Theoretical Computer Science 39 (1985) 135-154
North-Holland

AUTOMATIC SYNTHESIS OF TYPED Λ-PROGRAMS ON TERM ALGEBRAS*

Corrado BÖHM and Alessandro BERARDUCCI
Dipartimento di Matematica, Università degli Studi di Roma "La Sapienza", I-00185 Roma, Italy

Communicated by R. Milner
Received May 1984
Revised February 1985

Abstract. The notion of iteratively defined functions from and to heterogeneous term algebras is introduced as the solution of a finite set of equations of a special shape.

Such a notion has remarkable consequences: (1) Choosing the second-order typed lambda-calculus (Λ for short) as a programming language enables one to represent algebra elements and iterative functions by automatic uniform synthesis paradigms, using neither conditional nor recursive constructs. (2) A completeness theorem for Λ-terms with type of degree at most two and a companion corollary for Λ-programs have been proved. (3) A new congruence relation for the last-mentioned Λ-terms which is stronger than Λ-convertibility is introduced and proved to have the meaning of a Λ-program equivalence. Moreover, an extension of the paradigms to the synthesis of functions of higher complexity is considered and exemplified. All the concepts are explained and motivated by examples over integers, list- and tree-structures.

Introduction and summary

Automatic program synthesis will be considered in the following as a formal definition of a compiler translating from an algebraic specification language into a suitable high level language.

The data structures considered belong to the class of heterogeneous term (or absolutely free) algebras and the specifications of the functions to be computed belong to the simplest kind of recursive definitions on such algebras, the so-called 'iterative' definitions, thoroughly identified in the paper.

The high level language used is the second-order or polymorphic typed lambda-calculus (Λ). More precisely, heterogeneous term algebras correspond, by a Completeness Theorem (Theorem 6.3), to a natural restriction of the types of Λ, namely those of degree $\leqslant 2$. Every such type uniquely identifies the corresponding term algebra, together with its constructors or basic functions; moreover, it can be interpreted as an induction principle which may be used to systematically remove recursion from the iterative definitions of functions over the algebraic data structures. The two-way correspondence mentioned above induced a correspondence between

* This research has been supported partly by Grants of the Ministry of Public Instruction and partly by Contract No. 820076097 of the Consiglio Nazionale delle Ricerche, Italy.

elements of an algebra and Λ-terms. The formal definition of the compiler assumes in this paper the appearance of four paradigms the sufficiency of which is warranted again by the completeness Theorem.

It may be useful, at this point, to give the reader a better understanding of the line of research followed in this paper.

A theory of functionality is presently being developed, where traditional ideas of data structures are being reformulated in such a way that a structure is now seen as a functional operator missing some if its arguments (see [1]). For instance, an integer is akin to a for-loop missing its body, a boolean is akin to a conditional missing its pair of branches, etc. Such highly functional programming should be familiar to any programmer of APL (reduce operator) or ML (itlist function) in the context of list processing. But the ideas can be traced directly back to Church, since Church's numerals and booleans are the precursors of this very general view of data structures. What was missing was a proper view of types, since the Curry–Church types were too weak to support this notion. And, thus, the theory of functional of finite types developed by Gödel [6] had to rely on a primitive type 0 for integers. We had to wait until Girard's [4] development of second-order types to have a typed lambda-calculus where the type structure was rich enough to express the notion of a free algebra over a signature, permitting a uniform development of the standard data structures. Second-order types were invented independently by Reynolds [8] in 1974, and various typed lambda-calculi were experimented within De Bruijn's AUTOMATH project, but it was not until recently that the Curry–Howard isomorphism between types and propositions was understood in the computer science community as a fundamental notion. Constable's [2] popularization of Martin-Löf's work was certainly a key influence, along with the works of Fortune, Leivant and O'Donnell [3], Goad [5], and Takasu [9], among others.

More recently, Leivant [7] has done some research on second-order types that is related, but not identical. For example, it appears that he independently formulated the data synthesis paradigm.

In Section 1, a data system is defined as a (heterogeneous) term-algebra satisfying some (very weak) properties, and a data structure is defined as a carrier set of some data system.

In Section 2, the set of the iterative functions on arbitrary data structures is defined as the least set containing the basic operations of the data systems and closed under both explicit definitions and what we have called 'iterative definitions'.

It turns out that every primitive recursive function on natural numbers is iterative according to our definition.

By the way, we remark that the proposed definition is machine-independent and does not make use of any coding into natural numbers.

In Sections 3, 4, and 5 we have several 'synthesis paradigms' to represent any data structure and any iterative function inside second-order typed Λ-calculus (Λ).

It is important to underline that our approach is completely uniform and automatic.

Namely, considering Λ as a programming language for total functions, a computer may easily be programmed to translate specifications of data structures and iterative functions into Λ-terms representing the data and functions respectively.

We notice that the compiled Λ-programs for iteratively defined functions contain neither recursive nor conditional constructs. This is possible because, as already expressed in this Introduction, the computation of a program on a given input is driven by the input itself (which behaves like a functional).

In Section 6, a Completeness Theorem is proved according to which, given a closed Λ-term t, a (necessary and) sufficient condition for t to represent an element belonging to some data structure D is that the type of t represents D in Λ. As a consequence of the Completeness Theorem we develop, in Section 7, a method for proving Λ-program equivalence without induction. This method often enables one to eliminate basic operations from Λ-programs.

In Section 8 we illustrate by an example how to extend the 'Λ-program synthesis paradigm' to functions of higher complexity.

Section 9 proposes further data structures by merely specifying their Λ-types.

1. Heterogeneous algebras

1.1. Basic definitions

An algebra is a pair $A=\langle \mathscr{S}, \mathscr{G}\rangle$ in which

(1) $\mathscr{S}=\{S_\kappa\}_{\kappa\in K}$ is a family of nonempty sets each called a *carrier set* of the algebra A,

(2) $\mathscr{G}=\{g_\alpha\}_{\alpha\in\Omega}$ is a set of finitary operations (called *basic operations*), where each g_α is a mapping

$$g_\alpha: S_{k(1,\alpha)}\times\cdots\times S_{k(n(\alpha),\alpha)}\to S_{r(\alpha)} \quad (n(\alpha)\in N, k(1,\alpha),\ldots,k(n(\alpha),\alpha), r(\alpha)\in K).$$

Here, $n(\alpha)$ is the *arity* of g_α and $k(\kappa,\alpha)$ is the *index* of the carrier set of the κth argument of g_α.

