Math. Struct. in Comp. Science (1999), *vol.* 9, *pp.* 507–522. Printed in the United Kingdom
© 1999 Cambridge University Press

Stratified polymorphism and primitive recursion†

NORMAN DANNER‡ and DANIEL LEIVANT§

‡*Indiana University, Bloomington, IN 47405, U.S.A.*
Email: ndanner@member.ams.org

‡*Indiana University, Bloomington, IN 47405, U.S.A.*
Email: leivant@cs.indiana.edu

Received 10 October 1998; revised 5 April 1999

Natural restrictions on the syntax of the second-order (*i.e.*, polymorphic) lambda calculus are of interest for programming language theory. One of the authors showed in Leivant (1991) that when type abstraction in that calculus is stratified into levels, the definable numeric functions are precisely the super-elementary functions (level $\mathscr{E}_4$ in the Grzegorczyk Hierarchy). We define here a second-order lambda calculus in which type abstraction is stratified to levels up to ω^ω, an ordinal that permits highly uniform (and finite) type inference rules. Referring to this system, we show that the numeric functions definable in the calculus using ranks $< \omega^\ell$ are precisely Grzegorczyk's class $\mathscr{E}_{\ell+3}$ ($\ell \geqslant 1$). This generalizes Leivant (1991), where this is proved for $\ell = 1$. Thus, the numeric functions definable in our calculus are precisely the primitive recursive functions.

1. Introduction

The second-order (*i.e.*, polymorphic) lambda calculus $\mathbf{2}\boldsymbol{\lambda}$, due to Girard and Reynolds (Girard 1972; Reynolds 1974), has emerged as an important and powerful theoretical framework for functional programming languages and for formal methods in program specification and development. However, its abstraction power suffices by itself (without using recursors or fixpoint operators) to define all provably recursive functions of second-order arithmetic, thus far exceeding the computational complexity relevant to computer science. One is naturally led to a quest for natural restrictions on $\mathbf{2}\boldsymbol{\lambda}$ that would preserve its usefulness as a foundational tool in programming languages, by remaining computationally strong and with a finitary and streamlined syntax. Much of the strength of $\mathbf{2}\boldsymbol{\lambda}$ arises from the self-referential (sometimes called 'impredicative') use of the second-order quantifier: a term of type $\forall R.\tau$ can be applied to the type $\forall R.\tau$ itself. Techniques for blocking this kind of circularity have been analyzed extensively in metamathematics, notably in regard to Predicative Analysis and Ramified Second-Order Logic. It is natural to adapt to $\mathbf{2}\boldsymbol{\lambda}$ the notion of stratification developed there. In the finitely stratified variant of

† The research reported in this paper was partially supported by NSF grants CCR-9309824 and DMS-9870320. The proof figures were made with Makoto Tatsuta's `proof.sty` package, version 3.0.

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

2λ, introduced by Statman (Statman 1981) and independently by Leivant (Leivant 1989; Leivant 1991), type variables are indexed by natural numbers acting as 'ranks', and this indexing is extended to all types in such a way that a type involving quantification over a rank k variable is assigned a rank $> k$. Since rank-k quantifiers are interpreted to range over types of rank $\leqslant k$, the self-referential typing above is prevented. Whereas **2λ** captures the provably total functions of second-order arithmetic, Leivant (1991) showed that the finitely stratified variant of **2λ** captures exactly the super-elementary functions, that is, class $\mathscr{E}_4$ in Grzegorczyk's hierarchy of primitive recursive functions.

In this paper, we generalize the latter definability result by extending stratification to transfinite ranks $< \omega^\omega$. The key idea is to view quantification at limit ranks cumulatively, in the same spirit as the limit generalization rules in Feferman (1964) and Schütte (1977). For example, a term E may be assigned the type $\forall R : \omega.\sigma$ provided it can be assigned type σ when the type variable R is assigned an arbitrary rank $< \omega$ (the precise sense of this will be clear from the formal definitions below). This view of limit-abstraction forces us to use a semantic (Curry-style) type assignment system, as opposed to the ontological (Church-style) type system used in Leivant (1991).

Transfinite stratification buys us the ability to repeat the process of definition by iteration. Suppose that a function f is represented by the λ-term F of type $\nu_0 \to \nu_0$, where $\nu_\alpha =_{\mathrm{df}} \forall R : \alpha.(R \to R) \to (R \to R)$ is the rank-α type of the Church numerals. If g is defined from f by iteration, $g(x) =_{\mathrm{df}} f^{[x]}(x)$, then g is represented by the term $\lambda a.aFa$, which can be assigned the type $\nu_1 \to \nu_0$. However, since the input and output types of G are different, we cannot similarly type $\lambda a.aGa$. Transfinite ranks allow us to assign to G the type $\nu_\omega \to \nu_\omega$, which then permits the typing of $\lambda a.aGa$. Our main result (Theorem 4.8), announced in Leivant (1989), is that exactly the functions of $\mathscr{E}_{\ell+3}$ are definable using stratification ranks $< \omega^\ell$ ($\ell \geqslant 1$). Therefore, exactly the primitive recursive functions are definable using ranks $< \omega^\omega$ (Corollary 4.9).

A word is in order about our choice of ω^ω as the upper bound on ranks. To be feasibly implemented, a stratified type system must use simple notations for the ranks, and a finitary calculus of type inference for them. The ordinals up to ω^ω have such notations, in the guise of finite sequences of natural numbers, under the (reversed) lexicographic ordering. For instance, we shall use $(5, 0, 2, 3)$ as a notation for $\omega^3 \cdot 3 + \omega^2 \cdot 2 + 5$. Moreover, the simplicity of this notational system makes it possible to represent fundamental sequences of ordinals by simple symbolic expressions, enabling the definition of a finitary, simple, and highly uniform calculus of type assignment. It is satisfying that this stratification system leads to the definability of exactly the primitive recursive functions, which is a class far more natural than the class $\mathscr{E}_4$ corresponding to finite stratification.

In Leivant (1991) we announced further extensions of our present result to higher ordinals, but the exact definition of the type inference calculi needed is subtle, and the existence of a useful type inference for ranks of these transfinite heights is questionable. Restricting attention to simple ordinal notations is also relevant to programming language implementation. We comment further on these issues in our concluding section.

The plan of the paper is as follows. We start with a description of our ordinal notations and type assignment calculus. We then define in Section 3 our notion of numeric function definability, and show that the functions in $\mathscr{E}_{\ell+4}$ can be defined using notations for

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

ordinals $< \omega^\ell$. To show the converse implication, in Section 4, we start by defining functions that normalize our type assignment derivations, and analyze their complexity in the Grzegorczyk Hierarchy. Since function values can be extracted from type derivations for λ-expressions that define them, we can use the latter to conclude that the functions defined using only ranks $< \omega^\ell$ reside in Grzegorczyk's class $\mathscr{E}_{\ell+4}$. This concludes the proof of our main results, Theorem 4.8 and its Corollary 4.9. Finally, in Section 5 we suggest research directions of potential theoretical and practical interest.

