**Stratified polymorphism**

**Extended summary**

Daniel Leivant
School of Computer Science, Carnegie Mellon

**Synopsis**

We consider a spectrum of predicative type abstraction disciplines based on type quantification with stratified levels. These lie in the vast middle ground between parametric abstraction and full impredicative abstraction. Stratified polymorphism has an attractive, unproblematic, semantics, and has the potential of offering new approaches to type inference, without sacrificing useful expressive power.

We show that the functions representable in the finitely-stratified $\lambda$-calculus are precisely the superelementary functions, i.e. $\epsilon_4$ in Grzegorczyk’s subrecursive hierarchy.

We also define methods of transfinite stratification, and we show that stratification up to $\varepsilon_0$ has a simple finitary representation, making it a potentially useful concept in programming language design. We prove that the functions representable by stratified polymorphism up to $\varepsilon_0$ are precisely the primitive recursive functions.

Finally, we point out that these results imply that the equality problem for finitely stratified $\lambda$-calculus is not super-elementary, and that the equality problem for the calculus stratified up to $\varepsilon_0$ is not primitive recursive.

**Introduction**

Type disciplines for programming languages attempt to strike a balance between three, often conflicting, aims: expressive power, simplicity and methodological coherence, and user friendly implementability. The trade off between such aims can be seen in the two main paradigms of polymorphic typing: parametric and quantificational. The parametric discipline of ML is user friendly by virtue of its (in practice) fast type inference mechanism, but it lacks some of the power of full type quantification, and it suffers from certain anomalies [Myc84, Pey87]. On the other hand, the impredicative quantificational discipline has great expressive power, well beyond current programming needs, but it is probably too powerful to allow computationally feasible user friendly facilities, such as type inference.

We discuss here another ingredient to the design of programming type disciplines, *stratification of type abstraction*, which engenders a whole spectrum of disciplines between parametric polymorphism and full quantificational polymorphism. It therefore has the potential of both clarifying theoretical issues of polymorphic typing, and of providing effective language design tools. The idea of stratifying abstraction into levels goes back to the Ramified Type Theory of [Rus08, WR10], whose purpose was to circumvent the semantic antinomies. It was revived in the 1950’s (e.g. [Kre60, Wan54, Wan62]) in relation to *Predicative Analysis*, a semi-constructive foundation of Mathematics. Stratification of type abstraction in the polymorphic $\lambda$-calculus, and related typed programming languages, was first suggested by Statman [Sta81]. As a programming language design concept, it has the potential of allowing great expressive power without the full computational penalties that seem to plague unstratified (impredicative) type quantification.

Second order type quantification generates types of the form $\tau = \forall \lambda. \tau$, where the quantification is over all types, including $\tau$ itself. The aim of stratification is to prevent such circularity. Types are stipulated to fall into levels, with the base level consisting exactly of those types whose definition involves no type quantification. The next level consists of types whose definition may use quantification over types of level $n \in \omega$, whereas by definition $\text{level}(\tau) > \text{level}(\tau) = n$. The construction of levels can proceed into the transfinite, by taking the union of lower levels at limit ordinals $\lambda$: in $\forall \lambda. \tau$, the variable $\lambda$ ranges over types of level $\alpha < \lambda$. We show that this extension, albeit transfinite, has a potentially useful finite presentation.
This paper addresses two issues of stratified polymorphism. In §§1.2, we show that the numeric functions defined in the finitely stratified polymorphic \(\lambda\)-calculus are precisely the super-elementary functions. This answers a problem left open in [StaSl], where it is stated (without proof) that these functions fall between the super-elementary and the primitive recursive functions.

In §3 we define systems of transfinite stratification. In particular, we define a parametric, finitely presentable form, which is easily implementable in programmable languages. We show that the parametric form, for arbitrary levels, collapses to level \(\omega^\omega\). We then show that the functions representable in the parametrically stratified \(\lambda\)-calculus are precisely the primitive recursive functions.

In §4, we point out that our characterization results imply that the equality problem for finitely stratified \(\mu\)-calculus is not super-elementary, and that the equality problem for the calculus stratified up to \(\omega^\omega\) is not primitive recursive. We conclude with open problems and direction for future research in §5.

1. The finitely stratified Polymorphic \(\lambda\)-Calculus

1.1. Finite stratification

The finitely stratified polymorphic lambda calculus, \(S2\lambda^\omega\), is similar to Girard-Reynolds' second order lambda calculus \(2\lambda\) ([Gir72],[Rey74] exposition in [FLO83]), except that types are classified into levels 0, 1, ... .

Type expressions \(\tau\) and their levels \(\text{level}(\tau)\) are defined inductively:

- There is a denumerable supply of type variables of level \(k\), \(\delta, \delta_1, \delta_2, \ldots\), for each level \(k = 0, 1, \ldots\). We omit the level superscript when it is irrelevant or clear from the context. A type variable of level \(k\) is also a type expression of level \(k\).

