# Contracting Proofs to Programs

Daniel Leivant

July 25, 1989
CMU-CS-89-170

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*To appear in*
*Piergiorgio Odifreddi (Editor)*, **Logic and Computer Science**, *Academic Press.*

## Abstract

This work describes a family of homomorphisms that contract natural deductions into typed $\lambda$-expressions, with the property that a convergence proof for an untyped program for function $f$ is contracted to a typed program for $f$. The main novelties, compared to previous works on extracting algorithms from proofs, are the reading of deductions themselves as programs, and that instead of a constructive reading of $\exists$, we use a Leibnitzian view of objects as sets of properties. The method is based on the observation that object-level components of natural deductions can be ignored computationally. It is applicable to every formalism in which there are no axioms or rules for objects in general, only for properties. Formalisms of this type include Peano's first order axiomatization of arithmetic (in its original form, with a primitive predicate identifier $N$), second and higher order logic (in which data types, like the natural numbers, can be defined explicitly), and various variants of fixpoint extensions of first order logic.

Among the technical offshoots of the method are very simple and transparent proofs of Girard's Theorem, that the provably recursive functions of second order arithmetic are all representable in the second order $\lambda$-calculus, and of Gödel's "Dialectica" Theorem, that the provably recursive functions of first order arithmetic are all computable using primitive recursion in all finite types.

This research was partially supported by ONR Grant N00014-84-K-0415, and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under Contract Number F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFCS), Wright-Patterson AFB, Ohio 45433-6543.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR, DARPA, or the U.S. Government.

## Introduction

Most techniques for extracting the algorithmic contents of proofs fall into three methods. The *interpretational method* maps a formula $\varphi$ to a formula $\varphi^c$ that renders the constructive contents of $\varphi$. One shows that if $\varphi$ is provable constructively, then $\varphi^c$ is true. This is the method underlying Kleene's realizability interpretations, Gödel's "Dialectica" interpretation, and Kreisel-Goodman's Theory of Constructions.

The *proof normalization* method is based on the special form taken by normal proofs in semantic directed calculi for constructive theories, such as natural deductions for Intuitionistic (Heyting's) First Order Arithmetic. The key property used is *existential instantiation*: a normal proof of a formula $\exists x.\varphi$ must have, as premise of its last inference, $\varphi[t/x]$ for some term $t$.

The *proof combinatorics* method attempts to interpret a natural deduction proof as comprising an algorithm. The method is motivated by Curry and Howard's "formula as type" isomorphism [CF58,How80,Lau70,deB70], and underlies the programming language of Per Martin-Löf [Mar79], the family of PRL systems [Con86], and Coquand and Huet's Calculus of Constructions [CH88, Coq].

Our method is a variant, originating in [Lei83], of the proof combinatorics method. It is based on the observation that object-level components of natural deductions can be ignored computationally, leading to a *homomorphism* from proofs to typed programs, that extracts the computational contents of deductions. The underlying rationale goes back to Leibnitz: in a general setting where objects have a priori no special property, reasoning and computing are with respect to objects' properties, rather than over the objects themselves. The fact that an object $x$ has property $P$ will be reflected in the combinatorics of (potential) proofs of $P(x)$. For instance, if $N$ is a suitable rendition of the property of being a natural number, then a proof of $N(x)$ will have a structure that reflects the construction of $x$.

This approach is applicable to every formalism in which there are no axioms or rules for objects in general, only for properties. Formalisms of this type include Peano's first order axiomatization of arithmetic (in its original form, with a primitive predicate identifier $N$), second and higher order logic (in which data types, like the natural numbers, can be defined explicitly), and various variants of fixpoint extensions of first order logic.

Our approach differs in important respects from previous works on proofs as programs, such as Martin-Löf's Type Theory (MLTT). We disregard abstraction over individual objects, whereas object abstraction is at the core of the MLTT style. The constructive contents of proofs rests, for us, in the combinatorics of abstraction and application within proofs, in contrast to MLTT, where the constructive contents lies mainly in a constructive interpretation of existential quantification. Our approach is committed to a "semantic" view of typing (types as properties) as opposed to the "ontological" view of (most variants of) MLTT, where objects come equipped with their type. Finally, our method is particularly suited for reasoning about functions that are partial with respect to data types.

Among the technical offshoots of the method are very simple and transparent proofs of Girard's Theorem, that the provably recursive functions of second order arithmetic are all representable in the second order $\lambda$-calculus, and Gödel's "Dialectica" Theorem, that the provably recursive functions of first order arithmetic are all computable using primitive recursion in all finite types.

We outline the use of our contraction homomorphisms for three types of calculi. In §§1–4 we present the method and some of its ramifications and applications for pure second order logic, which is mapped to Girard-Reynolds's pure second order $\lambda$-calculus. We start with this instance of the method because the target formalism is a pure $\lambda$ calculus. In §5 we exhibit the method for a variant of Peano's Arithmetic and similar "generative axiomatizations" of inductively generated data types. §6 outlines applications to "inductive axiomatizations", based on first order inductive definitions. Finally, §7 touches on the contraction of second order proofs with restricted forms of comprehension, to second order typed $\lambda$-expressions with restrictions on type arguments. Since we deal here with functions provable in second order *logic*, restricting comprehension leads to computational classes well below the provably recursive functions of first order arithmetic, such as the primitive recursive functions, the elementary, and the super-elementary functions [Lei89, Lei$\alpha$, Lei$\beta$, Lei$\gamma$]. A number of technical elaborations are factored out into appendices, to avoid distraction from the main development.

The main results of §§1–3 were reported in [Lei83]. They were rediscovered by Krivine and Parigot [Kri86, KP87]. Closely related results are described in [Gir89] (see discussion following Theorem V below). The main results of §§4,5 are contained in §2 of [Lei84], and were reported in December 1983 at the Workshop on Logic in Computer Science at Brooklyn College of CUNY.

**Acknowledgements.** I am grateful to Phokion Kolaitis, Georg Kreisel, Michel Parigot, Jonathan Seldin, and Paul Taylor for comments on a preliminary version of this work. Research partially supported by ONR grant N00014-84-K-0415 and by DARPA grant F33615-87-C-1499, ARPA Order 4976, Amendment 20.

# 1. A contraction homomorphism from second order deductions to λ-programs

## 1.1. Natural deductions for minimal second order logic

We use a Gentzen-Prawitz style natural deduction calculus [Gen34, Pra65], **M2L**, for minimal second order logic, with implication and universal quantification as the only logical constants. The inference rules are:

$$\rightarrow\mathbf{I}: \quad \frac{\begin{array}{c}\chi^i \\ \dots \\ \varphi\end{array}}{\chi \rightarrow \varphi} \quad \text{(occurrences } \chi^i \text{ of } \chi \text{ are closed)} \qquad \rightarrow\mathbf{E}: \quad \frac{\chi \rightarrow \varphi \quad \chi}{\varphi}$$

$$\forall\mathbf{I}: \quad \frac{\varphi}{\forall x \varphi} \quad (x \text{ not free in assumptions}) \qquad \forall\mathbf{E}: \quad \frac{\forall x.\, \varphi}{\varphi[t/x]}$$

$$\forall^2\mathbf{I}: \quad \frac{\varphi}{\forall R.\varphi} \quad (R \text{ not free in assumptions}) \qquad \forall^2\mathbf{E}: \quad \frac{\forall R\, \varphi}{\varphi[\lambda \hat{u}.\psi/R]}$$

*Derivations* are defined as usual. When appropriate, we use numbered lists of formulas as concrete syntax for derivations.

We consider computability in the equational style of Herbrand-Gödel (see Appendix I below). For the rest of this paper, a *program* is a finite set of defining equations for function identifiers, possibly by (simultaneous) recursion. Given a program $\mathcal{P}$, **M2L** + $\mathcal{P}$ will denote the extension of **M2L** with the rule:

$$\mathcal{P}: \quad \frac{\varphi[t/x]}{\varphi[t'/x]} \qquad t = t' \text{ or } t' = t \text{ is a substitution instance of an equation in } \mathcal{P}$$

## 1.2. A homomorphism from deductions to λ-expressions

Curry and Howard's isomorphism has formulas correspond to types and deductions to λ-expressions. In defining the isomorphism for quantifiers, Howard was led to defining richer type structures, with dependent type operations. These were discovered independently by de Bruijn for the AUTOMATH project [deB70], and further developed by Martin-Löf, lending evidence to their naturalness and utility.

We pursue a dual approach: rather than enriching the type structure to match logic, we impoverish logic to match the type structure. The co-domain of our homomorphism is Girard-Reynolds' second order (polymorphic) typed $\lambda$ calculus, $2\lambda$ [Gir72, Rey74] (see expositions in [Sce] or [FLO83]). We write $E : \tau$ for "$E$ is of type $\tau$." We use the same identifiers for relational variables of **M2L** and for type variables of $2\lambda$.

For a formula $\varphi$, let the type $\underline{\varphi}$ be obtained by deleting from $\varphi$ the first order components: $\underline{R(\hat{t})} =_{Df} R$; $\underline{\psi \rightarrow \chi} =_{Df} \underline{\psi} \rightarrow \underline{\chi}$; $\underline{\forall x.\psi} =_{Df} \underline{\psi}$; $\underline{\forall R.\psi} =_{Df} \forall R.\underline{\psi}$. For example, if $\varphi \equiv \forall R(\forall z(R(z) \rightarrow R(sz)) \rightarrow R(\hat{0}) \rightarrow R(\hat{t}))$ (stating that $t$ denotes a natural number), then $\underline{\varphi} = \forall R.((R \rightarrow R) \rightarrow (R \rightarrow R))$, i.e. the Fortune-O'Donnell type of the natural numbers [FLO83].

By straightforward induction on formulas we have:

**Lemma 1..1** *If* $\varphi \equiv \psi[\lambda\hat{x}.\chi\ /R]$, *then* $\underline{\varphi} = \underline{\psi}[\underline{\chi}/R]$. $\dashv$

Next we define a mapping $\kappa$ that assigns to each derivation $\Pi$ of **M2L**$+\mathcal{P}$ deriving a formula $\varphi$ (where $\mathcal{P}$ is a program), an expression $\kappa\Pi$ of $2\lambda$, of type $\underline{\varphi}$.

| | |
|---|---|
| $\Pi \equiv \varphi^i$ (open assumption $\varphi$ labeled by $i$) | $\kappa\Pi =_{Df} x_i^{\underline{\varphi}}$ (variable of type $\underline{\varphi}$) |
| $\Pi \equiv \dfrac{\begin{array}{c}\chi^i \\ \Delta \\ \varphi\end{array}}{\chi \to \varphi}$ | $\kappa\Pi =_{Df} \lambda x_i^{\underline{\chi}}.\kappa\Delta$ |
| $\Pi \equiv \dfrac{\Delta \quad \Theta}{\psi \to \varphi \quad \psi}{\varphi}$ | $\kappa\Pi =_{Df} (\kappa\Delta)(\kappa\Theta)$ |
| $\Pi \equiv \dfrac{\begin{array}{c}\Delta \\ \psi\end{array}}{\forall x.\psi}$ | $\kappa\Pi =_{Df} \kappa\Delta$ |
| $\Pi \equiv \dfrac{\begin{array}{c}\Delta \\ \forall x.\psi\end{array}}{\psi[t/x]}$ | $\kappa\Pi =_{Df} \kappa\Delta$ |
| $\Pi \equiv \dfrac{\begin{array}{c}\Delta \\ \psi[t/x]\end{array}}{\psi[t'/x]}$ (by $\mathcal{P}$) | $\kappa\Pi =_{Df} \kappa\Delta$ |
| $\Pi \equiv \dfrac{\begin{array}{c}\Delta \\ \psi\end{array}}{\forall R.\psi}$ | $\kappa\Pi =_{Df} \Lambda R.\kappa\Delta$ |
| $\Pi \equiv \dfrac{\begin{array}{c}\Delta \\ \forall R.\psi\end{array}}{\psi[\lambda\hat{x}.\chi/R]}$ | $\kappa\Pi =_{Df} (\kappa\Delta)\underline{\chi}$ |

By a straightforward induction on derivations we have

**Lemma 1..2** *If $\Pi$ derives $\varphi$ from the open assumptions $\psi_1^{i_1} \ldots \psi_k^{i_k}$, then $\kappa\Pi$ is of type $\underline{\varphi}$, with free variables $x_{i_1}^{\psi_1} \ldots x_{i_k}^{\psi_k}$, of types $\underline{\psi}_1 \ldots \underline{\psi}_k$, respectively.*

The induction step for the case of second order $\forall$ elimination uses Lemma 1..1. The induction step for second order $\forall$ introduction uses, for the syntactic correctness of $\kappa\Pi$, the condition on $\Pi$ that $R$ is not free in open assumptions.

$\kappa$ can be extended to the remaining logical constants, using their definition in terms of implication and universal quantification [Pra65]. For example, $\varphi \wedge \psi \equiv \forall R^0((\varphi \to \psi \to R) \to R)$, so a proof $\Pi$ of $\varphi \wedge \psi$ is obtained from a proof $\Delta$ of $\varphi$ and a proof $\Theta$ of $\psi$ by two uses of $\to$ elimination and a use of $\forall$ introduction, yielding $\kappa\Pi \equiv \Lambda R.\ \lambda u.u(\kappa\Delta)(\kappa\Theta)$ (where $u : \underline{\varphi} \to \underline{\psi} \to R$).

### 1.3. $\kappa$ commutes with reductions

Prawitz's reductions for **M2L**, take the following forms [Pra65]:

**Implication:** $\Pi \equiv \dfrac{\begin{array}{cc}[\psi^i] & \\ \Delta & \\ \varphi & \Theta \\ \psi \to \varphi & \psi\end{array}}{\varphi}$ reduces to $\Pi' \equiv \begin{array}{c}\Theta \\ [\psi^i] \\ \Delta \\ \varphi\end{array}$

**Object $\forall$:** $\Pi \equiv \begin{array}{c}\Delta \\ \varphi \\ \forall x.\varphi \\ \varphi[t/x]\end{array}$ reduces to $\Pi' \equiv \begin{array}{c}\Delta[t/x] \\ \varphi[t/x]\end{array}$

**Relation $\forall$:** $\Pi \equiv \begin{array}{c}\Delta \\ \varphi \\ \forall R.\varphi \\ \varphi[\lambda\hat{u}.\chi/R]\end{array}$ reduces to $\Pi' \equiv \begin{array}{c}\Delta[\lambda\hat{u}.\chi/R] \\ \varphi[\lambda\hat{u}.\chi/R]\end{array}$

**Lemma 1..3 (Homomorphism)** *Let $\Pi$ be an* **M2L** *derivation. If $\Pi$ reduces to $\Pi'$ by an application of one of Prawitz's rules above, then $\kappa\Pi$ $\beta$-converts (in $2\lambda$) to $\kappa\Pi'$. Consequently, if $\Pi$ reduces (by successive reductions) to $\Omega$, then $\kappa\Pi$ $\beta$-reduces to $\kappa\Omega$, and if $\kappa\Pi$ is $\beta$-normal, then $\Pi$ is normal.*

*Conversely, if $E = \kappa\Pi$ $\beta$-reduces to $E'$, then $E' = \kappa\Pi'$ for some $\Pi'$ such that $\Pi$ Prawitz-reduces to $\Pi'$. Consequently, if $\Pi$ is normal, then $\kappa\Pi$ is $\beta$-normal.* $\dashv$

For a comment on $\eta$-reductions see Appendix 9.1.

### 1.4. Extension of $\kappa$ to conjunction

Let [$2\lambda$+ pairing] be the extension of $2\lambda$ with pairing: types are generated also using the product construct, $\tau \times \sigma$, and expressions are generated also using pairing and projections: if $E : \tau$ and $F : \sigma$, then $\langle E, F\rangle : \tau \times \sigma$; and if $E : \tau_1 \times \tau_2$, then $\mathbf{j}_i E : \tau_i$ $(i = 1, 2)$. An additional reduction rule is: $\mathbf{j}_i\langle E_1, E_2\rangle$ reduces to $E_i$ $(i = 1, 2)$.

$\kappa$ has an obvious extension to $\kappa$ : **M2L** + conjunction $\rightarrow$ $2\lambda$ + pairing. Let the mapping $\varphi \mapsto \underline{\varphi}$ be extended by the clause $\underline{\psi \wedge \chi} =_{Df} \langle\underline{\psi}, \underline{\chi}\rangle$. Extend the definition of $\kappa$ by the clauses

$$\Pi \equiv \frac{\begin{array}{cc}\Delta_1 & \Delta_2\\ \varphi_1 & \varphi_2\end{array}}{\varphi_1 \wedge \varphi_2} \qquad \kappa\Pi =_{Df} \langle\kappa\Delta_1, \kappa\Delta_2\rangle$$

---

$$\Pi \equiv \frac{\Delta}{\varphi_1 \wedge \varphi_2\ /\ \varphi_i} \qquad \kappa\Pi =_{Df} \mathbf{j}_i\kappa\Delta$$

Lemma 1..3 applies then to the extended $\kappa$.

Alternatively, conjunction can be considered, within **M2L**, as a defined connective [Pra65]: $\varphi \wedge \psi \equiv \forall Q((\varphi \rightarrow \psi \rightarrow Q) \rightarrow Q)$. From this definition and from the **M2L** derivations of the conjunction rules for it we obtain:

$$\begin{aligned}\tau_1 \times \tau_2 &=_{Df} \forall Q((\tau_1 \rightarrow \tau_2 \rightarrow Q) \rightarrow Q)\\ \langle E^\tau, F^\sigma\rangle &=_{Df} \Lambda Q.\lambda a^{\tau\rightarrow\sigma\rightarrow Q}.\, aEF\\ \mathbf{j}_1 E^{\tau\times\sigma} &=_{Df} E\tau(\lambda x^\tau y^\sigma.\, x)\\ \mathbf{j}_2 E^{\tau\times\sigma} &=_{Df} E\sigma(\lambda x^\tau y^\sigma.\, y)\end{aligned}$$

### 1.5. Extension of $\kappa$ to first order $\exists$

Although $\exists$ is definable in **M2L** in terms of $\rightarrow$ and $\forall$ [Pra65], it can be included, at no cost, in **M2L** and in the definition of $\kappa$, by the following clauses.

$$\Pi \quad \equiv \quad \dfrac{\begin{array}{c}\Delta\\ \psi[t/x]\end{array}}{\exists x.\psi} \qquad\qquad \kappa\Pi =_{Df} \kappa\Delta$$

---

$$\Pi \quad \equiv \quad \dfrac{\begin{array}{cc}\Delta & \psi^i\\ \exists x\psi & \varphi\end{array}}{\varphi} \qquad\qquad \kappa\Pi =_{Df} (\kappa\Theta)[\kappa\Delta/x_i^{\psi}]$$

Prawitz's reduction for Gentzen's natural deduction rules for $\exists$ is mapped under $\kappa$ into identity, exactly as for the $\forall$ reduction:

**Object $\exists$:** $\quad \Pi \quad \equiv \quad \dfrac{\begin{array}{cc}\Delta & [\,\psi^i\,]\\ \psi[t/x] & \Theta\\ \exists x\psi & \varphi\end{array}}{\varphi}$ reduces to $\quad \Pi' \quad \equiv \quad \begin{array}{c}\Delta\\ [\,\psi^i[t/x]\,]\\ \Theta[t/x]\\ \varphi\end{array}$

We have $\kappa\Pi' = \kappa\Pi$. Similarly, the *permutative* for $\exists$ elimination [Pra65] are idempotent under $\kappa$.

