variable free in B can be bound in $A[B/y]$.)

Lemma 13 *If the path Π_1 in $G(Q_1)$ is good then so is Π in $G(Q)$.*

Proof: Part 1 is obvious. For Part 2 observe that a correct computation along Π_1 is modified only by inserting several passes from v-nodes to B and conversely, and these steps are

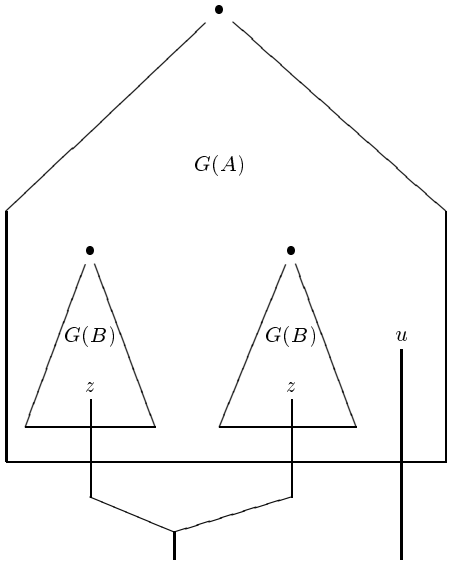

Figure 6: After reduction: $G(A[B/y])$

correct. The computation is good, because compressions and decompressions remain the same, and the path Π_1 is good (the $\forall$-cycles are essentially the same). ∎

Lemma 14 *Let a term $Q : \rho$ beta-reduce to $x : \rho$. Then there exists a good path Π in $G(Q)$.*

Proof: Immediate from Lemmas 11 and 13. ∎

5 Type embeddings

We say that type ρ can be *embedded* into another type τ, 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^ρ is a fresh variable. For a type σ, let $d(\sigma)$ be the depth of the tree $\overline{\sigma}$).

Lemma 15 *Let $Q = G(Fx)$, and let Π be a good path in the graph $G(Q)$. Let w be an arbitrary wire. The computation for w along Π must leave Fx through the top node at least once with the wire w not compressed, i.e., there must be a configuration of the form (Fx, s), where $s = (\pi \cdot w)$.*

Proof: We start at x with our wire w uncompressed. Each time it is compressed, a $\forall$-cycle begins, and all these $\forall$-cycles must end where they begin. The path Π must thus eventually leave Fx with wire w uncompressed. This must happen at the top node (no free variable of Fx is bound outside). ∎

Theorem 16 *If $\rho \leq \tau$ then $d(\rho) \leq d(\tau)$.*

Proof: Assume that $F : \rho \to \tau$ and $G : \tau \to \rho$ are such that $G \circ F =_\beta I$. Let $Q = G(Fx)$. By Lemma 14 there is a good path Π in Q. From Lemma 15, we have that every leaf w of $\overline{\rho}$ is a final segment of a leaf of $\overline{\tau}$. Thus, the depth of $\overline{\tau}$ must be at least equal to the depth of $\overline{\rho}$. ∎

Corollary 17 *System $\boldsymbol{\lambda}$**2U** cannot be defined within $\boldsymbol{\lambda}$**2**, by means of beta reductions.*

Proof: Suppose there is a type μ and there are operators **Fold** and **Unfold** implementing $\mu\alpha.\sigma$, where $\sigma \neq \alpha$ and α is free in σ. Then $\sigma[\mu/\alpha] \leq \mu$, contradicting Theorem 16. ∎

6 Conclusions

If we write $\preceq_\beta$ and $\preceq_{\beta\eta}$ to denote, respectively, beta and beta-eta interpretability, then the results of the present paper can be symbolically presented as follows. First, we have that iterators are syntactic sugar:

$$\boldsymbol{\lambda}\mathbf{2} \subseteq \boldsymbol{\lambda}\mathbf{2I}, \boldsymbol{\lambda}\mathbf{2J} \preceq_\beta \boldsymbol{\lambda}\mathbf{2}.$$

Propositions 5 and 6 give the following:

$$\boldsymbol{\lambda}\mathbf{2R}, \boldsymbol{\lambda}\mathbf{2Rec} \preceq_{\beta\eta} \boldsymbol{\lambda}\mathbf{2U} \preceq_{\beta\eta} \boldsymbol{\lambda}\mathbf{2R}, \boldsymbol{\lambda}\mathbf{2Rec}.$$

Finally, the meaning of Corollary 17 is

$$\boldsymbol{\lambda}\mathbf{2U} \not\preceq_\beta \boldsymbol{\lambda}\mathbf{2}.$$

We are not at all satisfied with these results, because the picture they give is highly incomplete. There are two obvious questions that should be asked:

1. Can **λ2U** be interpreted within **λ2R**(**λ2Rec**) by means of beta reductions only?
2. Can **λ2U** be interpreted within **λ2** with help of beta-eta reductions?