2. A polymorphic lambda calculus with transfinite ramification

2.1. *Ordinal expressions*

We define a calculus **S2$\boldsymbol{\lambda}$** for a second-order lambda calculus with types stratified into ranks up to ω^ω. As notations for ordinals up to ω^ω we use finite sequences of natural numbers, where $(d_0, \ldots, d_k)$ denotes $\sum_{i=k\ldots 0} \omega^i \cdot d_i$. *Numeric expressions* $t, s, u, v, \ldots$ are generated from 0 and numeric variables $\mathsf{m}, \mathsf{n}, \ldots$ using the identifier S for successor. *Ordinal expressions* ξ are of the form $(t_0, \ldots, t_\ell)$, where t_i is a numeric expression; a closed expression (no variables) is an *ordinal notation*. Substituting 0 for the free number variables in an ordinal expression ξ yields its *associated ordinal notation* $\bar{\xi}$. We dub ℓ the *level*[†] of ξ; thus the finite ordinals are of level 0, and in general the ordinals $< \omega^k$ have notations of level $< k$. We write $(0^\ell, t_\ell, \ldots)$ for $(0, \ldots, 0, t_\ell, \ldots)$ with ℓ head 0's. Note that $(0^\ell, t)$ is a level-ℓ expression. We use ξ, η (sometimes with marks and scripts) for ordinal expressions, and ζ for ordinal notations.

If $\xi = (t_0, \ldots, t_k)$, then $S\xi$ will stand for $(St_0, \ldots, t_k)$. Note that ordinal expressions fall into four classes:

— (0^ℓ) (where $\ell \geqslant 1$), all denoting 0.
— Successors: $S\xi$.
— Limits: $(0^\ell, St, \vec{u})$, where $\ell > 0$. If v is a numeric expression, we write $(\xi|v)$ for the ordinal expression $(0^{\ell-1}, v, t, \vec{u})$; under the intended interpretation of the ordinal expressions, this corresponds to the v'th element of the standard fundamental sequence of ξ.
— Undetermined: $(0^\ell, \mathsf{n}, \vec{t})$, where $\ell \geqslant 0$.

The *order* on ordinal expressions is the transitive closure of all instances of

$$(t'_0, \ldots, t'_{\ell-1}, u, \vec{v}) < (t_0, \ldots, t_{\ell-1}, Su, \vec{v})$$
$$(t'_0, \ldots, t'_{\ell-1}, 0, \vec{v}) < (t_0, \ldots, t_{\ell-1}, Su, \vec{v}).$$

Note that this is a partial order. For instance, $(0, \mathsf{n})$ and $(S0)$ are incomparable.

Lemma 2.1.

1 If $\xi < \eta$, then $\xi^* < \eta^*$ for any substitution instances ξ^* and η^* of ξ and η, respectively.

[†] Note that the phrase 'level' is used here for ordinals, not for types. In Leivant (1991) the phrase 'level' is used for what we call 'rank' here.

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

2 If ξ is a limit and $\eta < \xi$, there is a numeric expression v, calculable in linear time from ξ and η, such that $\eta < (\xi|v)$.

2.2. *Types and statements*

We refer to a semantic (*i.e.*, Curry style) second-order lambda calculus ramified by ordinal expressions. We posit a countable collection of *type variables*. The *types* are generated by the following:

— Every type variable is a type.
— If σ and τ are types, then so is $\sigma \to \tau$.
— If τ is a type, R a type variable, and ξ an ordinal expression, then $\forall R : \xi.\, \tau$ is a type.

We use the usual conventions and abbreviations for types. In particular, $\to$ will associate to the right, parentheses are used in concrete syntax to override this convention, and we write $\sigma_1, \ldots, \sigma_k \to \tau$ for $\sigma_1 \to \ldots \to \sigma_k \to \tau$.

A *type assignment* is a phrase $E : \tau$, where E is a λ-expression and τ a type. A *rank assignment* is a phrase $R : \xi$ where R is a type variable and ξ an ordinal expression. A *type environment* is a finite set Γ of type assignments for distinct λ-variables. A *rank environment* is a finite set Θ of rank assignments for distinct type variables. An *environment* is the union of a type environment and a rank environment. We say that two (type-) [rank-] environments are *disjoint* if their domains are disjoint.

A *type statement* is a phrase $\Theta, \Gamma \vdash E : \tau$, where Γ is a type-environment that assigns a type for (at least) each λ-variable free in E, and Θ is a rank environment that assigns a rank for (at least) each type variable in the domain of Γ[†]. We use the common notational conventions for environments and type statements (for example, listing and writing Γ, Γ' for $\Gamma \cup \Gamma'$).

2.3. *Inferences*

To implement type stratification, we wish to assign ranks to types. However, since ranks of free type variables are stated in rank environments, we refer to pairs (Θ, τ) rather than types τ on their own. Also, since type expressions are not linearly ordered, we cannot assign a maximum to the ranks occurring in (Θ, τ), and consider instead the entire set of such ranks. Thus, we define

$$\begin{aligned} \underline{\mathrm{ord}}(\Theta, \tau) =_{\mathrm{df}} & \{\xi : R \text{ free in } \tau \text{ and } R : \xi \text{ is in } \Theta\} \\ & \cup \\ & \{S\xi : \tau \text{ has a subtype of the form } (\forall R : \xi).\sigma\ \}. \end{aligned}$$

The reason for using ordinal expressions as our complexity measure, rather than ordinal notations (which are well-ordered), is to allow a finitely presentable rule for introducing types whose complexity is a limit ordinal. Following Feferman's and Schütte's rules for

[†] Note that we incorporate here into the definition the properties of environments often referred to as *consistency* and *adequacy*.

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

limit generalization in Second-Order Arithmetic and Logic (Feferman 1964; Schütte 1977), one might wish to infer $\Theta, \Gamma \vdash E : (\forall R : \xi.\tau)$ (ξ a limit) from the infinitely many premises $\Theta, R : (\xi | S^k 0)$, $\Gamma \vdash E : \tau$ $(k = 0, 1, \ldots)$. We formulate instead a uniform variant of this rule, which can be expressed finitely by replacing the infinitely many premises $R : (\xi | S^k 0)$ with the single premise $R : (\xi | S^r \mathsf{m})$ (r fixed, m a numeric variable), from which all but a finite number of the closed premises can be inferred. Our *inference rules* are thus the following schemas. We stipulate everywhere that type statements are correct, as per the definition of type statements above.