The clause of $\kappa$ for $\exists$-elimination is most transparent for a formulation of $\exists$-elimination as an instantiation rule. See Appendix 9.2.

## 2. Convergence proofs as polymorphic $\lambda$-programs

### 2.1. A second order statement of convergence

Let $N(x) \equiv_{Df} \forall R.\forall u\,(R(u) \to R(\mathrm{s}(u)) \to R(\mathbf{0}) \to R(x)$. $N$ defines a copy of the natural numbers in every structure $S$ that satisfies Peano's Third and Fourth Axioms, $\forall x, y.\ \mathrm{s}x = \mathrm{s}y \to x = y$, and $\forall x.\ \mathrm{s}x \neq \mathbf{0}$, which guarantee that the denotations of the numerals, $\bar{n} =_{Df} \mathrm{s}^{[n]}\mathbf{0} \equiv \mathrm{s}(\mathrm{s}(\cdots \mathrm{s}(\mathbf{0})\cdots))$, are all distinct.

If $\mathcal{P}$ is a program, we also write $\mathcal{P}$ for the conjunction of the universal closures of the equations in $\mathcal{P}$. The totality of the numeric function $f$ computed by $\mathcal{P}$ is expressed by the formula $\forall x \exists y\, T(\bar{e}_{\mathcal{P}}, x, y)$, where $e_{\mathcal{P}}$ is a Gödel number for $\mathcal{P}$ (under some canonical coding), and $T$ is Kleene's *computability* predicate: $T(e, x, y)$ holds if $y$ codes a completed computation of the program coded by $e$ on input $\bar{x}$ [Kle52]. The following is a variant of Dedekind's century old observation that the standard structure of the natural numbers is characterized by second order quantification. For a function identifier $\mathbf{f}$ of arity $r$, let $N^1(\mathbf{f}) \equiv_{Df} \forall \hat{z}.N(\hat{z}) \to N(\mathbf{f}\hat{z})$, where $\hat{z} = (z_1 \ldots z_r)$ and $N(\hat{z}) \equiv_{Df} N(z_1) \wedge \cdots \wedge N(z_r)$.

**THEOREM I** *Let $\mathcal{P}$ be a program, with target $\mathbf{f}$, that computes the function $f$. The following are equivalent:*

1. $f$ is total over $\omega$: $\mathcal{P}$ yields an output for every input.
2. $\mathcal{P} \models N^1(\mathbf{f})$, with respect to standard models: every standard second order structure in which $\mathcal{P}$ is true, satisfies $N^1(\mathbf{f})$.

**Proof.** Assume, without loss of generality, that $\mathbf{f}$ is unary.

Assume (1). Then, for every numeral $\bar{x}$, there is a completed symbolic computation $\mathcal{C}$ of $\mathcal{P}$ for input $\bar{x}$, with the final equation of the form $\mathbf{f}(\bar{x}) = \bar{z}$, for some numeral $\bar{z}$. $\mathcal{C}$ preserves equality in any model of the formula $\mathcal{P}$. Therefore, in every such model, if $x$ is the denotation of a numeral, then $fx$ is the denotation of a numeral. Since in a standard second order structure every element satisfying $N$ is the denotation of a numeral, this establishes (2).

Conversely, assume (2). Consider the structure $S$ with universe $|S| = \omega \cup \{\perp\}$ of the natural numbers augmented with an object $\perp$, and where for each function identifier $\mathbf{g}$ occurring in $\mathcal{P}$,

$$\mathbf{g}^S(x_1 \ldots x_r) = \begin{cases} y & \text{if } \mathcal{P} \text{ yields } \mathbf{g}(\bar{x}_1 \ldots \bar{x}_r) = \bar{y} \\ \perp & \text{otherwise (including if some } x_i \text{ is } \perp) \end{cases}$$

Then, by (2), $S \models \forall x.N(x) \to N(\mathbf{f}x)$. In $S$, $N$ is satisfied exactly by the elements of $\omega$, so $\mathbf{f}^S$ maps natural numbers to natural numbers. By the definition of $\mathbf{f}^S$ this implies that $\mathcal{P}$ has a completed computation for any numeral as input. $\dashv$

The second condition cannot be extended to all Henkin models of some formal calculus for second order logic, because that would make (2) equivalent to formal provability, an r.e. property.

### 2.2. Provably recursive functions of second order logic

The equivalence stated in Theorem I leads to an equivalence between provability conditions in second order arithmetic (analysis) and in second order logic, respectively.

Let **C2L**, **I2L** **M2L** be *classical, intuitionistic,* and *minimal second order logic*, respectively. Let **C2A**, **I2A**, and **M2A**, be the classical, intuitionistic and minimal variants of second order arithmetic. See e.g. [Schü77] for a detailed description of these formalisms.

We say that a program $\mathcal{P}$ is *standardized* if it contains recursion equations for the predecessor function, $\mathbf{pred}(\mathbf{0}) = \mathbf{0}$, $\mathbf{pred}(\mathbf{s}x) = x$. It is easy to see that every model of a standardized program satisfies Peano's Third Axiom, and the Fourth Axiom formulated as $\forall x.\mathbf{s}x = \mathbf{0} \rightarrow \forall x.x = bf\mathbf{0}$. We restrict attention to standardized programs to gain simplicity without sacrificing generality (any program can be trivially expended to a standardized one with no change of semantics), though the restriction can probably be bypassed.

**THEOREM II** *Let $\mathcal{P}$ be a standardized program, with target* **f**. *The following are equivalent.*

1. *$\mathcal{P}$ is provably total in classical second order arithmetic:* $\vdash_{\mathbf{C2A}} \forall \hat{x} \exists y\, T(\hat{e}_{\mathcal{P}}, \hat{x}, y)$.

2. *$\mathcal{P}$ is provably total in* **I2A**.

3. *$\mathcal{P}$ is provably total in* **I2L**: $\mathcal{P} \vdash_{\mathbf{I2L}} N^1(\mathbf{f})$.

4. *$\mathcal{P}$ is provably total in* **M2L**.

5. *$\mathcal{P}$ is provably total in* **C2L**.

**Proof.** For simplicity, say $f$ is unary. The implication from (1) to (2) is well known. It falls out from any one of the double negation translations of classical into intuitionistic second order arithmetic (see e.g. [Tro73]), combined with the closure of the latter under Markov's Rule [Gir72,§6.2.1].

Assume (2), towards proving (3). Let $Dfn[T]$ and $Dfn[U]$ be defining clauses for Kleene's $T$ predicate and for the graph $U$ of Kleene's result extraction function [Kle52]. Since $\mathcal{P}$ is standardized, (2) implies

$$\mathcal{P}, Dfn[T]^N \vdash_{\mathbf{I2L}} N(x) \rightarrow \exists y.N(y) \wedge T(\hat{e}_{\mathcal{P}}, x, y),$$

where $Dfn[T]^N$ is $Dfn[T]$ with universal quantifiers restricted to $N$. Since $Dfn[T]$ implies $Dfn[T]^N$ trivially,

$$\mathcal{P}, Dfn[T] \vdash_{\mathbf{I2L}} N(x) \rightarrow \exists y.N(y) \wedge T(\bar{e}_{\mathcal{P}}, x, y).$$

Similarly,

$$\mathcal{P}, Dfn[U] \vdash_{\mathbf{I2L}} N(y) \rightarrow \exists r.N(r) \wedge U(y, r),$$

and

$$\mathcal{P}, Dfn[T], Dfn[U] \vdash_{\mathbf{I2L}} N(x) \rightarrow T(\bar{e}_{\mathcal{P}}, x, y) \rightarrow N(y) \rightarrow U(y, r) \rightarrow \mathbf{f}x = r.$$

Combining these, we get

$$\mathcal{P}, Dfn[T], Dfn[U] \vdash_{\mathbf{I2L}} N(x) \rightarrow N(\mathbf{f}x).$$

The second order existential closures of $Dfn[T]$ and $Dfn[U]$ are both provable in **I2L**. So the last statement simplifies to

$$\mathcal{P} \vdash_{\mathbf{I2L}} N(x) \rightarrow N(\mathbf{f}x),$$

proving (3).

(3) implies (4), since falsehood, $\perp$, is definable in **M2L** by $\forall R.R$.

(4) implies (5) trivially.

Assume (5), towards proving (1). For second order formulas $\varphi$, with function identifiers from $\mathcal{P}$, we define an interpretation $\bar{\varphi}$ of $\varphi$ in second order arithmetic. For an arithmetic term $t$, let $t \simeq z$ abbreviate the first order formula "$z$ is the numeric value of $t$ with respect to $\mathcal{P}$", as in [Kle69]. Let $\bar{\varphi}$ arise from $\varphi$ by replacing each atomic subformula $Q(t_1 \ldots t_k)$ by $\exists u_1 \cdots u_k \, (t_1 \simeq u_1 \wedge \cdots \wedge t_k \simeq u_k \wedge Q(u_1 \ldots u_k))$. By induction on derivations, if $\mathcal{P} \vdash_{\mathbf{C2L}} \psi$ then $\vdash_{\mathbf{C2A}} \bar{\psi}$, proving (1). $\dashv$

Note that the theorem provides a method of reasoning about computable function convergence without reference to existential quantifiers, and without coding mechanisms.

### 2.3. Lambda representation of numbers and functions

Consider the numeral $\bar{2} \equiv \mathbf{ss0}$. If $\Pi$ is the direct proof of $N(\bar{2})$, then $\kappa\Pi$ is easily seen to be $\Lambda R.\lambda s^{R \rightarrow R}.\lambda z^R.s(s(z))$. More generally,

**THEOREM III** *Let $\bar{k} \equiv \mathbf{s}^{[k]}\mathbf{0}$. There is a unique normal* **M2L** *deduction $\Pi_k$ of the formula $N(\bar{k})$, for which $\kappa\Pi_k = \Lambda R \lambda s^{R \rightarrow R} z^R.\, s^{[k]}(z)$. (This is the Fortune-O'Donnell k'th numeral [FLO83], of which the untyped form is Church's k'th numeral [Chu33].)*

*More generally, if $t$ is a closed term over* **0**,**s** as well as the identifiers of $\mathcal{P}$, *and $\Pi$ is a $\beta$-normal derivation in* **M2L** $+ \mathcal{P}$ *of $N(t)$, then $\mathcal{P}$ derives $t = \bar{x}$ for some $x$, and $\kappa\Pi$ is the Fortune-O'Donnell numeral for $\bar{x}$.*

**Proof.** A normal derivation of $N(t)$ must end with three introductions (possibly interleaved with instances of $\mathcal{P}$), whose premise derivation $\Theta$ derives $R(t')$ from the assumptions $R(\mathbf{0})$ and $\forall z.R(z) \rightarrow R(\mathbf{s}z)$, where $R$ is a relational variable, and $\mathcal{P} \vdash t' = t$. An induction on such derivations $\Theta$ shows that $\kappa\Theta = s^{[k]}z$, where $\mathcal{P}$ derives $\bar{k} = t' = t$, $s$ is a variable of type $R \rightarrow R$ (corresponding to the assumption $\forall z.R(z) \rightarrow R(\mathbf{s}z)$), and $z$ is a variable of type $R$ (corresponding to the assumption $R(\mathbf{0})$). $\dashv$

**THEOREM IV (Numeric Function Representation)** *Let $\mathcal{P}$ be a program, with target* **f**, *that computes the function $f$. If $\Pi$ is an* **M2L** $+ \mathcal{P}$ *derivation of $N_1(\mathbf{f})$, then $\kappa\Pi$ $\beta$-represents $f$ in* $2\lambda$. *Hence, if $E$ is a $2\lambda$ expression such that $E =_{\beta\eta} \kappa\Pi$, then $E$ $\beta\eta$-represents $f$ in $2\lambda$.*

**Proof.** For simplicity, say $f$ is unary. For every $k \geq 0$ the derivation

$$\Delta_k \equiv \dfrac{\begin{array}{cc} \Pi & \\ N_1(f) & \Pi_k \\ N(\bar{k}) \rightarrow N(\mathbf{f}\bar{k}) & N(\bar{k}) \end{array}}{N(\mathbf{f}\bar{k})}$$

reduces to a normal derivation $\Delta'_k$ of $N(\mathbf{f}\bar{k})$. So we have

$$\begin{array}{rcll} (\kappa\Pi)\bar{k} & = & (\kappa\Pi)(\kappa\Pi_k) & \\ & = & \kappa\Delta_k & \\ & =_\beta & \kappa\Delta'_k & \text{by Lemma 1..3} \\ & =_\beta & \kappa\Pi_{fk} & \text{by Theorem III} \\ & =_\beta & \overline{fk} & \end{array}$$

$\dashv$

Combining Theorem IV with Theorem II we have

**THEOREM V (Girard [Gir72])** *All the provably total computable functions of classical second order arithmetic are representable in* $2\lambda$. $\dashv$

Developments similar to this are reported in [Kri86, KP87] and [Gir89, Chapter 15]. In the latter, Girard treats proofs in **I2A** (= $\mathbf{HA}_2$) of formulas of the form $\forall x.N(x) \rightarrow \exists y.N(y) \wedge T(e, x, y)$. In that setting, the presence of Peano's Third and Fourth Axioms requires additional non-trivial considerations (whereas in our setting the problem is eliminated trivially by considering standardized programs).

The converse of Theorem V was obtained by Girard as a consequence of his proof that $2\lambda$ has the normalization property. A corollary of Theorem V is:

**THEOREM VI (Girard)** *The normalization property of* $2\lambda$ *is not provable in second order arithmetic.*

**Proof.** Let $\iota =_{Df} \forall R.(R \to R) \to (R \to R)$. Fix a canonical (primitive recursive) enumeration, $E_1, E_2, \ldots$, of the $2\lambda$-expressions of type $\iota \to \iota$. Let $f(n,m) =_{Df} value(E_n\bar{m})$. Suppose the normalization property of $2\lambda$ were provable in C2A; then $f$ were provably total in C2A, and therefore representable in $2\lambda$, by Theorem V. Then $g(x) =_{Df} 1 + f(x,x)$ were also representable, by $E_k$ say, yielding $value(E_k\bar{k}) = g(k) = 1 + f(k,k) = 1 + value(E_k\bar{k})$, a contradiction. ⊣

## 2.4. Examples of function representations

1. *Successor.*

   A straightforward derivation of $N(x) \to N(\mathbf{s}x)$ shows that, for a unary relation $R$ containing $\mathbf{0}$ and closed under $\mathbf{s}$, $x \in R$ and therefore $\mathbf{s}x \in R$ (see Appendix 10.1). This yields the $\lambda$-expression $\lambda n^{\iota} \Lambda R \lambda s^{R \to R} z^{R}.\, s(nRsz)$, where $\iota =_{Df} \forall R.(R \to R) \to (R \to R)$. The untyped form is $\lambda n \lambda sz.\, s(nsz)$, which is Church's representation of the successor function in the untyped $\lambda$-calculus.

   An alternative derivation of $N(x) \to N(\mathbf{s}x)$ instantiates $N(x)$ to the predicate $\lambda u.R(\mathbf{s}\,u)$. This derivation is mapped under $\kappa$ to the $\lambda$-expression $\lambda n^{\iota} \Lambda R \lambda s^{R \to R} z^{R}.\, nRs(sz)$, from which the untyped form $\lambda n \lambda sz.\, ns(sz)$ falls out. The combinatory form of this representation of the successor function, $BW(BB)$, was discovered by Kearns [Kea70] (see [CHS72], p. 213, fn.6).

2. *Addition.*

   Given $\mathbf{a}(x,\mathbf{0}) = x$; $\mathbf{a}(x,\mathbf{s}\,y) = \mathbf{s}\,\mathbf{a}(x,y)$, the formula $N(\mathbf{a}(x,y))$ can be derived from $N(x)$ and $N(y)$ as follows. $N(y)$ instantiates to $\forall z(R(\mathbf{a}(x,z)) \to R(\mathbf{a}(x,\mathbf{s}z))) \to R(\mathbf{a}(x,\mathbf{0})) \to R(\mathbf{a}(x,y))$. The premise of this formula follows from the program, yielding by detachment $R(\mathbf{a}(x,\mathbf{0})) \to R(\mathbf{a}(x,y))$. From $N(x)$ and $\mathbf{a}(x,\mathbf{0}) = x$ obtain $R(\mathbf{a}(x,\mathbf{0}))$, hence $R(\mathbf{a}(x,y))$. This derivation is mapped under $\kappa$ to
   $\lambda n^{\iota} m^{\iota} \Lambda R \lambda s^{\iota \to \iota} z^{\iota}.\, mR(\lambda u^{R}.su)(nRsz)$, which $\eta$-reduces to $\lambda n^{\iota} m^{\iota} \Lambda R \lambda s^{\iota \to \iota} z^{\iota}.\, mRs(nRsz)$. The untyped form is Church's representation of addition, $\lambda nm \lambda sz.\, ms(nsz)$.

   An alternative derivation $\Pi$ of $N(\mathbf{a}(x,y))$ from $N(x)$ and $N(y)$ is detailed in Appendix 10.2. We get $\kappa\Pi \equiv \lambda n^{\iota} m^{\iota}.m\iota(\lambda u^{\iota} \Lambda R \lambda s^{R \to R} z^{R}.\, s(uRsz))n$. Here $\Pi$ instantiates a relational variable to a second order formula, so $\kappa\Pi$ contains the quantified type $\iota$ as the argument of a type application. The untyped form is $\lambda nm.m(\lambda usz.s(usz))n$, which too contains higher abstraction, in that the first argument of $m$ is an abstraction term.

3. *Multiplication.*

   Given a program for $\mathbf{m}$, denoting multiplication, a deduction $\Pi$ of $N(\mathbf{m}(x,y))$ from $N(x)$ and $N(y)$ is detailed in Appendix 10.3. We have $\kappa\Pi = \lambda n^{\iota} m^{\iota} \Lambda R \lambda s^{R \to R} z^{R}.\, nR(\lambda v^{R}.mR(\lambda u^{R}.su)v))z$, which $\eta$-reduces to $\lambda n^{\iota} m^{\iota} \Lambda R \lambda s^{R \to R} z^{R}.\, nR(mRs)z$. The untyped form is $\lambda nm \lambda sz.\, n(ms)z$, again the standard $\lambda$-representation of multiplication.

4. *Exponentiation and super-exponentiation.*

We leave the construction of these examples to the reader. The interesting point is that the representation obtained for exponentiation contains a type argument with $\rightarrow$, and that the representation obtained for super-exponentiation contains a type argument with $\forall$. These are essential: the former because exponentiation is not representable in the simply typed calculus (with fixed types for input and output) [Schw76, Sta79], and the latter because super-exponentiation is not representable in the simply typed calculus even allowing change of type from input to output [FLO83].

