Fundamenta Informaticae 28 (1996) 197–209
IOS Press

Positive Recursive Type Assignment*

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

Abstract. We consider several different definitions of type assignment with positive recursive types, from the point of view of their typing ability. We discuss the relationships between these systems. In particular, we show that the class of typable pure lambda terms remains the same for different type disciplines involving positive type fixpoints, and that type reconstruction is decidable.

Keywords: Lambda calculus, positive recursive types, type constraints.

1. Introduction

This paper's objective is to study "Curry style" type assignment with recursive types. By "Curry style", as opposed to fully typed "Church style", we mean the approach where types are assigned, as predicates, to pure lambda terms. A type assignment system proves judgements of the form $E \vdash M : \tau$, where M is a term of untyped lambda calculus, τ is a type, and E is a set of type assumptions about free variables (a type environment).

By recursive types we mean "type fixpoints" represented by expressions of the form $\mu\alpha\tau$, where τ is a type and α is a type variable. Such a fixpoint type is identified with its unfolding $\tau[\mu\alpha\tau/\alpha]$. For instance, if τ is $(\alpha \to \beta) \to \beta$ then $\mu\alpha\tau$ is equivalent to $((\mu\alpha\tau) \to \beta) \to \beta$, and this equivalence allows to derive e.g., $E \vdash z(xy)(yx) : \beta$, where the environment E consists of the type assumptions: $(x : \mu\alpha\tau)$, $(y : (\mu\alpha\tau) \to \beta)$ and $(z : \beta \to \beta \to \beta)$.

A type containing occurrences of the constructor μ is often represented as an infinite regular tree obtained by unwinding all its fixpoints. But the notion of equivalence between types induced by such presentation is stronger than the above "syntactic" equivalence: different types may unwind to the same tree (see [4]).

Type assignment with arbitrary fixpoints was studied, for instance, in the papers [2], [3] and [4]. See also [1], for a survey of major properties. However, arbitrary recursive types are in a sense too strong. Let σ be $\mu\alpha(\alpha \to \alpha)$. One can easily show that all terms can be assigned the type σ. This means that all terms are typable.

In order to keep the class of typable terms within the bounds of strong normalization property, one imposes the positivity condition: $\mu\alpha\tau$ is a legal type only if α does not occur in τ to the left of an odd number of arrows. Mendler ([7]) proves (essentially) that the

*Partly supported by NSF Grant CCR-9113196, KBN Grant 2 P301 031 06 and by ESPRIT BRA7232 "Gentzen". A preliminary version of this paper has been presented at the Symposium "Mathematical Foundations of Computer Science" Prague, Czech Republic, 1995.

positivity condition is sufficient for strong normalization, and that it is the weakest possible: adding an arbitrary negative fixpoint always allows to type a non-normalizable term.

In the present paper we concentrate on the positive recursive types. The goal is to compare the existing approaches to positive recursive type assignment, with respect to their typing ability. We first consider the approach based on the fixpoint constructor μ as described above. But there are two ways of understanding the equivalence between a fixpoint and its unfolding. One possibility is to provide rules to replace a typing $M : \mu\alpha\tau$ with $M : \tau[\mu\alpha\tau/\alpha]$, and conversely (folding and unfolding at the top level). The other way is to identify these types entirely, which generates a congruence on the set of all types, and then to replace freely all congruent types with each other.

An entirely different approach is to use "type constraints", i.e., to introduce type constants, and postulate equations of the form e.g. "$c = (c \to \beta) \to \beta$". Such c then behaves in a similar way to $\mu\alpha((\alpha \to \beta) \to \beta)$. This is the approach of [2] and [7].

As we mentioned before, recursive types are conveniently presented by regular trees. We would like to do the same with positive recursive types, but as noted in [3] and [4], the tree model is not adequate here. The difficulty is that a positive fixpoint may unwind to the same tree as a negative one, which means introducing "bad behaviour" in an implicit way. To avoid this we propose in Section 4. to use labelled trees to preserve the polarity information.

Our main result is that all the four type assignment systems are of equal typing power, in the following sense: a pure lambda term is typable in one of these systems iff it is typable in any other (but not necessarily into the same type). We also show how to use our tree representation to prove that self-application is not typable (Example 4.1.), and that typability is decidable (Corollary 5.1.).

Section 2. introduces the type assignment systems based on the constructor μ. Section 3. contains some technical results needed for the next Section 4., which introduces our trees. The last Section 5. deals with type constraints and contains our main result, Theorem 5.1.

2. The Basic Type Assignment Systems

The positive recursive types are defined by mutual induction, together with the notion of positive and negative free variables. For a type τ, we use the notation $FV_+(\tau)$ and $FV_-(\tau)$ to denote the sets of variables occurring positively (resp. negatively) in τ.

- Type variables are types, and we have $FV_+(\alpha) = \{\alpha\}$ and $FV_-(\alpha) = \emptyset$;
- If σ and τ are types then $(\sigma \to \tau)$ is a type, satisfying $FV_+(\sigma \to \tau) = FV_+(\tau) \cup FV_-(\sigma)$ and $FV_-(\sigma \to \tau) = FV_-(\tau) \cup FV_+(\sigma)$;
- If σ is a type and α is a type variable, such that $\alpha \notin FV_-(\sigma)$, then $(\mu\alpha\sigma)$ is a type, and we define $FV_+(\mu\alpha\sigma) = FV_+(\sigma) - \{\alpha\}$, and $FV_-(\mu\alpha\sigma) = FV_-(\sigma)$.