If $n(\alpha)=0$ (i.e., if g_α is a nullary operation), we identify g_α with a selected element of $S_{r(\alpha)}$.

An algebra with only one carrier is called a '*homogeneous algebra*', an algebra with more than one carrier is called a '*heterogeneous algebra*'.

1.2. Term algebras

A term algebra (or absolutely free algebra) in an algebra $\langle \mathscr{S}, \mathscr{G}\rangle$ such that, whenever $g_\alpha\in\mathscr{G}$, $g_\beta\in\mathscr{G}$, and $g_\alpha(x_1,\ldots,x_{n(\alpha)})=g_\beta(y_1,\ldots,y_{n(\beta)})$, we have $\alpha=\beta$ and $x_\kappa=y_\kappa$ for all $\kappa=1,2,\ldots,n(\alpha)=n(\beta)$.

Hence, in a term algebra equality means formal identity.

Given a carrier $S\in\mathscr{S}$ and an element $x\in S$ we say that x is a generator iff it is is not in the codomain of any basic operation $g\in\mathscr{G}$; in particular, a generator is not a nullary basic operation.

There are algebras with no generators at all; for example, the natural numbers with successor and zero as basic operations (see example (a) of Section 1.4). Different is the case of the algebra of lists considered in example (b) of Section 1.4; here, the elements $a_1, \ldots, a_n$ of the list

$$\text{cons}(a_1, \ldots, \text{cons}(a_n, \text{nil}) \ldots)$$

are generators of the algebra and not basic operations.

Since, in our treatment, the generators play the role of arbitrary objects, we shall often call them 'parameters'.

The presence of the parameters is quite irrelevant in the theory and in the proofs we shall give, so that it is possible to ignore them at first reading; nonetheless, we have included them in our discussion for their relevance in providing examples of familiar data structures and algorithms which do not depend on the particular nature of the parameters.

For the notion of 'iterative definition' (see Section 2.1) to make sense it will be essential that the carriers in which the parameters range are kept distinguished from the other carriers.

This motivates the following definition of data system (we owe the name 'data system' to Leivant [7]).

1.3. Data systems and data structures

From now on we shall only be interested in term algebras such that there is no carrier which contains both parameteric and non parametric elements and in which, moreover, there are only finitely many carriers and basic operations. Such an algebra will be called a *data system* and its carriers will be called data structures. A data structure will be called *parametric* if its elements are parametric, and *proper* otherwise.

Given a data system $D = \langle \mathscr{S}, \mathscr{G} \rangle$ we shall often denote by $\{A_\eta\}_{\eta \in J}$ the family of its parametric data structures and by $\{P_\iota\}_{\iota \in I}$ the family of its proper data structures.

So, we write $D = \langle \mathscr{S}, \mathscr{G} \rangle = \langle \{A_\eta\}\eta_{\in J} \cup \{P_\iota\}_{\iota \in I}\{g_\alpha\}_{\alpha \in \Omega} \rangle$ and we assume that each basic operation g_α is a mapping

$$g_\alpha : A_{j(1,\alpha)} \times \cdots \times A_{j(p(\alpha),\alpha)} \times P_{i(1,\alpha)} \times \cdots \times P_{i(m(\alpha),\alpha)} \to P_{r(\alpha)}$$

where $\alpha \in \Omega$, $p(\alpha)$, $m(\alpha) \in N$, $j(1, \alpha), \ldots, j(p(\alpha), \alpha) \in J$, $i(1, \alpha), \ldots, i(m(\alpha), \alpha)$, $r(\alpha) \in I$.

For simplicity, we have placed all the parametric data structures A_η to the left of the nonparametric ones; of course, this restriction is inessential. If there are no parameters, i.e., if $p(\alpha) = 0$, then all the A_η's disappear.

Remarks 1.1. Below we list some consequences of our definitions:

(a) A data structure $S \in \mathscr{S}$ is parametric iff it is not the codomain of any basic operation $g_\alpha \in \mathscr{G}$.

(b) A data system is completely determined up to isomorphism by the cardinality of its parametric data structures and by its 'language', that is, by the symbols by which we denote its data structures, its functions, and the respective domains and codomains of its functions.

(c) If in a data system $D = \langle \mathcal{S}, \mathcal{G} \rangle$ there are no parameters, i.e., if all the data structures $S \in \mathcal{S}$ are proper, then there must be some nullary operation $g \in \mathcal{G}$.

For example, there is no data system with language $\langle \{B\}, \{g: B \to B\} \rangle$.
Here, we implicitly assume that infinitely long terms such as $g(g(g(\ldots$ are not allowed; therefore, it is possible to make definitions by induction on the complexity of the elements of a data structure.

(d) Both the arguments and the values of a basic operation are elements of some data structure and not functions themself; so, a function $g: (B \to B) \to B$ with domain $B \to B$ is not allowed as a basic operation. This restriction will be crucial in the proof of the Completeness Theorem (Theorem 6.3).

1.4. Examples: natural numbers, lists, trees and forests

By Remark 1.1(c) we can give examples of data systems by just exhibiting their respective languages (we are not interested in the cardinality of the parameters).

(a) *Natural numbers*:

$$D = \langle \{N\}, \{s: N \to N, o: N\} \rangle.$$

(b) *Lists*:

$$D = \langle \{A, L\}, \{\text{cons}: A \times L \to L, \text{nil}: L\} \rangle,$$

where A (a set of atoms) is a parametric data structure (since it is not the codomain of 'cons', neither the codomain of 'nil'), and L (a list of atoms) is a proper data structure.

(c) *Trees and forests*:

$$D = \langle \{A, T, F\}, \{\text{span}: A \times F \to T, \text{join}: T \times F \to F, \text{empty}: F\} \rangle,$$

where A (a set of atoms) is a parametric data structure, and T (finite nonempty trees with nodes labelled by atoms) and F (forests) are proper data structures.

The intended meaning is

$$\begin{array}{c} a \\ / \quad \backslash \\ t_1 \cdots t_n \end{array} = \text{span}(a, \text{join}(t_1, \ldots, \text{join}(t_n, \text{empty}) \ldots)) \quad (n \geq 0).$$

So, we see that trees (which are obtained 'spanning' atoms to forests) and forests (i.e., finite sequences of trees) are defined in terms of each other.

In the following we shall sometimes write for short s, o, c, n, sp, j, e instead of s, o, cons, nil, span, join, empty, respectively.

2. Functions on data structures

Since a data system D is a term algebra, we can define functions on its data structures by some set of recursive equations and prove termination by induction on the length of the inputs (since a term is a finite object).