Initial:

$$\Theta, \Gamma, x : \tau \vdash x : \tau$$

Object Abstraction (λI):

$$\frac{\Theta, \Gamma, x : \sigma \vdash E : \tau}{\Theta, \Gamma \vdash \lambda x.E : \sigma \to \tau}$$

Object Application (λE):

$$\frac{\Theta, \Gamma \vdash E : \sigma \to \tau \qquad \Theta', \Gamma' \vdash F : \sigma}{\Theta, \Theta', \Gamma, \Gamma' \vdash EF : \tau}$$

Type Abstraction: ($\forall$I):

$$\frac{\Theta, R : \xi, \Gamma \vdash E : \tau}{\Theta, \Gamma \vdash E : \forall R : \xi.\tau}$$

(R not free in the derived statement)

Limit Abstraction ($\forall \mathrm{I}^{\mathrm{lim}}$):

$$\frac{\Theta, R : (\xi | (S^r \mathsf{m})), \Gamma \vdash E : \tau}{\Theta, \Gamma \vdash E : \forall R : \xi.\tau}$$

(R and m not free in the derived statement, ξ a limit ordinal expression, and $r \geqslant 0$ is fixed)

The type variable R in the last two rules is dubbed the *eigen-variable* of the inference.

Type Application ($\forall$E):

$$\frac{\Theta, \Gamma \vdash E : (\forall R : \xi)\tau}{\Theta, \Theta', \Gamma \vdash E : \{\sigma / R\}\tau}$$

where σ, dubbed the *eigen-type* of the inference, is free for R in τ. This rule is subject to the following proviso:

If ξ is a successor or of the form (0^ℓ), then $\eta \leqslant \xi$ for every $\eta \in \underline{\mathrm{ord}}((\Theta, \Theta'), \sigma)$; if ξ is a limit or undetermined, then $\eta < \xi$ for every $\eta \in \underline{\mathrm{ord}}((\Theta, \Theta'), \sigma)$.

Note that Θ' is needed in order to assign ranks to free type variables in σ, and that by our stipulation on statements $\Theta \cup \Theta'$ is a rank environment, *i.e.*, with no type variable assigned different ranks.

We use Π and Σ to range over derivations. A *level-ℓ* derivation is one in which all ordinal expressions have level $\leqslant \ell$. Note that weakening (adding hypotheses) is a derived rule for this system, and we will often use this fact silently; for example, we may use a derivation Π of $\Theta, \Gamma, x : \sigma \vdash E : \tau$ when we have proved that Π is a derivation of $\Theta, \Gamma \vdash E : \tau$. The following lemma is immediate.

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

Lemma 2.2. Type derivations remain correct under simultaneous substitution of numeric expressions for numeric variables.

3. Function representation

3.1. *Definition of function representation*

For each $p \in \mathbb{N}$ we use $\bar{p}$ to denote the *Church numeral for p*, $\bar{p} =_{\mathrm{df}} \lambda s\, \lambda z.\, s^{[p]} z$, where the superscript $[p]$ stands for p-fold iteration. For each ordinal expression ξ, the *canonical* ξ *statement* for $\bar{p}$ is $\vdash \bar{p} : v_\xi$, where $v_\xi =_{\mathrm{df}} \forall R : \xi.(R \to R) \to (R \to R)$. For each p and ξ, let $\Delta_{p,\xi}$ be the standard (normal) derivation of $\vdash \bar{p} : v_\xi$.

Lemma 3.1. If $\Theta, \Gamma \vdash a : v_\xi$ is derivable, then $\Theta, \Gamma \vdash a : v_\eta$ is derivable whenever $\eta < \xi$.

Proof. Suppose Π is a derivation of $\Theta, \Gamma \vdash a : v_\xi$, and let R be a fresh type variable. Then the following is a derivation of $\Theta, \Gamma \vdash a : v_\eta$:

$$\dfrac{\dfrac{\begin{matrix}\Pi \\ \Theta, R : \eta, \Gamma \vdash a : v_\xi\end{matrix}}{\Theta, R : \eta, \Gamma \vdash a : (R \to R) \to (R \to R)}\ \forall \mathrm{E}}{\Theta, \Gamma \vdash a : v_\eta}\ \forall \mathrm{I}$$

□

We write

$$\dfrac{\Theta, \Gamma \vdash a : v_\xi}{\Theta, \Gamma \vdash a : v_\eta}\ \mathrm{Dcr}$$

for the derived inference rule justified by Lemma 3.1.

A function $f : \mathbb{N}^r \to \mathbb{N}$ is *skewly representable* (or, simply, *representable*) *(at level ℓ)* if there is a λ-term F that represents f in the (untyped) λ-calculus, and a level-ℓ type derivation of $\vdash F : v_{\xi_1}, \ldots, v_{\xi_r} \to v_\eta$ for some $\xi_1, \ldots \xi_r, \eta$; we abbreviate the latter as $\vdash F : \xi_1, \ldots, \xi_r \to \eta$. An r-ary function f is *step-down representable (at level ℓ)* if it is representable by some λ-term F for which there is a level-ℓ derivation of $\vdash F : \xi_1, \ldots, \xi_r \to \eta$, where $\xi_i = (0^\ell, S^{q+k_i}\mathsf{n})$ and $\eta = (0^\ell, S^q \mathsf{n})$, for some $q, k_1, \ldots, k_r \geqslant 0$ and numeric variable n. If $k_1 = \cdots = k_r = 0$, so that $\vdash F : \eta, \ldots, \eta \to \eta$, then f is *strictly representable (at level ℓ)*.

Lemma 3.2. If a unary function f is step-down representable at level ℓ, then it is strictly representable at level $\ell + 1$.

Proof. Suppose F represents f, and let Π be a level-ℓ derivation of $\vdash F : (0^\ell, S^{q+r}\mathsf{n}) \to (0^\ell, S^r \mathsf{n})$. Let Π' be obtained from Π by replacing every notation $(d_0, \ldots, d_\ell)$ in Π with $(d_0, \ldots, d_\ell, \mathsf{m})$ (where m is a fresh numeric variable). Π' is a correct derivation of $\vdash F : (0^\ell, S^{q+r}\mathsf{n}, \mathsf{m}) \to (0^\ell, S^r \mathsf{n}, \mathsf{m})$, because all ordinal expressions in Π are of level $\leqslant \ell$. Set $\xi =_{\mathrm{df}} (0^{\ell+1}, S\mathsf{m})$, whence $(\xi | u) = (0^\ell, u, \mathsf{m})$. In particular, Π' is a derivation of $\vdash F : (\xi | S^{q+r}\mathsf{n}) \to (\xi | S^r \mathsf{n})$. Then $\lambda a.Fa$ represents f, and the following is a derivation of

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

$\vdash \lambda a.Fa : \xi \to \xi$.

$$\dfrac{\dfrac{\dfrac{\Pi'}{\vdash F : v_{(\xi|S^{q+r}\mathsf{n})} \to v_{(\xi|S^{r}\mathsf{n})}} \quad \dfrac{a : v_\xi \vdash a : v_\xi}{a : v_\xi \vdash a : v_{(\xi|S^{q+r}\mathsf{n})}}\,\mathrm{Dcr}}{a : v_\xi \vdash Fa : v_{(\xi|S^{r}\mathsf{n})}}}{\dfrac{a : v_\xi, R : (\xi|S^{r-1}\mathsf{n}) \vdash Fa : (R \to R) \to R \to R}{a : v_\xi \vdash Fa : v_\xi}\,\forall^{\lim}\mathrm{I}}\,\forall\mathrm{E}$$
$$\vdash \lambda a.Fa : v_\xi \to v_\xi$$

□

Lemma 3.3. Suppose a unary function f is defined by $f(x) = g^{[x]}(2)$. If g is strictly representable at level ℓ, then f is step-down representable at level ℓ.