The notational conventions are as follows: the operator μ is of higher priority than arrow, and arrows associate to the right. Unnecessary parentheses are omitted. That is, e.g., the notation "$\mu\alpha\sigma \to \tau \to \rho$" is equivalent to "$((\mu\alpha\sigma) \to (\tau \to \rho))$". Since μ binds variables, we allow for alpha conversion, i.e., we identify types that are the same except for a renaming of their bound variables. Also the notion of substitution, denoted $\tau[\sigma/\alpha]$, is as usual, with a possible alpha conversion of τ. A *type environment* is a set E of pairs of the form $(x : \sigma)$, where x is an object variable and σ is a type, such that if $(x : \sigma), (x : \sigma') \in E$ then $\sigma = \sigma'$. Thus, an environment is a finite partial function from variables into types. If E is an environment then $E(x : \sigma)$ is an environment such that

$$E(x : \sigma)(y) = \begin{cases} \sigma, & \text{if } x \equiv y; \\ E(y), & \text{if } x \not\equiv y. \end{cases}$$

The terms of the pure λ-calculus are defined as usual by the grammar

$$M ::= x \mid (M\,M) \mid (\lambda x\, M)$$

A *judgement* is an expression of the form $E \vdash M : \tau$ where E is a type environment. We consider two basic systems to derive judgements involving positive recursive types. Both these systems contain the ordinary simple type assignment rules:

(Var) $\quad E \vdash x : \sigma \qquad$ if $(x : \sigma)$ is in E

(App) $\quad \dfrac{E \vdash M : \tau \to \sigma,\ E \vdash N : \tau}{E \vdash (MN) : \sigma}$

(Abs) $\quad \dfrac{E(x : \tau) \vdash M : \sigma}{E \vdash (\lambda x.M) : \tau \to \sigma}$

The simplest way to introduce positive recursive types is to extend this core system by the following two rules:

(Fold) $\quad \dfrac{E \vdash M : \tau[(\mu\alpha\tau)/\alpha]}{E \vdash M : \mu\alpha\tau}$

(Unfold) $\quad \dfrac{E \vdash M : \mu\alpha\tau}{E \vdash M : \tau[(\mu\alpha\tau)/\alpha]}$

We use the notation $\vdash_{FU}$ to denote derivability in the above system. Our second system, which will be referred to by the notation $\vdash_\sim$, has only one new rule in addition to (Var), (App) and (Abs):

($\sim$) $\quad \dfrac{E \vdash M : \sigma,\ \ \sigma \sim \tau}{E \vdash M : \tau}$

Here, the relation $\sim$ is the smallest congruence on types satisfying

$$\mu\alpha\tau \sim \tau[(\mu\alpha\tau)/\alpha].$$

Clearly, rules (Fold) and (Unfold) are then seen as special cases of rule ($\sim$), and we have the obvious implication:

$$\text{If } \ E \vdash_{FU} M : \tau \ \text{ then } \ E \vdash_\sim M : \tau.$$

The converse implication does not hold, for the simple reason that

$$(x : \mu\alpha(\beta \to \alpha) \to \beta) \ \not\vdash_{FU} \ x : (\beta \to \mu\alpha(\beta \to \alpha)) \to \beta,$$

while the two types are obviously $\sim$-related. Note that the above example shows also that typing under $\vdash_{FU}$ is not closed under eta reduction. Indeed, a correct $\vdash_{FU}$-judgement is obtained from the above if x is replaced by $\lambda y.xy$ at the right hand side. On the other hand, it is not difficult to prove that $E \vdash_\sim M : \tau$ implies $E \vdash_\sim M' : \tau$, for all M' such that $M \twoheadrightarrow_\eta M'$.

However, there is more between $\vdash_\sim$ and $\vdash_{FU}$ than just eta reduction. For instance, we can derive

$$(x : \mu\alpha(\mu\beta(\alpha \to \beta) \to \alpha)) \ \vdash_\sim \ x : \mu\alpha((\alpha \to \mu\beta(\alpha \to \beta)) \to \alpha),$$

while, for all eta expansions X of x,

$$(x : \mu\alpha(\mu\beta(\alpha \to \beta) \to \alpha)) \ \not\vdash_{FU} \ X : \mu\alpha((\alpha \to \mu\beta(\alpha \to \beta)) \to \alpha).$$

The latter claim follows from the following Lemma.

Lemma 2.1. Let the *arity* of a fixpoint expression $\mu\alpha\rho$ be the number of free occurrences of α in ρ. If X is an eta expansion of a variable x, and if $x : \tau \vdash_{FU} X : \sigma$ then τ and σ have fixpoint subtypes of the same arities.

An easy proof by induction on the length of X is left to the reader. (Note that a proper expansion X of x must be of the form $\lambda y_1 \ldots y_n . X' Y_1 \ldots Y_n$, where X' is an eta expansion of x and each Y_i is an eta expansion of y_i.)

Two important properties of the positive recursive type assignment are as follows:

Theorem 2.1. Let $E \vdash M : \tau$, where $\vdash$ is either $\vdash_\sim$ or $\vdash_{FU}$. Then:

1. The term M is strongly normalizable;
2. If $M \twoheadrightarrow_\beta N$ then also $E \vdash N : \tau$.

Proof:
Part (1) follows from Mendler's paper [7], because of the equivalence between type constraints and recursive type assignment (see Theorem 5.1.). For part (2), we need to prove the following claim:

$$\text{If } E(x : \tau) \vdash M : \sigma \text{ and } E \vdash N : \tau \text{ then } E \vdash M[N/x] : \sigma.$$

This goes by an easy induction on M. Details are left to the reader. □

We conclude this section with a remark on the power of recursive types. Let $\tau = \mu\alpha(\beta \to \alpha)$. We have $\tau \sim (\beta \to \tau)$ and one can easily derive $(x : \tau) \vdash \lambda y. x : \tau$. This means that the combinator $\mathbf{K} \equiv \lambda xy. x$ has type $\tau \to \tau$. If $\mathbf{2} \equiv \lambda fx. f(fx)$ then of course we have $\mathbf{2} : (\tau \to \tau) \to \tau \to \tau$ and also $\mathbf{2} : ((\tau \to \tau) \to \tau \to \tau) \to (\tau \to \tau) \to \tau \to \tau$. It follows that $\mathbf{22K} : \tau \to \tau$. On the other hand we have:

Lemma 2.2. The term **22K** is untypable in the second order polymorphic lambda calculus (System **F**).

Sketch of proof: It is easy to see that the type σ assigned to **K** must have the form:

$$\sigma = \forall\vec{\alpha}(\tau \to \forall\vec{\beta}(\rho \to \tau')),$$

where τ' is an instance of τ. Using the ordinary tree representation of polymorphic types, we can say, in particular, that the rightmost path of τ' must be at least as long as the rightmost path in τ. The vector $\vec{\beta}$ of quantified variables may be empty, but $\vec{\alpha}$ cannot. This is because **22K** reduces to $\mathbf{2}(\lambda x.\mathbf{K}(\mathbf{K}x))$, and types of the two copies of **K** must be different. In addition, one of the variables $\vec{\alpha}$, say α, must occur at the end of the rightmost path in τ — otherwise the composition of the two **K**'s is ill-typed. The same α must remain at the end of the rightmost path in the instance τ', at the same depth.