A positive answer to question (1) and/or a negative answer to question (2) would give us the right to conclude that *"recursion is stronger than iteration"*, because we could derive either **λ2R** $\not\preceq_\beta$ **λ2I** or **λ2R** $\not\preceq_{\beta\eta}$ **λ2I**. In fact, we believe in a negative answer to question 2:

Conjecture 18 **λ2U** $\not\preceq_{\beta\eta}$ **λ2**.

It is not difficult to see that Theorem 16 is no longer true if η-reductions are permitted. Thus, a proof of Conjecture 18 will require a finer analysis of how the particular wires are transmitted along paths in polymorphic terms. (We conjecture that different wires should be passed through different leaves of $\overline{\tau}$.) In addition, in the presence of η-reductions, one good path is not enough and one has to consider a system of paths, one for each wire.

Acknowledgements

Numerous discussions with Mariangiola Dezani, Simona Ronchi Della Rocca, Sergei Soloview, Roberto Di Cosmo and Adolfo Piperno have greatly influenced this work. The idea of using paths in terms was once suggested to the second author by Simona Ronchi Della Rocca as a possible tool for some other problem. A discussion with Vincent Danos was essential for choosing the proper approach. Jerzy Tiuryn has read an early draft of this paper. Laurent Regnier pointed out some bugs in the initial version and suggested means to correct them. We are greatly indebted to Ralph Matthes who pointed out an essential gap in our paper and to Peter Lee for his effort to correct the proceedings at the last moment.

References

[1] Asperti, A., Danos, V., Laneve, C., Regnier, L., Paths in the lambda-calculus. *Proc 9th IEEE Symposium on Logic in Computer Science*, IEEE, 1994, pp. 426-436.

[2] Asperti, A., Guerrini, S., *The Optimal Implementation of Functional Programming Languages*, Cambridge Tracts in Theoretical Computer Science **45**, Cambridge University Press, Cambridge, 1998.

[3] van Bakel, S., Liquori, L., Ronchi Della Rocca, S., Urzyczyn, P., Comparing cubes of typed and type assignment systems, *Annals of Pure and Applied Logic* **86**, 1997, 267–303.

[4] Böhm, C., Berarducci, A., Automatic synthesis of typed Λ-programs on term algebras, *Theoretical Computer Science*, **39**(2/3), 1985, 135–154.

[5] Bruce, K., Longo, G., Provable isomorphisms and domain equations in models of typed languages, *Proc. 17th Symposium on Theory of Computing*, ACM, 1985, 263–272.

[6] 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.

[7] Danos, V., Regnier, L., Proof-nets and the Hilbert space, in *Advances in Linear Logic* (Girard, Lafont and Regnier, Eds.), London Mathematical Society Lecture Notes, 222, Cambridge University Press, Cambridge, 1995.

[8] Dezani, M., Characterization of normal forms posessing an inverse in the $\lambda\beta\eta$-calculus, *Theoretical Computer Science* **2**, 1976, 323–337.

[9] Di Cosmo, R., *Isomorphisms of types: from lambda-calculus to information retrieval and language design*, Birkhauser, 1995.

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

[11] Girard, J.-Y., Geometry of Interaction I: Interpretation of System **F**, *Proc. Logic Colloquium'88* (Ferro, Bonotto, Valentini and Zanardo, Eds.), Elsevier Science, 1989, pp. 221–260.

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

[13] Harper, R. and Honsell, F. and Plotkin, G., A framework for defining logics, *Journal of the ACM*, **40**(1), 1993, 143–184.

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

[15] 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.

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

[17] Leivant, D., Functions over free algebras definable in the simply typed lambda calculus, *Theoretical Computer Science*, **121**, 1993, 309–321.

[18] Margaria, I., Zacchi, M., Right and left invertibility in λ-β-*calculus*, *R.A.I.R.O. Informatique Théorique/Theoretical Informatics*, **17** (1), 1983, 71–88.

[19] Matthes, R., *Extensions of System* F *by Iteration and Primitive Recursion on Monotone Inductive Types*, PhD Thesis, University of Munich, 1998.

[20] 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.

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

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

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

[24] Reynolds, J.C., Polymorphism is not set-theoretic, *Proc. Semantics of data types*, LNCS 173, Springer-Verlag, Berlin, 1984, pp. 145–156.

[25] Reynolds, J.C., Plotkin, G.D., On functors expressible in the polymorphic typed lambda calculus, *Information and Computation*, **105**(1), 1993, 1–29.

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

[27] Urzyczyn, P., Positive recursive type assignment, *Fundamenta Informaticae*, **28**(1-2), 1996, 197–209.

[28] Urzyczyn, P., Recursive types and type embeddings, Research Report TR 99-01 (256), Warsaw University, 1999.

[29] 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.), LNCS 389, Springer-Verlag, Berlin 1989, pp. 118–127.