Thus, starting from the basic operations we can define new (total) functions in terms of already defined ones.

However, since we deal in general with heterogeneous algebras, it may happen that we cannot define only one function at a time; we have rather to define simultaneously a whole family of functions $\{f_\iota\}_{\iota\in I}$ (one for each proper data structure of the system) in terms of a family $\{h_\alpha\}_{\alpha\in\Omega}$ of already given functions (one for each basic operation of the data system).

The following definition should now be clear.

2.1. *Iterative definitions*

Given a data system

$$D=\langle\{A_\eta\}_{\eta\in J}\cup\{P_\iota\}_{\iota\in I},\{g_\alpha\}_{\alpha\in\Omega}\rangle$$

and a family of proper data structures $\{Q_\iota\}_{\iota\in I}$ (possibly belonging to some other data systems) we say that a family $\{f_\iota\}_{\iota\in I}$ of unary functions $f_\iota: P_\iota\to Q_\iota$ is iteratively defined in terms of a family $\{h_\alpha\}_{\alpha\in\Omega}$ of functions

$$h_\alpha: A_{j(1,\alpha)}\times\cdots\times A_{j(p(\alpha),\alpha)}\times Q_{i(1,\alpha)}\times\cdots\times Q_{i(m(\alpha),\alpha)}\to Q_{r(\alpha)}$$

(the indexes are as in Section 1.3) if for each $\iota\in I$ and for each element $g_\alpha(a_1,\ldots,a_{p(\alpha)},x_1,\ldots,x_{m(\alpha)})\in P_\iota$ we have

$$f_\iota(g_\alpha(a_1,\ldots,a_{p(\alpha)},x_1,\ldots,x_{m(\alpha)}))$$
$$=h_\alpha(a_1,\ldots,a_{p(\alpha)},f_{i(1,\alpha)}(x_1),\ldots,f_{i(m(\alpha),\alpha)}(x_{m(\alpha)})). \tag{1}$$

Explanation

Since P_ι is a proper data structure, for each $x\in P_\iota$ there exist $\alpha\in\Omega$, $a_1\in A_{j(1,\alpha)},\ldots,a_{p(\alpha)}\in A_{j(p(\alpha),\alpha)}$, $x_1\in P_{i(1,\alpha)},\ldots,x_{m(\alpha)}\in P_{i(m(\alpha),\alpha)}$ such that $x=g_\alpha(a_1,\ldots,a_{p(\alpha)},x_1,\ldots,x_{m(\alpha)})$.

Thus, for every $x\in P_\iota$ we can compute the value of $f_\iota(x)$ using the equation above and the values of the functions $f_{i(1,\alpha)},\ldots,f_{i(m(\alpha),\alpha)}$ at arguments of less complexity.

If $m(\alpha)=0$, we have the induction basis, that is, $f_\iota(g_\alpha(a_1,\ldots,a_{p(\alpha)}))=h_\alpha(a_1,\ldots,a_{p(\alpha)})$; in particular, if there are no parameters, $f_\iota(g_\alpha)=h_\alpha$.

Iteration versus recursion

It is important to notice that $x_1,\ldots,x_{m(\alpha)}$ do not appear as arguments of h_α in equation (1) above. This is the reason for the name 'iterative definitions' instead of 'primitive recursive definitions'.

Of course, an approach based on recursion rather than iteration is also possible, but the advantage of an iterative definition of a function is that it can be very simply translated into a Λ-term representing that function (see Section 5.2) once the data structures have been suitably represented (generalizing the representation of the natural numbers due to Church).

The well-known fact that primitive recursion on natural numbers can be reduced to iteration by means of a pairing function can easily be generalized to arbitrary algebras. Hence, we do not lose expressive power by restricting ourselves to iteration.

In particular, the predecessor function and, more generally, the inverse functions of the basic operations of any algebra (which can obviously be defined by recursion) can also be defined by iteration.

Remark 2.1 (*the identity function iteratively defined*). Let us notice that, if for all $\alpha \in \Omega$ we set (in equation (1)) $h_\alpha = g_\alpha$, then each of the iteratively defined functions f_ι turns out to be the identity function on P_ι.

Nonunary functions

Iterative definitions are extended to nonunary functions f replacing equation (1) by

$$f_\iota(\vec{y}, g_\alpha(a_1, \ldots, a_{p(\alpha)}, x_1, \ldots, x_{m(\alpha)})) = h_\alpha(\vec{y}, a_1, \ldots, a_{p(\alpha)}, f_{i(1,\alpha)}(\vec{y}, x_1), \ldots, f_{i(m(\alpha),\alpha)}(\vec{y}, x_{m(\alpha)})), \tag{2}$$

where $\vec{y}$ is a sequence of variables ranging over a sequence of data structures (the role of $\vec{y}$ is not to be confused with the role of the parameters $a_1, \ldots, a_{p(\alpha)}$).

2.2. *Examples*: *some iterative schemes*

In the following examples the functions f_ι are iteratively defined in terms of the functions h_α (the codomains are left unspecified).

(a) Take $D = \langle\{N\}, \{s: N \to N, o: N\}\rangle$ and define $f_1(s(x)) = h_1(f_1(x))$, $f_1(o) = h_2$ (the usual scheme of iteration on natural numbers).

(b) Take $D = \langle\{A, L\}, \{\text{cons}: A \times L \to L, \text{nil}: L\}\rangle$ and define $f_1(\text{cons}(a, y)) = h_1(a, f_1(y))$, $f_1(\text{nil}) = h_2$ (this scheme is also known as 'tail recursion')

(c) Take $D = \langle\{A, T, F\}, \{\text{span}: A \times F \to T, \text{join}: T \times F \to F, \text{empty}: F\}\rangle$ and define $f_1(\text{span}(a, y)) = h_1(a, f_2(y))$, $f_2(\text{join}(x, y)) = h_2(f_1(x), f_2(y))$, $f_2(\text{empty}) = h_3$.

2.3. *Iterative functions*

The following clauses inductively define what it means for a function $f: S_1 \times \cdots \times S_m \to S$ (with $S_1, \ldots, S_m$, S data structures) to be 'iterative'.

(a) A basic operation g (of some data system D) is an iterative function.

(b) Constant functions and projection functions $U_i^n(x_1, \ldots, x_n) = x_i$ are iterative functions. Iterative functions are closed under composition, that is, if $f(x_1, \ldots, x_n) = h(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ and h, $h_1, \ldots, h_m$ are iterative, then f is iterative.