Now let us consider the type of the second **2** in **22K**. It must have the form $\forall\vec{\gamma}(\sigma_0 \to \varrho)$, where σ can be obtained from σ_0 by instantiating $\vec{\gamma}$. After excluding the impossible case that σ_0 is just a variable, one concludes that it begins with $\forall\vec{\alpha}$ and has occurrences of α at the same places as σ does. In particular there is an occurrence of α at some depth n at the rightmost path of the left subtree of σ_0 and at depth $n+1$ at the rightmost path of the right subtree of σ_0. Now, σ_0 is the type of f in $f(fx)$ and ϱ is the type of $\lambda x. f(fx)$. No matter what is the type of x, we can note that the asymetry of σ_0 is doubled in ϱ, and thus the rightmost path in ϱ must be of length at least $n+3$. Although $\forall\vec{\gamma}(\sigma_0 \to \varrho)$ may still be a good type for **2**, a term of this type cannot be composed with itself, as the positions of α cannot be changed by just instantiating $\vec{\gamma}$. □

It follows that recursive types cannot be replaced by polymorphic types as long as typability is concerned: the class of terms typable with positive type fixpoints is not a subset of polymorphically typable terms. In Section 4. we show that the converse inclusion is also not true.

3. Paths in Types

This section is devoted to some technical preparation that will be used later. We begin with a simple observation: we can get rid of the "degenerate" fixpoint $\mu\alpha\alpha$, because it unfolds only to itself: $\alpha[\mu\alpha\alpha/\alpha]$ is just $\mu\alpha\alpha$ again. More formally:

Lemma 3.1. Let $E \vdash M : \tau$, and let γ be a fresh type variable (not occurring in E and τ). Let E' and τ' be obtained from E and τ by replacing each occurrence of $\mu\alpha\alpha$ with an occurrence of γ. Then $E' \vdash M : \tau'$, and this judgement can be derived without using any occurrence of $\mu\alpha\alpha$.

Proof:
Easy. □

The above lemma allows us from now on to exclude $\mu\alpha\alpha$ from the set of types occurring in our consideration. This does not affect our results as long as we only ask about typability/nontypability.

Let us now remark that the relation $\sim$ on types can be seen as equality generated by a reduction relation $\rightarrow_\mu$, given by the following rule:

$$\mu\alpha\tau \ \rightarrow_\mu \ \tau[(\mu\alpha\tau)/\alpha].$$

A useful observation is that such a reduction relation makes a *regular combinatory reduction system* in the terminology of Klop [5], and thus we have:

Lemma 3.2. The relation $\rightarrow_\mu$ has the Church-Rosser property.

Proof:
See Theorem 3.11 in [5]. □

The next lemma states a basic technical fact about the relation $\sim$:

Lemma 3.3. If $\tau \sim \sigma$ then $\tau[\rho/\alpha] \sim \sigma[\rho/\alpha]$.

Proof:
Induction w.r.t. the definition of $\sim$. □

A *path* is an arbitrary, finite or infinite, word over the alphabet $\{L, R\}$ (understood as "left" and "right"). The empty path is denoted by ε. A finite path w is *positive* iff the number of L's in w is even, otherwise we say that w is *negative*. We write $w \subseteq v$ when w is a prefix of v. If τ is a type, and w is a finite path then $\tau[w]$ will denote the *subtype of τ rooted at w*, defined as follows:

$$\begin{aligned} \tau[\varepsilon] &= \tau; \\ (\tau \rightarrow \sigma)[Lw] &= \tau[w]; \\ (\tau \rightarrow \sigma)[Rw] &= \sigma[w]; \\ (\mu\alpha\tau)[w] &= \tau[w], \text{ for } w \neq \varepsilon. \end{aligned}$$

Note that the definition of $\tau[w]$ depends on the choice of bound variables of τ — it is not invariant under alpha conversion. That's why we have to be careful in the choice of bound variables. In what follows we assume that all types under consideration have their bound variables chosen so that there is no confusion between bound and free variables.
The following are basic properties of subtypes:

Lemma 3.4.

1. Assume that neither α nor any free variable of σ is bound in τ, and that $\tau[\sigma/\alpha][w]$ is defined. Then either $\tau[w]$ is defined and $\tau[\sigma/\alpha][w] = \tau[w][\sigma/\alpha]$, or $w = w_0 w_1$, where $\tau[w_0] = \alpha$, and $\tau[\sigma/\alpha][w] = \sigma[w_1]$.

2. For each τ, w, v, it holds that $\tau[wv] = \tau[w][v]$, provided either side is defined.

Proof:
Part (1) follows by induction w.r.t. τ, part (2) by induction w.r.t. the length of w. (For (1), note that it may happen that $\tau[\sigma/\alpha][w]$ is defined, while $\tau[w]$ is not.) □

If $\tau[w]$ is defined then we say that w is a *node* of τ. The *trace* of a path w in τ, denoted $trace(w,\tau)$ is the sequence of nodes visited by traversing τ down from the root towards leaves along path w, with the additional rule that whenever a μ-bound variable is encountered, one returns to the appropriate μ-binding. For instance, $trace(RRL, \beta \to \mu\alpha(\beta \to \alpha))$ is the sequence $(\varepsilon, R, RR, R, RL)$. Formally, $trace(w,\tau)$ is defined as follows. First, $trace(\varepsilon,\tau) = \varepsilon$. Now assume that $trace(w,\tau)$ has already been defined, and let v be the last node of it. In order for $trace(wL,\tau)$ to be defined, we require that vL is a node of τ. There are two cases:

Case 1: Let $\tau[vL]$ be a variable α which is μ-bound. That is, assume vL is a leaf of τ and there is $v' \subseteq v$ with $\tau[v'] = \mu\alpha\rho$. Take the longest such v', and define $trace(wL,\tau) = trace(w,\tau); vL; v'$, where the semicolon denotes concatenation for sequences of nodes.

