SCHEDAE INFORMATICAE
VOLUME 12 2003

Defining Recursors by Solving Equations in Second-Order Lambda Calculus

Zdzisław Spławski
Faculty of Informatics and Management, Wrocław University of Technology, Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland
e-mail: splawski@ci.pwr.wroc.pl
and *Institute of Computer Science, Wrocław University, Przesmyckiego 20, 51-151 Wrocław*

Abstract. Positive recursive (fixpoint) types can be added to the polymorphic (Church-style) lambda calculus $\boldsymbol{\lambda 2}$ (System $\mathbf{F}$) in several different ways, depending on the choice of the elimination operator. Known extensions of $\boldsymbol{\lambda 2}$ fall into two equivalence classes with respect to mutual interpretability by means of beta-eta reductions, and elimination operators for fixpoint types can be classified accordingly as either "iterators" or "recursors". Systems with iterators can be defined within $\boldsymbol{\lambda 2}$ by means of beta reductions, and it is conjectured that systems with recursors cannot.

In this paper we define the general form of mutual iteration scheme in $\boldsymbol{\lambda 2}$ and we show that the explicit solution for particular functions defines recursors within $\boldsymbol{\lambda 2}$, though proof of this fact requires much more than beta reductions, namely parametricity. We propose a convenient *equational* inference rule which can be used instead of parametricity for proving equational properties of polymorphic functions, defined by iterators.

Keywords: Typed lambda calculus, inductive definitions, equational reasoning.

1. Introduction

In [11] we addressed the question of the interpretability of positive recursive types within the polymorphic lambda calculus $\boldsymbol{\lambda 2}$, known also as System $\mathbf{F}$. Polymorphic lambda calculus has been discovered independently by Girard [3] and Reynolds [9]. It has long been known, see e.g. [1, 4], that various recursively defined data types can be defined within $\boldsymbol{\lambda 2}$ with help of beta reductions. That is, for every $\mu\alpha.\tau$ (all occurences of α in τ must be positive) 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 $\boldsymbol{\lambda 2}\mathbf{I}$ of recursive types for which strong normalization was proved in Mendler's journal article [7]. (Thus, the strong normalization result of [7] may also be obtained by translation to $\boldsymbol{\lambda 2}$.)

The names "iterative" and "recursive" were suggested by the difference between *iteration* and *recursion* over natural numbers. Recall that Gödel's system $\mathbf{T}$ is typically defined with a *recursor*

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

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

$$\left\{ \begin{array}{rcl} \mathbf{R}_\sigma MN\mathbf{0} & \Rightarrow & N \\ \mathbf{R}_\sigma MN(\mathbf{S}\,k) & \Rightarrow & M\langle k, \mathbf{R}_\sigma MNk\rangle \end{array} \right.$$

An alternative is to choose the *iterator*

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

with the reduction rules

$$\left\{ \begin{array}{rcl} \mathbf{It}_\sigma MN\mathbf{0} & \Rightarrow & N \\ \mathbf{It}_\sigma MN(\mathbf{S}\,k) & \Rightarrow & M(\mathbf{It}_\sigma MNk) \end{array} \right.$$

In a sense, these variants of system $\mathbf{T}$ 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 $\mathbf{It}_{\mathbf{Nat}\times\sigma}$ 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.

It is well known (see e.g. [4]) that natural numbers with a polymorphic iterator can be defined in $\boldsymbol{\lambda 2}$ as Church numerals. It is also known how to define recursion for natural numbers (and for some other algebraic types) in terms of iteration. Unfortunately, the second reduction rule for the defined recursor can be proved in $\boldsymbol{\lambda 2}$ only "pointwise", i.e. for concrete numerals and again a single reduction step is simulated by possibly many steps. In $\boldsymbol{\lambda 2}$ exactly the integer functions that are provably total in second order Peano arithmetic are representable.

Below we show how to find the definition of a recursor for natural numbers in terms of an iterator by solving a system of two functional equations, given by a mutual iteration schema.