- If \(\tau\) and \(\sigma\) are type expressions, of levels \(p\) and \(q\) respectively, then \(\tau \rightarrow \sigma\) is a type expression of level \(\text{max}(p, q)\).

- If \(\tau\) is a type expression, of level \(p\), then \(\forall \delta. \tau\) is a type expression, of level \(\text{max}(p, q+1)\).

Thus, the level of a type expression \(\tau\) is the largest of \(\text{level}(t)\) for \(t\) free in \(\tau\) and \(1 + \text{level}(t)\) for \(t\) bound in \(\tau\).

We then show that the parametric form, for arbitrary levels, collapses to level \(\omega^\omega\). We show that the functions representable in the parametrically stratified \(\lambda\)-calculus are precisely the primitive recursive functions.

There is a denumerable supply of individual variables, \(x', x_0', x_1', \ldots\), for each type expression \(\tau\). \(\tau\) is the type of \(x'\). (We omit type superscripts when irrelevant or clear from the context.) An individual variable of type \(\tau\) is also an object expression of type \(\tau\).

- If \(E\) is an expression of type \(\tau\), then \(\lambda x'. E\) is an expression of type \(\sigma \rightarrow \tau\).

- If \(E\) is an expression of type \(\tau \rightarrow \sigma\), and \(F\) an expression of type \(\tau\), then \(EF\) is an expression of type \(\sigma\).

- If \(E\) is an expression of type \(\tau\), then \(\lambda t. E\) is an expression of type \(\forall t. \tau\).

- If \(E\) is an expression of type \(\forall t. \tau\), and \(\text{level}(\sigma) \leq k\), then \(E\sigma\) is an expression of type \(\tau[\sigma/1]\). \(\tau[\sigma/1]\) is the result of simultaneously substituting \(\sigma\) for all free occurrences of \(t\) in \(\tau\), after renaming bound variable in \(\tau\) to avoid binding of variables free in \(\sigma\). Note that if \(\text{level}(\sigma) > k\) then \(E\sigma\) is not legal).

\(S2\lambda^\omega\) is the restriction of \(S2\lambda\) where the type of any expression has level \(\leq n\). Thus, \(S2\lambda^0\) allows no type quantification, and is equivalent to \(1\lambda\).

Clearly, the parametric polymorphism of \(\mu\)-, as well as its extension defined in [KTU88] (without recursive types, in both cases), are contained in \(S2\lambda^1\).

1.2. Example

Suppose \(\lambda t_1 x_1' \cdots x_m' E\) is an expression of type \(\tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow \tau\), where \(\tau_i \equiv \tau[\sigma_i/1\] (\(i = 1, \ldots, m\)). Let \(E'\) be the same as \(E\), except that all free occurrences of \(x_i'\) are replaced by \(x_i'\) and \(\sigma_i\). Then \(E'\) is also of type \(\tau\). \(\lambda t. E'\) is of type \(\forall t. \tau\), and \(\lambda t. E'\) is of type \(\forall t. \tau\). \(\forall t. \tau\) is of type \(\forall t. \tau\). Note that \(\lambda t_1 x_1' \cdots x_m' E\) and \(\lambda t_1 x_1' \cdots x_m' E\) have the same underlying untyped \(\lambda\)-expression. Thus quantification over types of level 0 makes it possible, in some cases, to change a typing in \(1\lambda\) where the type of the "output" differs from the type of the "inputs" into a typing where these types are all identical.

1.3. Model Theory

The full polymorphic \(\lambda\)-calculus has a problematic and complex model theory. For instance, there is no set-theoretic...
2. Computational calibration of finitely stratified polymorphism

2.1. All super elementary functions are represented at finite levels

The Grzegorczyk class $E_k$ (for all $k \geq 0$) is generated by composition and bounded recursion from zero, successor, the projection functions, and the function $F_k$, where $F_0 = \text{odd}$, and $F_{k+1}(x) = F_k^2(x)$ ($F^a$ being the $n$th iterate of $F$). $E_3$ is Kalmar’s class of ELEMENTARY functions, and the functions in $E_4$ are dubbed SUPER-ELEMENTARY. We have $PR = \cup k E_k$ [Grz53]. (For exposition see eg [Ros84].)

In $\lambda$, CHURCH’S NUMERALS OVER TYPE $\tau$ are the expressions

$$n = \text{df } \lambda s. \tau \rightarrow \tau \rightarrow s. \text{s}^n \text{z}, \quad n = 0, 1, \ldots$$

The type of these numerals is $\nu[\tau] = \text{df } (\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$. An expression $E$ of type $\nu[\tau] \rightarrow \cdots \rightarrow \nu[\tau] \rightarrow \nu[\tau]$ represents $a$ $k$-ary recursive function $F$ (at type $\tau$) if the conditions $F_n \ldots n = m$ and $E(n) \ldots m = s, m = \text{gn } m$ are equivalent.