Case 2: Otherwise take $trace(wL,\tau) = trace(w,\tau); vL$.

Of course, $trace(wR,\tau)$ is defined analogously. For an infinite path w, we define $trace(w,\tau)$ as the appropriate limit of $trace(w',\tau)$, for finite $w' \subseteq w$.

Lemma 3.5.

1. If $\sigma \sim \tau$ then $trace(w,\sigma)$ is defined iff $trace(w,\tau)$ is defined.
2. Let $\sigma[w]$ be defined. Then the sequence $trace(wv,\sigma)$ is defined iff $trace(v,\sigma[w])$ is defined.

Proof:
(1) An easy induction w.r.t. the definition of $\sim$.
(2) It is easy to see that $trace(v,\sigma[w])$ is always a suffix of $trace(wv,\sigma)$. □

If we think of a recursive type as of an infinite tree then the above lemma states that equivalent types correspond to the same (unlabelled) tree. The main lemma of this section is as follows:

Lemma 3.6. If w is a negative path then $\tau \not\sim \tau[w]$.

Proof:
Assume the contrary, and first observe that $trace(w^\infty,\tau)$, and also $trace(w^\infty,\tau[w])$, must be defined because of Lemma 3.5. For convenience, we will identify each node x of $\tau[w]$ with the node wx of τ. We claim that, under this convention, the infinite sequence $trace(w^\infty,\tau[w])$ equals to the final segment of $trace(w^\infty,\tau)$ obtained by cutting off $trace(w,\tau)$.

Indeed, one can prove by induction w.r.t. the length of u, that for each finite prefix u of w^∞, the sequences $trace(u,\tau[w])$ and $trace(wu,\tau)$ are the same. The only nontrivial case is when the last node x on $trace(wu,\tau)$ is a result of a return from some leaf y of τ. Then $\tau[y] = \beta$, for some β which is μ-bound in τ. The induction step now follows from the fact that the binding place x must lie within $\tau[w]$, as otherwise β would be free in $\tau[w]$ and $trace(w^\infty,\tau[w])$ would break here — a contradiction with Lemma 3.5. Note that this argument implies a general observation that all returns on our path occur below w.

Let us now analyze the shape of $trace(w^\infty,\tau)$ in some detail. There is only finitely many nodes of τ, thus there are k_1, k_2, such that $trace(w^{k_1},\tau)$ and $trace(w^{k_2},\tau)$ end with the same node v. Take the least such pair (k_1,k_2) (lexicographically).

The construction of a trace is determined at each step only by the path and the current node. Thus, each trace can be seen as a computation history of a finite automaton. It follows that the sequence of nodes on $trace(w^\infty,\tau)$ after w^{k_1} must be the same as after w^{k_2}, i.e., that $trace(w^\infty,\tau)$ is periodic after the last node of $trace(w^{k_1},\tau)$. The period is determined

by $w^{k_2-k_1}$. A cycle is possible only if there are returns. Thus there must be nodes v_0, v_1, such that $v_0 \subseteq v \subseteq v_1$ and we have $\tau[v_0] = \mu\alpha\rho$ and $\tau[v_1] = \alpha$, with v_0 following v_1 on $trace(w^\infty, \tau)$ somewhere between the last node of $trace(w^{k_1}, \tau)$ and the last node of $trace(w^{k_2}, \tau)$. (In other words we have $trace(w^\infty, \tau) = trace(w^{k_1}, \tau); \ldots; v_1; v_0; \ldots; v; \ldots; v_1; v_0; \ldots; v; \ldots)$. We choose the highest (shortest) such v_0 (there can be more than one to choose from), and we now look at the trace of w^∞ as a periodic sequence starting with the first occurrence of v_0 after $trace(w^{k_1}, \tau)$. Note that, except returns to v_0, there may be also other returns on each period. All of them are caused by μ-bindings occurring below v_0. We shall show now how to eliminate such returns.

Suppose $\tau[u_0] = \mu\beta\sigma$, $\tau[u_1] = \beta$, with $v_0 \subseteq u_0 \subseteq u_1$, and suppose that a return $\ldots; u_1; u_0; \ldots$ occurs on $trace(w^\infty, \tau)$ during each period. Let τ' be obtained from τ by unfolding this occurrence of $\mu\beta\sigma$ to $\sigma[\mu\beta\sigma/\beta]$.

Let $u_1 = u_0u'$. The sequence $trace(w^\infty, \tau')$ can be obtained from $trace(w^\infty, \tau)$ in the following way. Initial segments are the same, until in τ we reach the return from u_1 to u_0 (call it an "upper phase"). Then a "lower phase" begins, when each node u_0x on $trace(w^\infty, \tau)$ is replaced by u_1x on $trace(w^\infty, \tau')$. The lower phase continues until a return is reached to a node which is above u_0. Then we have again an "upper phase": identical nodes on the corresponding parts of $trace(w^\infty, \tau')$ and $trace(w^\infty, \tau)$. This goes on until another return to u_0 is reached, and we begin another "lower phase".

The useful thing is that $trace(w^\infty, \tau')$ has less returns on each period than $trace(w^\infty, \tau)$, and still we have $\tau' \sim \tau$ and $\tau'[w] \sim \tau[w]$, since both equivalences result from a single unfolding. (Recall that $w \subseteq v_0$, and thus our unfolding is done within $\tau[w]$.) The node v_0 is still the top return node as described before. Repeating the above, we can eliminate all returns below v_0, and thus we can assume from now on that our cycle has only returns to v_0 (not necessarily always from v_1, but always from nodes labelled α). In a similar way, we can also assume that there are no returns before the first period begins. Indeed, call a return *unimportant* iff it occurs only finitely many times on the trace. By unfolding an appropriate μ-subtype, one decreases the number of unimportant returns, eventually obtaining some τ' with $\tau' \sim \tau'[w]$ and no unimportant returns on $trace(w^\infty, \tau')$.