To find explicit solutions for a function $f : \mathbf{Nat} \to \sigma$ given by the iteration scheme:

$$\begin{cases} f(0) &= g \\ f(n+1) &= h(f(n)) \end{cases}$$

or the primitive recursion scheme:

$$\begin{cases} f(0) &= g \\ f(n+1) &= h\langle n, f(n)\rangle \end{cases}$$

where g and h are known functions of appropriate types, we define: $f \equiv \lambda n.\mathbf{It}_\sigma\, h\, g\, n$ and $f \equiv \lambda n.\mathbf{R}_\sigma\, h\, g\, n$, respectively.

The mutual iteration scheme for functions $f_1 : \mathbf{Nat} \to \sigma_1$ and $f_2 : \mathbf{Nat} \to \sigma_2$ looks as follows:

$$\begin{cases} f_1(\mathbf{0}) &= g_1 \\ f_1(\mathbf{S}\, n) &= h_1\langle f_1 n, f_2 n\rangle \\ f_2(\mathbf{0}) &= g_2 \\ f_2(\mathbf{S}\, n) &= h_2\langle f_1 n, f_2 n\rangle \end{cases}$$

To eliminate mutual iteration we can define $f_1 \equiv \mathbf{fst} \circ F$, $f_2 \equiv \mathbf{snd} \circ F$, where $F : \mathbf{Nat} \to \sigma_1 \times \sigma_2$ is defined by the iteration scheme:

$$\begin{cases} F(\mathbf{0}) &= \langle g_1, g_2\rangle \\ F(\mathbf{S}\, n) &= (\lambda z^{\sigma_1 \times \sigma_2}.\langle h_1 z, h_2 z\rangle)\,(F\, n) \end{cases}$$

which can be solved explicitly:

$F \equiv \mathbf{It}_{\sigma_1 \times \sigma_2}\,(\lambda z^{\sigma_1 \times \sigma_2}.\langle h_1\, z, h_2\, z\rangle)\,\langle g_1, g_2\rangle$.

Now, notice that for $\sigma_1 \equiv \mathbf{Nat}$, $g_1 \equiv \mathbf{0}$, and $h_1 \equiv \mathbf{S}\circ\mathbf{fst}$ function f_1 defines identity on $\mathbf{Nat}$. Substituting identity for f_1 in the defining equations for f_2 we see that they turn to the primitive recursion scheme. This method of reducing recursion to iteration was used in [2]. Abstracting with respect to all (also type) variables we can define[1]:

$\mathbf{R} \equiv \Lambda\beta.\lambda h.\lambda g.\lambda n.\mathbf{snd}\,(\mathbf{It}\,(\mathbf{Nat} \times \beta)(\lambda z^{\mathbf{Nat}\times\beta}.\langle \mathbf{S}\,(\mathbf{fst}\, z), h\, z\rangle)\,\langle \mathbf{0}, g\rangle\, n)$.

[1]Remember that product type with both projections is definable in $\boldsymbol{\lambda 2}$.

Below we will use this method to define in $\boldsymbol{\lambda 2}$ recursors for any truly recursive extension of $\boldsymbol{\lambda 2}$.

2. Two fixpoint extensions of $\lambda 2$

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.

2.1. System $\boldsymbol{\lambda 2}$J

In system $\boldsymbol{\lambda 2}$**J** we have $\mathbf{in}_{\mu\alpha.\tau} : \tau[\mu/\alpha] \to \mu\alpha.\tau$, with an eliminator of type:

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

and a reduction rule:

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

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 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:
- $p_\alpha(\alpha) = 1$;
- $p_\alpha(\sigma \to \rho) = 1 + \max(p_\alpha(\sigma), p_\alpha(\rho))$;
- $p_\alpha(\forall\beta\sigma(\beta,\alpha)) = 1 + p_\alpha(\sigma(\beta,\alpha))$;
- $p_\alpha(\mu\beta.\sigma(\beta,\alpha)) = 1 + \max(p_\alpha(\sigma(\beta,\alpha)), p_\beta(\sigma(\beta,\alpha)))$[2].

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
   $\mathcal{M}^\tau_{\varphi,\psi} \equiv \mathcal{A}^\tau_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\,y^\tau.y$.

[2]For $\boldsymbol{\lambda 2}$**J** 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 $\boldsymbol{\lambda 2}$**Rec** below.