Proof. Suppose G represents g, and Π is a derivation of $\vdash G : v_\xi \to v_\xi$. Then $\lambda a.aG\bar{2}$ represents (in the untyped λ-calculus) f, with the following type derivation:

$$\dfrac{\dfrac{\dfrac{a : v_{S\xi} \vdash a : v_{S\xi}}{a : v_{S\xi} \vdash a : (v_\xi \to v_\xi) \to (v_\xi \to v_\xi)}\,\forall\mathrm{E} \quad \dfrac{\Pi}{a : v_{S\xi} \vdash G : v_\xi \to v_\xi}}{a : v_{S\xi} \vdash aG : v_\xi \to v_\xi} \quad \dfrac{\Delta_{2,\xi}}{a : v_{S\xi} \vdash \bar{2} : v_\xi}}{a : v_{S\xi} \vdash aG\bar{2} : v_\xi}$$
$$\vdash \lambda a.aG\bar{2} : v_{S\xi} \to v_\xi$$

□

3.2. *Representation of primitive recursive functions*

Let the functions G_r $(r \geqslant 0)$ be defined by $G_0(x, y) = x + y$, $G_1(x) = x^2$, $G_{r+1}(x) = G_r^{[x]}(2)$ $(r \geqslant 1)$. The *r'th Grzegorczyk class*, $\mathscr{E}_r$, consists of the closure under composition and bounded recurrence of the constant, projection and successor functions along with $G_0, \ldots, G_{r-1}$. $\mathscr{E}_3$ is the set of *elementary functions,* and $\mathscr{E}_4$ of *super-elementary functions.* $\bigcup_r \mathscr{E}_r$ is the set of primitive recursive functions. For details, see Rose (1984, Section 2.2)†.

Lemma 3.4. $G_{\ell+3}$ is step-down representable at ℓ $(\ell \geqslant 0)$.

Proof. Note, first, that G_0, G_1, and G_2 can be defined by composition from addition, multiplication and the exponential function $\exp_2 : x \mapsto 2^x$. These three functions are represented (in the untyped λ-calculus) by the terms $\lambda ab.\lambda sz.as(bsz)$, $\lambda ab.\lambda s.a(bs)$, and $\lambda a.a\bar{2}$, respectively, which are strictly representable at level 0 (see the Appendix).

We proceed to prove the lemma by induction on ℓ. As noted, G_2 is strictly representable at level 0, so by Lemma 3.3, G_3 is step-down representable at level 0, establishing the base case. For the induction step, assume that $G_{\ell+3}$ is step-down representable at level ℓ. By Lemma 3.2, $G_{\ell+3}$ is strictly representable at level $\ell + 1$, and so $G_{\ell+4}$ is step-down representable at level $\ell + 1$, by Lemma 3.3. □

Lemma 3.5. The functions step-down representable at level ℓ are closed under composition.

† In Rose (1984) $G_1(x)$ is defined as $x^2 + 2$ for a technical reason, but the classes are the same.

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

Proof. We show that if the unary functions h_0, h_1 and binary function f are step-down representable, then so is the function $g(x) =_{\mathrm{df}} f(h_0(x), h_1(x))$; the general case is similar.

Suppose that the functions above are represented by H_0, H_1, and F, respectively, with level-ℓ derivations for

$$\vdash H_i : (0^\ell, S^{k_i+q_i}\mathsf{n}) \to (0^\ell, S^{k_i}\mathsf{n}) \qquad (i = 0, 1)$$

and

$$\vdash F : (0^\ell, S^{p+j_0}\mathsf{n}), (0^\ell, S^{p+j_1}\mathsf{n}) \to (0^\ell, S^{p}\mathsf{n}).$$

We use these derivations to build a type derivation for the term $\lambda x.\, F(H_0 x)(H_1 x)$, which represents $g(x)$.

In the derivations for H_0 and H_1 we substitute $S^{k_1+q_1+p+j_0+j_1}\mathsf{n}$ and $S^{k_0+q_0+p+j_0+j_1}\mathsf{n}$, respectively, for n to obtain level-ℓ derivations for $\vdash H_0 : (0^\ell, S^{k+q_0+q_1}\mathsf{n}) \to (0^\ell, S^{k+q_1}\mathsf{n})$ and $\vdash H_1 : (0^\ell, S^{k+q_0+q_1}\mathsf{n}) \to (0^\ell, S^{k+q_0}\mathsf{n})$, where $k =_{\mathrm{df}} k_0 + k_1 + p + j_0 + j_1$. By Lemma 3.1 these derivations yield derivations for $\vdash H_i : (0^\ell, S^{k+q}\mathsf{n}) \to (0^\ell, S^{p+j_i}\mathsf{n})$ $(i = 0, 1)$, where $q =_{\mathrm{df}} q_0 + q_1$, from which we readily get derivations for $x : v_{(0^\ell, S^{k+q}\mathsf{n})} \vdash H_i x : v_{(0^\ell, S^{p+j_i}\mathsf{n})}$ $(i = 0, 1)$. Combining the latter two derivations with the given derivation for F, we have a level-ℓ derivation for $x : v_{(0^\ell, S^{k+q}\mathsf{n})} \vdash F(H_0 x)(H_1 x) : v_{(0^\ell, S^{p}\mathsf{n})}$, from which one use of Object Abstraction yields a derivation of $\vdash \lambda x.F(H_0 x)(H_1 x) : v_{(0^\ell, S^{k+q}\mathsf{n})} \to v_{(0^\ell, S^{p}\mathsf{n})}$. □

Lemma 3.6. Every super-elementary function is step-down representable at level 0.

Proof. The proof is identical to the proof of Lemma 2.12 in Leivant (1991), adapted to our current Curry-style formalism. □

Lemma 3.7. Every function in $\mathscr{E}_r$ is definable by composition from $G_0, \ldots, G_{r-1}$ and elementary functions.

Proof. See Rose (1984, Theorem 3.1). □

Proposition 3.8. Every function in $\mathscr{E}_{\ell+4}$ is step-down representable at level ℓ $(\ell \geqslant 0)$. Consequently, every primitive recursive function is step-down representable.