Assume now that v_0 is positive. Take any type σ, and let x be the longest prefix of w^∞ which is a node of σ. We claim that if $\tau \twoheadrightarrow_\mu \sigma$ then $\sigma[x] = \alpha$ and x is positive. The proof is by induction. For the induction step, consider an arbitrary reduction $\sigma \to_\mu \sigma'$. The leaf x is left unchanged in σ', unless the appropriate μ-binding of α was unfolded (because no other returns are possible). In the latter case the path w^∞ chooses a new α-leaf which extends x by a positive path y.

In exactly the same way we prove that x must be negative if $\tau[w] \twoheadrightarrow_\mu \sigma$. Thus, it should be impossible to have a type σ satisfying both $\tau \twoheadrightarrow_\mu \sigma$ and $\tau[w] \twoheadrightarrow_\mu \sigma$. However such a type must exist by the Church-Rosser property — a contradiction. □

4. Labelled Trees

Arbitrary recursive types are often represented as their unfoldings to infinite regular trees, see e.g. [3] and [4]. This representation can be useful, although it is not fully adequate, as two different types can sometimes be unfolded to the same tree. However, when one restricts attention to positive fixpoints only, the ordinary tree model becomes just inconsistent. Consider the type $\tau = \mu\alpha((\alpha \to \beta) \to \beta)$. There is nothing really wrong if it is represented by the same tree as $\mu\alpha((((\alpha \to \beta) \to \beta) \to \beta) \to \beta)$, but the serious problem is different. Let $\sigma = (\tau \to \beta)$. Then we have $\tau \sim (\sigma \to \beta)$ and both τ and σ unfold to exactly the same tree. Now, τ is a "negative" subtype of σ and equating these two types results in a calculus which is no longer strongly normalizable (see [7]).

A tree-like representation of positive recursive types is useful because it allows sometimes for graph-theoretical reasoning about type inference (especially when one wants to prove non-

typability). Since the polarity information is lost after unfolding, we must keep it visible by adding extra informations to our trees. That's why we propose to use labelled trees instead. In what follows a *tree* is defined as a function $T : \mathrm{Dom}(T) \to \mathrm{TVar} \cup \mathcal{L}$, such that

- TVar denotes the set of all type variables;
- the symbol $\mathcal{L}$ stands for a set of *labels*;
- $\mathrm{Dom}(T)$ is a nonempty downward closed subset of $\{L, R\}^*$, i.e., $wv \in \mathrm{Dom}(T)$ implies $w \in \mathrm{Dom}(T)$;
- for each $w \in \mathrm{Dom}(T)$, either $T(w) \in \mathcal{L}$ and $wR, wL \in \mathrm{Dom}(T)$, or $T(w)$ is a variable and w is a leaf.

By $T|_w$ we denote the *subtree* of T *rooted at* w, i.e., a tree with domain $\{v : wv \in \mathrm{Dom}(T)\}$, defined by $T|_w(v) = T(wv)$. A tree is called *regular* iff it has only a finite number of different subtrees. (In particular there is only a finite number of labels.)

Let T and T' be regular trees such that $\mathrm{Dom}(T) = \mathrm{Dom}(T')$. We identify T with T' if the equality $T(w) = T'(w)$ holds *for almost every* $w \in \mathrm{Dom}(T)$, including necessarily all the leaves. That is, from now on, we write $T = T'$ even if there is a finite number of differences in the labelling of T and T', but only at the internal nodes. In particular, this means that internal labelling of finite trees can be ignored altogether, so our finite trees are equivalent to ordinary finite types.

A node w is *positive* iff it has an even number of L's, otherwise it is called *negative*. Now we would like to define a positive tree as one that satisfies $T(w) \neq T(wv)$, for each w and each negative v. This would lead to difficulties because of our convention to identify trees with labels equal almost everywhere, so we must relax this condition as follows: a tree T is called *positive* iff $T(w) \neq T(wv)$ holds for almost every pair (w, v), with negative v (provided both sides are defined).

If T_1 and T_2 are trees then $T_1 \to T_2$ denotes the only tree T satisfying $T|_L = T_1$ and $T|_R = T_2$. Note that, due to our convention, the label $T(\varepsilon)$ does not matter.

For our *tree assignment system*, we define a *tree environment* as a finite partial function from variables to positive regular trees, and we use the notation $\mathcal{E}(x : T)$ as for ordinary environments. The rules are as follows:

$$\text{(Var)} \qquad \mathcal{E} \vdash x : S \qquad \text{if } (x : S) \text{ is in } \mathcal{E}$$

$$\text{(App)} \qquad \frac{\mathcal{E} \vdash M : T \to S,\ \mathcal{E} \vdash N : T}{\mathcal{E} \vdash (MN) : S}$$

$$\text{(Abs)} \qquad \frac{\mathcal{E}(x : T) \vdash M : S}{\mathcal{E} \vdash (\lambda x.M) : T \to S}$$

We use the symbol $\vdash_t$ to denote derivability in the tree assignment systems. Clearly, if $\mathcal{E} \vdash_t M : T$ then T must be positive and regular. The above system behaves very much like the ordinary simple assignment system; in particular it is easy to show that $\mathcal{E} \vdash_t M : T$ implies $\mathcal{E} \vdash_t M' : T$, for all M' such that $M \twoheadrightarrow_{\beta\eta} M'$.

In order to find a translation from recursive type assignment to tree assignment, we need the following definition. Let τ be a type and w be a path. We define a type $\tau(w)$ as follows:

$$\begin{aligned}
\tau(\varepsilon) &= \tau;\\
(\tau \to \sigma)(Lw) &= \tau(w);\\
(\tau \to \sigma)(Rw) &= \sigma(w);\\
(\mu\alpha\tau)(w) &= \tau(w)[\mu\alpha\tau/\alpha], \text{ if } w \neq \varepsilon \text{ and } \tau(w) \text{ is defined};\\
(\mu\alpha\tau)(wv) &= (\mu\alpha\tau)(v), \text{ if } \tau[w] = \alpha.
\end{aligned}$$

The only possibility when two cases of the above definition can overlap is when $\tau[w] = \alpha$ and we want to define $(\mu\alpha\tau)(w)$. Thus, to see that the above definition is correct, it suffices to prove by induction w.r.t. τ that:

$$\text{if } \tau[w] = \alpha \text{ and } \alpha \text{ is free in } \tau \text{ then also } \tau(w) = \alpha,$$

(with no ambiguity possible). Then we obtain $(\mu\alpha\tau)(w) = \mu\alpha\tau$, either way.