5. *Ackermann's Function.*

Let $k$ be defined by $\mathbf{k}(\mathbf{0}, x, y) = \mathbf{s}x$, $\mathbf{k}(\mathbf{s}q, \mathbf{0}, y) = y$, $\mathbf{k}(\mathbf{s}q, \mathbf{s}x, y) = \mathbf{k}(q, \mathbf{k}(\mathbf{s}q, x, y), y)$. From a straightforward derivation $\Pi$ of $N(\mathbf{k}(q, x, y)$ from $N(q)$, $N(x)$ and $N(y)$ we get

$$\kappa\Pi \equiv \lambda q^{\iota} n^{\iota} m^{\iota}.\, q\xi\, (\lambda u^{\xi} a^{\iota} b^{\iota}.\, a\iota(\lambda c^{\iota}.\, ubc)b)\, (\lambda d^{\iota} e^{\iota}.\, \Lambda R.\lambda s^{R\rightarrow R} z^{R}.\, s(dRsz))mn,$$

where $\xi =_{Df} \iota \rightarrow (\iota \rightarrow \iota)$. Note that $\kappa\Pi$ contains type arguments with negative type quantification. This is essential, for otherwise Ackermann's function would have been primitive recursive, by [Lei89]. $\kappa\Pi$ $\eta$-converts to

$$\lambda q^{\iota} n^{\iota} m^{\iota}.\, q\xi\, (\lambda u^{\xi} a^{\iota} b^{\iota}\, a\iota ub)b)(\lambda d^{\iota} e^{\iota}\, \Lambda R\lambda s^{R\rightarrow R} z^{R}.\, s(dRsz))\, mn$$

of which the untyped form is

$$\lambda qnm.\, q\, (\lambda uab.aubb)(\lambda de\lambda sz.s(dsz))mn.$$

6. *Predecessor,*

Given $\mathbf{p0} = \mathbf{0}$, $\mathbf{p}(\mathbf{s}x) = x$, a deduction $\Pi$ of $N(x) \rightarrow N(\mathbf{p}(x))$ is given in Appendix 10.4. We have, for $\kappa\Pi$ in $2\lambda+$ pairing,

$$\kappa\Pi \equiv \lambda n^{\iota} \Lambda R\lambda s^{R\rightarrow R} z^{R}.\, \mathbf{j}_1(n(R \times R)(\lambda u^{R\times R}.\langle \mathbf{j}_2 u, s(\mathbf{j}_2 u)\rangle \langle z, z\rangle)).$$

The untyped form is $\lambda n\Lambda R\lambda sz.\, \mathbf{j}_1(n(\lambda u.\langle \mathbf{j}_2 u, s(\mathbf{j}_2 u)\rangle \langle z, z\rangle)$, i.e. Kleene's representation of the predecessor.

## 2.5. Higher order representations

The Function Representation Theorem lifts, without change of proof, to higher order logic. For $2 \leq k \leq \omega$, let **MkL** be minimal $k$-th order logic, i.e. the generalization of **M2L** to $k$-th order relations and quantification over them, and let $\mathbf{k}\lambda$ be the $k$-th order fragment of Girard's system $F_{\omega}$, restricted to $\rightarrow$ and $\forall$. The homomorphism $\kappa$ is extended to $\kappa$ : **MkL** $\rightarrow$ $\mathbf{k}\lambda$ by setting $\underline{\forall X.\varphi} =_{Df} \forall X.\underline{\varphi}$, for $X$ of any order $\geq 2$, and defining $\kappa$ for a derivation $\Pi$ ending with a $\forall$ of order $> 2$ similarly to the definition for order 2.

**THEOREM IV'** *Let $\mathcal{P}$ be a program, with target* **f**, *that computes the function $f$. Let $k \geq 2$. If $\Pi$ is a derivation in* **MkL** $+ \mathcal{P}$ *of* $N^1(\mathbf{f})$, *then $\kappa\Pi$ represents $f$ in* $\mathbf{k}\lambda$.

## 3. Programs over data systems

We generalize the representation of natural numbers and numeric functions to representation of objects and functions in systems of inductively generated data types.

### 3.1. Inductive data types

The (pure applicative) *types* are generated by the clauses: $o$ is a type; if $\sigma$ and $\tau$ are types then so is $\sigma \to \tau$. The *orders* of types are defined by $order(o) =_{Df} 0$; $order(\tau \to \sigma) =_{Df} \max(1+order(\tau), order(\sigma))$.

Let $L$ be a *functional vocabulary*, i.e. a set of identifiers, the *primitives*, each associated a type. A *computation space* $B = B(L)$ is the set of *canonical expressions*, i.e. the closed terms of type $o$ in the initial algebra over $L$. $B$ is *non-trivial* if $L$ has at least one identifier of type $o$ and one identifier of order 1. For $L = \{\mathbf{0}, \mathbf{s}\}$, $B(L)$ is the set of numerals, and is the simplest non-trivial computation space. $L$ is of *order* $k$ if the types of its identifiers are all of order $\leq k$.

Suppose $L = \{\mathbf{c}_1, \ldots, \mathbf{c}_k\}$ is of order 1. In a structure whose vocabulary contains $L$, the denotation of canonical expressions is defined by the formula

$$\begin{aligned} D_L(x) &\equiv \forall R(C_1 \to C_2 \to \cdots \to C_k \to Rx), \text{ where} \\ C_i &\equiv \forall z_1 \ldots z_r.\, R(z_1) \to \cdots \to R(z_r) \to R(\mathbf{c}_i(z_1)\cdots(z_r)), \\ &\qquad (\text{and } type(\mathbf{c}_i) = o^r \to o \equiv o \to \cdots \to o \to o) \end{aligned}$$

### 3.2. Canonical representation in the $\lambda$-calculus

**THEOREM VII (Object representation)** *Let $L$ be of order 1, $t \in B(L)$. There is a unique normal* **M2L** *deduction $\Pi$ of the formula $D_L(t)$. Thus, $\kappa\Pi$ can be used as a canonical representation $rep(t)$ of $t$ in $2\lambda$.*

*Moreover, if $t$ is a closed term over the primitives of $L$ and the identifiers of program $\mathcal{P}$, and $\Pi$ is a $\beta$-normal derivation in* **M2L** $+ \mathcal{P}$ *of $D_L(t)$, then $\mathcal{P}$ derives $t = x$ for some $x \in B(L)$, and $\kappa\Pi = rep(x)$.*

The representation in $2\lambda$ obtained here for canonical expressions is identical to the representation defined, more directly, in [BB85].

**Examples.**

1. *Booleans:* $L = \{\top : o, \bot : o\}$. $Bool(x) \equiv D_L(x) \equiv \forall R.R\bot \to R\top \to Rx$.
   From the proofs of $Bool(\top)$ and $Bool(\bot)$ we obtain $rep(\top) = \Lambda R\lambda tf.\, t$, and $rep(\bot) = \Lambda R\lambda tf.\, f$. The untyped forms are Church's representations of the booleans [Bar81]. A similar representation is obtained for any $L$ of order 0.

2. *Words over a finite alphabet* $\Sigma$ can be viewed as "generalized numerals" obtained using the elements of $\Sigma$ as distinct successor functions. Suppose $\Sigma = \{\mathbf{0}, \mathbf{1}\}$.
Let $L = \{\epsilon : o, \mathbf{0} : o \to o, \mathbf{1} : o \to o\}$.

$$W(x) \equiv D_L(x) \equiv \forall R.\, \forall z.(R(z) \to R(\mathbf{0}z)) \to \forall z.(R(z) \to R(\mathbf{1}z)) \to R(\epsilon) \to R(x).$$

If $t \equiv \mathbf{s}_1 \cdots \mathbf{s}_n\epsilon$ ($\mathbf{s}_i \in \{\mathbf{0}, \mathbf{1}\}$), and $\Pi$ is the normal proof of $W(t)$, then
$rep(t) = \kappa\Pi = \Lambda R\, \lambda s_0^{R\to R}\, s_1^{R\to R}\, z^R.\, s_{\mathbf{s}_1} \cdots s_{\mathbf{s}_n} z.$

3. *Binary trees:* $L = \{\mathbf{0} : o,\ \mathbf{p} : (o \to o \to o)\}$. We have

$$BT(x) \equiv D_L(x) \equiv \forall R.\, \forall uv.(R(u) \to R(v) \to R(\mathbf{p}uv)) \to R(\epsilon) \to R(x).$$

The $\lambda$-representation for the canonical expressions, obtained via $\kappa$, is
$rep(t) = \Lambda R \lambda p^{R\to R\to R} e^R.\, \tilde{t}$, where $\tilde{t}$ is $t$ with $\epsilon$ replaced by $e$ and $\mathbf{p}$ by $p$.

4. *Data binary trees.* One $\lambda$-representation of binary trees over an alphabet $\Sigma$ is obtained by taking $L = \{\epsilon : o, \mathbf{a} : o \to o \to o\ (\mathbf{a} \in \Sigma)\}$. For instance, a three node tree with $\mathbf{a}$ at the root, and $\mathbf{b}$ and $\mathbf{c}$ at the leaves, is represented by $\mathbf{a}(\mathbf{b}\epsilon\epsilon)(\mathbf{c}\epsilon\epsilon)$, which is mapped to the $\lambda$-expression $\Lambda R \lambda a^{R\to R\to R} b^{R\to R\to R} c^{R\to R\to R} \ldots e^R.\, a(bee)(cee)$.

### 3.3. Representation of data types of higher order

If $L$ is of order $> 1$, then terms are generated possibly via function-denoting terms. The explicit definition of $D_L$ is then expressed using quantifiers over functions.

Let **M2L**$^f$ be an extension of **M2L** with quantifiers over functions of all types, but with no new existence axioms (such as comprehension or definition by recursion). The only inference rules for quantifiers over functions are the trivial ones:

$$\forall^\tau\mathbf{I}: \quad \frac{\varphi}{\forall x^\tau.\varphi} \quad (x \text{ not free in assumptions}) \qquad\qquad \forall^\tau\mathbf{E}: \quad \frac{\forall x^\tau.\, \varphi}{\varphi[t/x]} \quad (t : \tau)$$

Clearly, **M2L**$^f$ is a conservative extension of **M2L**.

For a type $\tau$, a relational variable $R$, and a term $t : \tau$, define the formula $R^{[\tau]}(t)$ by

$$\begin{aligned} R^{[o]}(t) &\equiv R(t) \\ R^{[\rho\to\sigma]}(t) &\equiv \forall u^\rho.\, R^{[\rho]}(u) \to R^{[\sigma]}(tu). \end{aligned}$$

For example,

$$R^{[(o\to o)\to o]}(a) \equiv \forall u^{o\to o}.\, (\forall v^o\, (R(v) \to R(uv)) \to R(au)).$$

Given $L = \{\mathbf{c}_1 : \tau_1, \ldots, \mathbf{c}_k : \tau_k\}$, we can now define

$$D_L(x) \equiv \forall R.R^{[\tau_1]}(\mathbf{c}_1) \to \cdots \to R^{[\tau_k]}(\mathbf{c}_k) \to R(x).$$

**THEOREM VIII (Higher type object representation)** *Let $t \in B(L)$. There is a unique normal* M2L$'$ *deduction $\Pi_t$ of $D_L(t)$. Thus, $\kappa\Pi_t$ is a unique representation of $t$ in $2\lambda$.* $\dashv$

**Example.** *Words over an infinite alphabet* may be generated from an infinite collection of unary functions. Such collection is obtained from a primitive $\mathbf{g}$ of type $(o \to o) \to (o \to o)$: with $L = \{\mathbf{g} : (o \to o) \to (o \to o), \mathbf{s} : o \to o, \epsilon : o\}$, the expressions $\mathbf{g}^{[i]}\mathbf{s}$, of type $o \to o$, can be used as distinct successor functions. We have

$$\begin{aligned} D_L(x) \equiv \forall R.(\forall z^{o\to o}\,(\forall u^0.\,(R(u) \to R(zu)) \to \quad \forall u^0.\,(R(u) \to R(\mathbf{g}zu))) \\ \to \forall u^o\,(R(u) \to R(\mathbf{s}u)) \\ \to R(\epsilon) \\ \to R(x)). \end{aligned}$$

Every $t \in B(L)$ is of the form $\mathbf{s}_{n1} \cdots \mathbf{s}_{nr}\epsilon$, where $\mathbf{s}_i \equiv \mathbf{g}^{[i]}\mathbf{s}$. If $\Pi$ is the normal proof of $D_L(t)$, then

$$rep(t) = \kappa\Pi = \Lambda R\,\lambda g^{(R\to R)\to R\to R}\lambda s^{R\to R}\lambda z^{R}.\ (g^{[n1]}s)\cdots(g^{[nr]}s)z.$$

The representation of trees over finite alphabet can be adapted to infinite alphabets in a similar fashion.

### 3.4. Data systems

A *data system* is the setting in which inductively generated data types are used most often in programming, i.e. a finite collection $\Delta$ of data types $D_1 \ldots D_k$, defined by (possibly) simultaneous induction, that is, by a finite set of clauses of the form

$$D_1(\hat{z}_1) \to \ldots \to D_k(\hat{z}_k) \to D_i(t),$$

where each $\hat{z}_j$ is a tuple of variables, and $t$ is a term over $L \cup \hat{z}_1 \cup \ldots \cup \hat{z}_k$. The data types may include, for instance, natural numbers, booleans, even numbers, lists of numbers, alphanumeric words, finite data types, and grammatical notions. A data system $\Delta$ is *well-parsed* if, for each $D \in \Delta$ and each $t \in B(L)$, $D(t)$ can be derived by at most one sequence of closure conditions. For example, { $Even(\mathbf{0})$; $Even(x) \to Odd(\mathbf{s}x)$; $Odd(x) \to Even(\mathbf{s}x)$ } is well-parsed, but adding the clause $Even(x) \to Even(\mathbf{ss}x)$ would yield a non-well-parsed system.

In a well-parsed data system, if $D_i(t)$ is true then it has a unique derivation from the closure conditions. A data type $D_i$ of $\Delta$ can be defined explicitly by

$$\begin{aligned} D_i(x) \equiv \forall R_1 \ldots R_k. \\ \text{conjunction of all defining clauses (universally quantified)} \\ \text{for } \Delta \text{ (with } R_i \text{ replacing } D_i) \\ \to R_i(x). \end{aligned}$$

The uniqueness of the direct derivation of $D_i(t)$ implies:

**Proposition 3..1** *Suppose $\Delta = \{D_i\}_i$ is a well-parsed data system. For each i and $t \in D_i$ there is a unique normal deduction of* **M2L** *of $D_i(t)$.*

As for data types, this proposition yields canonical representations in $2\lambda$ for canonical expressions. These representations are dependent on the defining clauses, and are relative to each particular data type. For example, if $\Delta$ is $\{N(\mathbf{0});\ N(x) \to N(\mathbf{s}x));\ Even(\mathbf{0});\ Even(x) \to Even(\mathbf{ss}x)\}$, then **ss0** is represented by $\Lambda RQ\lambda s^{R\to R} z^R t^{Q\to Q} w^Q.\ ssz$ as element of $N$, and by $\Lambda RQ\lambda s^{R\to R} z^R t^{Q\to Q} w^Q.\ tw$ as an element of *Even*.

### 3.5. Representing type inheritance

In general, type containment is not a decidable property (see Appendix III). However, when an inclusion $D_0 \subseteq D_1$ is provable by a proof $\Pi$ of **M2L**, then $\kappa\Pi$ is a $\lambda$-expression $C_{D_0 \to D_1}$ that converts the representation of $t \in D_0$ with respect to $D_0$ to the representation of $t$ with respect to $D_1$: if $F$ represents $t$ with respect to $D_0$ then $C_{D_0 \to D_1} F$ represents $t$ with respect to $D_1$. In programming parlance, the expression $C_{D_0 \to D_1}$ represents a *coercion* of one data type in another.

### 3.6. Function Representation

**THEOREM IX (Function Representation)** *Let $\mathcal{P}$ be a program over a data system $\Delta$, with target* **f**, *that computes $f : o^r \to o$. If $\Pi$ is an* **M2L** $+ \mathcal{P}$ *derivation of*

$$\forall x_1 \ldots x_r.\ (\, D_1(x_1) \to \cdots D_r(x_r) \ \to\ D_0(\mathbf{f}(\vec{x}))\,),$$

*then $\kappa\Pi$ represents in $2\lambda$ the restriction of $f$ to $D_1 \times \cdots \times D_r$: if $E_i : \underline{D_i}$ $\lambda$-represents $a_i \in D_i$, modulo the canonical $\lambda$-representation of $D_i$ $(i = 1, \ldots, r)$, then $(\kappa\Pi)E_1 \cdots E_r$ $\lambda$-represents $fa_1 \ldots a_r \in D_0$, modulo the canonical $\lambda$-representation of $D_0$.*

**Proof.** Similar to Theorem IV. ⊣

**Examples**

1. *Negation.* Let $\mathcal{P}$ be the program $\mathbf{neg}(\top) = \bot; \mathbf{neg}(\bot) = \top$. Then a normal $\Pi$ deriving $Bool(x) \to Bool(\mathbf{neg}(x))$ uses instantiation of the relational quantifier in $Bool(x)$ to $\lambda u.R(\mathbf{neg}(u))$. We obtain $rep(\mathbf{neg}) = \kappa\Pi = \lambda x^\beta \Lambda R \lambda t^R f^R.xRft$, where $\beta = \forall R.R \to R \to R$.

2. *Conjunction.* Let $\mathcal{P}$ be the program $\mathbf{conj}(\top)(x) = x$; $\mathbf{conj}(\bot)(x) = \bot$. A straightforward normal deduction $\Pi$ of $Bool(x) \to Bool(y) \to Bool(\mathbf{conj}(x)(y))$ yields $\kappa\Pi = \lambda x^\beta y^\beta \Lambda R \lambda t^R f^R.\ xR(yRtf)f$. Note that this cannot be obtained if the program for **conj** consists of the four equations $\mathbf{conj}(\top)(\top) = \top$; $\mathbf{conj}(\top)(\bot) = \bot$; $\mathbf{conj}(\bot)(\top) = \bot$; $\mathbf{conj}(\bot)(\bot) = \bot$. The straightforward normal deduction $\Pi$ of $Bool(x) \to Bool(y) \to Bool(\mathbf{conj}(x)(y))$ using the latter program yields $\kappa\Pi = \lambda x^\beta y^\beta \Lambda R \lambda t^R f^R.\ xR(yRtf)(yRff)$.

3. *Concatenation.* Consider the representation in §3.2 of words over the alphabet $\{\mathbf{0}, \mathbf{1}\}$. Define the concatenation function by $\mathbf{cat}(\epsilon, y) = y$, $\mathbf{cat}(\mathbf{0}x, y) = \mathbf{0}(\mathbf{cat}(x, y))$, $\mathbf{cat}(\mathbf{1}x, y) = \mathbf{1}(\mathbf{cat}(x, y))$. If $\Pi$ is the straightforward proof of $W(x) \to W(y) \to W(\mathbf{cat}(x, y))$, then $\kappa\Pi$ gives the representation $\lambda x^{\tau} y^{\tau} \Lambda R \lambda u^{R \to R} v^{R \to R} e^{R}.\, xRuv(yRuve)$, where $\tau =_{Df} (R \to R) \to (R \to R) \to R \to R$. This expression is the standard $\lambda$-representation of the pairing function. Note the similarity to the representation of addition for the numerals.