(c) Iterative functions are closed under iterative definitions. We can consider iterative functions as a generalization of the primitive recursive functions (see Section 2.1) to arbitrary data structures.

2.4. *Examples*: *list concatenation and preorder traversal of a tree*

In the following examples we refer to the data systems defined in Section 1.4.

(a) We define the iterative function $\mathrm{cat}: L \times L \to L$ (list concatenation; usually called append) as follows: first define $f: L \times L \to L$ iteratively by $f(y, \mathrm{cons}(z, w)) = \mathrm{cons}(z, f(y, w))$, $f(y, \mathrm{nil}) = y$. Then define $\mathrm{cat}(x, y) = f(y, x)$.

(b) We give an iterative definition of the function preorder: $T \to L$ which, when applied to a tree $x \in T$, gives the list preorder (x) consisting of all the nodes of the tree x ordered according to a 'preorder traversal'. First define $f_1: T \to L$, $f_2: F \to L$ iteratively by $f_1(\mathrm{span}(a, y)) = \mathrm{cons}(a, f_2(y))$, $f_2(\mathrm{join}(x, y)) = \mathrm{cat}(f_1(x), f_2(y))$, $f_2(\mathrm{empty}) = \mathrm{nil}$. Then define $\mathrm{preorder}\,(x) = f_1(x)$.

3. Second-order Λ-calculus

Our purpose is now to represent any data structure and any iterative function in second-order Λ-calculus (Λ for short). We refer to [8, 3] for a description of Λ.

It is important to underline that our approach is completely uniform and automatic: we can represent in Λ whatever data structure and, most importantly, given an iterative function f, we can define by a mere syntactical inspection of the definition of f, a typed Λ-term $\underline{f}$ which represents f in Λ.

Of course, the iterative functions form a rather small subset of all the functions representable in Λ; in Section 8 we shall consider the possibility of extending our automatic method to functions of higher complexity.

3.1. *Names*

In order to carry out our program, we first make an inessential enlargement of the language of Λ by adding, for each parametric data structure A, a 'parametric' Λ-type $\boldsymbol{A}$ called the 'name' of A and, for each parametric element $a \in A$, a 'parametric' Λ-term $\boldsymbol{a}$, of Λ-type $\boldsymbol{A}$, called the name of a.

Parametric Λ-types $\boldsymbol{A}$ and parametric Λ-terms $\boldsymbol{a}$ must be considered as closed Λ-types and closed Λ-terms respectively, and it is not allowed to abstract with respect to them. We will instead abstract with respect to variables of parametric types.

Note that a parametric Λ-term $\boldsymbol{a}$ always behaves as an argument and never as a function (i.e., it cannot be applied to any Λ-term). For notational convenience, before defining Λ-representations, we first give 'names' also to proper data structures P and proper elements $x \in P$.

(a) For each proper data structure P we choose a type variable $\boldsymbol{P}$ (in Λ) called the name of P.

(b) For each basic operation

$$g_\alpha : A_{j(1,\alpha)} \times \cdots \times A_{j(p(\alpha),\alpha)} \times P_{i(1,\alpha)} \times \cdots \times P_{i(m(\alpha),\alpha)} \to P_{r(\alpha)}$$

(of some data system) we choose a Λ-variable $\boldsymbol{g}_\alpha$ of Λ-type

$$\theta_\alpha \equiv \boldsymbol{A}_{j(1,\alpha)} \to \cdots \to \boldsymbol{A}_{j(p(\alpha),\alpha)} \to \boldsymbol{P}_{i(1,\alpha)} \to \cdots \to \boldsymbol{P}_{i(m(\alpha),\alpha)} \to \boldsymbol{P}_{r(\alpha)}$$

called the name of g_α.

The name of a proper element $x \in P$ is inductively defined as follows:

(c) If $x = g_\alpha(a_1, \ldots, a_p, x_1, \ldots, x_{m(\alpha)})$, the name $\boldsymbol{x}$ of x is the open Λ-term $(\boldsymbol{g}_\alpha \boldsymbol{a}_1 \cdots \boldsymbol{a}_{p(\alpha)} \boldsymbol{x}_1 \cdots \boldsymbol{x}_{m(\alpha)})$. It can be checked that $\boldsymbol{x}$ has type $\boldsymbol{P}$. For example, the name of the natural number $s(s(o))$ is the open Λ-term $\boldsymbol{s}(\boldsymbol{s}\boldsymbol{o})$ where $\boldsymbol{s}$, $\boldsymbol{o}$ are Λ-variables of Λ-type $\boldsymbol{N} \to \boldsymbol{N}$ and $\boldsymbol{N}$ respectively.

4. Representation of data structures in Λ

Once we have defined names, we can define representations. More precisely, given a data system

$$D = \langle \mathcal{S}, \mathcal{G} \rangle = \langle \{A_1, \ldots, A_l, P_1, \ldots, P_n\}, \{g_1, \ldots, g_k\} \rangle,$$

we shall define for each data structure $S \in \mathcal{S}$ a closed Λ-type $\underline{S}$, called the representation of S, and, for each element $x \in S$, a closed Λ-term $\underline{x}$ of Λ-type $\underline{S}$, called the representation of x. We shall say that $\underline{S}$ is defined by the type synthesis paradigm, and $\boldsymbol{x}$ by the data synthesis paradigm. We shall now explain how to define these representations.

Definition 4.1 (*type synthesis paradigm*). The representation of a parametric data structure A_j is its own name, that is $\underline{A}_j \equiv \boldsymbol{A}_j$. The representation of a proper data structure P_i is given by

$$\underline{P}_i \equiv \Delta \boldsymbol{P}_1 \cdots \Delta \boldsymbol{P}_n (\theta_1 \to \cdots \to \theta_k \to \boldsymbol{P}_i),$$

where $\theta_1, \ldots, \theta_k$ are the types of the Λ-variables $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_k$ as defined in (b) of Section 3.1.

Examples 4.2. (a) If $D = \langle \{N\}, \{s: N \to N, o: N\} \rangle$, then $\underline{N} \equiv \Delta \boldsymbol{N}((\boldsymbol{N} \to \boldsymbol{N}) \to \boldsymbol{N} \to \boldsymbol{N})$.