The next lemma shows the properties of $\tau(w)$ which are most important for our needs. Note that in part (1) there is no problem with capturing names of bound variables (cf. Lemma 3.4.).

Lemma 4.1.

1. If $\tau(w)$ is defined then $\tau[\sigma/\alpha](w) = \tau(w)[\sigma/\alpha]$;
2. If $\tau[w] = \alpha$ then $\tau[\sigma/\alpha](wv) = \sigma(v)$;
3. If $\tau(wv)$ is defined then so is $\tau(w)$ and $\tau(wv) = \tau(w)(v)$;
4. If $\tau \sim \sigma$ and $\tau(w)$ is defined then so is $\sigma(w)$ and $\tau(w) \sim \sigma(w)$;
5. For each τ, w, there is a type τ', such that $\tau' \sim \tau$ and $\tau(w) = \tau'[w]$;
6. If w is negative then $\tau \not\sim \tau(w)$.

Proof:
The proof of (1) is by induction with respect to τ. In case of fixpoint, there is an inner induction w.r.t. the length of w. For (2), we proceed in a similar way, using part (1) of Lemma 3.4. Also part (3) is proved by the same induction pattern, and in the case of fixpoint we use parts (1) and (2) plus the following easy observation: if $\sigma[w]$ is defined then so is $\sigma(w)$.

Part (4) follows by induction w.r.t. the definition of $\sim$. For the base step ($\mu\alpha\tau \sim \tau[(\mu\alpha\tau)/\alpha]$), we exploit parts (1) and (2). Induction steps are easy, but for the case of fixpoint, we need to show (again by induction) that if $\tau[w] = \alpha$ and $\sigma \sim \tau$ then $\sigma[w] = \alpha$, provided α is free in τ.

To show part (5), we again proceed by induction w.r.t. the definition of $\tau(w)$, but the induction hypothesis must require in addition that no variable free in τ or τ' is bound in τ'. For the proof we use Lemma 3.3. and Lemma 3.4. Finally, part (6) is an immediate consequence of (5) and Lemma 3.6. □

Now, for an arbitrary type τ, we define its tree representation T_τ, by $T_\tau(w) = [\tau(w)]_\sim$. If we identify the equivalence class of a variable with the variable itself, then we can say that this definition is correct:

Lemma 4.2.

1. T_τ is a positive regular tree, for each type τ. (The set $\mathcal{L}$ of labels is the set of equivalence classes of non-variable types.)
2. If $\tau \sim \sigma$ then $T_\tau = T_\sigma$, and conversely.

Proof:
(1) The domain of T_τ is downward closed because of part (3) of Lemma 4.1. Then we can show by an easy induction on τ that T_τ is a binary tree (for each node there is either two sons or none) and that variables occur as labels of all leaves and only leaves. (Recall that we assumed no type $\sim$-equivalent to $\mu\alpha\alpha$ is allowed.) This tree is positive, because $\tau(w) \sim \tau(wv)$ contradicts parts (3) and (6) of Lemma 4.1. To see that T_τ is also regular, one proves by induction that for each type τ the set $Z_\tau = \{[\tau(w)]_\sim : w \in \{L, R\}^*\}$ is finite. Indeed, we have $Z_{\tau\to\sigma} \subseteq Z_\tau \cup Z_\sigma \cup \{[\tau \to \sigma]_\sim\}$, and $Z_{\mu\alpha\tau} \subseteq \{[\rho[\mu\alpha\tau/\alpha]]_\sim : [\rho]_\sim \in Z_\tau\}$.

(2) The "if" part is immediate from Lemma 4.1.(4). The converse follows from the fact that almost all corresponding labels in T_τ and T_σ are $\sim$-related, and from the following observation: $\tau \sim (\tau(L) \to \tau(R))$, provided the rhs is defined. □

Note that the equality of trees $T_\tau = T_\sigma$ is quite different than the relation $\approx$ of [4] (identical tree unfoldings), because the latter identifies more types than $\sim$.

If E is a type environment then by $\mathcal{T}_E$ we denote the corresponding tree environment, given by $\mathcal{T}_E(x) = T_{E(x)}$, for all x. The main result of this section is:

Proposition 4.1. If $E \vdash_\sim M : \tau$ then $\mathcal{T}_E \vdash_t M : T_\tau$.

Proof:
Induction w.r.t. the length of derivation, by cases depending on the last rule used. The base step (rule (Var)) is obvious. Case (App) and (Abs) follow because $T_{\sigma\to\tau} = T_\sigma \to T_\tau$ (recall that the labels need only be equal almost everywhere). Finally, case ($\sim$) is a consequence of Lemma 4.2.(2). □

The converse of Lemma 4.1. will follow from Theorem 5.1. An advantage of the tree assignment is demonstrated by the following example:

Example 4.1. There is no type τ such that $\vdash_\sim \lambda x.\, xx : \tau$.

Proof:
Suppose the contrary. By Proposition 4.1., there is a positive regular tree T, such that our tree assignment derives $\vdash \lambda x.\, xx : T$. Let S be the left subtree of T. One can easily see that we must have $S = S \to S'$, for some S', and thus all labels on the leftmost path in S are the same. This contradicts the positivity condition. □

Together with Lemma 2.2., the above proves that recursive types and quantificational polymorphism are orthogonal with respect to their typing power. Indeed, $\lambda x.\, xx$ is easily typable in System **F**.

5. Type Constraints

In this section we consider positive type constraints, in the spirit of [7]. If TConst is a fixed set of *type constants* then we define *simple types with constants* with help of the grammar:

$$\tau ::= \alpha \mid c \mid \tau \to \tau,$$

where the metavariables α and c range over type variables and type constants, respectively. The subtype notation $\tau[w]$ is used also for types with constants and has the obvious meaning. A *type constraint* is an equation of the form "$c = \tau$", where $c \in$ TConst and τ is a simple type with constants. A *system of constraints* is a finite set of constraints such that all the left-hand sides are different constants.