An expression $E$, of type $\nu[\tau_1] \rightarrow \nu[\tau_2] \rightarrow \cdots \rightarrow \nu[\tau_r] \rightarrow \nu[\tau]$, represents a $k$-ary recursive function $F$ if the conditions $F_n \ldots n = m$ and $E(n) \ldots m = s, m = \text{gn } m$ are equivalent. For a typed $\lambda$-calculus $L$ we write Strict$[L]$ and Skew$[L]$ for the set of functions representable strictly, askew respectively, in $L$.

Lemma 2.1 (Church) $0$, $S$, addition, and multiplication are in Strict$[\lambda]$.

Proof. See eg [Schw76] or [FLO83].

In $S2\lambda^w$ we define the $l$-LEVEL $n$'TH NUMERAL, $\text{n}_l$, to be the expression $\lambda t. \text{\lambda s} t \rightarrow \text{s}^n \text{z}$, $l = 0, 1, \ldots$. The $l$-level numerals are thus of type $l = \forall l' \cdot (l' \rightarrow l') \rightarrow (l' \rightarrow l')$. An expression $E$ represents strictly (at level $l$) a $k$-ary recursive function $F$ if the conditions $F_n \ldots n = m$ and $E^n \ldots m = s, m$ are equivalent. An expression $E$ represents askew (at level $l$) a $k$-ary recursive function $F$ if for suitable types $\tau_1 \ldots \tau_k$ of level $l$, the conditions $F_n \ldots n = m$ and $E(n) \ldots m = s, m$ are equivalent. An expression $E$ represents slantwise (at levels $l_1, l_2$) a $k$-ary recursive function $F$ if the conditions $F_n \ldots n = m$ and $E^n \ldots m = s, m$ are equivalent. For a stratified second order typed $\lambda$-calculus $L$ we write Slant$[L]$ for the set of functions representable slantly in $L$.

A function $f$ is defined by iteration from $g$ and $h$ if

$$f(0, x) = g(x), \quad (1)$$
$$f(Sy, x) = h(f(y, x), x). \quad (2)$$

Proposition 2.2 If a function $f$ is defined by iteration from $g$ and $h$ in $\text{Strict}[S2\lambda^4]$, then $f \in \text{Skew}[S2\lambda^4]$.

Proof. Suppose $E_g$ and $E_h$ strictly represent $g$ and $h$ in $S2\lambda^4$ at types $\tau$ and $\tau'$ respectively. Without loss of generality $\tau' = \tau$ (they both can be embedded in a common larger type, where the representation takes place). Then $f$ is represented by the expression $Ef = \text{df } \lambda \lambda n. n(\lambda u. E_{A\mu})(E_{\lambda m})$. (Note that the type of $n$ here is larger than the type of $m$ and the type of the result.)

Proposition 2.3 If $f$ is defined by iteration from $0$, $S$, addition, and multiplication, then $f \in \text{Skew}[\lambda]$.

Proof. By Lemmas 2.1 and 2.2.

Proposition 2.4 Skew$[\lambda] = \text{Strict}[S2\lambda^1]$.

Proof. Skew$[\lambda] \supset \text{Strict}[S2\lambda^1]$ is straightforward by induction on $\lambda$-expressions. Skew$[\lambda] \subset \text{Strict}[S2\lambda^1]$ is the same as Example 1.2.

Proposition 2.5 If $f$ is defined by two iterations from $0$, $S$, addition, and multiplication, then $f \in \text{Skew}[S2\lambda^1]$.

Proof. By Lemmas 2.4 and 2.2.

Proposition 2.6 Skew$[S2\lambda^4] \subset \text{Slant}[S2\lambda^{k+1}]$.

Proof. Same as Example 1.2.

Proposition 2.7 If $f$ is defined by composition from $g \in \text{Slant}[S2\lambda^4]$ and $h \in \text{Slant}[S2\lambda^4]$ then $f \in \text{Slant}[S2\lambda^{k+1}]$. 
represents expression it can be easily modified to have input at level \( k \) and output at level 0, and then modified (by lifting all levels) to an expression \( E'_k \) that represents \( g \) with input at level \( k + l \) and output at level 0. Now, if \( E_k \) represents \( h \) with input at level \( l \), then the expression \( \lambda h^{r&.} E_k(E'_k,n) \) is well typed and represents \( f \) at level \( k + l \). ♦

**Lemma 2.8** Suppose \( C \) contains all functions defined from 0, \( S \), + and \( \times \) by two iterations, and is closed under composition with such functions. Then \( C \) is closed under bounded recursion.