Proof. $G_0, \ldots, G_{\ell+2}$ are all step-down level-ℓ representable by Lemmas 3.6 and 3.4. Also, by Lemma 3.6, all elementary functions are representable, and by Lemma 3.5, the step-down level-ℓ representable functions are closed under composition. The Proposition then follows by Lemma 3.7. □

4. Calibration of function representation

In this section we establish the converse of Proposition 3.8, that is, if a function is step-down representable at level ℓ, then it is in $\mathscr{E}_{\ell+4}$ (Proposition 4.7). This is done by defining functions in the Grzegorczyk hierarchy that normalize level-ℓ type derivations (modulo coding of syntax). These normalizing functions are defined in terms of simpler detour-contraction functions, where only detours of particular ordinal-levels are contracted. The key ideas are: (1) reduction and normalization functions contract detours 'from the bottom up', and (2) uniform normalization functions $\underline{\mathrm{norm}}_\ell$ serve to contract all detours of all levels $\leqslant \ell$.

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

4.1. *Variable separations*

To streamline the statement and discussion of reductions on derivations, we separate in derivations unrelated uses of type variables and of numeric variables. First, we can rename occurrences of type variables so that each type variable is either free in the entire derivation or is the eigen-variable of exactly one type abstraction or limit abstraction. We call such derivations *type-separated*. Note that in a type-separated derivation no variable occurs both free and bound in the same type statement.

Next, we rename occurrences of numeric variables so that each numeric variable is either free in the entire derivation, or the eigen-variable of at most one limit abstraction. Moreover, we can ensure that if a variable m occurs in u in an ordinal expression $\xi = (\vec{t}, u, \vec{v})$, and also in u' in $\xi' = (\vec{t'}, u', \vec{v'})$, then $\vec{t}$ and $\vec{t'}$ are vectors of equal length, and $\vec{v'}$ and $\vec{v}$ are identical vectors of numeric expressions. This is possible because renaming occurrences of m in all ordinal expressions ξ' that do not satisfy the condition above does not disturb the order on ordinal expressions, and hence the correctness of the derivation. We call a derivation that satisfies this condition *rank-separated*.

4.2. *Detours and contractions*

We stipulate that all derivations are type-separated and rank-separated. An *object detour* in a type derivation is an object abstraction followed by an object application, and is contracted by

$$\dfrac{\dfrac{\overset{\Pi}{\Theta, \Gamma, x : \sigma \vdash F : \tau}}{\Theta, \Gamma \vdash \lambda x.F : \sigma \to \tau} \quad \overset{\Sigma}{\Theta', \Gamma' \vdash E : \sigma}}{\underset{\Pi'}{\Theta, \Theta', \Gamma, \Gamma' \vdash (\lambda x.F)E : \tau}} \quad \Rightarrow \quad \underset{\Pi'}{\overset{\{\Sigma/x\}\Pi}{\Theta, \Theta', \Gamma, \Gamma' \vdash \{E/x\}F : \tau}}$$

Here $\{\Sigma/x\}\Pi$ is obtained from Π by:

(a) deleting the type assignment $x : \sigma$ from all environments;
(b) substituting E for the free occurrences of x, which must now all be in the succedent of type statements;
(c) joining Θ', Γ' to all environments; and
(d) replacing each Initial type statement of Π for x, now of the form $\Theta_0, \Theta', \Gamma_0, \Gamma' \vdash E : \sigma$ by a copy of Σ weakened by Θ_0, Γ_0.

A *type detour* is a type abstraction followed by a type application; it is contracted by substituting the eigen-type for the eigen-variable in the succedent of conclusions:

$$\dfrac{\dfrac{\overset{\Pi}{\Theta, R : \xi, \Gamma \vdash E : \tau}}{\Theta, \Gamma \vdash E : (\forall R : \xi.\, \tau)}}{\underset{\Pi'}{\Theta, \Theta', \Gamma \vdash E : \{\sigma/R\}\tau}} \quad \Rightarrow \quad \underset{\Pi'}{\overset{\{\Theta', \sigma/R\}\Pi}{\Theta, \Theta', \Gamma \vdash E : \{\sigma/R\}\tau}}$$

Here $\{\Theta', \sigma/R\}\Pi$ is obtained from Π by

(a) deleting the rank assignment $R : \xi$ from all environments;

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

(b) substituting σ for all occurrences of R (which are all free, since the given derivation is type-separated); and
(c) joining Θ' to all antecedents.

A *limit detour* is a limit abstraction followed by a type application; it is contracted by first appealing to Lemma 2.1 to find v so that for every $\eta \in \underline{\mathrm{ord}}(\Theta, \sigma)$, $\eta < (\xi|v)$, and then substituting v for m and σ for R in Π:

$$\dfrac{\dfrac{\begin{matrix}\Pi \\ \Theta, R : (\xi|S^r\mathsf{m}), \Gamma \vdash E : \tau\end{matrix}}{\Theta, \Gamma \vdash E : (\forall R : \xi.\, \tau)}}{\begin{matrix}\Theta, \Theta', \Gamma \vdash E : \{\sigma/R\}\tau \\ \Pi'\end{matrix}} \quad \Rightarrow \quad \begin{matrix}\{v/\mathsf{m}\}\{\Theta', \sigma/R\}\Pi \\ \Theta, \Theta', \Gamma \vdash E : \{\sigma/R\}\tau \\ \Pi'\end{matrix}$$

By assumption, Π is rank-separated, so $\{v/\mathsf{m}\}\{\Theta', \sigma/R\}\Pi$ is a correct derivation.

Note that contractions preserve the derived λ-expression modulo β-conversion, and preserve the derived type.

A *detour statement* in a type derivation is the leftmost premise of the last statement inferred in one of the detours listed above. The *type* of a detour statement $\Theta, \Gamma \vdash E : \tau$ is (Θ, τ) and its *rank* is $\underline{\mathrm{rk}}(\Theta, \tau) =_{\mathrm{df}} \max\{\overline{\xi} : \xi \in \underline{\mathrm{ord}}(\Theta, \tau)\}$; recall that this is defined in terms of only the type variables that occur in τ. A type derivation is *normal* if it has no detour. Since a derivation may have detours of rank zero (which must necessarily be object detours), a derivation of detour rank zero need not be normal (this accounts for the formulation of Lemma 4.2 below).

The following is straightforward by induction on derivations.

Lemma 4.1. In a normal type derivation all λ-terms are normal.

4.3. *Rank reduction*

We posit some canonical numeric coding of the syntax of type derivations. The basic syntactic operations on derivations, such as detecting detours, identifying their structure, and performing a reduction, are then mirrored by elementary functions on the corresponding numeric codes. We also assume that the code of syntactic objects (such as type statements and type derivations) is elementary in the number of symbol-occurrences therein. All these conditions are trivially satisfied by all standard coding schemes for syntax. In this section we prove the following statement.

Lemma 4.2. There is a super-elementary function $\underline{\mathrm{rnkdec}}$ that maps (by repeated reductions) each derivation of detour rank > 0 to a derivation of lower detour rank, and each derivation of detour rank 0 to a normal derivation.