A system of constraints $\mathcal{C}$ determines an equivalence relation $\sim_\mathcal{C}$ as the smallest congruence satisfying $c \sim_\mathcal{C} \tau$, whenever "$c = \tau$" is in $\mathcal{C}$. Such a system $\mathcal{C}$ is *positive* iff, for all constants c, the equivalence $c \sim_\mathcal{C} \tau$ implies that $\tau[w] = c$ may only hold for positive w (Mendler's condition **P**). This positivity condition can be also presented in a more "syntactic" way. For this, consider the quasi-order $\prec_\mathcal{C}$, generated on the set of constants in $\mathcal{C}$ by the condition that $d \prec_\mathcal{C} c$ must hold whenever there is a constraint "$c = \tau$" in $\mathcal{C}$, such that $\tau[w] = d$, for some w. The following is Proposition 10 of Mendler's paper [7].

Lemma 5.1. (Mendler) A system $\mathcal{C}$ of constraints is positive iff, for every $c \in$ TConst, the equivalence class $[c]$ of the equivalence relation generated by the quasi-order $\prec_\mathcal{C}$ can be partitioned into two subsets, denoted $[c]^+$ and $[c]^-$, in such a way that $c \in [c]^+$, and whenever "$c = \tau$" is a constraint in $\mathcal{C}$ and $\tau[w] = d$ then:

— if $d \in [c]^+$ then w is positive, and

— if $d \in [c]^-$ then w is negative.

Type inference for simple types with constants is defined relative to a given positive set of constraints $\mathcal{C}$. The rules are the ordinary (Var), (App), (Abs) and in addition:

$$(\sim_{\mathcal{C}}) \qquad \frac{E \vdash M : \sigma, \quad \sigma \sim_{\mathcal{C}} \tau}{E \vdash M : \tau}$$

We use the symbol $\vdash_{\mathcal{C}}$ to denote type assignment with help of the system of constraints $\mathcal{C}$. It was shown by Mendler that typable terms have strong normalization property iff $\mathcal{C}$ is positive. An analogous result can be obtained (as shown by Marz, [6]) for generalized constraints, i.e., arbitrary equations of the form "$\tau = \sigma$", with an appropriate generalized notion of positivity.

We are particularly interested in derivations of a specific simple shape. We write $E \vdash^{\bullet}_{\mathcal{C}} M : \tau$ if $E \vdash_{\mathcal{C}} M : \tau$ can be derived so that rule $(\sim_{\mathcal{C}})$ is used only at the "top level", i.e., in one of the two restricted forms:

$$\frac{E \vdash M : c}{E \vdash M : \tau} \qquad \frac{E \vdash M : \tau}{E \vdash M : c}$$

where "$c = \tau$" is a constraint in $\mathcal{C}$.

Our aim is now to translate the tree assignment system of Section 4. into type inference with constraints. More precisely, we show the following result.

Proposition 5.1. If $\mathcal{E} \vdash_t M : T$ then there exists a system $\mathcal{C}$ of constraints such that $E \vdash^{\bullet}_{\mathcal{C}} M : \tau$, for some E and τ.

Proof:
For each positive regular tree T, let c_T be a new constant. Our system $\mathcal{C}$ of constraints consists of all equations of the form:

$$\begin{aligned} c_{T\to S} &= c_T \to c_S; \\ c_T &= \alpha, \quad \text{if } T(\varepsilon) = \alpha. \end{aligned}$$

Since there is only a finite number of trees used in any derivation, we actually have only a finite number of constants and thus $\mathcal{C}$ is finite. To show that $\mathcal{C}$ is positive, we first prove the following claim: if $\tau \sim_{\mathcal{C}} \sigma$ with $\tau[w] = c_T$ and $\sigma[wv] = c_S$ then $T|_v = S$. The proof of this claim is by induction w.r.t. the definition of $\sim_{\mathcal{C}}$. To avoid difficulties created by transitivity we note that if $(\tau_1 \to \tau_2) \sim_{\mathcal{C}} (\sigma_1 \to \sigma_2)$ then $\tau_1 \sim_{\mathcal{C}} \sigma_1$ and $\tau_2 \sim_{\mathcal{C}} \sigma_2$ (in a smaller number of steps). Now, if $c_T \sim_{\mathcal{C}} \tau$, and $\tau[w] = c_T$ then $T = T|_w$, and thus w must be positive, because almost all labels $T(w^n)$ must be equal.

Now assume that $\mathcal{E} \vdash_t M : T$, and define E so that $E(x) = c_S$, whenever $\mathcal{E}(x) = S$. We prove $E \vdash^{\bullet}_{\mathcal{C}} M : c_T$ by induction w.r.t. the number of steps needed to derive $\mathcal{E} \vdash_t M : T$. Note that rule $(\sim_{\mathcal{C}})$ is only used to replace $c_{T\to S}$ by $c_T \to c_S$ and conversely, and thus we have a $\vdash^{\bullet}_{\mathcal{C}}$ derivation indeed. □

Our last step is from constraints back to recursive types. For this it is convenient to extend our language so that both constants and fixpoints may occur in types. Thus, we allow for all types constructed according to the following grammar:

$$\tau ::= \alpha \mid c \mid \tau \to \tau \mid \mu\alpha\tau,$$

with only positive μ-bindings. We generalize appropriately the notion of a constraint and for a given system of constraints $\mathcal{C}$, we redefine the relation $\sim_{\mathcal{C}}$ as the smallest congruence satisfying both conditions:

$$\begin{aligned} \mu\alpha\tau &\sim_{\mathcal{C}} \tau[(\mu\alpha\tau)/\alpha]; \\ c &\sim_{\mathcal{C}} \tau, \qquad \text{for "}c = \tau\text{" in } \mathcal{C}. \end{aligned}$$

Of course, $\mathcal{C}$ is always assumed to be positive. The meaning of "$E \vdash_{\mathcal{C}} M : \tau$" for the extended language should now be clear. We also use the notation $E \vdash^{\bullet}_{\mathcal{C}} M : \tau$ if the judgement $E \vdash M : \tau$ is derivable with help of rules (Var), (App), (Abs), (Fold), (Unfold), and the two restricted forms of ($\sim_{\mathcal{C}}$), as above. Note that $\vdash_{\emptyset}$ is just $\vdash_{\sim}$, and that $\vdash^{\bullet}_{\emptyset}$ is equivalent to $\vdash_{FU}$. We are going to eliminate one constant at a time. The main technical lemma is as follows:

Lemma 5.2. Let $\mathcal{C}$ be a positive system of n constraints ($n > 0$), and assume that $E \vdash_{\mathcal{C}} M : \sigma$. There exists a positive system $\mathcal{D}$ of $n-1$ constraints, such that $E' \vdash_{\mathcal{D}} M : \sigma'$, for some E' and σ'. In addition, if $E \vdash^{\bullet}_{\mathcal{C}} M : \sigma$ then $E' \vdash^{\bullet}_{\mathcal{D}} M : \sigma'$.