Consequently, if \( f \in P^R \) is bounded by some \( g \in E_k \), where \( k \geq 3 \), then \( f \in E_k \).

**Proof.** See [Ros84] §1.3, proof of Theorem 3.1. ♦

The following Lemma is stated in [Sta81].

**Lemma 2.9** \( E_4 \subseteq \text{Slant}(S2\lambda^w) \).

**Proof.** By Lemmas 2.5 and 2.6, \( \text{Slant}(S2\lambda^w) \) contains the functions defined by two iterations from 0, \( S \), + and \( \times \). By Lemma 2.7 \( \text{Slant}(S2\lambda^w) \) is closed under composition. So, by Lemma 2.8 it is also closed under bounded recursion. Since \( F_4 \) is defined from addition by two iterations, we also have \( F_4 \in \text{Slant}(S2\lambda^w) \). So \( E_4 \subseteq \text{Slant}(S2\lambda^w) \). ♦

### 2.2. \( \text{Slant}(S2\lambda^w) = E_4 \)

The **redex-level** of a redex \( EF \) or \( E\sigma \) is the level of the type of \( E \). The **eigen-level** of an expression \( E \) is the level of the type of \( E \). The **overall-level** of an expression \( E \) is the largest of the levels of the types of subexpressions of \( E \) (\( E' \) included).

For \( l \geq 0 \), the **l-degree** \( dgr(\tau, l) \) of a type \( \tau \) is the count of negative nestings of \( \rightarrow \) that have \( \forall \tau^n, m \geq l \), in their scope.

The **l-degree** of \( E \), \( dgr(E, l) \), is the l-degree of its type. The **redex-level** of \( E \), \( \text{rdx.lvl}(E) \), is the largest among the levels of redexes in \( E \). The **redex-degree** of \( E \), \( \text{rdx.dgr}(E) \), is the largest among the degrees of the maximum-level redexes in \( E \). The **redex-multiplicity** of \( E \), \( \text{rdx.mlt}(E) \), is the number of redexes in \( E \) whose level is \( \text{rdx.lvl}(E) \) and whose degrees is \( \text{rdx.dgr}(E) \).

A **critical redex** of an expression \( E \) is a redex subexpression \( \text{type of } F \) of \( E \), with level \( l = \text{rdx.lvl}(E) \) and degree \( d = \text{rdx.dgr}(E) \), and such that if \( G \) is a redex in \( F \) then \( \text{lvl}(G) < l \), or \( \text{rdx.lvl}(G) = l \) and \( \text{rdx.dgr}(G) < d \).

**Lemma 2.10** Let \( l = \text{rdx.lvl}(E), d = \text{rdx.dgr}(E), n = \text{rdx.mlt}(E) \). Suppose \( E \) converts to \( E' \) by a contraction of a critical redex. Set \( l' = \text{rdx.lvl}(E'), d' = \text{rdx.dgr}(E'), n' = \text{rdx.mlt}(E') \). Then \( (l', d', n') \leq (l, d, n) \) under lexicographic ordering.

**Proposition 2.11** For each \( k < \omega \) there is a function \( B_k \) such that if \( E \) is an expression of \( S2\lambda^k \) then \( \text{dpth(normal}(E)) \leq B_k(\text{dpth}(E)), \) where \( \text{dpth}(E) \) is the depth of the syntax tree of expression \( E \), and \( \text{norm}(E) \) is the normal form of \( E \).

**Proof.** Let \( \text{rdc}(E) \) be the function that returns the expression to which \( E \) converts by contracting the leftmost critical redex in \( E \), if \( E \) is not normal; and returns \( E \) if \( E \) is normal. Let \( \text{trdc}(E) \) be the function that returns the expression to which \( E \) converts by contracting all type redexes in \( E \) of level \( \text{lvl}(E) \) (and returns \( E \) if \( E \) is normal).

Define

\[
\begin{align*}
\text{rdc.mlt}(E, 0) &= E \\
\text{rdc.mlt}(E, k+1) &= \text{rdc.mlt}(\text{rdc.mlt}(E, k)) \\
\text{prev.dgr}(E) &= \text{rdc.mlt}(E, \text{rdx.mlt}(E)) \\
\text{rdc.dgr}(E, 0) &= E \\
\text{rdc.dgr}(E, k+1) &= \text{prev.dgr}(\text{rdc.dgr}(E, k)) \\
\text{prev.lvl}(E) &= \text{trdc}(\text{rdc.dgr}(E, \text{rdx.dgr}(E))) \\
\text{norm}(E) &= \text{prev.lvl}(E) \quad (9)
\end{align*}
\]

(9)

If \( \text{rdx.lvl}(E) \leq i \), then \( E \) converts to \( \text{norm}(E) \), and \( \text{norm}(E) \) is normal. This is proved by primary induction on \( i \), secondary induction on \( d = \text{rdx.dgr}(E, l) \), and ternary induction on \( \text{rdx.lvl}(E, l, d) \), and using Lemma 2.10.