### 3.7. Generic types and functions

A data type can be parameterized by an unspecified base type, as in *pairs of Q's*, which is defined by

$$Pair_Q(x) \equiv \forall R. \forall u, v. (Q(u) \to Q(v) \to R(\mathbf{p}uv)) \to R(x).$$

There can be no closed proof of a formula $P_Q(t)$, but there is a trivial derivation $\Pi$ of $P_Q(\mathbf{p}t_1t_2)$ from the open assumptions $Q(t_1)$ and $Q(t_2)$. We have $\kappa\Pi = \Lambda R. \lambda c^{Q \to Q \to R} c x_1^Q x_2^Q$, where $x_1$ and $x_2$ are free variables, corresponding to the assumptions $Q(t_1)$ and $Q(t_2)$. Note that neither $\Pi$ nor $\kappa\Pi$ can be closed with respect to $Q$, because they have free variables of type $Q$.

Similarly, the type of *lists of Q's* is defined by

$$List_Q(x) \equiv \forall R. \forall u, v. (Q(u) \to R(v) \to R(\mathbf{p}uv)) \to R(\epsilon) \to R(x).$$

Let $\Pi$ be the straightforward derivation of $List_Q(\langle t_1 \dots t_n \rangle)$ from the open assumptions $Q(t_1), \dots, Q(t_n)$ (where $\langle t_1 \dots t_n \rangle \equiv \mathbf{p}t_1(\mathbf{p} \cdots (\mathbf{p}t_n\epsilon) \cdots)$). Then $\kappa\Pi = \Lambda R. \lambda c^{Q \to R \to R} e^R. c x_1^Q (c x_2^Q (\cdots c x_n^Q e)$, where $x_1 \dots x_2$ are free variables.

**THEOREM IX'** *Let $\mathcal{P}$ be a program over a data system $\Delta$, with target* $\mathbf{f}$, *that computes $f : o' \to o$. Suppose $\Pi$ is an* **M2L** $+ \mathcal{P}$ *derivation of*

$$\forall x_1 \dots x_r.\, D_1(x_1) \to \cdots D_r(x_r) \to D_0(\mathbf{f}(\vec{x})),$$

*where some of the $D_i$'s above are parameterized by relational variables, say by $Q$. Then $\kappa\Pi$ is an expression of $2\lambda$, with $Q$ a free type variable, such that, for any data type $D$, if $E_i : \underline{D_i[D/Q]}$ $\lambda$-represents $a_i \in D_i[D/Q]$, modulo the canonical $\lambda$-representation of $D_i[D/Q]$ $(i = 1, \dots, r)$, then $(\kappa\Pi)[\underline{D}/Q]E_1 \cdots E_r$ $\lambda$-represents $fa_1 \dots a_r \in D_0[D/Q]$, modulo the canonical $\lambda$-representation of $D_0[D/Q]$.*

**Example.** For $List_Q$ define the function **cat** by $\mathbf{cat}(\epsilon, z) = z$, $\mathbf{cat}(\mathbf{p}(x, y), z) = \mathbf{p}(x, \mathbf{cat}(y, z))$. If $\Pi$ is the straightforward proof of $List_Q(x) \to List_Q(y) \to List_Q(\mathbf{cat}(x, y))$, then $\kappa\Pi$ gives the representation $\lambda x^{\tau} y^{\tau} \Lambda R \lambda u^{Q \to R \to R} e^R.\, xR(\lambda v^Q w^R. uvw)(yRue)$ for the generic concatenation function, where $\tau =_{Df} \forall R. (Q \to R \to R) \to R \to R$.

# 4. Representation of recursive order 2 functionals

## 4.1. Recursive functionals

We refer to Kleene's recursive functionals [Kle59]. Kechris and Moschovakis [KM77] gave an equivalent definition, less dependent on coding. Their approach was developed by Kolaitis [Kol85], who eliminated coding altogether, showing that Kleene's notion is equivalent to a purely applicative notion of recursive functionals. This is summarized in Appendix I below.

We consider functionals of types whose order is $\leq 2$, and focus on the simplest type of order 2, $(o \to o) \to o$. Let $N^2(F) \equiv_{\mathrm{Df}} \forall g.\, N^1(g) \to N(Fg)$.

**THEOREM X** *Let $\mathcal{P}$ be an order 2 program, with target* **F**. *The following are equivalent:*

1. $\mathcal{P}$ is total for total arguments: if $g$ is a total numeric function, then $\mathcal{P}$ converges on input $g$.
2. $\mathcal{P} \models N^2(\mathbf{F})$ with respect to standard second order quantification.

**Proof.** The proof is similar to the proof of Theorem I. ⊣

## 4.2. Provably recursive functionals

For a formalism **S**, let $\mathbf{S}^f$ be **S** augmented with the trivial rules of §3.3 for function quantification. Let $T^f$ be Kleene's $T$ predicate, modified to code computation with partial functions as input: $T^f(e, g, y)$ asserts that $y$ codes a completed computation of the program coded by $e$, where the input function variable is evaluated as $g$. $T^f$ is primitive recursive in its arguments.

**THEOREM XI** *Let $\mathcal{P}$ be a standardized order 2 program, with target* **F**, *that computes the numeric functional $F$, say of type $(o \to o) \to o$. The following are equivalent.*

1. *The program $\mathcal{P}$ is provably total in* $\mathbf{C2A}^f$ *:*
$$\vdash_{\mathbf{C2A}^f} \forall g \exists y\, T^f(\bar{e}_{\mathcal{P}}, g, y).$$
2. *$\mathcal{P}$ is provably total in* $\mathbf{I2A}^f$ *.*
3. *$\mathcal{P}$ is provably total in* $\mathbf{I2L}^f$ *:* $\mathcal{P} \vdash_{\mathbf{I2L}^f} N^2(\mathbf{F})$.
4. *$\mathcal{P}$ is provably total in* $\mathbf{M2L}^f$ *.*

5. *$\mathcal{P}$ is provably total in* $\mathbf{C2L}^f$.

**Proof.** Similar to the proof of Theorem II. ⊣

### 4.3. Functional representation theorem

We refer to a strong notion of functional representability in $\lambda$ calculi. To keep notation uncluttered, we give the definition for numeric functionals of type $(o \to o) \to o$.

Let $g$ be a partial function from numbers to numbers, $x$ a fixed $\lambda$-variable of type $\iota \to \iota$. Define a notion of $\lambda$-computability with oracle $g$, as follows. For $\mathbf{2}\lambda$ expressions $E, E'$, we write $E \succ_{[g/x]} E'$ if, for some $n$ for which $g$ is defined, $E'$ comes from replacing in $E$ a subexpression of the form $x\bar{n}$ by $\overline{gn}$ ($\bar{m}$ is the $m$'th Fortune-O'Donnell numeral). The relation $=_{\beta,[g/x]}$ is the joint reflexive, symmetric and transitive closure of $\beta$-reductions and $\succ_{[g/x]}$.

An expression $\lambda x^{\iota\to\iota}.E^{\iota}$ *represents* the numeric functional $F : (o \to o) \to o$, if for every partial function $g$ from numbers to numbers, $F(g) = z$ iff $E =_{\beta,[g/x]} \bar{z}$. In particular, if $G : \iota \to \iota$ $\lambda$-represents the function $g$, then $(\lambda x.E)G =_\beta \overline{F(g)}$.

**THEOREM XII (Numeric Functional Representation)** *Let $\mathcal{P}$ be an order 2 program, with target* **F**, *that computes the functional $F$ of type $(o \to o) \to o$. If $\Pi$ is an* $\mathbf{M2L} + \mathcal{P}$ *derivation of $N^2(\mathbf{F})$, then $\kappa\Pi$ represents $F$ in $\mathbf{2}\lambda$.*

**Proof.** The proof is similar to the proof of Theorem IV, using Theorem XI. ⊣

Combining this with Theorem XI we have

**THEOREM XIII** *All the provably total computable functionals of classical second order arithmetic are representable in $\mathbf{2}\lambda$.*

As for functions of order 1, Theorem XII can be used to derive $\lambda$ representations of various functionals. Also, the discussion of generic functions can be generalized to generic functionals of any finite order. Generic functionals of order 2 play an important role in functional programming; an example is the functional **map** satisfying $\mathbf{map}(f, \langle x_1 \ldots x_r \rangle) = \langle fx_1 \ldots fx_r \rangle$. I.e., **map** is defined by the program $\mathbf{map}(f, \epsilon) = \epsilon$; $\mathbf{map}(f, \mathbf{cons}(x, y)) = \mathbf{cons}(fx, \mathbf{map}(f, y))$.

## 5. Contraction to programs for generative axiomatizations

In this section we apply our main method to generative axiomatizations of data types, such as Peano's formalism for the natural numbers. Here the co-domain of the proof-to-program homomorphism is no longer a pure $\lambda$-calculus. However, proofs and programs are far shorter and more readable than the analogous proofs and programs in pure second order logic and pure $\lambda$-calculi.

### 5.1. Generative axiomatizations of the natural numbers

The generative style is the simplest and most easy-to-use formalization style for reasoning about inductively generated data types. It is illustrated by Peano's original axiomatization of Arithmetic, using a primitive constant $N$. This consists of two main groups.

1. The *generative axioms*, $N(0)$ and $\forall x.N(x) \rightarrow N(\mathrm{s}x)$;
2. The principle of Induction: $\forall x.N(x) \rightarrow \forall R.Closed[R] \rightarrow R(x)$, where $Closed[R] =_{Df} R(\mathbf{0}) \wedge \forall x.R(x) \rightarrow R(\mathrm{s}x)$. In the absence of set quantification, the induction axiom is replaced by the induction schema (for $\mathcal{C}$), $\forall x.N(x) \rightarrow Closed[\lambda x.\varphi] \rightarrow \varphi[x]$, where $\varphi$ ranges over a class $\mathcal{C}$ of formulas.

The two closure conditions for $N$ guarantee that $N$ contains the denotation of all numerals. Induction forces the extension of $N$ to be the minimal set closed under these conditions, at least with respect to definable sets. (Peano's third and fourth axioms enforce inequality between all numerals; we return to this in the next subsection.)

The second order generative axiomatization of $N$, **M2LP** (**P** for Peano), differs from (Minimal) Second Order Arithmetic in using $N$ explicitly (variables are intended to range over possibly non-numeric objects), and in not having Peano's third and fourth axioms. A first order variant **M1LP** of **M2LP** is obtained by replacing the Induction Axiom by the induction schema for all first order formulas in the language. **C2LP** and **C1LP** are the classical variants of **M2LP** and **M1LP**.

**THEOREM II'** *Let $\mathcal{P}$ be a standardized program, with target* **f**, *that computes a numeric function $f$. The conditions of Theorem II are also equivalent to:*

6. $N^1(\mathbf{f})$ is provable in **M2LP** + $\mathcal{P}$.
7. $N^1(\mathbf{f})$ is provable in **C2LP** + $\mathcal{P}$.

**THEOREM XIV** *Let $\mathcal{P}$ and* **f** *be as above. The following conditions are equivalent:*

(i) *$\mathcal{P}$ is provably total in Peano Arithmetic based on Minimal Logic.*

(ii) $\mathcal{P}$ *is provably total in (classical) Peano Arithmetic.*

(iii) $\mathcal{P}$ *is provably total in the classical variant* **C1LP** *of* **M1LP**, *i.e.*, $\mathcal{P} \vdash_{\mathbf{C2LP}} N^1(\mathbf{f})$.

(iv) $\mathcal{P}$ *is provably total in* **M1LP**.

**Proof.** Analogous to the proof of Theorem II. ⊣

### 5.2. Generative axiomatizations in general

The paradigm of Peano's axiomatization applies to any data system $\Delta$. Relational identifiers are used to denote the data types, and two groups of axioms define them implicitly:

1. The *generative axioms*, consisting of the data type's closure conditions formulated for the data identifiers;
2. For each data type, an induction principle. For instance, if the data types are $D_1$ and $D_2$, both unary say, with closure condition $Closed[D_1, D_2]$, then the induction axiom for $D_1$ is $\forall R_1, R_2.Closed[R_1, R_2] \rightarrow \forall x.D_1[x] \rightarrow R_1(x)$.

We write **M2L***D* and **M1L***D* for the extensions of **M2L** and **M1L** as above (the induction axioms being formulated as schemas in **M1L***D*).

One may add a third group of *separation axioms*, modeled after Peano's Third and Fourth Axioms, and implying that all canonical expressions have distinct values (or they all have identical value, if the Fourth Axiom is formulated as $\forall x.(\mathbf{s}x = \mathbf{0} \rightarrow \mathbf{s0} = \mathbf{0})$, or as $\forall x.(\mathbf{s}x = \mathbf{0} \rightarrow \forall x.x = \mathbf{0})$). In §2.2 we observed that these axioms may be replaced by the definition of the predecessor function. For arbitrary $B = B(L)$, the third group can be dispensed with in the presence of equations for *destructor functions* for the function primitives: for each $\mathbf{c} \in L$ define functions $\check{\mathbf{c}}_i$ ($i = 1 \ldots arity(\mathbf{c})$) by $\check{\mathbf{c}}_i(\mathbf{c}(x_1, \ldots, x_r)) = x_i$, $\check{\mathbf{c}}_i(\mathbf{c}'(\hat{z})) = \mathbf{c}'(\hat{z})$ ($\mathbf{c}' \not\equiv \mathbf{c}$, $arity(\hat{z}) = arity(\mathbf{c}')$). This suffices if there is at most one $\mathbf{c} \in L$ of arity 0. Otherwise, we add defining equations also for a *discriminator function* $\check{\mathbf{c}}$, for each constant primitive $\mathbf{c} \in L$: $\check{\mathbf{c}}(\mathbf{c}, x) = \mathbf{c}$, $\check{\mathbf{c}}(\mathbf{c}'(\hat{z}), x)) = x$ for $\mathbf{c}' \in L$, $\mathbf{c}' \not\equiv \mathbf{c}$.

Generative axiomatizations are of interest even for trivial data types, such as the booleans: for $L = \{\bot, \top\}$ ($arity(\bot) = arity(\top) = 0$), the generative axioms are $B(\bot)$ and $B(\top)$, and the induction principle is $\forall R.R(\bot) \rightarrow R(\top) \rightarrow \forall x.B(x) \rightarrow R(x)$.

### 5.3. Reductions and normalization for generative axiomatizations

Prawitz [Pra65,Pra71] defined induction-reductions on natural deductions for first order arithmetic, which reduce the complexity of the eigen-term of induction, if that term is **0** or a successor

term. Using the abbreviation $CS[\varphi] \equiv_{Df} \forall z.\varphi[z] \rightarrow \varphi[sz]$, the reductions are:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{N(0) \rightarrow CS[\varphi] \rightarrow \varphi[0] \rightarrow \varphi[0]}\ \textit{Induction} \qquad N(0)}{CS[\varphi] \rightarrow \varphi[0] \rightarrow \varphi[0]} \qquad \Theta_1}{\varphi[0] \rightarrow \varphi[0]} \qquad CS[\varphi]}{\varphi[0]} \qquad \Theta_2 \quad \varphi[0]$$

reduces to

$$\dfrac{\Theta_2}{\varphi[0]}$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{N(st) \rightarrow CS[\varphi] \rightarrow \varphi[0] \rightarrow \varphi[st]}\ \textit{Induction}}{CS[\varphi] \rightarrow \varphi[0] \rightarrow \varphi[st]} \qquad \Theta_1}{\varphi[0] \rightarrow \varphi[st]} \qquad CS[\varphi]}{\varphi[st]} \qquad \Theta_2 \quad \varphi[0]$$

$$\dfrac{\dfrac{\forall z.\, N(z) \rightarrow N(sz)}{N(t) \rightarrow N(st)} \qquad \Theta_3}{N(st)} \qquad N(t)$$

reduces to

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{N(t) \rightarrow CS[\varphi] \rightarrow \varphi[0] \rightarrow \varphi[t]}\ \textit{Induction} \qquad \Theta_3}{CS[\varphi] \rightarrow \varphi[0] \rightarrow \varphi[t]} \qquad N(t)}{\varphi[0] \rightarrow \varphi[t]} \qquad \Theta_1 \quad CS[\varphi]}{\varphi[t]} \qquad \Theta_2 \quad \varphi[0]$$

$$\dfrac{\Theta_1}{\varphi[t] \rightarrow \varphi[st]}$$

$$\varphi[st]$$

The definition of similar reductions for arbitrary data types is straightforward. Also, the proofs in [Pra65,Pra71] readily generalize to establish:

**THEOREM XV** *For each data type D, the calculi* **M2L***D* **M1L***D have the strong normalization property (with respect to the reductions of* **M2L** *plus the ones stipulated above for these calculi).*
⊣

### 5.4. The deduction-as-program homomorphism for generative proofs

We wish to extend the homomorphism $\kappa$ of §1 to a mapping from derivations of **M2LP** to expressions of an extended $\lambda$-calculus. Let $\mathbf{2\lambda P}$ be $2\lambda$ modified as follows.

1. The *type structure* is augmented with a type constant $N$.
2. The formation rules for *expressions* are supplemented with three constant expressions: $\mathbf{0}$, of type $N$; $\mathbf{s}$, of type $N \to N$; and $\mathbf{R}$, of type $N \to \forall R.(R \to R) \to R \to R$. (The latter is a polymorphic recursor operator).
3. The *reduction operations* are augmented with the reductions

$$\begin{array}{rll} \mathbf{R}0\tau E^{\tau\to\tau}F^{\tau} & \text{reduces to} & F \\ \mathbf{R}(\mathbf{s}G^{N})\tau E^{\tau\to\tau}F^{\tau} & \text{reduces to} & E(\mathbf{R}G\tau EF) \end{array}$$

The mapping $\varphi \mapsto \underline{\varphi}$ is extended by the clause $\underline{N(t)} =_{Df} N$. $\kappa$ is extended to $\kappa$ : **M2LP** $\to$ $\mathbf{2\lambda P}$, as follows.

- If $\Pi$ is the derivation consisting of the single axiom $N(\mathbf{0})$, then $\kappa\Pi =_{Df} \mathbf{0}$;
- If $\Pi$ consists of the single axiom $\forall x.N(x) \to N(\mathbf{s}x)$, then $\kappa\Pi =_{Df} \mathbf{s}$;
- If $\Pi$ consists of the Induction Axiom,

$$\forall x.\, N(x) \to \forall R\,(\forall u.\,(R(u) \to R(\mathbf{s}u)) \to R(\mathbf{0}) \to R(x)),$$

then $\kappa\Pi =_{Df} \mathbf{R}$.

One easily verifies that this extended mapping $\kappa$ maps induction reductions in **M2LP** to recursion reductions in $\mathbf{2\lambda P}$.