(b) If $D = \langle \{A, L\}, \{\text{cons}: A \times L \to L, \text{nil}: L\} \rangle$, then

$$\underline{\boldsymbol{A}} \equiv \boldsymbol{A}, \qquad \underline{\boldsymbol{L}} \equiv \Delta \boldsymbol{L}((\boldsymbol{A} \to \boldsymbol{L} \to \boldsymbol{L}) \to \boldsymbol{L} \to \boldsymbol{L}).$$

(c) $D = \langle \{A, T, F\}, \{\text{span}: \{A \times F \to T, \text{join}: T \times F \to F, \text{empty}: F\} \rangle$, then

$$\underline{\boldsymbol{A}} \equiv \boldsymbol{A}, \qquad \underline{\boldsymbol{T}} \equiv \Delta \boldsymbol{T} \Delta \boldsymbol{F}((\boldsymbol{A} \to \boldsymbol{F} \to \boldsymbol{T}) \to (\boldsymbol{T} \to \boldsymbol{F} \to \boldsymbol{F}) \to \boldsymbol{F} \to \boldsymbol{T}),$$

$$\underline{\boldsymbol{F}} \equiv \Delta \boldsymbol{T} \Delta \boldsymbol{F}((\boldsymbol{A} \to \boldsymbol{F} \to \boldsymbol{T}) \to (\boldsymbol{T} \to \boldsymbol{F} \to \boldsymbol{T}) \to \boldsymbol{F} \to \boldsymbol{F}).$$

Definition 4.3 (*data synthesis paradigm*). The representation of a parametric data element $a \in A$ is its own name, that is, $\underline{a} \equiv \boldsymbol{a}$. The representation of a nonparametric element of a nonparametric element $x \in O_i$ is given by

$$\underline{x} \equiv \Lambda \boldsymbol{P}_1 \cdots \Lambda \boldsymbol{P}_n \lambda \boldsymbol{g}_1 \cdots \lambda \boldsymbol{g}_k . \boldsymbol{x},$$

where $\boldsymbol{x}$ is inductively defined in (c) of Section 3.1.

On other words, the Λ-representation $\underline{x}$ on an element x of a data structure is the abstraction closure of its 'name' $\boldsymbol{x}$. It follows from the definition that $\underline{x}\boldsymbol{P}_1 \cdots \boldsymbol{P}_m \boldsymbol{g}_1 \cdots \boldsymbol{g}_k =_\Lambda \boldsymbol{x}$, where "$=_\Lambda$" means Λ-convertibility. It can easily be checked that if x belongs to a data structure S, then $\underline{x}$ has Λ-type $\underline{S}$.

Examples 4.4. With reference to Section 1.4 we have, for example:

(a) $\underline{s(s(o))}\boldsymbol{N}\boldsymbol{s}^{N\to N}\boldsymbol{o}^N =_\Lambda \boldsymbol{s}(\boldsymbol{so})$

(that is, $\underline{s(s(o))} \equiv \Lambda \boldsymbol{N} \lambda \boldsymbol{s}^{N\to N} \lambda \boldsymbol{o}^N . \boldsymbol{s}(\boldsymbol{so})$). Here $\boldsymbol{s}$ and $\boldsymbol{o}$ are variables of Λ not to be confused with the successor and the zero functions.

(b) $\underline{c(a_1, c(a_2, n))}\boldsymbol{L}\boldsymbol{c}^{A\to L\to L}\boldsymbol{n}^L =_\Lambda \boldsymbol{c}\boldsymbol{a}_1(\boldsymbol{c}\boldsymbol{a}_2\boldsymbol{n})$.

(c) $\underline{\mathrm{sp}(a_1, j(\mathrm{sp}(a_2, e), e))}\boldsymbol{T}\boldsymbol{F}\mathbf{sp}^{A\to F\to T}\boldsymbol{j}^{T\to F\to F}\boldsymbol{e}^F =_\Lambda \mathbf{sp}\ \boldsymbol{a}_1(\boldsymbol{j}(\mathbf{sp}\ \boldsymbol{a}_2\boldsymbol{e})\boldsymbol{e})$.

5. Representation of functions in Λ

Once we have represented data, we can represent functions.

Definition 5.1 (*representability of functions in* Λ). Given a function $f: S_1 \times \cdots \times S_m \to S$ (with $S_1, \ldots, S_m$, S data structures) we say that f is Λ-representable if there exists a Λ-term $\underline{f}$ of Λ-type $\underline{S}_1 \to \cdots \to \underline{S}_m \to \underline{S}$ such that, for all $x_1 \in S_1, \ldots, x_m \in S_m$,

$$\underline{f}\,\underline{x}_1 \cdots \underline{x}_m = \underline{f(x_1, \ldots, x_m)},$$

where all the underlined Λ-terms, except $\underline{f}$, are defined by the data synthesis paradigm.

For a given Λ-representable function f there are, in general, infinitely many nonconvertible Λ-terms which represent f; however, we shall see that not only is each iterative function f Λ-representable, but, once we are given a definition of f, we can effectively find one standard representation $\underline{f}$ of f.

Let us first consider the case of the basic operations.

5.1. Representation of the basic operations

Let

$$D = \langle \mathcal{S}, \mathcal{G} \rangle = \langle \{A_1, \ldots, A_b, P_1, \ldots, P_n\}, \{g_1, \ldots, g_k\} \rangle$$

be a data system and let

$$g_\alpha : A_{j(1,\alpha)} \times \cdots \times A_{j(p(\alpha),\alpha)} \times P_{i(1,\alpha)} \times \cdots \times P_{i(m(\alpha),\alpha)} \to P_{r(\alpha)} \in \mathcal{G}$$

be a basic operation ($\alpha \in \{1, \ldots, k\}$).

Define a closed Λ-term $\underline{g_\alpha}$ of Λ-type

$$\underline{A_{j(1,\alpha)}} \to \cdots \to \underline{A_{j(p(\alpha),\alpha)}} \to \underline{P_{i(1,\alpha)}} \to \cdots \to \underline{P_{i(m(\alpha),\alpha)}} \to \underline{P_{r(\alpha)}}$$

as follows.

Definition 5.2 (*basic operation synthesis paradigm*)

$$(\underline{g_\alpha} u_1 \cdots u_{p(\alpha)} v_1 \cdots v_{m(\alpha)}) \boldsymbol{P}_1 \cdots \boldsymbol{P}_n \boldsymbol{g}_1 \cdots \boldsymbol{g}_k$$
$$=_\Lambda \boldsymbol{g}_\alpha u_1 \cdots u_{p(\alpha)} (v_1 \boldsymbol{P}_1 \cdots \boldsymbol{P}_n \boldsymbol{g}_1 \cdots \boldsymbol{g}_k) \cdots (v_{m(\alpha)} \boldsymbol{P}_1 \cdots \boldsymbol{P}_n \boldsymbol{g}_1 \cdots \boldsymbol{g}_k),$$

where $u_1, \ldots, u_{p(\alpha)}, v_1, \ldots, v_{m(\alpha)}$ are variables of type

$$\underline{A_{j(1,\alpha)}}, \ldots, \underline{A_{j(p(\alpha),\alpha)}}, \underline{P_{i(1,\alpha)}}, \ldots, \underline{P_{i(m(\alpha),\alpha)}}$$

respectively.