A classification of \( P^R \) by depth of recursion was defined in [Hei61, Ax165]. Let \( R_0 = \text{rdc} E_0 \), and let \( R_k+1 \) consist of the functions defined by at most one recursion and by compositions from functions in \( R_k \). It is known that \( R_k = E_{k+1} \) for \( k \geq 2 \) [Schw69, Mu87].

For each \( i \), the function \( \text{norm} \) is defined using composition and two recursions from the functions \( \text{rdc} \) and \( \text{rdx.mlt} \). We have \( \text{rdc}(x) \leq x + x \) and \( \text{rdx.mlt}(x) < x \). So \( \text{norm} \in R_3 = E_4 \). ♦
THEOREM I  $\text{Slant}(S2\lambda^\omega) = \mathcal{E}_4$.

Proof.  $\text{Slant}(S2\lambda^\omega) \supseteq \mathcal{E}_4$ by Proposition 2.9.

For the converse, suppose that $E$ represents slantly in $S2\lambda^\omega$ a (unary) function $f$. Then, by Lemma 2.11, there is a function $B \in \mathcal{E}_4$ such that $f(n) \leq \text{length}(E) \leq B(n)$. Since $f$ is clearly primitive recursive, it follows from Lemma 2.8 that $f \in \mathcal{E}_4$. ♦

The functions representable in $S2\lambda^k$ are exactly those defined from 0 and successor by three iterations (allowing explicit definitions) followed by $k$ composition. The proof is omitted here.

3. Transfinite stratification

The rationale of stratifying types into finite levels lifts to levels labeled by any (countable) ordinal $\alpha$: a type has level $\alpha$ iff it is definable using quantification of levels up to $\alpha$. In particular, if a level is a limit ordinal then it is the union of the lower levels. At first blush such direct appeal to transfinite syntax may seem irrelevant to programming language design. However, we exhibit in §3 a simple type formalism based on the idea of transfinite typing but using strictly finitary syntax.

Let $\theta$ be a countable limit ordinal. We define a stratification $S2\lambda^\theta$ of $S2\lambda$ for ordinals $\prec \theta$. The typing rules are as for the system $S2\lambda^\omega$ above, except for an additional rule for limit ordinals $\xi \in \theta$, with countably many premises:

\begin{align}
\frac{\{ \eta \vdash E : \tau[\xi^0] \}_0^{\xi} }{ \eta \vdash E : \tau[\xi] } \quad (10)\end{align}

Here $t$ (with any level index) is not free in the range of $\eta$, and $\tau[\xi]$ denotes the type $\tau$ where the formal type variable identifier $t$ is indexed with level $\beta$.

DERIVATIONS are well founded trees of typing statements (i.e. every branch is finite), related by the type inference rules.

The use of infinite inferences with a $\xi$-list of premises, $\xi$ limit, can be reduced under mild assumptions to infinite inferences with $\omega$-lists of premises. Suppose the countable ordinal $\theta$ is given as a system: for every limit ordinal $\xi \in \theta$ there is given an increasing function from $\omega$ into $\xi$ whose limit is $\xi$. We write $\xi$ for that function too, so $\xi = \lim_{i \in \omega} \xi(i)$. \{\xi(i)\}_i \omega is a FUNDAMENTAL SEQUENCE for $\xi$. Reformulate the Type Union Rule as follows, under conditions as above.

\begin{align}
\frac{ \eta \vdash E : \tau[\xi^0] \}_0^{\omega} }{ \eta \vdash E : \tau[\xi] } \quad (11)\end{align}

Again, derivations are well-founded infinitary deduction figures. It is easy to see that, assuming given a system of fundamental sequences for ordinal $\prec \theta$, the two variants of Type Union are equivalent. The variant based on fundamental sequences is the basis of a parameterized version of the Union Rule, discussed in §3. The fundamental-sequence variant is similar to the use of $\omega$ rules in formalisms for arithmetic, for instance, inferring $\forall \lambda \phi(\lambda)$ from the $\omega$-sequences of premises $\phi[\xi]$. A fundamental difference between these two uses of infinitary rules, however, is that the $\omega$-rule in Arithmetic (even with proof figures of order type $\prec \omega^2$) yields all true arithmetic formulas as theorems, and is therefore not contained in any sound finitary formalism. By contrast, the Type Union Rule is infinitary only at a "control" level, and the resulting formalism is contained in the sound formalism of $2\lambda$ (where "sound", for a typed $\lambda$-calculus, means "ensuring conversion to normal form"):  

Proposition 3.1 Let $\theta$ be a countable ordinal. If a $\lambda$-expression $E$ can be typed in $S2\lambda^\theta$, then it can be typed in $2\lambda$.