For detours of a fixed rank ζ, we will define a reduction function by recursion on the complexity of the type of the detours. That complexity cannot be defined as simply the number of type constructors, because the contraction of a type detour can increase the number of type constructors in other detours. Instead, we only start counting type constructors from subtypes of rank ζ. The proviso on the Type Application rule ensures that this measure does not increase by a contraction of a type or limit detour. Thus, for

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

ordinal notations ζ we define $\underline{\mathrm{dg}}_\zeta(\Theta,\sigma)$, the ζ-*degree of* (Θ,σ), to be the number of type constructors in σ 'once σ has rank ζ':

$$\underline{\mathrm{dg}}_\zeta(\Theta,R)=\begin{cases}1 & \text{if } \underline{\mathrm{rk}}(\Theta,R)\geqslant\zeta\\ 0 & \text{otherwise}\end{cases}$$

$$\underline{\mathrm{dg}}_\zeta(\Theta,\sigma\to\tau)=\dot{\mathbf{s}}(\max\{\underline{\mathrm{dg}}_\zeta(\Theta,\sigma),\underline{\mathrm{dg}}_\zeta(\Theta,\tau)\})$$

$$\underline{\mathrm{dg}}_\zeta(\Theta,\forall R:\eta.\sigma)=\begin{cases}\underline{\mathrm{dg}}_\zeta((\Theta,R:\eta),\sigma)+1, & \overline{\eta+1}\geqslant\zeta\\ \dot{\mathbf{s}}\,\underline{\mathrm{dg}}_\zeta((\Theta,R:\eta),\sigma), & \text{otherwise}\end{cases}$$

where

$$\dot{\mathbf{s}}(x)=\begin{cases}x+1, & \text{if } x>0\\ 0, & \text{otherwise.}\end{cases}$$

The ζ-*degree* of a detour $\Theta,\Gamma\vdash E:\tau$ is the value $\underline{\mathrm{dg}}_\zeta(\Theta,\tau)$. A ζ-*d-detour* is a detour of rank ζ and ζ-degree d. The *critical* ζ-*d-detour* of a derivation Π is the topmost leftmost detour of rank ζ and ζ-degree d. The ζ-*degree* of a derivation Π is the maximum among the ζ-degrees of detours in Π. It is easy to check the following.

Lemma 4.3. Let Π be a derivation of detour rank ζ and ζ-degree d, with m ζ-d-detours and s symbols. Let Π' be the result of reducing the critical ζ-d-detour in Π. Then the detour degree of Π' is $\leqslant\zeta$, its ζ-degree is $\leqslant d$, its number of ζ-d-detours is $m-1$, and it has $\leqslant s^2$ symbols. □

Define $\underline{\mathrm{szred}}(d,m,h)$ by

$$\underline{\mathrm{szred}}(0,m,s)=s$$

$$\underline{\mathrm{szred}}(d+1,m,s)=\underline{\mathrm{szred}}(d,s^{2^m},s^{2^m})$$

Note that $\underline{\mathrm{szred}}$ is defined by a single recurrence from elementary functions, and is therefore super-elementary (see, for example, Rose (1984) or Schwichtenberg (1969)). The function $\underline{\mathrm{szred}}$ will bound the size of derivations as detours are contracted.

Lemma 4.4. Let Π be as in Lemma 4.3. Then Π converts by repeated reductions of critical detours to a type derivation Π^*, whose detour rank is $<\zeta$ and with $\leqslant\underline{\mathrm{szred}}(d,m,s)$ symbols.

Proof. The proof is by induction on d. If $d=0$, then Π has no detour of rank ζ, so we may take $\Pi^*=\Pi$. If $d>0$, then using Lemma 4.3 m times leads to a derivation Π'' with detour rank $\leqslant\zeta$, ζ-degree $<d$, and $\leqslant s^{2^m}$ symbols. Thus s^{2^m} is also a bound on the number of ζ-$(d-1)$-detours. By the induction assumption, Π'' converts by repeated reductions of critical detours to a derivation Π^* of detour rank $<\zeta$, and with $\leqslant\underline{\mathrm{szred}}(d-1,s^{2^m},s^{2^m})=\underline{\mathrm{szred}}(d,m,h)$ symbols. □

We now prove Lemma 4.2. Let $\underline{\mathrm{dgdec}}$ be the unary function defined by the following process, modulo numeric coding of syntax. First, identify the detour rank ζ of the derivation Π coded by the input, as well as its ζ-degree d, and number m of ζ-d-detours. Perform m successive reductions of ζ-d-critical detours, resulting in a derivation of detour rank $\leqslant\zeta$ and ζ-degree $<d$.

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

Now let rnkdec be computed by identifying the input's detour rank ζ and ζ-degree d, and iterating dgdec d times. The function dgdec is defined from elementary functions by composition and a single recurrence, and is therefore super-elementary. The function rnkdec is defined from elementary functions and the super-elementary dgdec by a single recurrence. By Lemma 4.4 (and our proviso on the numeric coding scheme used), that recurrence is bounded by a super-elementary function. Thus rnkdec is also super-elementary, completing the proof of Lemma 4.2.

4.4. *Normalization*

Lemma 4.5. For each $\ell \geqslant 0$ there are a function $\underline{\mathrm{red}}_\ell$ in $\mathscr{E}_{\ell+4}$ and a function $\underline{\mathrm{norm}}_\ell$ in $\mathscr{E}_{\ell+5}$, such that (modulo numeric coding of syntax)

1 $\underline{\mathrm{red}}_\ell$ maps by iterated reductions each type derivation of detour rank

$$\leqslant (d_0, \ldots, d_{\ell-1}, Sd_\ell, d_{\ell+1}, \ldots, d_r)$$

to a derivation of detour rank

$$\leqslant (0^\ell, d_\ell, d_{\ell+1}, \ldots, d_r).$$

2 $\underline{\mathrm{norm}}_\ell$ maps by iterated reductions each type derivation of detour rank

$$\leqslant (d_0, \ldots, d_\ell, d_{\ell+1}, \ldots, d_r)$$

to a derivation of detour rank

$$\leqslant (0^{\ell+1}, d_{\ell+1}, \ldots, d_r).$$

Proof. The proof is by induction on ℓ. For $\ell = 0$ we let $\underline{\mathrm{red}}_0$ be rnkdec. The function $\underline{\mathrm{norm}}_0$ is defined by identifying the input's detour rank ζ and ζ-degree d, and iterating $\underline{\mathrm{red}}_0$ d times. Thus $\underline{\mathrm{norm}}_0$ is defined from super-elementary functions by a single recurrence, and is therefore in $\mathscr{E}_5$.