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 $\tau = \forall\psi.\sigma(\psi,\alpha)$ we define

$\mathcal{M}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\forall\psi.\sigma(\psi,\varphi)}\Lambda\psi.\mathcal{M}^{\sigma(\psi,\alpha)}_{\varphi,\psi} z(y\psi)$, and

$\mathcal{A}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\forall\psi.\sigma(\psi,\psi)}\Lambda\psi.\mathcal{A}^{\sigma(\psi,\alpha)}_{\varphi,\psi} z(y\psi)$.

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:

$\mathcal{M}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\mu_1}.\mathbb{J}_{\mu_1}(\mu_2)(\lambda x^{\sigma(\mu_2,\varphi)}.\mathbf{in}_{\mu_2}(\mathcal{M}^{\sigma(\mu_2,\alpha)}_{\varphi,\psi} zx))y$,

$\mathcal{A}^{\tau}_{\varphi,\psi} \equiv \lambda z^{\varphi\to\psi}\, y^{\mu_2}.\mathbb{J}_{\mu_2}(\mu_1)(\lambda x^{\sigma(\mu_1,\psi)}.\mathbf{in}_{\mu_1}(\mathcal{A}^{\sigma(\mu_1,\alpha)}_{\varphi,\psi} zx))y$.

In each case the parameter p_α decreases and our definition is well-founded.

2.2. System $\boldsymbol{\lambda}$2Rec

The recursor and the reduction scheme for **$\boldsymbol{\lambda}$2Rec** are as follows:

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

The symbol $\mathcal{M}^{\tau}_{\mu,\mu\times\sigma}$ stands here again for an appropriately defined "lifting" combinator, which for cases (1) – (4) is defined as in **$\boldsymbol{\lambda}$2J**, 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

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

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

Notice that in the definition of $\mathcal{M}^{\mu\beta.\sigma(\beta,\alpha)}$ we use combinators $\mathcal{M}^{\sigma}$ defined with respect to α and to β. The difference between **Rec** and $\mathbb{J}$ is similar to the difference between **R** in Gödel's system **T** and the iterator **It** that we discussed in the Introduction.

3. Interpretation of $\boldsymbol{\lambda}$2Rec in $\boldsymbol{\lambda}$2J

Now we shall use the method by which we found in the Introduction the definition of a recursor for natural numbers in terms of iterator to interpret **$\boldsymbol{\lambda}$2Rec** in **$\boldsymbol{\lambda}$2J**. The mutual iteration scheme in **$\boldsymbol{\lambda}$2J** for functions $f_1 : \mu \to \sigma_1$ and $f_2 : \mu \to \sigma_2$ looks as follows:

$$\begin{cases} f_1(\mathbf{in}_\mu\, x) &= h_1\,(\mathcal{M}^\tau_{\mu,\sigma_1\times\sigma_2}\,(\lambda z^\mu.\langle f_1\, z, f_2\, z\rangle)\, x) \\ f_2(\mathbf{in}_\mu\, x) &= h_2\,(\mathcal{M}^\tau_{\mu,\sigma_1\times\sigma_2}\,(\lambda z^\mu.\langle f_1\, z, f_2\, z\rangle)\, x) \end{cases}$$

To eliminate mutual iteration we can define $f_1 \equiv \mathbf{fst} \circ F$ and $f_2 \equiv \mathbf{snd} \circ F$, where $F : \mu \to \sigma_1 \times \sigma_2$ is defined by the iteration scheme:

$F(\mathbf{in}_\mu\, x) = (\lambda z^{\sigma_1\times\sigma_2}.\langle h_1\, z, h_2\, z\rangle)\,(\mathcal{M}^\tau_{\mu,\sigma_1\times\sigma_2}\, F\, x)$

which can be solved explicitly:

$F \equiv \mathbb{J}_\mu(\sigma_1 \times \sigma_2)(\lambda z^{\tau(\sigma_1\times\sigma_2)}.\langle h_1\, z, h_2\, z\rangle)$.