Proof Sketch. Erase the level labels in the derivation in $S2\lambda^\theta$, and for applications of Type Union take the first (or any other) premise. (A precise proof proceeds by transfinite induction on the (ordinals of) proofs.) ♦

3.1. Parametric stratification

While the expression typed in the infinitary calculus $S2\lambda^\theta$ are all typed within $2\lambda$, we consider subsystems of $S2\lambda^\theta$. We consider first

The formulation above of transfinite stratification uses infinite type derivations, and even though such derivations can be described and manipulated in a coded form within finitary theories such as Peano Arithmetic, they seem of not direct relevance to programming methodology. We consider here a strictly finitary variant of transfinite typing. The idea is simply to require that the premises of the Type Union Rule be uniform for all ordinals in the fundamental sequence considered:

\begin{align}
\frac{ \{ \eta \vdash E : \tau[\xi^0] \}_0^{\omega} }{ \eta \vdash E : \tau[\xi] } \quad (12)\end{align}
where \(i\) is a variable ranging over \(\omega\). More precisely, one uses ordinal terms, in which numeric variables are used\(^1\).

We write \(\mathcal{P}2\lambda^\Theta\) for the variant of \(S2\lambda^\Theta\) where the Type Union Rule is parametric, as above.

### 3.2. Parametric ordinal notations over \(\omega^\omega\)

We outline a simple system of notations, sufficient for the ordinal machinery for \(\mathcal{P}2\lambda^\omega\). Define SIMPLE NUMERIC TERMS inductively: 0 and every NUMERIC VARIABLE \(i\) are simple numeric terms, and if \(s\) is such a term then so is \(Ss\). Thus every term \(s\) is of the form \(S^{\ell_i}S\), where \(\ell_i \in \{0, l_0, l_1, \ldots\}\) is the root of \(s\), and \(\ell\) is the LENGTH of \(s\). Write \(s \prec s'\) if \(s\) and \(s'\) have the same root, and \(\text{length}(s) < \text{length}(s')\). A (LEVEL) LABEL \(\alpha\) is a tuple \((\alpha_k, \ldots, \alpha_0)\) of simple numeric terms. For \(\alpha\) as above, set \(\text{length}(\alpha) = \ell_k\), \(\text{head}(\alpha) = \alpha_k\), and \(\text{tail}(\alpha) = \alpha_0\) (\(\alpha_k, \ldots, \alpha_0\)). The intention is that \(\alpha\) is a notation for the ordinal \(\omega^\omega \cdot \alpha_k + \cdots \cdot \omega^0 \cdot \alpha_0 < \omega^\omega\). We say that \(\alpha\) is a LIMIT LEVEL if \(\alpha_0 = 0\).

We define a relation \(\alpha \prec \alpha'\) which holds if, regardless of the numeric value assigned to the numeric variables \(i\), the ordinal denoted by \(\alpha\) precedes the ordinal denoted by \(\alpha'\). This is defined inductively by

- \(\text{length}(\alpha) < \text{length}(\alpha')\) and \(\text{head}(\alpha') > 0\), or
- \(\text{length}(\alpha) < \text{length}(\alpha')\), \(\text{head}(\alpha') = 0\), and \(\alpha < \text{tail}(\alpha')\), or
- \(\text{length}(\alpha) = \text{length}(\alpha')\) and \(\text{head}(\alpha) \prec \text{head}(\alpha')\), or
- \(\text{length}(\alpha) = \text{length}(\alpha')\), \(\text{head}(\alpha) = \text{head}(\alpha')\) (syntactic identity), and \(\text{tail}(\alpha) < \text{tail}(\alpha')\).

This order is the one to be used for the level condition on the Type Application Rule.

For a limit level \(\alpha\) and a simple numeric term \(s\), define

\[
\alpha(s) = \langle (\alpha_{k}, \ldots, \alpha_{r+1}, (\alpha_{r}, -1), s, 0 \ldots 0)\rangle,
\]

where \(\alpha_{r}, \ldots, \alpha_{r+1} = \cdots = \alpha_{0} = 0\), and \(k = \text{length}(\alpha)\).

This is the notational variant of the fundamental sequences, needed to state the parametric Type Union Rule.

\(^{1}\) An alternative is to simply use the infinite fundamental-sequence version of the Type Union Rule, with the stipulation that correct type derivations must have identical subderivations above each instance of the Type Union Rule. However, the point of parametric typing is to avoid infinite syntax altogether.