$\mathbf{1\lambda P}$ is defined like $\mathbf{2\lambda P}$, except that, in place of $\mathbf{R}$, there is for each type $\tau$ a constant $\mathbf{R}_{\tau}$, of type $N \to (\tau \to \tau) \to \tau \to \tau$, and the *reductions* are augmented with

$$\begin{array}{rll} \mathbf{R}_{\tau}0E^{\tau\to\tau}F^{\tau} & \text{reduces to} & F \\ \mathbf{R}_{\tau}(\mathbf{s}G^{N})E^{\tau\to\tau}F^{\tau} & \text{reduces to} & E(\mathbf{R}_{\tau}GEF) \end{array}$$

A mapping $\kappa$ : **M1LP** $\longrightarrow$ **1$\lambda$P** is now defined as for the second order formalisms **M2LP** and **2$\lambda$P** above, except that an instance of induction, with eigen-formula $\varphi$, is mapped to the recursor operator $\mathbf{R}_{\underline{\varphi}}$. The Function Representation Theorem between these formalisms is derived as for the pairs [**M2L / 2$\lambda$**] and [**M2LP / 2$\lambda$P**].

More generally, let **2$\lambda D$** be **2$\lambda$** modified as follows.

1. The *type structure* is extended with a type constant $D$.
2. The formation rules for *expressions* are supplemented with constant expressions $\mathbf{c}_i$ of type $D \to D \to \cdots \to D$ ($r_i + 1$ occurrences), for $i = 1 \ldots k$, and a constant $\mathbf{R}$, of type $D \to \forall R.R_1 \to \cdots R_k \to R$, where $R_i \equiv_{\mathrm{Df}} R \to R \to \cdots \to R$ ($r_i + 1$ occurrences).
3. The *reduction operations* are supplemented with $k$ reduction rules, of which the $i$'th recurse over expressions $\mathbf{c}_i E_1 \ldots E_{r_i}$.

**1$\lambda D$** is defined analogously to **1$\lambda$P**. The homomorphism $\kappa$ is extended as before, to $\kappa$ : **M2L$D$** $\to$ **2$\lambda D$** and $\kappa$ : **M1L$D$** $\to$ **1$\lambda D$**.

### 5.5. Generative proofs as programs

The object representation obtained from the extended mappings $\kappa$ is trivial: for a numeral $\bar{k} \equiv \mathbf{s}^{[k]}\mathbf{0}$, the unique normal proof of the formula $N(\bar{k})$ is obtained by $k$ applications of Peano's Second Axiom, $\forall z.N(z) \to N(\mathrm{sz})$, suitably instantiated, to the axiom $N(\mathbf{0})$. $\kappa$ maps this derivation simply to the expression $\mathbf{s}^{[k]}\mathbf{0}$ itself! Similarly, the representation of a canonical expression $t$ of any functional vocabulary is $t$ itself.

The Function Representation Theorem holds as for **M2L** and **2$\lambda$**, and it leads to representations in **2$\lambda$P** which use the recursor operator **R** extensively, permitting substantial economy in the size of the typed programs obtained, and in their computation time requirements. For instance, the function $\min(x, y)$ can be defined by recursion of type 2 so that the execution time is the size of the smaller argument [Col88]. The reformulation of $\kappa$ for generative axiomatizations of data types has also the advantage of being applicable to first order formalisms.

The functions defined in the $\lambda$-calculus expanded with **0**, **s** and $\mathbf{R}_\tau$ (for all $\tau$), are the *functions defined by primitive recursion in all finite types* [Göd58]. The Function Representation Theorem for $\kappa$ : **M1LP** $\longrightarrow$ **1$\lambda$P** therefore implies:

**THEOREM XVI (Gödel** [Göd58]) *Every provably recursive function of Peano Arithmetic is defined by primitive recursion in finite types.*

**Proof.** Suppose a program $\mathcal{P}$ for $f$ is provably total in Peano Arithmetic. By Theorem XIV there is a derivation $\Pi$ of $N^1(\mathbf{f})$ in **M1LP** + $\mathcal{P}$. Then $\kappa\Pi$ is an expression of **1$\lambda$P**, which represents $f$. That is, $f$ is primitive recursive in finite type. $\dashv$

The converse of the theorem above also holds [Göd58]. One proof uses the provability in Peano's Arithmetic, for each bound $b$, of the normalization theorem for terms of $\mathbf{1}\lambda\mathbf{P}$ of types of order $\leq b$.

In addition to its simplicity, the proof above of Gödel's Theorem permits us to view the deduction of totality of $f$, in a natural formalism, as itself being an annotated primitive recursive program for $f$. Also, the proof permits an immediate generalization of the theorem. Taking in place of the natural numbers an arbitrary data type $D$, we obtain

**THEOREM XVII** *Let $D$ be a data type. The functions provably total in* $\mathbf{C1L}D$ *are precisely the functions definable in* $\mathbf{1}\lambda D$*, i.e. the $D$-primitive recursive functions in all finite types.* ⊣

# 6. Inductive axiomatizations

## 6.1. Inductive definitions

A master method for generating sets predicatively is first order positive inductive definitions (see e.g. [Mos74, Acz77]). Suppose $\Phi$ is a positive first order operator: $\Phi R \equiv \lambda\hat{u}.\varphi[R](\hat{u})$, where $arity(R) = arity(\hat{u})$, and $\varphi$ is first order, with no negative occurrences of $R$. Then $\Phi$ is monotone: $R \subseteq Q$ implies $\Phi R \subseteq \Phi Q$. The chain $\Phi^\xi =_{Df} \Phi(\cup_{\eta<\xi}\Phi^\eta)$ ($\xi$ an ordinal) is non-decreasing, since $\Phi$ is monotone, and therefore reaches a fixpoint, denoted $\mu R.\Phi \equiv \mu R.\lambda\hat{u}.\varphi$. The monotonicity also easily implies that the resulting fixpoint is minimal, i.e. contained in every fixpoint of $\Phi$. The fixpoint $\mu R.\Phi$ is explicitly defined by

$$\forall R.(\Phi R \subseteq R \ \rightarrow R(\hat{x})),$$

where

$$(\Phi R \subseteq R) \ \equiv \ \forall\hat{u}.\varphi[R] \rightarrow R(\hat{u}).$$

We say that $\varphi$ is *well-parsed* if for every tuple $\hat{t}$ of closed terms (with $arity(\hat{t}) = arity(R)$), there is at most one normal **M2L** deduction (up to renaming of variables) of $(\mu R.\Phi)[\hat{t}]$.

For a well-parsed inductive definition, we obtain a canonical representation of $\mu R.\Phi$ in $2\lambda$: if $\Pi_1$ is a proof of $(\mu R.\Phi)[\hat{t}]$, then $\kappa\Pi_1$ will represent $\hat{t}$ as an element of $\mu R.\Phi$. Given this object representation, one obtains function representations as for generative definitions.

In particular, this implies an alternative representation of inductively generated data types, and — more generally — of data systems. Suppose a data system $\{D_i\}_{i=1..k}$ is generated by closure conditions $C_{i1} \ldots C_{ir_i}$ ($i = 1 \ldots k$). The data types are then the minimal solutions of a set of simultaneous equivalences of the form

$$D_i(x) \equiv \ cond_{i1} \ \vee \ \cdots \ \vee \ cond_{ir_i},$$

where, if say $C_{ij} \equiv D_1(u) \wedge D_2(v) \rightarrow D_i(\mathrm{f}(u, v))$, then $cond_{ij} \equiv \exists u, v.\, D_1(u) \wedge D_2(v) \wedge x = \mathrm{f}(u, v)$.

**Lemma 6..1** *If a data system is well-parsed (in the sense of §3.4), then its inductive definition is well-parsed.* ⊣

In particular, the set of the natural numbers can be defined as the inductive closure of

$$x = \mathbf{0} \ \vee \ \exists y.\, (R(y) \wedge x = \mathrm{s}y).$$

Using Prawitz's definition of the existential quantifier in terms of $\rightarrow$ and $\forall$ [Pra65], we have

$$\begin{aligned} x = \mathbf{0} \vee \exists y.\, (R(y) \wedge x = \mathrm{s}y) \ &\leftrightarrow \ \exists y.\, (R(y) \wedge x = \mathrm{s}y \,.\vee\ x = \mathbf{0}) \\ &\leftrightarrow \ \forall Q(\forall y.\, ((R(y) \wedge x = \mathrm{s}y \vee .\, x = \mathbf{0}) \ \rightarrow Q(x)) \ \rightarrow Q(x)) \\ &\leftrightarrow \ \forall Q(\forall y.\, (R(y) \ \rightarrow Q(\mathrm{s}y)) \ \rightarrow \ Q(\mathbf{0}) \ \rightarrow Q(x)) \end{aligned}$$

An explicit inductive definition of the natural numbers is therefore

$$M(x) \equiv \forall R(\forall u(\forall Q(\forall y.\,(R(y) \to Q(\mathbf{s}y)) \to Q(\mathbf{0}) \to Q(u)) \to R(u)) \to R(x)).$$

**Lemma 6..2** *For natural numbers k, there is a unique normal* **M2L** *derivation* $\Pi_k$ *of* $M(\tilde{k})$. ⊣

The *k'th pure inductive numeral*, $\tilde{k}$, fall out as the image under $\kappa$ of $\Pi_k$. Explicitly,

$$\begin{aligned} \tilde{0} &=_{Df} \Lambda R.\lambda q.0_{R,q} &&: \forall R.\sigma[R] \to R \\ \widetilde{k+1} &=_{Df} \Lambda R.\lambda q.(k+1)_{R,q} &&: \forall R.\sigma[R] \to R, \end{aligned}$$

where

$$\begin{aligned} \sigma[R] &=_{Df} (\forall Q.(R \to Q) \to Q \to Q) \to R \\ 0_{R,q} &=_{Df} q(\Lambda Q.\lambda s^{R\to Q} z^{Q}.z) &&: R \\ (k+1)_{R,q} &=_{Df} q(\Lambda Q.\lambda s^{R\to Q} z^{Q}.sk_{R,q}) &&: R \end{aligned}$$

Pure inductive numerals, and similar representations of data types, can be used as basis for function representation, but they do not seem to have, in and by themselves, any advantage over the more direct representation of data types. However, when inductive definitions are used as a basis for *axiomatization* of data types, useful forms of object and function representations do emerge.

## 6.2. Inductive Axiomatizations

The salient properties of $D \equiv \mu R.\Phi$ are the closure property, $\Phi D \subseteq D$, and the minimality property: for every relation $R$ of the proper arity, if $\Phi R \subseteq R$, then $D \subseteq R$.

Converting generative axiomatizations into inductive axiomatizations is a trivial change of notation. However, the combinatorics of proofs changes, resulting in different $\lambda$-representations. For the same reason, distinctions between logically equivalent forms of closure and minimality are also of interest.

Closure can be stated either as an axiom,

$$\begin{aligned} \textbf{Closure}(D)\textbf{:} \qquad & \Phi D \subseteq D, \\ \text{i.e.,} \qquad & \forall \hat{x}.\varphi[D](\hat{x}) \to D(\hat{x}), \end{aligned}$$

or as an inference rule,

$$\frac{(\Phi D)[\hat{t}]}{D(\hat{t})}.$$

A straightforward induction axiom is

$$\textbf{Induction}(D): \qquad \forall R.\, \Phi R \subseteq R \;\rightarrow\; D \subseteq R,$$
$$\text{i.e.,} \qquad \forall R.((\lambda \hat{u}.\varphi[R] \subseteq R) \;\rightarrow\; \forall \hat{x}.\, D(\hat{x}) \rightarrow R(\hat{x}))).$$

A first order *Induction Schema* for $D$ is defined similarly.

The minimality of $D$ can be proved from *Induction*($D$) using the monotonicity of $\Phi$. A statement of induction with a built-in monotonicity condition, akin to the one in [Men87], is

$$\textbf{M-Induction}(D): \qquad \forall X.\, (\forall Y.\, (Y \subseteq X \;\rightarrow\; \Phi Y \subseteq X) \;\rightarrow\; D \subseteq X).$$

The converse of the closure principle is derivable from either form of induction. However, it is also useful to consider that converse separately, as a *Co-Closure Rule*:

$$\frac{D(\hat{t})}{\Phi D(\hat{t})}.$$

This rule is *strictly weaker* than the induction axioms above, since it is consistent with the interpretation of $D$ as any fixpoint of $\Phi$, not necessarily the minimal one.

Let **M2LI** (**I** for "inductive") be the extension of **M2L** with the $\mu$ operator for positive formulas in the language, with the Closure Axiom (or Closure Rule), and with the M-Induction Axiom, for all fixpoints $D$. **M2LJ** is the variant of **M2LI** using Induction rather than M-Induction. **M1LJ** is like **M2LJ**, but with the Induction Schema rather than the Induction Axiom.

Let **M2L**$\mu$ be like **M2LI**, but with the Co-Closure Rule in place of induction. The variant of **M2L**$\mu$ based on classical logic is similar to the "first order programming logic" of Cartwright [Car84], who reports his experience that most of the interesting facts about recursive programs are provable therein, that is, without the full power of induction!

### 6.3. Reductions and normalization

Since closure for $D$ and induction for $D$ can be viewed as $D$-introduction and $D$-elimination rules, it is natural to define *closure-reductions* that eliminate instances of Closure followed by M-Induction or Induction. Let $\varphi$, $\Phi$ and $R$ be as above.

Given

$$\Pi \equiv \frac{\Theta_1' \quad \Theta_2 \quad \Phi D(\hat{t})}{D \subseteq \chi \qquad D(\hat{t})}{\chi[\hat{t}]},$$

where $\chi$ abbreviates $\lambda\hat{u}.\chi$, and

$$\Theta_1' \equiv \frac{\textit{M-Induction} \qquad\qquad \Theta_1}{\forall R.(R \subseteq \chi \rightarrow \Phi R \subseteq \chi) \rightarrow D \subseteq \chi \qquad \forall R.(R \subseteq \chi \rightarrow \Phi R \subseteq \chi)}{D \subseteq \chi},$$

the *Closure Reduction* for **M2LI** maps $\Pi$ to

$$\frac{\dfrac{\Theta_1' \quad \Theta_1}{D \subseteq \chi \qquad \forall R.(R \subseteq \chi \rightarrow \Phi R \subseteq \chi) \quad D \subseteq \chi \rightarrow \Phi D \subseteq \chi}{\Phi D \subseteq \chi} \quad \Theta_2 \quad \Phi D(\hat{t}) \quad D(\hat{t})}{\chi[\hat{t}]}$$

*Closure Reductions* for **M2LJ** are defined as follows. Recall that $\Phi = \lambda\hat{u}.\varphi[R]$ is positive in $R$. Let $\Xi_\Phi[Q, S]$ be the straightforward derivation of $\Phi Q \subseteq \Phi S$ from $Q \subseteq S$, defined by induction on $\varphi$. The least trivial case of that induction is for $\varphi$ of the form $\mu P.\psi[R, P]$. By induction assumption, there is a derivation leading from the assumption $Q \subseteq S$ to $\psi[Q, \mu P.\psi[S, P]] \subseteq \psi[S, \mu P.\psi[S, P]]$. By *Closure* the latter is $\subseteq \mu P.\psi[S, P]$. By *Induction* this implies $\mu P.\psi[Q, P] \subseteq \mu P.\psi[S, P]$.

Now, given

$$\Pi \equiv \frac{\Theta_1' \quad \Theta_2 \quad (\Phi D)[\hat{t}]}{D \subseteq \chi \qquad D(\hat{t})}{\chi[\hat{t}]},$$

where

$$\Theta_1' \equiv \frac{\forall R(\Phi R \subseteq R \rightarrow D \subseteq R) \qquad \Theta_1}{\Phi\chi \subseteq \chi \rightarrow D \subseteq \chi \qquad \Phi\chi \subseteq \chi}{D \subseteq \chi},$$

we let $\Pi$ reduce to

$$\dfrac{\dfrac{\Theta_1 \qquad \dfrac{\dfrac{\Theta_1'}{D \subseteq \chi}}{\Xi_\Phi[D,\chi]} \qquad \dfrac{\Theta_2}{(\Phi D)[\hat{t}]}}{\Phi D \subseteq \Phi\chi}}{(\Phi\chi)[\hat{t}]} \qquad \dfrac{\Phi\chi \subseteq \chi}{}}{\chi[\hat{t}]}$$

Closure Reductions for M2L$\mu$ are defined trivially.

**THEOREM XVIII** *The calculi* **M2LI**, **M2LJ**, **M1LJ**, *and* **M2L**$\mu$ *all have the strong normalization property with respect to the reductions of* **M2L** *and the corresponding variants of Closure Reduction.* ⊣

A proof can be modeled after [Men87], where a strong normalization theorem for a $\lambda$-calculus analogous to **M2LJ** is proved, using Girard's method [Gir72]. (The argument is also outlined in [Kri87].)

### 6.4. The contraction homomorphism for inductive formalisms

We now adapt the homomorphism $\kappa$ to the formalisms above for positive inductive definitions.

#### 6.4.1. $\lambda$-expressions for M2LI

Let **2λI** be **2λ** modified as follows. (The system **2λI** is akin to the fixpoint formalism of [Men87].)

1. The *type structure* is extended with a fixpoint operator: if $\tau$ is a type where type variable $R$ has no free negative occurrences, then $\mu R.\tau$ is a type. We write $\tau[\sigma]$ for $\tau[\sigma/R]$; in particular, $\tau[R] \equiv \tau$.
2. The formation rules for *expressions* are supplemented with constants: For each type $\delta \equiv \mu R.\tau[R]$, a *closure constant* $\mathrm{C}_\delta$, of type $\tau[\delta] \to \delta$, and an *induction constant* $\mathrm{I}_\delta$, of type $\forall Q.(\forall R.((R \to Q) \to \tau[R] \to Q) \to \delta \to Q)$.
3. The *reductions* are supplemented with Closure Reduction: for arbitrary type $\sigma$, and expressions $E : \forall R.((R \to \sigma) \to \tau[R] \to \sigma)$, and $F : \tau[\delta]$,

$$\mathrm{I}_\delta\sigma E(\mathrm{C}F) \qquad \text{reduces to} \qquad E\delta(\mathrm{I}_\delta\sigma E)F$$

The definition of $\varphi \mapsto \underline{\varphi}$ is augmented by the clause $\underline{\mu R.\lambda\hat{u}.\varphi} =_{Df} \mu R.\underline{\varphi}$. A homomorphism $\kappa : \mathbf{M2LI} \to 2\lambda\mathbf{I}$, extending the homomorphism $\kappa : \mathbf{M2L} \to 2\lambda$, is defined by assigning $\mathbf{C}_{\underline{\mu R.\varphi}}$ to the closure axiom for $D \equiv \mu R.\lambda\hat{u}.\varphi$, and $\mathbf{I}_{\underline{\mu R.\varphi}}$ to the M-Induction axiom for $D$.