Now, take $\sigma_1 \equiv \mu$, $h_1^{\tau(\mu\times\sigma_2)\to\mu} \equiv \mathbf{in}_\mu \circ (\mathcal{M}^\tau_{\mu\times\sigma_2,\mu}\, \mathbf{fst})$ and define (abstracting w.r.t. all variables):

$\mathbf{Rec}_\mu \equiv \Lambda\beta\lambda y^{\tau(\mu\times\beta)\to\beta}\lambda z^\mu.\mathbf{snd}\,(\mathbb{J}_\mu\,(\mu\times\beta)\,(\lambda v^{\tau(\mu\times\beta)}.\langle \mathbf{in}_\mu(\mathcal{M}^\tau_{\mu\times\beta,\mu}\,\mathbf{fst}\, v), y\, v\rangle)\, z)$.

To prove properties of $\mathbf{Rec}_\mu$ one needs parametric polymorphism [10] as formalized e.g. in [8]. But it is also possible to do it using the following uniqueness rule (U):

$$\frac{\Gamma, x : \tau[\mu/\alpha] \vdash F\,(\mathbf{in}_\mu\, x) = M\,(\mathcal{M}^\tau_{\mu,\sigma}\, F\, x)}{\Gamma \vdash F = \lambda y^\mu.\mathbb{J}_\mu \sigma\, M\, y}\ (U)$$

together with the equation $\langle \mathbf{fst}\, N, \mathbf{snd}\, N\rangle = N$ for N of appropriate type.

The uniqueness rule is much simpler than parametricity. It is equational, hence easier to implement in proof systems. For proofs of equality between two functions the following equivalent rule could be more useful:

$$\frac{\begin{array}{c}\Gamma, x : \tau[\mu/\alpha] \vdash F\,(\mathbf{in}_\mu\, x) = M\,(\mathcal{M}^\tau_{\mu,\sigma}\, F\, x) \\ \Gamma, x : \tau[\mu/\alpha] \vdash G\,(\mathbf{in}_\mu\, x) = M\,(\mathcal{M}^\tau_{\mu,\sigma}\, G\, x)\end{array}}{\Gamma \vdash F = G}\ (U'),$$

where $x \notin FV(M)$.

4. Interpretability of $\boldsymbol{\lambda}$2 extensions by type fixpoints

Mendler [6] introduced another extension of $\boldsymbol{\lambda}\mathbf{2}$, which we call $\boldsymbol{\lambda}\mathbf{2R}$. One more extension by retract types, called by us $\boldsymbol{\lambda}\mathbf{2U}$, can be obtained by adding to $\boldsymbol{\lambda}\mathbf{2}$ two operators: $\mathbf{Fold} : \sigma[\mu\alpha.\sigma/\alpha] \to \mu\alpha.\sigma$ and $\mathbf{Unfold} : \mu\alpha.\sigma \to \sigma[\mu\alpha.\sigma/\alpha]$ with the reduction rule: $\mathbf{Unfold}\,(\mathbf{Fold}\,M) \Rightarrow M$.

If we write $\preceq_\beta$ and $\preceq_{\beta\eta}$ to denote, respectively, beta and beta-eta interpretability, then the results reported in [11] can be symbolically presented as follows. Iterators are just syntactic sugar: $\boldsymbol{\lambda}\mathbf{2} \subseteq \boldsymbol{\lambda}\mathbf{2I}, \boldsymbol{\lambda}\mathbf{2J} \preceq_\beta \boldsymbol{\lambda}\mathbf{2}$.

The three systems with recursors are definable in each other by means of beta-eta reductions: $\boldsymbol{\lambda}\mathbf{2R}, \boldsymbol{\lambda}\mathbf{2Rec} \preceq_{\beta\eta} \boldsymbol{\lambda}\mathbf{2U} \preceq_{\beta\eta} \boldsymbol{\lambda}\mathbf{2R}, \boldsymbol{\lambda}\mathbf{2Rec}$.

System $\boldsymbol{\lambda}\mathbf{2U}$ cannot be defined within $\boldsymbol{\lambda}\mathbf{2}$ by means of beta reductions: $\boldsymbol{\lambda}\mathbf{2U} \not\preceq_\beta \boldsymbol{\lambda}\mathbf{2}$, and we conjectured that $\boldsymbol{\lambda}\mathbf{2U} \not\preceq_{\beta\eta} \boldsymbol{\lambda}\mathbf{2}$.