Proposition 5.3. $\underline{g_\alpha}$ *represents* g_α *in* Λ, *that is, for every* $a_1, \ldots, a_p, x_1, \ldots, x_m$ *in the domain of* g_α,

$$\underline{g_\alpha}\ \underline{a_1} \cdots \underline{a_p}\ \underline{x_1} \cdots \underline{x_m} =_\Lambda \underline{g_\alpha(a_1, \ldots, a_p, x_1, \ldots, x_m)}.$$

(*Here,* p *is* $p(\alpha)$ *and* m *is* $m(\alpha)$.)

Proof

$$\underline{g_\alpha}\ \underline{a_1} \cdots \underline{a_p}\ \underline{x_1} \cdots \underline{x_m}$$

by Definition 5.2

$$=_\Lambda \Lambda \boldsymbol{P}_1 \cdots \Lambda \boldsymbol{P}_n \lambda \boldsymbol{P}_n \lambda \boldsymbol{g}_1 \cdots \boldsymbol{g}_k . \boldsymbol{g}_\alpha \underline{a_1} \cdots \underline{a_p} (\underline{x_1} \boldsymbol{P}_1 \cdots \boldsymbol{P}_n \boldsymbol{g}_1 \cdots \boldsymbol{g}_k)$$
$$\cdots (\underline{x_m} \boldsymbol{P}_1 \cdots \boldsymbol{P}_n \boldsymbol{g}_1 \cdots \boldsymbol{g}_k)$$

by Definition 4.3

$$=_\Lambda \Lambda \boldsymbol{P}_1 \cdots \Lambda \boldsymbol{P}_n \lambda \boldsymbol{g}_1 \cdots \lambda \boldsymbol{g}_k . \boldsymbol{g}_\alpha \boldsymbol{a}_1 \cdots \boldsymbol{a}_p \boldsymbol{x}_1 \cdots \boldsymbol{x}_m$$

by Definition 4.3 again

$$\equiv \underline{g_\alpha(a_1, \ldots, a_p, x_1, \ldots, x_m)}.$$

Examples 5.4. With reference to Section 1.4 and Definition 5.2 we have, for example,

$$(\underline{s} v^{\underline{N}}) \boldsymbol{N} \boldsymbol{s}^{\boldsymbol{N} \to \boldsymbol{N}} \boldsymbol{o}^{\boldsymbol{N}} =_\Lambda s(v \boldsymbol{N} \boldsymbol{s} \boldsymbol{o}),$$

$$\underline{o} \boldsymbol{N} \boldsymbol{s}^{\boldsymbol{N} \to \boldsymbol{N}} \boldsymbol{o}^{\boldsymbol{N}} =_\Lambda \boldsymbol{o},$$

$$(\underline{c} u^{\underline{A}} v^{\underline{L}}) \boldsymbol{L} \boldsymbol{c}^{\boldsymbol{A} \to \boldsymbol{L} \to \boldsymbol{L}} \boldsymbol{n}^{\boldsymbol{L}} =_\Lambda \boldsymbol{c} u (v \boldsymbol{L} \boldsymbol{c} \boldsymbol{n}),$$

$$\underline{n} \boldsymbol{L} \boldsymbol{c}^{\boldsymbol{A} \to \boldsymbol{L} \to \boldsymbol{L}} \boldsymbol{n}^{\boldsymbol{L}} =_\Lambda \boldsymbol{n},$$

$$(\mathrm{sp}\ u^{\underline{A}} v^{\underline{F}}) \boldsymbol{TF} \mathbf{sp}^{\boldsymbol{A} \to \boldsymbol{F} \to \boldsymbol{T}} \boldsymbol{j}^{\boldsymbol{T} \to \boldsymbol{F} \to \boldsymbol{F}} \boldsymbol{e}^{\boldsymbol{F}} =_\Lambda \mathbf{sp}\ u (v \boldsymbol{TF} \mathbf{sp}\ \boldsymbol{je}).$$

5.2. Representation of iterative functions in Λ

Let

$$D = \langle \{A_1, \ldots, A_l, P_1, \ldots, P_n\}, \{g_1, \ldots, g_k\} \rangle$$

be a data system.

Suppose that a family $\{f_\iota | \iota = 1, \ldots, n\}$ of unary functions $f_\iota : P_\iota \to Q_\iota$ is iteratively defined in terms of a family of already given functions $\{h_\alpha | \alpha = 1, \ldots, k\}$ (as in equation (1)) and suppose that each h_α is Λ-representable by a Λ-term $\underline{h}_\alpha$, say.

Define $\underline{f}_\iota$ as follows.

Definition 5.5 (*program synthesis paradigm*). For each $\iota \in \{1, \ldots, n\}$ define a Λ-term $\underline{f}_\iota$ of Λ-type $\underline{P}_\iota \to \underline{Q}_\iota$ by

(a) $\underline{f}_\iota \equiv \lambda v^{P_\iota}.v\underline{Q}_1 \cdots \underline{Q}_n \underline{h}_1 \cdots \underline{h}_k$.

Similarly in case of nonunary functions (see equation (2)) define $\underline{f}_\iota$ by

(b) $\underline{f}_\iota \vec{u} =_\Lambda \lambda v^{P_\iota}.v\underline{Q}_1 \cdots \underline{Q}_n(\underline{h}_1 \vec{u}) \cdots (\underline{h}_k \vec{u})$,

where $\vec{u}$ is a sequence of Λ-variables.

We claim that $\underline{f}_\iota$ represents f_ι in Λ. Let us prove it only in case of unary functions $f_\iota : p_\iota \to Q_\iota$.

Theorem 5.6. *For each $x \in P$,*

$$\underline{f}_\iota \underline{x} =_\Lambda \underline{f_\iota(x)}$$

where $\underline{x}$ and $\underline{f_\iota(x)}$ are given by the data synthesis paradigm (Definition 4.3) and $\underline{f}_\iota$ is given by Definition 5.5(a).

Proof. Let $x \in P_\iota$. The proof follows by structural induction on x.