Proof:
Choose a constraint "$c = \tau$" from $\mathcal{C}$. Now, for every type ρ, let $\rho[(\mu\alpha\tau[\alpha/c])/c]$ be denoted by ρ'. This is correct, because $\mathcal{C}$ is positive, and thus c cannot occur negatively in τ. The new system $\mathcal{D}$ is obtained from $\mathcal{C}$ by removing "$c = \tau$", and replacing every other equation "$d = \rho$" by "$d = \rho'$". To see that $\mathcal{D}$ is positive, one proves that the equivalence relation determined by $\prec_D$ is contained in that determined by $\prec_C$, and that the partition of equivalence classes given by Lemma 5.1. remains correct after removing c.

On the other hand we can prove by induction that $\sigma_1 \sim_C \sigma_2$ implies $\sigma'_1 \sim_D \sigma'_2$. In particular, c' is $\mu\alpha\tau[\alpha/c]$ and τ' is $\tau[(\mu\alpha\tau[\alpha/c])/c]$, the latter equal to $\tau[\alpha/c][(\mu\alpha\tau[\alpha/c])/\alpha]$.)

The hypothesis of the lemma is now shown by induction w.r.t. the number of steps to derive $E \vdash_{\mathcal{C}} M : \sigma$. In case of $E \vdash^{\bullet}_{\mathcal{C}} M : \sigma$, we have just to note that a "top level" application of the constraint "$c = \tau$" becomes replaced by an application of either (Fold) or (Unfold).□

The above lemma applied repeatedly allows one to get rid of all constants and constraints at the cost of introducing μ's. Let us note here that the result of this process is not unique and depends on the order in which the constants are eliminated. (Take the constrains "$c = d \to c$" and "$d = c \to d$" as an example.)

We have completed the loop connecting our systems, and we can now state our main result: each of them is of the same power w.r.t. typability of pure lambda terms.

Theorem 5.1. For every pure lambda term M, the following conditions are equivalent:

1. $E \vdash_{FU} M : \tau$, for some E and τ;
2. $E \vdash_{\sim} M : \tau$, for some E and τ;
3. $\mathcal{E} \vdash_t : T$, for some $\mathcal{E}$ and T;
4. $E \vdash_{\mathcal{C}} : \tau$, for some positive system $\mathcal{C}$, and some E and τ.

Proof:
That (1) implies (2) is an obvious inclusion. The implication from (2) to (3) follows from Proposition 4.1., and the implication from (3) to (4) is given by Proposition 5.1. Condition (4) implies (2) because by Lemma 5.2., we can eliminate step by step all occurrences of constants, resulting in a derivation of the form $E' \vdash_{\emptyset} M : \tau'$. Finally, (3) implies (1) because Proposition 5.1. guarantees a derivation in the special form $E \vdash^{\bullet}_{\mathcal{C}} M : \tau$, and the elimination process of Lemma 5.2. will result in $E' \vdash^{\bullet}_{\emptyset} M : \tau'$. □

Corollary 5.1. It is decidable whether a given term M is typable with respect to $\vdash_{\sim}$.

Proof:
Due to the equivalence of conditions (1) and (3) of Theorem 5.1., we ask whether M is typable in the tree assignment system. (Compare this to the use of arbitrary regular trees in [4] to prove properties of type assignment with unrestricted recursive types.) First we follow essentially a standard algorithm for ordinary finite types. We assign an unknown t_P to each subterm P of M (with different occurrences of non-variable subterm counted as different subterms). Then we set equations of the form: $t_P = t_Q \to t_{PQ}$ and $t_{\lambda x.P} = t_x \to t_P$. It remains to solve these equations over positive regular trees. It is not difficult to show that a solution exists iff our equations form a positive system of constraints, and this is decidable in polynomial time. □

6. Conclusion

We have shown the equivalence of typability under different type disciplines involving the combination of simple types (arrow types) and recursion. We have also constructed an equivalent tree model of positive recursive types. We would like to extend these results to systems involving quantificational polymorphism. However, the generalization will not be immediate. An example difficulty is as follows: a type of the form $\forall\alpha\tau$, with α occurring free within a μ-binding, must be represented by a tree with an infinite number of occurrences of α. If equality of trees is taken almost everywhere, then the tree corresponding to $\tau[\rho/\alpha]$ is not well-defined.

References

[1] Barendregt, H.P.: "Lambda calculi with types", Chapter 1 in: S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum (eds.), *Handbook of Logic in Computer Science*, vol **2**, Oxford: Clarendon Press, 1992, 118–310.

[2] Breazu-Tannen, V. and Meyer, A.R.: "Lambda calculus with constrained types", R. Parikh (ed.), *Proc. Logics of Programs*, LNCS 193, Berlin, Springer-Verlag, 1985, 23–40.

[3] Cardone, F. and Coppo, M.: "Two extensions of Curry's type inference system", P. Oddifreddi (ed.), *Logic and Computer Science*, London: Academic Press, 1990, pp. 19–75.

[4] Cardone, F. and Coppo, M.: "Type inference with recursive types: syntax and semantics", *Information and Computation*, **92**(1),1991, 48–80.

[5] Klop, J.W.: *Combinatory Reduction Systems*, Amsterdam: Mathematisch Centrum, 1980.

[6] Marz, M.: "An algebraic view on recursive types", manuscript, Technische Hochschule Darmstadt, 1995.

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

[8] Urzyczyn, P.: "Positive recursive type assignment", J.Wiedermann and P. Hájek (eds.), *Proc. MFCS 1995*, LNCS 969, Berlin, Springer-Verlag, 1995, 382–391.