**Lemma 6..3** *$\kappa$ maps closure reductions of* **M2LI** *to closure reductions of* $2\lambda\mathbf{I}$. ⊣

### 6.4.2. $\lambda$-expressions for M2LJ

The formalism $2\lambda\mathbf{J}$ is defined like $2\lambda\mathbf{I}$, with two changes. First, for each $\delta = \mu R.\tau$, in place of a constant $\mathbf{I}_\delta$ we have a constant $\mathbf{J}_\delta$, of type $\forall R.((R \to \tau[R]) \to \delta \to R)$. Then, the closure reductions of $2\lambda\mathbf{J}$ are modified accordingly. We need expressions $X_{\pi,\alpha,\beta}$, that correspond to the derivations $\Xi_\phi[Q,S]$ used in §6.3 to formulate closure reductions for **M2LJ**, i.e. such that $\kappa\Xi_\phi[A,B] = X_{\underline{\phi},\underline{A},\underline{B}}$. We define these expressions explicitly. Fix a type variable $R$. For $\pi$ in which $R$ occurs only positively, we define $X_{\pi,\alpha,\beta}$ of type $(\alpha \to \beta) \to \pi[\alpha/R] \to \pi[\beta/R]$. To proceed inductively, we define these expressions together with dual expressions $\bar{X}_{\nu,\alpha,\beta}$, of type $(\alpha \to \beta) \to \nu[\beta] \to \nu[\alpha]$, for $\nu$ in which $R$ occurs only negatively.

$$
\begin{aligned}
X_{R,\alpha,\beta} &=_{Df} \lambda x^{\alpha\to\beta}.\, x \\
\bar{X}_{P,\alpha,\beta} \equiv X_{P,\alpha,\beta} &=_{Df} \lambda x^{\alpha\to\beta} y^{P}.\, y \quad (P \text{ other than } R) \\
X_{\nu\to\pi,\alpha,\beta} &=_{Df} \lambda x^{\alpha\to\beta} y^{\nu[\alpha]\to\pi[\alpha]} z^{\nu[\beta]}.\, X_{\pi,\alpha,\beta} x (y (\bar{X}_{\nu,\alpha,\beta} x z)) \\
\bar{X}_{\pi\to\nu,\alpha,\beta} &=_{Df} \lambda x^{\alpha\to\beta} y^{\pi[\beta]\to\nu[\beta]} z^{\pi[\alpha]}.\, \bar{X}_{\nu,\alpha,\beta} x (y (X_{\pi,\alpha,\beta} x z)) \\
X_{\forall P.\pi,\alpha,\beta} &=_{Df} \Lambda P.\, X_{\pi,\alpha,\beta} \\
\bar{X}_{\forall P.\nu,\alpha,\beta} &=_{Df} \Lambda P.\, \bar{X}_{\nu,\alpha,\beta} \\
X_{\mu P.\pi[R,P],\alpha,\beta} &=_{Df} \lambda x^{\alpha\to\beta} y^{\mu P.\pi[\alpha,P]}. \\
&\qquad \mathbf{J}_{\mu P.\pi[\alpha,P]} (\lambda z^{\pi[\alpha,\mu P.\pi[\beta,P]]}.\, \mathbf{C}_{\mu P.\pi[\beta,P]} (X_{\pi[R,\mu P.\pi[\beta,P]],\alpha,\beta}\, x\, z))\, y \\
\bar{X}_{\mu P.\nu[R,P],\alpha,\beta} &=_{Df} \lambda x^{\alpha\to\beta} y^{\mu P.\nu[\beta,P]}. \\
&\qquad \mathbf{J}_{\mu P.\nu[\beta,P]} (\lambda z^{\nu[\beta,\mu P.\nu[\alpha,P]]}.\, \mathbf{C}_{\mu P.\nu[\alpha,P]} (\bar{X}_{\nu[R,\mu P.\nu[\alpha,P]],\alpha,\beta}\, x\, z))\, y
\end{aligned}
$$

*Closure reductions* for **M2LJ** are now defined as follows. For type $\sigma$, and expressions $E : \tau[\sigma] \to \sigma$ and $F : \tau[\delta]$, let

$$\mathbf{J}_\delta \sigma E(\mathbf{C}_\delta F) \qquad \text{reduce to} \qquad E(X_{\tau,\delta,\sigma}(\mathbf{J}_\tau \sigma E)F).$$

A homomorphism $\kappa : \mathbf{M2LJ} \to 2\lambda\mathbf{J}$ is defined as for **M2LI** and $2\lambda\mathbf{I}$.

**Lemma 6..4** *$\kappa$ maps closure reductions of* **M2LJ** *to closure reductions of* $2\lambda\mathbf{J}$. ⊣

### 6.4.3. Closure-free $\lambda$-expressions

The meaning of $\delta = \mu R.\tau$ is conveyed in the calculi above partly by the type-changing constant $\mathbf{C}_\delta$. One can instead consider the types $\delta$ and $\tau[\delta]$ as interchangeable, and dispense with $\mathbf{C}_\delta$.

Let $2\lambda\mu$ be $2\lambda$ modified as follows.

1. The *type structure* is extended with a fixpoint operator, as for $2\lambda\mathbf{I}$.
2. The type-correctness of functional application is liberalized: Let the relation $=_\mu$ between types be the symmetric and transitive closure of substituting $\tau[\delta]$ for $\delta$ in types (where $\delta = \mu R.\tau$). If $E : \sigma \to \rho$, and $F : \sigma'$, where $\sigma =_\mu \sigma'$, then $EF$ is a correctly typed expression, of type $\rho$.

A homomorphism $\kappa$ : **M2L**$\mu \to 2\lambda\mu$, is defined by extending $\kappa$ : **M2L** $\to 2\lambda$. Here, if $\Pi$ is a derivation of $\varphi$, then $\kappa\Pi$ is of a type $=_\mu \underline{\varphi}$. The additional clause is: if $\Pi$ is $\Pi_0$ extended with an instance of the Closure Rule or the Co-Closure rule, then $\kappa\Pi =_{Df} \kappa\Pi_0$.

Let $2\lambda\mathbf{I}^-$ be the following extension of $2\lambda\mu$.

1. For each $\delta = \mu R.\tau$ there is an induction constant $\mathbf{I}_\delta$, as for $2\lambda\mathbf{I}$ (no closure constants).
2. The *reductions* are supplemented with *Closure Reductions*: for arbitrary type $\sigma$, expression $E$, of type $\forall R.((R \to \sigma) \to \tau[R] \to \sigma)$, and expression $F$, of type $\tau[\delta] =_\mu \delta$,

$$\mathbf{I}_\delta \sigma EF \qquad \text{reduces to} \qquad E\delta(\mathbf{I}_\delta \sigma E)F.$$

(I.e., the reductions of $2\lambda\mathbf{I}$, but with the constant $\mathbf{C}_\delta$ dropped.)

$2\lambda\mathbf{I}^-$ differs from $2\lambda\mathbf{I}$ in that the Closure constant is no longer needed once each type $\tau[\delta]$ (where $\delta = \mu R.\tau$) is identified with $\delta$. The calculus $2\lambda\mathbf{J}^-$ is a similar modification of $2\lambda\mathbf{J}$.

A homomorphism $\kappa$, from derivations of **M2LI** (formulated with the Closure Rule) to expressions of $2\lambda\mathbf{I}^-$, is defined by extending $\kappa$ : **M2L** $\to 2\lambda$. The two additional clauses are: $\mathbf{I}_{\underline{\mu R.\varphi}}$ is assigned to the M-Induction axiom for $D$; and if $\Pi$ is $\Pi_0$ extended with an instance of the Closure Rule, then $\kappa\Pi =_{Df} \kappa\Pi_0$. A homomorphism $\kappa$ : **M2LJ** $\to 2\lambda\mathbf{J}^-$ is defined similarly.

**Lemma 6..5** *$\kappa$ maps closure reductions of* **M2LI** *to closure reductions of* $2\lambda\mathbf{I}^-$, *and closure reductions of* **M2LJ** *are mapped to closure reductions of* $2\lambda\mathbf{J}^-$. ⊣

## 6.5. Representation of numerals

Given the inductive definition of $N$ in §6.1, the Closure Axiom for $N$, *Closure*($N$), is

$$\forall Q(\forall y.\,(N(y) \to Q(\mathrm{sy})) \to Q(\mathbf{0}) \to Q(x)) \to N(x).$$

**Lemma 6..6** *For natural numbers $k$, there is a unique normal derivation $\Pi_k$ of $N(\bar{k})$ in* **M2LI** *and in* **M2LJ**. *$\Pi_k$ is in fact a derivation in* **M2L** + *Closure*($N$) *(i.e. no induction axiom is used).* ⊣

(Note that the derivation $\Pi_k$ here is different from the derivation $\Pi_k$ of Lemma 6..2.)

We now define the *k'th inductive numerals*, as the image under $\kappa : \mathbf{M2L}\mu \to 2\lambda\mu$ of $\Pi_k$ (we use the same notation as for the pure inductive numerals above, but we have no further use for the latter).

$$\begin{aligned} \tilde{0} &=_{Df} \Lambda Q\lambda s^{N\to Q}z^{Q}.z, \\ \widetilde{k+1} &=_{Df} \Lambda Q\lambda s^{N\to Q}z^{Q}.s\tilde{k}. \end{aligned}$$

Disregarding types, these are

$$\begin{aligned} \tilde{0} &=_{Df} \lambda sz.\, z \\ \widetilde{k+1} &=_{Df} \lambda sz.\, s\tilde{k} \end{aligned}$$

Again we have function representation theorems, with respect to inductive numerals, for each one of the homomorphisms $\kappa : \mathbf{M2L}\mu \to 2\lambda\mu$, $\kappa : \mathbf{M2LI}^{-} \to 2\lambda\mathbf{I}^{-}$, and $\kappa : \mathbf{M2LJ}^{-} \to 2\lambda\mathbf{J}^{-}$.

**THEOREM XIX** *The functions representable in $2\lambda\mathbf{I}^{-}$ and $2\lambda\mathbf{J}^{-}$, with respect to the inductive numerals, are precisely the provably recursive functions of second order arithmetic.*

*The functions representable in $2\lambda\mu$ are precisely the functions provably recursive in Second Order Arithmetic, with Induction replaced by the weaker axiom $\forall x.(x = 0 \vee \exists y.x = \mathbf{s}y)$.*

The inductive numerals were discovered by Michel Parigot [Par88,Par89]. Their major advantage is that they enable a representation of the predecessor function, $\lambda x^{N}.xNI\tilde{0}$, where $I =_{Df} \lambda u^{N}.u$, which is computable, by $\beta$-reductions, in constant time. This expression is the image, under $\kappa : \mathbf{M2L}\mu \to 2\lambda\mu$, of the derivation

$$\dfrac{\dfrac{\dfrac{[N(x)]}{\forall Q.(\forall z(N(z) \to Q(\mathbf{s}z)) \to Q(0) \to Q(x))}}{\forall z(N(z) \to N(\mathbf{ps}z)) \to Q(\mathbf{p}0) \to Q(\mathbf{p}x))} \quad \dfrac{[N(z)]}{N(\mathbf{ps}z)}}{\forall z.(N(z) \to N(\mathbf{ps}z))} \quad \dfrac{\dfrac{[R(0)]}{R(\mathbf{p}0)}}{R(0) \to R(\mathbf{p}0)}}{\forall z.(N(z) \to R(\mathbf{s}z)) \to R(0) \to R(\mathbf{p}0)} \quad N(\mathbf{p}0)$$

$$\dfrac{N(\mathbf{p}x)}{N(x) \to N(\mathbf{p}x)}$$

This ease of representing the predecessor is due to the "cumulative" nature of the inductive numerals, in the sense that they contain all previous numerals as (easily extractable) subexpressions. This is similar to von Neumann's numerals in Set Theory (a numeral is the set of

smaller numerals), and to Scott's numerals in the $\lambda$-calculus, for which the existence of an easy representation of the predecessor is well known [Sco63,Bar81,§6.2.9]. Scott's numerals can be typed by recursive types, like the inductive numerals.

Inductive numerals can be alternatively perceived as the fixpoint of an attempt to define numerals stronger than Church's, to permit an easy representation of the predecessor function. Proving $N(x) \to N(\mathbf{p}x)$ for the predecessor function $\mathbf{p}$ is easily reduced to proving $\forall z.N(\mathbf{p}z) \to N(\mathbf{p}\mathbf{s}z)$. Since for $z \in N$ the implication is trivial, one is tempted to define

$$N_0(x) \quad =_{Df} \quad \forall R.\forall z.(N(z) \to R(\mathbf{s}z)) \to R(\mathbf{0}) \to R(x).$$

($N_0$ is equivalent in **M2L** to $N$.) The proof of $N_0(z) \to N(\mathbf{p}z)$ is easy, so one would like to have $N_0 \equiv N$, leading to the fixpoint definition for $N$ (in its second order form!).

## 6.6. Inductive representation of data types and of destructors

Not surprisingly, the development above for natural numbers generalizes straightforwardly to arbitrary data types and data systems. An inductive representation $rep(t)$ of data objects $t \in B(L)$ is obtained analogously to the inductive numerals, i.e. $rep(t)$ contains as subexpressions the representation of the subexpressions of $t$.

From this one readily obtains representations of the destructor and discriminator functions for $B(L)$ (cf. §5.2), each of which is computable in a constant number of steps.

## 7. Controlled Abstraction

The functions whose representation is obtained from the homomorphism $\kappa$ : **M2L** $\rightarrow 2\lambda$ constitute a vast class, the provably recursive functions of second order arithmetic. We survey in this section restricted forms of **M2L** and of $2\lambda$, to which correspond more restricted classes of computable functions. §§7.1 and 7.2 deal with restrictions whose effect fails to be drastic. In §7.3 we consider predicative variants of these formalisms, for which the corresponding classes of functions are far more feasibly computable.

### 7.1. Restricted nesting of quantification

Quantifier alternation underlies several important descriptive hierarchies, such as the arithmetical, analytic, and first order query hierarchies [Kle55,CH82]. In **M2L** and $2\lambda$, existential quantification is replaced by negative occurrences of universal quantification. We thus define the following classes of $2\lambda$ types. $\Sigma_0$ and $\Pi_0$ consist of the quantifier free types. If $\sigma \in \Sigma_n$ and $\pi \in \Pi_n$, then $(\sigma \rightarrow \pi) \in \Pi_n$; $(\pi \rightarrow \sigma) \in \Sigma_n$; $\sigma, \pi \in \Sigma_{n+1} \cap \Pi_{n+1}$; and $(\forall R.\sigma) \in \Pi_{n+1}$. A formula $\varphi$ is $\Sigma_n$ (respectively, $\Pi_n$) if the type $\underline{\varphi}$ is $\Sigma_n$ (respectively, $\Pi_n$).

Let $\mathbf{M2L}_k$ be **M2L** with comprehension restricted to $\Pi_k$ relations, and similarly for **C2L**. Let $2\lambda_k$ be $2\lambda$ where type arguments are restricted to be in $\Pi_k$. Then $\kappa$ maps proofs of $\mathbf{M2L}_k$ to expressions of $2\lambda_k$. This classification of second order formulas has a simple relation to the analytical hierarchy: every $\Pi^1_1$ relation is expressible as $\forall R \exists x.\,(N(x) \wedge \forall y.N(y) \rightarrow \psi[x,y])$, where $\psi$ is a first order formula of arithmetic (two number quantifiers are needed because the second order variable is relational). The latter formula is in $\Pi_2$. More generally, every $\Pi^1_k$ relation is expressible by a $\Pi_{k+1}$ formula of pure second order logic.

**THEOREM XX** *Let $\mathcal{P}$ be a program, with target* **f**, *that computes a numeric function $f$. If $\Pi$ is an $\mathbf{M2L}_k + \mathcal{P}$ derivation of $N^1(\mathbf{f})$, then $\kappa\Pi$ represents $f$ in $2\lambda_{k+1}$. Hence, every provably total computable functions of $\mathbf{C2A}_k$ is representable in $2\lambda_{k+1}$.*

**Proof.** Similar to the proof of Theorem IV. ⊣

### 7.2. Closed comprehension and stationary types

Among the difficulties of implementing programming languages with a full type quantification discipline is the potential proliferation of types, preventing effective predication of types at compile time[1]. This problem can be bypassed by restricting the allowable type arguments. Call a $\lambda$ expression *stationary* if all type arguments therein are type variables or closed type

[1] In some programming languages, such as ALPHARD [WLS76], the restriction is to explicitly disallow procedure definitions that would not permit prediction at compile time of all potential types in execution.

expressions. If $E$ is stationary, then no expression to which $E$ reduces contains a type argument not in $E$ [FLO83].

For a formalism **S**, let $\mathbf{S}^c$ be **S** denote the restriction of **S** where Comprehension and Induction (where present) are allowed only for eigen formulas without free relational variables.

**Lemma 7..1** [Fri81] *Let $\varphi$ be a formula of first order arithmetic. If $\varphi$ is a theorem of* **C2A**, *then $\varphi$ is a theorem of* $\mathbf{C2A}^c$.

From this we obtain:

**THEOREM XXI** [Lei81] *Every function $f$ provably total in* **C2A** *is representable in the stationary fragment of $2\lambda$. Hence, the normalization property of the stationary fragment of $2\lambda$ is not provable in Second Order Arithmetic.*

**Proof.** Suppose $\vdash_{\mathbf{C2A}} \forall x \exists y. T(\vec{e}_{\mathcal{P}}, x, y)$. By Lemma 7:.1 this implies $\vdash_{\mathbf{C2A}^c} \forall x \exists y. T(\vec{e}_{\mathcal{P}}, x, y)$. As in the proof of Theorem II, we obtain $\vdash_{\mathbf{M2A}^c} \forall x \exists y. T(\vec{e}_{\mathcal{P}}, x, y)$, from which also $\vdash_{\mathbf{M2L}^c+\mathcal{P}} N^1(\mathbf{f})$. If $\Pi$ is an $\mathbf{M2L}^c+\mathcal{P}$ derivation of $N^1(\mathbf{f})$, then $\kappa\Pi$ is a stationary expression of $2\lambda$ that represents $f$. ⊣

## 7.3. Predicative second order calculi

The impredicative nature of comprehension is bypassed in stratified higher order logic, where relations are classified into levels. Fixing an ordinal $\Theta$, the *levels* are the ordinals $\prec \Theta$. For each level $\alpha$ there are relational variables of level $\alpha$. The *level of a formula* $\varphi$ is the largest of $level(R)$ for $R$ free in $\varphi$ and $1 + level(R)$ for $R$ bound in $\varphi$. *$\Theta$-Ramified Minimal Second Order Logic,* **$\Theta$-RM2L**, has the same rules and axioms as **M2L**, except that relational $\forall$ elimination is restricted:

$$\frac{\forall R.\, \varphi}{\varphi[\lambda \hat{u}.\psi/R]} \qquad \text{where } level(\psi) \leq level(R).$$

Analogously, the *$\Theta$-stratified polymorphic $\lambda$-calculus,* **$\Theta$-R2$\lambda$**, is like $2\lambda$, except that ordinals $\prec \Theta$ are used as *levels* into which the type variables are classified. The *level of a type* $\tau$ is the largest of $level(R)$ for type variable $R$ free in $\tau$ and $1 + level(R)$ for $R$ bound in $\tau$. *Expressions* $E$ are defined as for $2\lambda$, except that if $E$ is an expression of type $\forall R.\tau$, then $E\sigma$ is a legal expression of type $\tau[\sigma/t]$ only under the proviso that $level(\sigma) \leq level(R)$.