Since P_ι is a proper data structure, there exists a basic operation $g_\alpha : A_{j(1,\alpha)} \times \cdots \times A_{j(m(\alpha),\alpha)} \times P_{i(1,\alpha)} \times \cdots \times P_{i(m(\alpha),\alpha)} \to P_{r(\alpha)}$ such that, for some $a_1, \ldots, a_p, x_1, \ldots, x_m$ in the domain of g_α, $x = g_\alpha(a_1, \ldots, a_p, x_1, \ldots, x_m)$. (Here, p is $p(\alpha)$ and m is $m(\alpha)$.)

Therefore, in Λ, by Proposition 5.3,

$$\underline{f}_\iota \underline{x} = \underline{f}_\iota(\underline{g}_\alpha \underline{a}_1 \cdots \underline{a}_p \underline{x}_1 \cdots \underline{x}_m) =$$

by Definition 5.5(a)

$$= \underline{g}_\alpha \underline{a}_1 \cdots \underline{a}_p \underline{x}_1 \cdots \underline{x}_m \underline{Q}_1 \cdots \underline{Q}_n \underline{h}_1 \cdots \underline{h}_1 \cdots \underline{h}_k =$$

by Definition 5.2

$$= \underline{h}_\alpha \underline{a}_1 \cdots \underline{a}_p(\underline{x}_1 \underline{Q}_1 \cdots \underline{Q}_n \underline{h}_1 \cdots h_k) \cdots (\underline{x}_m \underline{Q}_1 \cdots \underline{Q}_n \underline{h}_1 \cdots \underline{h}_k) =$$

by Definition 5.5(a)

$$= \underline{h}_\alpha \underline{a}_1 \cdots \underline{a}_p(\underline{f}_{i(1,\alpha)} \underline{x}_1) \cdots (\underline{f}_{i(m,\alpha)} \underline{x}_m) =$$

by induction hypothesis and since $\underline{h}_\alpha$ represents h_α in Λ

$$= \underline{h_\alpha(a_1, \ldots, a_p, f_{i(1,\alpha)}(x_1), \ldots, f_{i(m,\alpha)}(x_m))} \equiv$$

by equation (1)

$$\equiv \underline{f_\iota(g_\alpha(g_1, \ldots, a_p, x_1, \ldots, x_m))}$$

$$\equiv \underline{f_\iota(x)}.$$

For $m = 0$ we have the induction basis.

It can easily be checked that all the expressions above are well typed. □

Notice that the Λ-program $\underline{f}_\iota$ $(\iota = 1, \ldots, n)$ defined by the program synthesis paradigm (Definition 5.5) depends on ι only in the type of its argument variable.

Corollary 5.7 (fixed point equation). *Let* $D = \langle \{A_1, \ldots, A_l, P_1, \ldots, P_n, \{g_1, \ldots, g_k\}\rangle$ *be as above. Then, for all* $x \in P_\iota$,

$$\underline{x}\,\underline{P_1} \cdots \underline{P_n}\,\underline{g_1} \cdots \underline{g_k} =_\Lambda \underline{x}.$$

(Do not confuse the last expression with $\underline{x}\,\boldsymbol{P}_1 \cdots \boldsymbol{P}_n \boldsymbol{g}_1 \cdots \boldsymbol{g}_1 \cdots \boldsymbol{g}_k =_\Lambda \boldsymbol{x}$ which is nothing but the definition of $\underline{x}$ (Definition 4.3).)

Proof. Let us put, in Section 5.2, $h_1 = g_1, \ldots, h_k = g_k$.

Then, by Remark 2.1, each iteratively defined function $f_\iota : P_i \to Q_\iota$ is the identity function on P_ι, that is, $Q_\iota = P_\iota$ and $f_\iota(x) = x$.

Therefore, in Λ,

$$\underline{x} = \underline{f_\iota(x)} = \underline{f_\iota}\,\underline{x} =$$

by definition of $\underline{f}_\iota$ (Definition 5.5)

$$= \underline{x}\,\underline{P_1} \cdots \underline{P_n}\,\underline{g_1} \cdots \underline{g_k}. \qquad \square$$

Corollary 5.8 (representability of iterative functions). *Every iterative function is Λ-representable (and we have an automatic method to represent it).*

Proof. In the definition of iterative functions (Section 2.3), point (a) is settled by the basic operation synthesis paradigm (Definition 5.2) point (c) by the program synthesis paradigm (Definition 5.5), and point (b) is trivial. □

5.3. Examples: some Λ-programs

(a) With references to point (a) of Section 2.4 we have, by Definition 5.5(b),

$$\underline{f}v^L =_\Lambda \lambda u^L.u\underline{L}\ \underline{\text{cons}}\ v.$$

It follows that

$$\underline{\text{cat}}\ u^L v^L =_\Lambda \underline{f}vu =_\Lambda u\underline{L}\ \underline{\text{cons}}\ v,$$

hence

$$\underline{\text{cat}} \equiv \lambda u^L \lambda v^L.u\underline{L}\ \underline{\text{cons}}\ v.$$

(b) With reference to point (b) of Section 2.4 we have, by Definition 5.5(a),

$$\underline{\text{preorder}} \equiv \lambda v^T.v\underline{L}\ \underline{\text{cons}}\ \underline{\text{cat}}\ \underline{\text{nil}}.$$

6. Completeness

We raise the following question: given a closed Λ-type γ, which is the set of all the closed Λ-terms in normal form having type γ?

Notice that, in general, this set is recursively enumerable. Using the Curry–Howard homomorphism between proofs and Λ-terms we could rephrase this question in the following equivalent way: given a sentence in second-order (intuitionistic) logic, which is the set of all its closed normal proofs?

Of course, a false sentence has no proof at all; therefore, the corresponding Λ-type is not the type of any closed Λ-term. This is the case, for example, of the Λ-type $\Delta\beta((\beta\to\beta)\to\beta)$ (which corresponds to the false sentence $(\forall\beta)((\beta\to\beta)\to\beta)$).

Imposing suitable restrictions on the structure of γ, the Completeness Theorem below settles our question. The result is that the class of all closed normal forms of type γ consists exactly of all the Λ-representations of the elements of a suitable data structure which is uniquely determined by the relation $\underline{S}=\gamma$ ($\underline{S}$ is defined in Section 4).

In particular, if $\gamma=\underline{N}=\Delta N((N\to N)\to N\to N)$, our result says that the closed normal forms of type $\underline{N}$ are the Λ-representations of the natural numbers and (most importantly) no other closed normal form has type γ.