For the induction step, assume $\underline{\mathrm{red}}_\ell$ and $\underline{\mathrm{norm}}_\ell$ have been defined with the prescribed properties and complexities. Let $\underline{\mathrm{red}}_{\ell+1}$ be computed as follows. First apply $\underline{\mathrm{norm}}_\ell$ to the input derivation Π. If the detour rank of Π is $\leqslant (d_0, \ldots, d_{\ell-1}, Sd_\ell, d_{\ell+1}, \ldots, d_r)$, then the result is of rank $\leqslant (0^\ell, Sd_\ell, d_{\ell+1}, \ldots, d_r)$ by the induction assumption. Now apply rnkdec, yielding a derivation of rank $\leqslant (d'_0, \ldots, d'_{\ell-1}, d_\ell, d_{\ell+1}, \ldots, d_r)$, for some $d'_0, \ldots, d'_{\ell-1}$. Finally, applying $\underline{\mathrm{norm}}_\ell$ again, we obtain a derivation of rank $\leqslant (0^\ell, d_\ell, d_{\ell+1}, \ldots, d_r)$. The function $\underline{\mathrm{red}}_{\ell+1}$ is in $\mathscr{E}_{\ell+5}$, since it is defined by composition from $\underline{\mathrm{norm}}_\ell$, which is in $\mathscr{E}_{\ell+5}$ by the induction assumption, the function rnkdec, which is in $\mathscr{E}_4$ by Lemma 4.2, and elementary functions.

The function $\underline{\mathrm{norm}}_{\ell+1}$ is defined from $\underline{\mathrm{red}}_{\ell+1}$ analogously to the base case. Since the definition is by a single recurrence, and $\underline{\mathrm{red}}_{\ell+1}$ is in $\mathscr{E}_{\ell+5}$, it follows that $\underline{\mathrm{norm}}_{\ell+1}$ is in $\mathscr{E}_{\ell+6}$, as required. □

Lemma 4.6. For each $\ell > 0$ and $k > 0$ there is a function $\underline{\mathrm{normlz}}_{\ell,k}$ in $\mathscr{E}_{\ell+4}$ that maps (by iterated reductions) every derivation whose detour rank is $\leqslant (0^\ell, S^{[k]}0)$ to a normal derivation.

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

Proof. Let

$$\underline{\mathrm{normlz}}_{\ell,k} = \underline{\mathrm{rnkdec}} \circ \underline{\mathrm{red}}_{\ell}^{[k]}$$

The lemma is now proved by induction on k, using Lemma 4.5. Since a derivation of detour rank zero need not be normal, this last application of $\underline{\mathrm{rnkdec}}$ is necessary. □

Note that the properties we stipulate for our syntax-coding scheme guarantee that there is an elementary function $\underline{\mathrm{extr}}$ such that if Π is a derivation of $\vdash \bar{p} : v_{\xi}$, then $\underline{\mathrm{extr}}(\Pi) = p$.

Proposition 4.7. Every level-ℓ skewly representable function is in $\mathscr{E}_{\ell+4}$.

Proof. Without loss of generality, let f be binary. Suppose f is represented by the λ-term F, with a level-ℓ derivation Π of $\vdash F : \xi_1, \xi_2 \to \eta$. For $p_1, p_2 \in \mathbb{N}$, let Π_{p_1,p_2} be the derivation

$$\dfrac{\dfrac{\Pi \qquad \Delta_{p_1,\xi_1}}{\vdash F : v_{\xi_1}, v_{\xi_2} \to v_{\eta} \quad \vdash \bar{p}_1 : v_{\xi_1}} \qquad \Delta_{p_2,\xi_2}}{\vdash F\bar{p}_1 : v_{\xi_2} \to v_{\eta} \qquad \vdash \bar{p}_2 : v_{\xi_2}}$$
$$\vdash F\bar{p}_1\bar{p}_2 : v_{\eta}$$

The detour rank of Π_{p_1,p_2} is of the form $(d_0, \ldots, d_{\ell-1}, S^{[k]}0)$ for some k. Then by Lemma 4.6, $f(p_1, p_2) = \underline{\mathrm{extr}}(\underline{\mathrm{normlz}}_{\ell,k+1}(\Pi_{p_1,p_2}))$, and so f is in $\mathscr{E}_{\ell+4}$. □

We are now ready to conclude our main results.

Theorem 4.8. Let f be a function over the natural numbers. The following conditions are equivalent:

1 f is in the Grzegorczyk class $\mathscr{E}_{\ell+4}$.
2 f is level-ℓ step-down representable in **S2λ**.
3 f is level-ℓ representable (*i.e.*, skewly representable) in **S2λ**.

Proof. (1) implies (2) by Proposition 3.8. (2) implies (3) by the definitions (skew representation is more general than step-down representation). (3) implies (1) by Proposition 4.7. □

Corollary 4.9. Let f be a function over the natural numbers. The following conditions are equivalent:

1 f is primitive recursive.
2 f is step-down representable in **S2λ**.
3 f is strictly representable in **S2λ**.
4 f is representable (*i.e.*, skewly representable) in **S2λ**.

Proof. (1) implies that $f \in \mathscr{E}_{\ell}$ for some ℓ, which by Proposition 3.8 implies (2). (2) implies (3) by Lemma 3.2. (3) implies (4) by the definitions, and (4) implies (1) by Proposition 4.7. □

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

5. Concluding remarks

We have defined a finitely presented and highly uniform calculus of stratified polymorphism, and related it to Grzegorczyk's natural classification of the primitive recursive functions. This raises several questions of interest for further exploration.

An extension of the formalism to all ordinals $< \varepsilon_0$ is fairly natural, for instance using as ordinal notations the hereditarily finite sequences built on top of the natural numbers. But a more subtle extension is needed in order to obtain more powerful function representation. This is apparent already with the Ackermann-like function $G_\omega(x) =_{\mathrm{df}} G_{x+1}(x)$, defined by the λ-term $A = \lambda a.(\mathsf{S}a)(\lambda f \lambda b.bf\bar{2})(\lambda c.\mathsf{M}cc)a$ (here S and M are the terms representing successor and multiplication, given in the Appendix). To type the term $J = \lambda f \lambda b.bf\bar{2}$, one must assign f a type $v_\xi \to v_\xi$. Then b must get a type v_η with $\eta > \xi$, whence J is of type $(v_\xi \to v_\xi) \to (v_\eta \to v_\xi)$. However, the variable a is to be instantiated by Church numerals, and its first argument must therefore have identical input and output types, which J does not. Thus $(\mathsf{S}a)J$ cannot be typed in a formalism moulded after **S2$\boldsymbol{\lambda}$**. The authors have been exploring extensions of the type system, exceeding the mould of **S2$\boldsymbol{\lambda}$**, in which all ε_0-recursive functions, in particular, G_ω above, can be typed; for example, one such extension allows quantification over numeric variables in the types, with types such as $\forall k.v^{\omega^k} \to v^{\omega^k}$. It might be worth recalling, though, that a natural sub-calculus of **2$\boldsymbol{\lambda}$** that characterizes exactly the ε_0-recursive functions exists, using simple syntactic restrictions rather than a stratification machinery (Leivant 1997). Extensions of **S2$\boldsymbol{\lambda}$** beyond ε_0 are increasingly likely to require infinitary type-inference calculi, and would have little interest in programming language design.

An area related to λ-calculi with stratified type abstraction is the proof theoretic systems that are mapped onto them via natural Schönfinkel–Curry–Howard morphisms, as in the framework of Leivant (1990, Section 7.3). These are systems of minimal ramified second-order logic[†], in which a proof of totality of f is mapped to a typing of a λ-term representing f. Thus, these formalisms provide a natural framework in which to reason about subrecursive functions, but it is yet to be put to use.