### 3.3. Provably correct type derivations

We have defined \(\mathcal{P}2\lambda\) as a restriction of the infinitary system \(S2\lambda\). Another class of natural restrictions is obtained by requiring the infinite type derivations to be provably correct in a suitable meta-theory (e.g. Peano Arithmetic). Here "provably correct" means that the derivation, \(\Delta\), say, is recursive, and that for a suitable index the theory proves the correctness of local inferences in \(\Delta\), as well as tree-induction on \(\Delta\) (see ...).

If \(T\) is such a meta-theory, we write \(\mathcal{P}2\lambda^\Theta[T]\) for that formalism (modulo a coding of \(\Theta\) in the language of \(T\)). Details here are similar to analogous restrictions of arithmetic with the \(\omega\) rule.

Note that

\[
\mathcal{P}2\lambda^\Theta \subseteq S2\lambda^\Theta|\mathcal{P}\mathcal{A}_0,
\]

where \(\mathcal{P}\mathcal{A}_0\) is \(\mathcal{P}\mathcal{A}\) without Induction. Thus, all results for \(S2\lambda^\Theta\) in \(\S 2\) lift to \(\mathcal{P}2\lambda^\Theta\).

### 3.4. Basic relations between typing calculi

Note that

\[
\mathcal{P}2\lambda^\Theta \subseteq S2\lambda^\Theta[\mathcal{P}\mathcal{A}_0] \subseteq S2\lambda^\Theta,
\]

where \(\mathcal{P}\mathcal{A}_0\) is \(\mathcal{P}\mathcal{A}\) without Induction. Thus, all results for \(S2\lambda^\Theta\) in \(\S 2\) lift to \(\mathcal{P}2\lambda^\Theta\).

**THEOREM II** Suppose \(E\) is a \(\lambda\)-expression that has a type derivation in \(\mathcal{P}2\lambda^\Theta\). Then \(E\) has a type derivation in in \(\mathcal{P}2\lambda^\omega\).

**Proof Sketch.** Replace in the given derivation every label \(\lambda(i_0)(i_1)\cdots(i_l)(\lambda\alpha)\) (where \(\lambda\) is any limit ordinal) by \(\omega^{k+1}(i_0)(i_1)(\cdots(i_l)(i_0) = \omega^k \cdot k + \cdots + \omega^0 \cdot i_0\). The result is a correct derivation. ♦

### 3.5. Placing the functions representable in \(S2\lambda^\Theta\)

We calibrate functions representable in \(S2\lambda^\Theta\) using transfinite recursion

**THEOREM III** Suppose that \(F\) is a numeric function represented in \(S2\lambda^\Theta\), even askew, by an expression of level \(\alpha\). Then \(F\) is \((\omega^3 \cdot \alpha)\)-recursive. ♦
Corollary 3.2 Suppose that $F$ is a numeric function represented in $S_2 \lambda^\omega$, even if we use, by an expression of level $\alpha = \omega^\beta$, where $\beta$ is a limit ordinal. Then $F$ is $\alpha$-recursive.

In particular, if $F$ is representable in $S_2 \lambda^\omega$ then $F$ is primitive recursive, and if $F$ is representable in $S_2 \lambda^\omega$ then $F$ is $\omega$-recursive. ♦

It is well known that the $\epsilon$-recursive functions are precisely the provably recursive functions of Peano Arithmetic. The connection between function representability in $S_2 \lambda$ and provably recursiveness in number theoretic formalisms can be stated in greater generality. The usual development of ordinal recursiveness in number theoretic formalisms can be proved in a formalism in which a suitable notion of transfinite induction may be adopted — we omit details for lack of space. The following is straightforward from Theorem III.

THEOREM IV Let $Z$ be a number theory (such as Peano Arithmetic or the arithmetical fragment of a formalism for Analysis or Set Theory). Suppose that $F$ is a numeric function represented, even askew, in $S_2 \lambda_\alpha[Z]$. Suppose that every instance of the schema of $\beta$-induction is provable in $Z$, where $\beta = (\omega^3 \cdot \alpha)$. Then $F$ is provably recursive (i.e. provably total) in $Z$.

In particular,

- The functions represented in $S_2 \lambda_\alpha[PA]$ are provably total in $PA_\alpha$ ($=_{df}$ Peano Arithmetic with Induction only over $\Pi_1$ formulas).
- The functions represented in $S_2 \lambda_\alpha[PA]$, are provably total in $PA$.
- The functions represented in $S_2 \lambda_\alpha[PA \setminus \text{Predicative Analysis}]$ are provably total in $PA$.