The idea of stratifying abstraction into levels goes back to Russel's Ramified Type Theory, 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 considered by Statman [Sta81].

The definition of a homomorphism $\kappa : \Theta\text{-}\mathbf{RM2L} \rightarrow \Theta\text{-}\mathbf{R2}\lambda$ is identical to $\kappa : \mathbf{M2L} \rightarrow \mathbf{2}\lambda$.

For the stratified formalisms above we now have the analog of Theorem IV:

**THEOREM XXII** *Let $\mathcal{P}$ be a program, with target* **f**, *that computes the numeric function $f$. If $\Pi$ is a derivation in $\Theta$-**RM2L** + $\mathcal{P}$ of $N^1(\mathbf{f})$, then $\kappa\Pi$ represents $f$ in $\Theta$-**R2**$\lambda$.*

The computational significance of this result arises from the relation between the functions representable in $\Theta$-**R2**$\lambda$, for various $\Theta$'s, and subrecursive classes. In [Lei89] we showed that for $\Theta = \omega$ these functions are exactly the super-elementary ones (Grzegorczyk's class $\mathcal{E}^4$), for $\Theta = \omega^\omega$ they are the primitive recursive functions, and for $\Theta = \epsilon_o$ they are the provably recursive functions of Peano's Arithmetic.

A subrecursive class smaller yet is obtained when comprehension is further restricted, for the formalisms [**M2L**+ conjunction] and [2$\lambda$+ pairing]. Let **M2L°** be [**M2L**+ conjunction], modified as follows. The relational variables are labeled as being of level 0 or level 1. A formula $\varphi$ is said to be of level 0 if it contains no $\forall$ nor $\rightarrow$, and of level 1 if it contains no $\forall$ binding a variable of level 1. Comprehension is restricted, allowing $\psi[\lambda\hat{u}.\chi/R]$ to be derived from $\forall R.\psi$ only if $level(\chi) \leq level(R)$. Let **2$\lambda$°** be a similar modification of **2**$\lambda$. The homomorphism $\kappa$ : [**M2L**+ conjunction]$\rightarrow$[2$\lambda$+ pairing], defined in §1, maps **M2L°** to **2$\lambda$°**. A proof of $N(x) \rightarrow N^1(\mathbf{f})$ in **M2L°** + $\mathcal{P}$, where $\mathcal{P}$, with target **f**, computes the function $f$, contracts under $\kappa$ into a representation of $f$ in **2$\lambda$°**. The computational significance of this is that the functions representable in **2$\lambda$°** are precisely the elementary functions, i.e. Grzegorczyk's class $\mathcal{E}^3$ [Lei$\gamma$].

# 8. Appendix I. Herbrand-Gödel Computability

## 8.1. Programs

As in §3.1, a *functional vocabulary* is a set of identifiers, the *primitives*, each assigned a type. The types are assigned orders: $order(o) =_{Df} 0$, $order(\tau \to \sigma) =_{Df} \max(1+order(\tau), order(\sigma))$. A *computation space* $B = B(L)$ is the set of *canonical expressions*, i.e. the closed terms of type $o$ in the initial algebra generated by $L$. $B$ is *non-trivial* if $L$ has at least one identifier of type $o$ and one identifier of order 1. $L$ is of *order* $k$ if the types of its identifiers are all of order $\leq k$

Let $V$ be an infinite denumerable set of identifiers, the *program variables*, each associated a type. Let $I$ be a set of identifiers, the *input identifiers*, each associated a type of order 1. (Note that only functions are used as input (oracles).) We assume that $L$, $V$ and $I$ are pairwise disjoint. The set $Term(L, I)$ of terms built on top of $L$ is defined inductively like $B(L)$, except that identifiers from $V$ and $I$ are used. A term in which no program variable applies to a term is *simple*. Thus, if $\mathbf{j}^{o\to o} \in I$ and $\mathbf{f}^{o\to o}, \mathbf{g}^{(o\to o)\to o} \in V$, then $\mathbf{sx}$, $\mathbf{s0}$ and $\mathbf{jx}$ are simple, but $\mathbf{f0}$, $\mathbf{s(f0)}$ and $\mathbf{g(j)}$ are not.

A *statement* over $L, I$ is an equation of the form $ft = s$, where $f \in V$, $t$ is simple, and $ft$, $s \in B(L)$ are of the same type. A *program* over $L$ is a tuple $\mathcal{P} = (\mathcal{P}_0, \mathbf{F}, \mathbf{j}_1, \ldots, \mathbf{j}_r)$, where $\mathbf{j}_1, \ldots, \mathbf{j}_r$ are distinct input identifiers, of respective types $\sigma_1 \ldots \sigma_r$ say, $\mathbf{F}$ is a variable of type $\sigma_1 \to \ldots \to \sigma_r \to o^k \to o$ for some $k \geq 0$, and $\mathcal{P}_0$ is finite set of statements over $L \cup \{\mathbf{j}_1, \ldots, \mathbf{j}_r\}$. $\mathbf{F}$ is the *target* of the program. The *order* of the program is the largest among the orders of types of variables in $\mathcal{P}$. Note that a program of order 1 must have an empty list of input identifiers.

## 8.2. Operational semantics

For $\tau$ of order $\leq 2$, the set $Fnctl^\tau$ of *functionals of type* $\tau$ is defined by induction on $\tau$. $Fnctl^o = B(L)$. $Fnctl^{\rho\to\sigma}$ consists of the partial mappings $F$ from $Fnctl^\rho$ to $Fnctl^\sigma$ that are *monotone*: if $f, g \in Fnctl^\rho$, $f \subseteq g$, then $Ff \subseteq Fg$.

A *valuation* of $I$ in $B$ is an assignment $\eta$ that yields for each identifier $\mathbf{j} \in I$ of type $\tau$ a type $\tau$ functional over $B$.

A program $(\mathcal{P}_0, \mathbf{F}, \mathbf{j}_1, \ldots, \mathbf{j}_r)$ induces the inference rule $\mathcal{P}$ of §2, which, for the special case where the formulas are equations, reads:

$$\mathcal{P}_0: \quad \frac{s[t/x] = q[t/x]}{s[t'/x] = q[t'/x]} \quad \text{where } t = t' \text{ or } t' = t \text{ is a substitution instance of an equation in } \mathcal{P}$$

Note that, in general, the terms $t$ and $t'$ may be of higher type.

We also stipulate a *rule of substitution:*

$$\textbf{Substitution:} \quad \frac{s = q}{s[t/x] = q[t/x]}$$

(This is a derived rule in presence of universal quantification rules.)

Finally, a valuation $\eta$ induces the rule

$$\eta: \quad \frac{s = q}{s' = q'}$$

Here $s'$ ($q'$) is $s$ (respectively, $q$) with possibly some subterm $\mathbf{j}(t)$, where $\mathbf{j} \in I$ and $t \in B$, replaced by $(\eta\mathbf{j})(t)$.

We write $\mathcal{P}_0 + \eta$ for the deductive equational calculus induced by $\mathcal{P}$ and $\eta$ as above. ($\mathcal{P}_0$ provides the computation rules, and $\eta$ is the input functions).

Let $(\mathcal{P}, \mathbf{F}^\tau, \mathbf{j}_1, \ldots, \mathbf{j}_r)$ be a program, $\eta$ a valuation for $I = \{\mathbf{j}_i, \ldots, \mathbf{j}_r\}$. Define a relation $[\mathcal{P}, \eta]$ over $B = B(L)$ by

$$(t_1 \ldots t_r)\ [\mathcal{P}, \eta]\ t_0 \quad \Leftrightarrow_{Df} \quad \mathcal{P}_0 + \eta \vdash \mathbf{F}(\mathbf{j}_1, \ldots, \mathbf{j}_r, t_1 \ldots t_k) = t_0.$$

A functional $F$ of order $\leq 2$ is *computed* by $(\mathcal{P}, \mathbf{F}, \mathbf{j}_1, \ldots, \mathbf{j}_r)$ if $[\mathcal{P}, \eta]$ is the graph of $F$.

**THEOREM XXIII** *Over any non-trivial computation space, the functionals of order 2 computable by programs of order 2 are precisely the recursive functionals in the sense of Kolaitis [Kol85].*

*Over the space of numerals, the functionals of order 2 computed by programs of order 2 are precisely the recursive functionals in the sense of Kleene.*

**Proof Outline.** It is straightforward to verify that all clauses in Kolaitis's definition of a *functional in canonical form* are legal statements of order 2. Kolaitis shows that every recursive functional is the inductive closure of a functional in canonical form. The inductive closure can be simulated by equations, over a non-trivial $B$, as in Kleene's simulation of the numeric minimalization operation $\mu$ by equations [Kle52].

The second part follows from the first by [Kol85]. ⊣

## 8.3. Coherence

The relation $[\mathcal{P}, \eta]$ need not be a function. If it is, for all valuations $\eta$, then we say that $\mathcal{P}$ is *coherent*. The problem of determining program coherence is, of course, undecidable.

However, it is easy to give a set of obviously coherent programs which contains a program for every computable function. First, we give a variant of Kleene's proof [Kle52] that computable functions over natural numbers have coherent programs.

**Lemma 8..1 (Coherence for numeric functions)** *There is a (linear time) decidable collection $P$ of coherent programs which is complete for the computable numeric functions: every program $\mathcal{P}$ can be converted into a program in $P$ which is denotationally equivalent to $\mathcal{P}$ if $\mathcal{P}$ is coherent.*

**Proof.** Every computable numeric function $f$ is definable as $f(\hat{x}) = \mu y.\ g(\hat{x}, y) = 0$, where $g$ is primitive recursive and therefore defined by a coherent system of recursion equations. $f$ is then defined by supplementing the system for $g$ with the following equations (with fresh function identifiers). $\mathbf{a}(u, \mathbf{s}v) = u$; $\mathbf{b}(\hat{x}, \mathbf{0}) = \mathbf{g}(\hat{x}, \mathbf{0})$, $\mathbf{b}(\hat{x}, \mathbf{s}u) = \mathbf{a}(\mathbf{g}(\hat{x}, \mathbf{s}u), \mathbf{b}(\hat{x}, u))$; $\mathbf{c}(u, \mathbf{0}) = u$; $\mathbf{f}(\hat{x}) = \mathbf{c}(u, \mathbf{b}(\hat{x}, u))$. Note that $\mathbf{b}(\hat{x}, y)$ is the same as $\mathbf{g}(\hat{x}, y)$ for values of $y$ up to and including the first zero of the function, and is undefined for larger values of $y$. ⊣

**Lemma 8..2** *For any language $L$, there is a (linear time) decidable collection of coherent programs which is complete for the computable functions of order 1 over $B(L)$.*

**Proof.** The case where $L$ does not have at least one constant primitive and one primitive of order 1 is trivial, since there are then a finite number of canonical expressions. The case where $L$ has at least one constant identifier $\mathbf{e}$, and one function identifier $\mathbf{q}$, say of type $o \to o \to o$, is reduced to coherence of numeric functions by a Gödel coding, as follows. A program $\mathcal{P}$ over $B(L)$ is mapped into a program $\mathcal{P}'$ over the natural numbers, that simulates $\mathcal{P}$ for codes of $B(L)$ in $\omega$. $\mathcal{P}'$ is mapped into a cohenerect program $\mathcal{P}''$, equivalent to $\mathcal{P}'$ if $\mathcal{P}'$ is coherent. $\mathcal{P}''$ can be formulated for an embedding of the natural numbers in $B(L)$, e.g. by letting $\mathbf{0} =_{Df} \mathbf{e}$ and $\mathbf{s} =_{Df} \lambda x.\mathbf{qxe}$. For any of the standard codings of $L$ in $\omega$, the decoding is primitive recursive in $B(L)$. Thus $f$ is computed by the coherent program $\mathcal{P}^o$ over $L$ obtained as the union (with sets of function variables suitably disjoint) of a coherent program that maps the input into a numerically coded form (simulated as above in $B(L)$), the coherent program $\mathcal{P}''$, and a coherent program that decodes the "numeric" output. The coherence of $\mathcal{P}^o$ follows from the coherence and disjointness of its constituents. ⊣

**Lemma 8..3 (Coherence)** *For each $L$, there is a (linear time) decidable collection of coherent programs which is complete for the computable functions of order $\leq 2$ over $B(L)$.*

**Proof Outline.** Repeat the proofs of Lemmas 8..1 and 8..3, but with function input, and using Kleene's [Kle59] definition of type 2 functionals. ⊣

# 9. Appendix II. Extensions of $\kappa$ in M2L

## 9.1. $\eta$-reductions

Consider $\eta$-reductions of $\lambda$-expressions: $\lambda x.Ex$ reduces to $E$. Some $\eta$-reductions might be described as the image under $\kappa$ of an additional reduction rule for derivations, say

$$\Pi \equiv \dfrac{\dfrac{\Delta}{\psi \to \varphi} \quad [\psi]}{\varphi}\,{\scriptstyle \psi \to \varphi} \qquad \text{reduces to} \qquad \Pi' \equiv \begin{matrix}\Delta \\ \psi \to \varphi\end{matrix}$$

However, a derivation $\Pi$ can be normal with respect to reductions of the kind above, and yet $\kappa\Pi$ would not be $\eta$-normal. For instance, the derivation

$$\Pi \equiv \dfrac{\dfrac{[\forall x.\psi]}{\psi[t/x] \to \varphi} \quad \psi[t/x]}{\varphi}\,{\scriptstyle \forall x.\psi \to \varphi}$$

is normal, yet $\kappa\Pi$ can be $\eta$-reduced.

## 9.2. Existential instantiation

The clauses of $\kappa$ for $\exists$ rules are completely analogous to the definition for $\forall$ rules, if existential elimination is formulated as an existential instantiation rule, dual to $\forall$ introduction. An $\exists$ instantiation rule for intuitionistic and for minimal logics, defined in [Lei73], is: if $\exists x.\varphi$ is derived from assumptions $\in \Gamma$, then infer $\varphi[\epsilon x.\{\Gamma \Rightarrow \varphi\}/x]$. Here $\epsilon$ is variable binding descriptor operator, akin to Hilbert's $\epsilon$ notation, with the sequent $\Gamma \Rightarrow \varphi$ as argument. The corresponding reduction is simply:

$$\textbf{Object } \exists\text{:} \quad \Pi \equiv \begin{matrix}\Delta \\ \varphi[t/x] \\ \exists x.\,\varphi \\ \varphi[\epsilon x.\{\Gamma \Rightarrow \varphi\}/x]\end{matrix} \qquad \text{reduces to} \qquad \Pi' \equiv \begin{matrix}\Delta \\ \varphi[t/x]\end{matrix}$$

(The reduction alters the derived formula, but this alteration is of no consequence in proofs of $\epsilon$-free formulas; see [Lei73].)

For a deductive calculus based on existential instantiation, the corresponding clause in the definition $\kappa$ is then obviously

for $\Pi \equiv \begin{matrix} \Delta \\ \exists x.\varphi \\ \varphi[\epsilon x.\{\Gamma \Rightarrow \varphi\}/x] \end{matrix}$ let $\kappa\Pi =_{Df} \kappa\Delta$

## 10. Appendix III. Natural deductions for totality of numeric function

### 10.1. Successor, first derivation

| | | | |
|---|---|---|---|
| 1. | $\forall z(R(z) \to R(\mathbf{s}z))$ | assumption | $s$ |
| 2. | $R(x) \to R(\mathbf{s}x)$ | 1, $z \mapsto x$ | $s$ |
| 3. | $N(x)$ | assumption | $n$ |
| 4. | $\forall z(R(z) \to R(\mathbf{s}z)) \to R(0) \to R(x)$ | 3, $\forall$ elimination | $nR$ |
| 5. | $R(0) \to R(x)$ | 1,4 | $nRs$ |
| 6. | $R(0)$ | assumption | $z$ |
| 7. | $R(x)$ | 5,6 | $nRsz$ |
| 8. | $R(\mathbf{s}x)$ | 2,7 | $s(nRsz)$ |
| 9. | $R(0) \to R(\mathbf{s}x)$ | 8, close 6 | $\lambda z.s(nRsz)$ |
| 10. | $\forall z(R(z) \to R(\mathbf{s}z)) \to R(0) \to R(\mathbf{s}x)$ | 9, close 1 | $\lambda sz.s(nRsz)$ |
| 11. | $\forall R.\forall z(R(z) \to R(\mathbf{s}z)) \to R(0) \to R(\mathbf{s}x)$ | 10 | $\Lambda R.\lambda nz.s(nRsz)$ |
| | $N(\mathbf{s}x)$ | same as 11 | |

### 10.2. Addition, second derivation

| | | | |
|---|---|---|---|
| 1. | $N(y)$ | assumption | $m$ |
| 2. | $\forall z(N(\mathbf{a}(x,z)) \to N(\mathbf{a}(x,\mathbf{s}z)))$ $\to N(\mathbf{a}(x,0)) \to N(\mathbf{a}(x,y))$ | 1, $R \mapsto \lambda z.R(\mathbf{a}(x,z))$ | $m\iota$ |
| 3. | $\forall z(R(z) \to R(\mathbf{s}z))$ | assumption | $s$ |
| 4. | $R(\mathbf{a}(x,z)) \to R(\mathbf{s}\mathbf{a}(x,z)))$ | 3 | $s$ |
| 5. | $N(\mathbf{a}(x,z))$ | assumption | $u$ |
| 6. | $\forall z(R(z) \to R(\mathbf{s}z)) \to R(0) \to R(\mathbf{a}(x,z))$ | 5 | $uR$ |
| 7. | $R(0) \to R(\mathbf{a}(x,z))$ | 6,3 | $uRs$ |
| 8. | $R(0)$ | assumption | $z$ |
| 9. | $R(\mathbf{a}(x,z))$ | 7,8 | $uRsz$ |
| 10. | $R(\mathbf{s}\mathbf{a}(x,z))$ | 4,9 | $s(uRsz)$ |
| 11. | $R(\mathbf{a}(x,\mathbf{s}z))$ | 10, equation | $s(uRsz)$ |
| 12. | $R(0) \to R(\mathbf{a}(x,\mathbf{s}z))$ | 11, close 8 | $\lambda z.s(uRsz)$ |
| 13. | $\forall z(R(z) \to R(\mathbf{s}z)) \to R(0) \to R(\mathbf{a}(x,\mathbf{s}z))$ | 12, close 3 | $\lambda sz.s(uRsz)$ |
| | $N(\mathbf{a}(x,\mathbf{s}z))$ | same as 13 | |
| 14. | $N(\mathbf{a}(x,z)) \to N(\mathbf{a}(x,\mathbf{s}z))$ | 13, close 5 | $\lambda usz.s(uRsz)$ |
| 15. | $\forall z.N(\mathbf{a}(x,z)) \to N(\mathbf{a}(x,\mathbf{s}z))$ | 14 | $\lambda usz.s(uRsz)$ |
| 16. | $N(\mathbf{a}(x,0)) \to N(\mathbf{a}(x,y))$ | 2,15 | $m\iota(\lambda usz.s(uRsz))$ |
| 17. | $N(x)$ | assumption | $n$ |
| 18. | $N(\mathbf{a}(x,0))$ | 17, equation | $n$ |
| 19. | $N(\mathbf{a}(x,y))$ | 16,18 | $m\iota(\lambda usz.s(uRsz))n$ |