Finally, a research direction of significant practical interest is type inference for the calculi considered. Given a typed calculus TC, the problem of *type inference* is to produce, given an untyped expression, a TC type for it (hopefully a 'most general type') if one exists, and an error message otherwise. The availability of a type inference algorithm for TC is essential in making it a useful programming environment, as it avoids the need for programmers to supply type information (which can become fairly involved). In designing the programming language ML, Robin Milner restricted the use of polymorphism to a rudimentary form precisely because that form supports a rather effective type inference algorithm (Milner 1985)[‡].

Since there is no type inference algorithm for **2$\boldsymbol{\lambda}$** (Wells 1994; Wells 1999), the question

[†] The restriction to minimal logic is not of central importance here, as, for instance, every proof of program convergence in a classical formalism is easily convertible to a proof in the corresponding minimal-logic formalism.

[‡] A slightly more general form of polymorphism supports a similar algorithm, but without significant loss of computational efficiency (Kfoury and Wells 1994).

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

of type inference for subformalisms, such as **S2λ**, is of obvious interest. The existence of such algorithms for stratified variants of **2λ**, even with finite levels, is still an open problem with no consensus conjecture. However, a decidability result for type inference in the finitely stratified calculus is likely to generalize to the calculus presented here. Moreover, the stratified versions of **2λ** offer interesting possibilities of delineating large classes of functional programs for which type inference would not only be decidable, but may be decidable by an algorithm of low computational complexity†.

Appendix. Type derivations for basic functions

We exhibit type derivations for terms representing successor, multiplication and exponentiation to base 2. The environments, which we omit for the sake of readability, contain the following assignments, as needed: $R : S\mathsf{n}$, $z : R$, $s : R \to R$, $a : v_{S\mathsf{n}}$. We also omit $\vdash$ (that is, we end up with a natural deduction display style).

— Successor:

$$\dfrac{\dfrac{\dfrac{\dfrac{a : v_{Sn}}{a : (R \to R) \to (R \to R)} \quad s : R \to R}{as : R \to R} \quad z : R}{asz : R}}{\dfrac{\dfrac{s : R \to R}{s(asz) : R}}{\dfrac{\lambda sz.s(asz) : (R \to R) \to (R \to R)}{\lambda sz.s(asz) : v_{Sn}}}}}{\lambda a.\lambda sz.s(asz) : v_{Sn} \to v_{Sn}}$$

— Multiplication:

$$\dfrac{\dfrac{\dfrac{a : v_{Sn}}{a : (R \to R) \to (R \to R)} \quad \dfrac{\dfrac{b : v_{Sn}}{b : (R \to R) \to (R \to R)} \quad s : R \to R}{bs : R \to R}}{a(bs) : R \to R}}{\dfrac{\dfrac{\lambda s.a(bs) : (R \to R) \to (R \to R)}{\lambda s.a(bs) : v_{Sn}}}{\lambda ab.\lambda s.a(bs) : v_{Sn} \to (v_{Sn} \to v_{Sn})}}$$

— Base-2 exponentiation: let $R_2 =_{\mathrm{df}} ((R \to R) \to (R \to R))$ and $R_3 =_{\mathrm{df}} R_2 \to R_2$.

$$\dfrac{\dfrac{\dfrac{a : v_{Sn}}{a : R_3} \quad \dfrac{\dfrac{\Delta_{2,(S\mathsf{n})}}{\bar{2} : v_{S\mathsf{n}}}}{\bar{2} : R_2}}{a\bar{2} : R_2}}{a\bar{2} : v_{Sn}}}{\lambda a.a\bar{2} : v_{Sn} \to v_{Sn}}$$

† A similar issue arises regarding the programming language ML, whose type inference is fast in practice, but has been proved to be exponential-time hard (Mairson *et al.* 1994; Kfoury *et al.* 1994)

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press

Acknowledgements

The authors are grateful to Fritz Henglein, Assaf Kfoury, Harold Mairson, Pawel Urzyczyn, Joe Wells and the anonymous referees for useful discussions and suggestions.

References

Feferman, S. (1964) Systems of predicative analysis. *J. Symbolic Logic* **29** (1) 1–30.

Girard, J.-Y. (1972) *Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'ordre supérieur*, Thèse de Doctorat d'Etat, Université Paris VII.

Kfoury A. J. and Wells, J. B. (1994) A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In: *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*.

Kfoury A. J., Tiuryn, J. and Urzyczyn, P. (1994) An analysis of ML typability. *Journal of the ACM* **41** (2).

Leivant, D. (1989) Stratified polymorphism. In: *Fourth Annual Symposium of Logic in Computer Science*, IEEE 39–47.

Leivant, D. (1990) Contracting proofs to programs. In: Odifreddi, P. (ed.) *Logic and Computer Science*, Academic Press.

Leivant, D. (1991) Finitely stratified polymorphism. *Inform. and Comput.* **93** (1) 93–113.

Leivant, D. (1997) Peano's Lambda Calculus: the functional abstraction implicit in arithmetic (manuscript). To appear in Anderson, C. A. and Zeleny, M. (eds.) Church Memorial Volume: Logic, Language, and Computation, Kluwer Academic Publishing. Abstract in *Bulletin of Symbolic Logic* **4** (1998) 451–452.

Mairson, H., Hillebrand, G. and Kanellakis, P. C. (1994) An analysis of the Core-ML language: expressive power and type reconstruction. In: 1994 International Colloquium on Automata, Languages, and Programming (ICALP). *Springer-Verlag Lecture Notes in Computer Science* **820** 83–106.

Milner, R. (1985) The standard ML core language. *Polymorphism* **2** (2).

Reynolds, J. C. (1974) Towards a theory of type structure. In: Programming Symposium (Proc. Colloq. Programmation, Paris, 1974). *Springer-Verlag Lecture Notes in Computer Science* **19** 408–425.

Rose, H. E. (1984) *Subrecursion: functions and hierarchies*, The Clarendon Press, Oxford University Press.

Schütte, K. (1977) *Proof Theory*, Springer-Verlag, Berlin. (Translated by J. N. Crossley from the revised German edition: *Grundlehren der Mathematischen Wissenschaften* **225**.)

Schwichtenberg, H. (1969) Rekursionszahlen und die Grzegorczyk-Hierarchie. *Arch. Math. Logik Grundlagenforsch* **12** 85–97.

Statman, R. (1981) Number theoretic functions computable by polymorphic programs. In: *Twenty-Second Annual Symposium on Foundations of Computer Science*, IEEE 279–282.

Wells, J. B. (1994) Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. In: *Ninth Annual Symposium of Logic in Computer Science*, IEEE.

Wells, J. B. (1999) Typability and type checking in System F are equivalent and undecidable (to appear in *Annals of Pure and Applied Logic*).

https://doi.org/10.1017/S0960129599002868 Published online by Cambridge University Press