2In keeping with our statement of $\alpha$-recursion, one might adopt ordinal notations of the form $(x, c)$, where $c = 0, 1$ or 2 according to whether the ordinal $\beta$, denoted by $x$, is a successor of a limit, where for $x$ denoting a successor ordinal $\gamma + 1$, $(x)(0)$ is a notation for $\gamma$, and where for $x$ denoting a limit ordinal $\lambda$, $(x)(0), i \in \omega$ are notations for a fundamental sequence for $\lambda$. Moreover, the set of notations should be recursive. $\omega$-induction is then the schema $(\forall(x, c) \in A)(c = 0 \rightarrow \varphi(x) \land (c = 1 \rightarrow \varphi((x)(0)) \rightarrow \varphi(x)) \land (c = 2 \rightarrow (\forall i \in \omega)[(x)(0)]) \rightarrow \varphi(x))(\forall(x, c) \in A)(c)(x))(\forall(x, c) \in A)(c)).$ (This is not the shortest statement possible, but it seems well suited for applications.)

3.6. Representability by transfinitely stratified expressions

Proposition 3.3 Suppose that $F$ is defined from $G$ and $H$ by recursion, where for a certain limit $\lambda G$ and $H$ are representable in $S_2 \lambda^\omega$ at each level $\lambda(\xi)$. Then $F$ is representable in $S_2 \lambda^\omega$. ♦

From this we get

Proposition 3.4 For $k \geq 4$, all functions in $\mathcal{E}^k$ are representable in $S_2 \lambda_{\omega^{k-3}}$. Hence, all primitive recursive functions are representable in $S_2 \lambda^\omega$. ♦

The converse holds too:

Proposition 3.5 For $k \geq 4$, all functions representable in $S_2 \lambda_{\omega^{k-3}}$ are in $\mathcal{E}^k$. Hence, all functions representable in $S_2 \lambda^\omega$ are primitive recursive.

Proof Sketch. Define, for each $k$ and $d$, a function in $\mathcal{E}^{k+3}$ that bounds the size blow-up generated by normalization, for all expression in $S_2 \lambda^\omega$ with up to depth $d$ nesting of $\rightarrow$ in types. ♦

Let $PRA$ be the standard formalism for Primitive Recursive Arithmetic. We conclude:

THEOREM V Let $F$ be a recursive numeric function. The following conditions are equivalent.

1. $F$ is primitive recursive;
2. $F$ is representable in $PRA = P2\lambda = P2\lambda^\omega$;
3. $F$ is representable in $S_2 \lambda^\omega[\text{PRA}]$.

4. Complexity of equality in stratified calculi

Given a typed $\lambda$-calculus $L$, the EQUALITY PROBLEM FOR $L$, is the problem of deciding, given two expression in $L$, whether they are $\beta - \eta$-equal. Statman [Sta79] showed that $E_{\lambda} \notin \mathcal{E}_3$.

THEOREM VI $Eq[S2\lambda^\omega] \in \mathcal{E}_5 - \mathcal{E}_4$. 

45
Proof. Extending the proof of Lemma 2.11 with one additional recursion yields a function in \( \mathcal{E}_5 \) that bounds the depth of the normal form of any expression in \( S2\lambda^\omega \). Thus \( Eq(S2\lambda^\omega) \in \mathcal{E}_5 \).

Suppose \( Eq(S2\lambda^\omega) \in \mathcal{E}_4 \). Let \( E_n \) be an enumeration in \( \mathcal{E}_4 \) of all expressions in \( S2\lambda^\omega \). The assumption implies that the function

\[
F(E_n) = \begin{cases} 
1 & \text{if } E_n \equiv \rho_0 \tilde{0} \\
0 & \text{otherwise}
\end{cases}
\]

is in \( \mathcal{E}_4 \), hence represented by some \( E_k \in S2\lambda^\omega \). But then \( E_k \equiv \rho_0 \tilde{0} \) iff \( E_k \not\equiv \rho_0 \tilde{0} \), a contradiction. ♦

A similar proof establishes:

**THEOREM VII** \( Eq(S2\lambda^\omega) \) is not primitive recursive (but is double recursive).

5. Open problems and research directions

1. Methodological problems, axiomatization: Are there finite type disciplines that capture exactly the nonuniform \( S2\lambda^\theta, \) for \( \Theta > \omega \)? If not, what is the complexity (in the arithmetical hierarchy, say) of the set of \( \lambda \)-expressions typed in these disciplines?

2. Stratification combined with other type paradigms.

3. Extensions of stratified quantification to higher kinds.

4. Combination of stratification with other predicative type disciplines, in particular Martin-Löf’s Type Theory (yielding a stratified version of Coquand-Huet’s Theory of Constructions) and Constable’s family of PRL languages.

5. Connections with proof theoretic calibrations of subrecursive hierarchies [Schw72].

6. Model theory of stratified disciplines. Identify class of models complete for functional equations, comparable to [Frt75] for \( 1\lambda \). Relations between size of models and level of stratification.

References