## 10.3. Multiplication

| | | | |
|---|---|---|---|
| 1. | $N(y)$ | assumption | $n$ |
| 2. | $\forall z(R(\mathbf{m}(x,z)) \to R(\mathbf{m}(x,\mathbf{s}z))) \to R(\mathbf{m}(x,0)) \to R(\mathbf{m}(x,y))$ | 1, $R \mapsto \lambda z.R(\mathbf{m}(x,z))$ | $nR$ |
| 3. | $N(x)$ | assumption | $m$ |
| 4. | $\forall u(R(\mathbf{a}(\mathbf{m}(x,z),u)) \to R(\mathbf{a}(\mathbf{m}(x,z),\mathbf{s}u))) \to R(\mathbf{a}(\mathbf{m}(x,z),0)) \to R(\mathbf{a}(\mathbf{m}(x,z),x))$ | 3 | $mR$ |
| 5. | $\forall z(R(z) \to R(\mathbf{s}z))$ | assumption | $s$ |
| 6. | $R(\mathbf{a}(\mathbf{m}(x,z),u)) \to R(\mathbf{sa}(\mathbf{m}(x,z),u))$ | 5 | $s$ |
| 7. | $R(\mathbf{a}(\mathbf{m}(x,z),u))$ | assumption | $u$ |
| 8. | $R(\mathbf{sa}(\mathbf{m}(x,z),u))$ | 6,7 | $su$ |
| 9. | $R(\mathbf{a}(\mathbf{m}(x,z),\mathbf{s}u))$ | 8,equation | $su$ |
| 10. | $R(\mathbf{a}(\mathbf{m}(x,z),u)) \to R(\mathbf{a}(\mathbf{m}(x,z),\mathbf{s}u))$ | 9, close 7 | $\lambda u.su$ |
| 11. | $\forall u.R(\mathbf{a}(\mathbf{m}(x,z),u)) \to R(\mathbf{a}(\mathbf{m}(x,z),\mathbf{s}u))$ | 10 | $\lambda u.su$ |
| 12. | $R(\mathbf{a}(\mathbf{m}(x,z),0)) \to R(\mathbf{a}(\mathbf{m}(x,z),x))$ | 11,4 | $mR(\lambda u.su)$ |
| 13. | $R(\mathbf{m}(x,z))$ | assumption | $v$ |
| 14. | $R(\mathbf{a}(\mathbf{m}(x,z),0))$ | 13, equation | $v$ |
| 15. | $R(\mathbf{a}(\mathbf{m}(x,z),x))$ | 12,14 | $mR(\lambda u.su)v$ |
| 16. | $R(\mathbf{m}(x,\mathbf{s}z))$ | 15, equation | $mR(\lambda u.su)v$ |
| 17. | $R(\mathbf{m}(x,z)) \to R(\mathbf{m}(x,\mathbf{s}z))$ | 16, close 13 | $\lambda v.mR(\lambda u.su)v$ |
| 18. | $\forall z.R(\mathbf{m}(x,z)) \to R(\mathbf{m}(x,\mathbf{s}z))$ | 17 | $\lambda v.mR(\lambda u.su)v$ |
| 19. | $R(\mathbf{m}(x,0)) \to R(\mathbf{m}(x,y))$ | 2,18 | $nR(\lambda v.mR(\lambda u.su)v)$ |
| 20. | $R(0)$ | assumption | $z$ |
| 21. | $R(\mathbf{m}(x,0)$ | 20, equation | $z$ |
| 22. | $R(\mathbf{m}(x,y))$ | 19,21 | $nR(\lambda v.mR(\lambda u.su)v)z$ |
| 23. | $R(0) \to R(\mathbf{m}(x,y))$ | 22, close 20 | $\lambda z.nR(\lambda v.mR(\lambda u.su)v)z$ |
| 24. | $\forall z(R(z) \to R(\mathbf{s}z)) \to R(0) \to R(\mathbf{m}(x,y))$ | 23, close 5 | $\lambda sz.nR(\lambda v.mR(\lambda u.su)v)z$ |
| 25. | $\forall R.\forall z(R(z) \to R(\mathbf{s}z)) \to R(0) \to R(\mathbf{m}(x,y))$ $N(\mathbf{m}(x,y))$ | 24 same as 25 | $\Lambda R.\lambda sz.nR(\lambda v.mR(\lambda u.su)v)z$ |

## 10.4. The predecessor function

Let $\varphi[z]$ abbreviate $R(\mathbf{p}(z)) \wedge R(z)$.

| | | | |
|---|---|---|---|
| 1. | $N(x)$ | assumption | $n$ |
| 2. | $\forall z(R(z) \to R(\mathbf{s}z))$ | assumption | $s$ |
| 3. | $R(0)$ | assumption | $z$ |
| 4. | $\forall z(\varphi[z] \to \varphi[\mathbf{s}z]) \to \varphi[0] \to \varphi[x]$ | 1 | $n(R \times R)$ |
| 5. | $\varphi[z]$ | assumption | $u$ |
| 6. | $R(z)$ | 5 | $\mathbf{j}_2 u$ |
| 7. | $R(\mathbf{p}\mathbf{s}z)$ | 6, equation | $\mathbf{j}_2 u$ |
| 8. | $R(\mathbf{s}z)$ | 2,6 | $s\mathbf{j}_2 u$ |
| 9. | $\varphi[\mathbf{s}z]$ | 7,8 | $\langle \mathbf{j}_2 u, s\mathbf{j}_2 u \rangle$ |
| 10. | $\forall z.\, \varphi[z] \to \varphi[\mathbf{s}z]$ | 9, close 5 | $\lambda u.\langle \mathbf{j}_2 u, s\mathbf{j}_2 u \rangle$ |
| 11. | $\varphi[0] \to \varphi[x]$ | 4,10 | $n(R \times R)\lambda u.\langle \mathbf{j}_2 u, s\mathbf{j}_2 u \rangle$ |
| 12. | $R(\mathbf{p}0)$ | 3, equation | $z$ |
| 13. | $\varphi[0]$ | 3, 12 | $\langle z, z \rangle$ |
| 14. | $\varphi[x]$ | 11,13 | $n(R \times R)\lambda u.\langle \mathbf{j}_2 u, s\mathbf{j}_2 u \rangle \langle z, z \rangle$ |
| 15. | $R(\mathbf{p}x)$ | 14 | $\mathbf{j}_1(n(R \times R)\lambda u.\langle \mathbf{j}_2 u, s\mathbf{j}_2 u \rangle \langle z, z \rangle)$ |
| 16. | $R(0) \to R(\mathbf{p}x)$ | 15, close 3 | $\lambda z.\mathbf{j}_1(n(R \times R)\lambda u.\langle \mathbf{j}_2 u, s\mathbf{j}_2 u \rangle \langle z, z \rangle)$ |
| 17. | $\forall z(R(z) \to R(\mathbf{s}z)) \to R(0) \to R(\mathbf{p}x)$ | 15, close 3 | $\lambda sz.\mathbf{j}_1(n(R \times R)\lambda u.\langle \mathbf{j}_2 u, s\mathbf{j}_2 u \rangle \langle z, z \rangle)$ |
| 18. | $N(\mathbf{p}x)$ | 17 | $\Lambda R.\lambda sz.\mathbf{j}_1(n(R \times R)\lambda u.\langle \mathbf{j}_2 u, s\mathbf{j}_2 u \rangle \langle z, z \rangle)$ |

## 11. Appendix IV. Type Containment

**Proposition 11..1** *It is not decidable for data systems $\Delta$, and $D_0, D_1 \in \Delta$, whether $D_0 \subseteq D_1$.*

**Proof.** Consider two context free grammars $G_0, G_1$. Let $L$ have the terminals of $G_0$ and $G_1$ as primitives of type $o$, plus additional primitives $\perp$ of type $o$, and $\mathbf{p}$ of type $o \to o \to o$.

For words $w$ of $G_0$, define $t_w$ by: $t_\epsilon =_{Df} \perp$, $t_{xw} =_{Df} \mathbf{p}xt_w$. (I.e., $t_w$ is $w$ turned into a list.) Define $\Delta$ to have $L$ as a set of primitives, and two data types, $D_0, D_1$, where $D_i$ has, for each production $x \Rightarrow w$ of $G_i$, the closure condition $D_i(x) \to D_i(t_w)$. Then $D_0 \subseteq D_1$ iff $G_0 \subseteq G_1$. Since the inclusion problem for context free grammars is not effectively decidable (see e.g. [HU79]), it follows that neither is the inclusion of data types in a data system. $\dashv$

## References


Acz77 Peter Aczel, *An introduction to inductive definitions*, in Jon Barwise (editor), **Handbook of Mathematical Logic**, North-Holland, Amsterdam, 1977, pp.739–782.

Bar81 Henk Barendregt, **The Lambda Calculus,** North Holland, Amsterdam, 1981, xiv+615pp.

BB85 Corado Böhm & A. Berarducci, *Automatic synthesis of typed $\lambda$-programs on term algebras*, **Theoretical Computer Science 39** (1985).

Car84 Robert Cartwright, *Recursive programs as definitions in first order logic*, **SIAM Journal of Computing 13** (1984) 374–408.

CF58 Haskell B. Curry and R. Feys, **Combinatory Logic,** North-Holland, Amsterdam-New York-Oxford, 1958.

CH82 Ashok Chandra and David Harel, *Structure and complexity of relational queries*, **Journal of Computer and System Sciences 25** (1982) 99–128. Preliminary version in **Twenty first Symposium on Foundations of Computer Science** (1980) 333–347.

CH88 Thierry Coquand & Gerard Huet, *The calculus of constructions*, **Information and Computation 76** (1988) 95–120.

CHS72 Haskell Curry, Roger Hindley and Jonathan Seldin, **Combinatory Logic** (Volume II), North-Holland, Amsterdam, 1972.

Chu33 Alonzo Church, *A set of postulates for the foundations of logic*, **Annals of Mathematics 34** (1933) 839–864.

Col88 Loïc Colson, *About primitive recursive algorithms*, Manuscript, December 1988.

Con86 Robert Constable & als., **Implementing Mathematics with the Nuprl Proof Development System,** Prentice Hall, Englewood Cliffs, 1986.

Coq Thierry Coquand, *Mathematical investigation of a Calculus of Constructions*, This Volume.

deB70 N.G. de Bruijn, *The mathematical language AUTOMATH, its usage and some of its extensions*, **Symposium on Automatic Demonstration**, Springer-Verlag (LNM # 125), Berlin, 1970, 29–61.

FLO83 S. Fortune, D. Leivant, and M. O'Donnell, *The expressiveness of simple and second order type structures*, **Journal of the ACM 30** (1983) 151–185.

Fri81 Harvey Friedman, *On the necessary use of abstract set theory*, **Advnces in Mathematics 41** (1981) 209–280.

Gen34 Gerhard Gentzen, *Unterschungen über das logische Schlissen*, **Mathematische Zeitschrift 39** (1934) 176–210.

Gir72 J.-Y. Girard, **Interpretation fonctionelle et elimination des coupures dans l'arithmetique d'ordre superieur,** Thèse de Doctorat d'Etat, 1972, Université Paris VII.

Gir89 J.-Y. Girard, **Proofs and Types** (with appendices by Paul Taylor and Yves Lafont), Cambridge University Press, Cambridge, 1989.

Göd58 Kurt Gödel, *Ueber eine bisher noch nicht benutzte Erweiterung des finiten standpunktes,* **Dialectica 12** (1958) 280–287.

HU79 J. Hopcroft & J. Ullman, **Introduction to Automata Theory, Languages and Computability,** Addison-Wesley, Reading (Mass.), 1979.

How80 William A. Howard, *The formulae-as-types notion of construction,* pp. 479-490 in [SH80].

Kea70 John Kearns, *Combinatory logic with discriminators,* **Journal of Symbolic Logic 34** (1970) 561–575.

Kle52 S.C. Kleene, **Introduction to Metamathematics,** Noordhoff, Groningen, 1952.

Kle55 S.C. Kleene, *Hierarchies of number theoretic predicates,* **Bulletin of the American Math. Soc. 61** (1955) 193–213.

Kle59 S.C. Kleene, *Recursive functionals and quantifiers of finite type I,* **Transactions of the American Mathematical Society 91** (1959) 1–51.

Kle69 S.C. Kleene, **Formalized Recursive Functions and Formalized Realizability,** Memoirs of the AMS #89 (1969).

KM77 A.S. Kechris and Y.N. Moschovakis, *Recrusion in Higher type,* in J. Barwise (ed.), **Handbook of Mathematical Logic,** North-Holland, Amsterdam, 1977, 681–737.

Kol85 P.G. Kolaitis, *Canonical forms and hierarchies in Generalized Recursion Theory,* in A. Nerode and R.A. Shore (eds.), **Recursion Theory,** Proceedings of Symposia in Pure Mathematics, volume 42, American Mathematical Society, Providence, 1985, 139–170.

KP87 Jean-Louis Krivine and Michel Parigot, *Programming with Proofs,* Manuscript. Presented at the Sixth Symposium on Computation Theory, Wendisch-Rietz, Nopvember 1987.

Kre60 Georg Kreisel, *La Prédicativité,* **Bull. Soc. Math. France 88** (1960) 371–391.

Kri86 Jean-Louis Krivine, *Programmation en arithmétique fonctionelle du second ordre,* Manuscript, 1986.

Kri87 Jean-Louis Krivine, *Un algorithme non typable dans le système F,* **C.R. Acad. Sci. Paris, Série I, 304** (1987) 123–126.

**Lau70** H. Lauchli, *An abstract notion of realizability for which intuitionistic predicate calculus is complete*, in A. Kino, J. Myhill, R.E. Vesley (eds.), **Intuitionism and Proof Theorey**, North-Holland, Amsterdam, 1970.

**Lei73** Daniel Leivant, *Existential instantiation in a system of natural deduction*, Mathematisch Centrum ZW 13-73, 1973, 1–36.

**Lei81** Daniel Leivant, *The complexity of parameter passing in polymorphic procedures*, **Thirteenth Annual Symposium on Theory of Computing**, ACM Press, Providence, 1981, 38-45.

**Lei83** Daniel Leivant, *Reasoning about functional programs and complexity classes associated with type disciplines*, **Twenty-fourth Annual Symposium on Foundations of Computer Science** (1983) 460-469.

**Lei84** Daniel Leivant, *Typing and abstraction in proofs and in programs*, privately circulated, 22pp., January 1984.

**Lei89** Daniel Leivant, *Stratified polymorphism*, Preliminary report, **Fourth Annual Symposium on Logic in Computer Science**, Computer Society Press of the IEEE, Washington, 1989, 39–47. Full paper to appear.

**Lei$\alpha$** Daniel Leivant, *Low abstraction and subrecursion*, in S. Buss and P. Scott (eds.), **Feasible Mathematics** (Proceedings of the June 1989 Workshop at Cornell), to appear.

**Lei$\beta$** Daniel Leivant, *Computationally based set existence principles*, in Wilfried Sieg (ed.), **Logic and Computing**, AMS Advances in Mathematics Series, to appear. (Preliminary version in the Proceedings of the COLOG'88 Conference.)

**Lei$\gamma$** Daniel Leivant, *Discrete polymorphism*, Manuscript, July 1989.

**Mar79** P. Martin-Löf, *Constructive mathematics and computer programming*, **Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science**, North-Holland, Amsterdam, 1979.

**Men87** N.P. Mendler, *Recursive types and type contraints in second-order Lambda Calculus*, **Proceedings, Symposium on Logic in Computer Science**, Computer Society Press of the IEEE, Washington, 1987, 30–36.

**Mos74** Yiannis Moschovakis, **Elementary Induction on Abstract Structures**, North-Holland, Amsterdam, 1974.

**Par88** Michel Parigot, *Programming with proofs: a second order type theory*, in H. Ganziger (ed.), **ESOP '88**, Springer Verlag (LNCS #300), Berlin, 1988, 145–159.

**Par89** Michel Parigot, *Recursive programming with proofs*, Manuscript, 1989.

**Pra65** Dag Prawitz, **Natural Deduction**, Almqvist and Wiksell, Uppsala, 1965.

**Pra71** Dag Prawitz, *Ideas and results of proof theory*, in J.E. Fenstad (ed.), **Proceedings of the Second Scandinavian Logic Symposium**, North-Holland, Amsterdam, 1971, 235-308.

**Rey74** J.C. Reynolds, *Towards a theory of type structures*, in **Programming Symposium (Colloque sur la Programmation Paris)**, Springer-Verlag (LNCS #19), Berlin, 1974, 408–425.

**Sce** Andre Scedrov, *A guide to polymorphic types*, This volume.

**Schw76** Helmut Schwichtenberg, *Definierbare Funktionen im Lambda-Kalkul mit Typen*, **Archiv Logik Grundlagenforsch. 17** (1976) 113–114.

**Schü77** Kurt Schütte, **Proof Theory**, Springer-Verlag (GMW #225), Berlin, 1977.

**Sco63** Dana Scott, *A system of functional abstraction*, Lecture Notes, University of Claifornia at Berkeley, 1962/63.

**SH80** J.P. Seldin and J.R. Hindley (eds.), **To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism**, Academic Press, London, 1980.

**Sta79** Richard Statman, *The type $\lambda$ calculus is not elementary recursive*, **Theoretical Computer Science 9** (1979) 73–81. Preliminary version appeared as **Proceedings of the Eighteenth Conference on Foundations of Computer Science**, Computer Society Press of the IEEE, Washington, 1977, 90–.

**Sta81** Richard Statman, *Number theoretic functions computable by polymorphic programs*, **Twenty Second Annual Symposium on Foundations of Computer Science**, IEEE Computer Society, Los Angeles, 1981, 279–282.

**Tro73** A.S. Troelstra, **Metamathematical Investigation of Intuitionistic Arithmetic and Analysis**, Springer-Verlag (LNM #344), Berlin, 1973.

**Wan54** Hao Wang, *The formalization of mathematics*, **Journal of Symbolic Logic 19** (1954) 241–266.

**Wan62** Hao Wang, *Some formal details on predicative set theories*, Chapter XXIV of **A survey of Mathematical Logic**, Science Press, Peking, 1962. Republished in 1964 by North Holland, Amsterdam. Republished in 1970 under the title **Logic, Computers, and Sets** by Chelsea, New York.

**WLS76** W. Wulf, R.L. London and M. Shaw, *Abstraction and verification in ALPHARD – Introduction to language and methodology*, Tech. Report, Carnegie-Mellon University, 1976.