Since $\boldsymbol{\lambda}\mathbf{2J} \preceq_\beta \boldsymbol{\lambda}\mathbf{2}$ we may conclude that $\boldsymbol{\lambda}\mathbf{2Rec}$ is also definable in $\boldsymbol{\lambda}\mathbf{2}$ using the uniqueness rule. But $\boldsymbol{\lambda}\mathbf{2Rec}$, $\boldsymbol{\lambda}\mathbf{2R}$ and $\boldsymbol{\lambda}\mathbf{2U}$ are mutually beta-eta interpretable, hence $\boldsymbol{\lambda}\mathbf{2R}$ and $\boldsymbol{\lambda}\mathbf{2U}$ are definable in $\boldsymbol{\lambda}\mathbf{2}$ using the uniqueness rule, as well.

5. Conclusions

In this paper we defined the general form of the mutual iteration scheme in $\boldsymbol{\lambda}\mathbf{2J}$ and we found explicit solutions in $\boldsymbol{\lambda}\mathbf{2J}$ for functions defined by this scheme. Since $\boldsymbol{\lambda}\mathbf{2J}$ is beta interpretable in $\boldsymbol{\lambda}\mathbf{2}$, we may conclude that our constructions were carried out in $\boldsymbol{\lambda}\mathbf{2}$. We showed that the explicit solution for particular functions defines recursors within $\boldsymbol{\lambda}\mathbf{2}$. We proposed a convenient equational inference rule which can be used instead of parametricity for proving equational properties of polymorphic functions, defined by iterators.

Similar method can be used to solve a system of functions defined by the mutual coiterator scheme and to build corecursor by coiterator.

It is known that recursors can be defined by iterators, and examples can be found in literature for concrete algebraic types. However, we have not found the construction for general case. The same applies to reducing the mutual iteration scheme to single iteration.

Polymorphic lambda calculus has been proposed as a "programming language" e.g. in [5]. All our constructions are algoritmizable, which makes it possible to "program" in $\boldsymbol{\lambda}\mathbf{2}$, using more familiar ways of defining functions and constructing equational proofs of equational properties.

6. References

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

[2] Böhm C.; *Reducing Recursion to Iteration by Means of Pairs and N-tuples*, in: M. Boscarol et al., (eds.), *Foundations of Logic and Functional Programming*, Lecture Notes in Computer Science, Vol. 306, Springer–Verlag, Berlin 1988, pp. 58–66.

[3] Girard J.-Y.; *Une extension de l'interpretation de Gödel á l'analyse, et son application á l'èlimination de coupures dans l'analyse et la thèorie des types*, in: J.E. Fenstad, (ed.), *Proceedings of the Second Scandinavian Logic Symposium*, North-Holland, Amsterdam 1971, pp. 63–92.

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

[5] Krivine J.-L., Parigot M.; *Programming with proofs*, Journal of Information Processing and Cybernetics, 26(3), 1990, pp. 149–167.

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

[7] Mendler N.P.; *Inductive types and type constraints in the second-order lambda calculus*, Annals of Pure and Applied Logic, 51(1/2), 1991, pp. 159–172.

[8] Plotkin G., Abadi M.; *A Logic for Parametric Polymorphism*, in: M. Bezem, J.F. Groote, (eds.), *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, Vol. 664, Springer–Verlag, Berlin 1993, pp. 361–375.

[9] Reynolds J.C.; *Towards a theory of type structure*, in: B. Robinet, (ed.), *Colloque sur la Programmation*, Lecture Notes in Computer Science, Vol. 19, Springer–Verlag, 1974, pp. 408–425.

[10] Reynolds J.C.; *Types, Abstraction and Parametric Polymorphism*, in: R.E.A. Mason, (ed.), *Information Processing 83*, North-Holland, Amsterdam 1983, pp. 513–523.

[11] Spławski Z., Urzyczyn P.; *Type Fixpoints: Iteration vs. Recursion*, in: *Proc. ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, 34(9), ACM Press, 1999, pp. 102–113.

Received November 18, 2002