Actually, the restrictions on γ can be expressed simply by $\gamma=\underline{S}$ for some data structure S.

For a more syntactical characterization of γ we need a definition. For simplicity, in the following we work in Λ without parameters (the general case requires some minor changes).

Definition 6.1 (*degree of a type*). For a Δ-free type α we define the degree $d(\alpha)$ of α as follows:

$$d(\alpha)=0 \quad \text{if } \alpha \text{ is a type variable,}$$

$$d(\alpha\to\beta)=\max\{1+d(\alpha), d(\beta)\}.$$

Proposition 6.2 (on the restrictions). *$\gamma=\underline{S}$ for some data structure S (without parameters) iff γ is a closed nonempty type (i.e., there is some closed Λ-term of type γ) and $\gamma=\Delta P_1\cdots\Delta P_n.\theta$ for some $n>0$, where θ is Δ-free and $d(\theta)\leq 2$ (in the case with parameters we must further require that θ does not contain any subtype $\alpha\to\beta$, where β is a parameter type).*

Actually, the restriction that γ be nonemtpy is unnecessary if we enlarge the definition of data system to enclose empty data structures (like $\langle\{B\}, \{g: B \to B\}\rangle$, see Remark 1.1(c)).

Since we do not want to prove the Proposition stated above because it is quite tedious and trivial, we shall assume directly that $\gamma = \underline{S}$ for some data structure S.

Theorem 6.3 (Completeness Theorem). *Let*

$$D = \langle \mathscr{S}, \mathscr{G} \rangle = \langle \{A_1, \ldots, A_l, P_1, \ldots, P_n\}, \{g_1, \ldots, g_k\} \rangle$$

be a data system. If $S \in \mathscr{S}$ is a data structure and t is a closed Λ-term of Λ-type $\gamma = \underline{S}$, then there exists a (unique) element $x \in S$ such that $\underline{x} =_\Lambda t$ (where $\underline{x}$ is given by Definition 4.3).

Proof (without parameters). Let so S be a proper data structure, say $S = P_\iota$, and let t be a closed Λ-term of Λ-type $\underline{P_\iota} \equiv \Delta \boldsymbol{P}_1 \cdots \Delta \boldsymbol{P}_n(\theta_1 \to \cdots \to \theta_k \to \boldsymbol{P}_\iota)$ (see Definition 4.1).

The proof will make essential use of the fact that $\theta_1, \ldots, \theta_k$ have degree 1 (check!) and P_ι has degree 0.

We have to prove that, for some $x \in P_\iota$,

$$t =_\Lambda \underline{x} \equiv \Lambda \boldsymbol{P}_1 \cdots \Lambda \boldsymbol{P}_n \lambda \boldsymbol{g}_1 \cdots \lambda \boldsymbol{g}_k . \boldsymbol{x}$$

(see Definition 4.3), or equivalently $t\boldsymbol{P}_1 \cdots \boldsymbol{P}_m \boldsymbol{g}_1 \cdots \boldsymbol{g}_k =_\Lambda \boldsymbol{x}$.

We are done if we have proved that there exists an $x \in P_\iota$ such that the normal form t' of $t\boldsymbol{P}_1 \cdots \boldsymbol{P}_n \boldsymbol{g}_1 \cdots \boldsymbol{g}_k$ is identical to $\boldsymbol{x}$ (indeed $\boldsymbol{x}$ is a normal form by point (c) of Section 3.1).

By Section 3.1 it follows that the requirement that t' is identical to $\boldsymbol{x}$ for some $x \in P_\iota$ amounts to fulfill all of the following points:

(a) t' is a normal form of Λ-type $\boldsymbol{P}$.

(b) t' has at most $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_k$ as free variables.

(c) t' is abstraction-free (i.e., neither "λ" nor "Λ" occur in t').

Since (a) is trivially satisfied and (b) follows from the hypothesis that t is a closed Λ-term, the only nontrivial point is (c).

Suppose, by contradiction, that t' is not abstraction-free. Since t' has a type P_ι of degree 0, t' is not an abstraction term, that is, $t' \neq \lambda x.N$ and $t' \neq \Lambda\gamma.N$.

Since t' is normal, t' has no subterms of the form $(\lambda x.N_1)N_2$ or of the form $(\Lambda\gamma.N)\alpha$.

The only possibility left is that t' has a subterm of the form $N_1(\lambda x.N_2)$ (or of the form $N_1(\Lambda\gamma.N_2)$).

If we consider the leftmost among such subterms, we can assume that N_1 is abstraction-free and that it is built up by application from a subset of the free variables $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_k$ of t' only.

Since the types $\theta_1, \ldots, \theta_k$ of $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_k$ have degree 1 and, since this property is hereditary w.r.t. application, the type of N_1 has also degree 1.

Since any Λ-term whose type has degree 1 can only be applied to a Λ-term whose type has degree 0, it follows that $N_1(\lambda x.N_2)$ (respectively $N_1(\Lambda\gamma.N_2)$) has no type; a contradiction. Thus, t' is abstraction-free. □

Corollary 6.4. *The Completeness Theorem extends to functional types; that is, if $S_1, \ldots, S_m, S$ are data structures and t is a closed Λ-term of Λ-type $\underline{S_1} \to \cdots \to \underline{S_m} \to \underline{S}$, then there is a (unique) function $f: S_1 \times \cdots \times S_m \to S$ such that t Λ-represents f (see Definition 5.1). (In general, f may not be iterative.)*

Proof. Choose $x_1 \in S_1, \ldots, x_m \in S_m$ and apply Theorem 6.3 to the Λ-term $(t\underline{x_1} \cdots \underline{x_m})$. □

7. An equivalence of programs in Λ

Given a Λ-representable function $f: S_1 \times \cdots \times S_m \to S$ (with $S_1, \ldots, S_m, S$ data structures), there are, in general, several nonconvertible Λ-terms which represent f in Λ. We say that two Λ-terms, or 'Λ-programs', are equivalent if they represent the same function. We shall give in Lemma 7.2 a sufficient condition for Λ-program equivalence, based on the fixed point equation (Corollary 5.7) and on the Completeness Theorem 6.3.

More precisely, we shall define an equivalence relation "$\simeq$" between Λ-terms such that if a, b are Λ-programs and $a \simeq b$, then a, b are equivalent Λ-programs. In this way we can prove equivalence of programs without using induction on the structure of the input.

Definition 7.1 (*the basic operations drop out*). Let a, b be Λ-terms. Define "$\simeq$" as the least congruence extending $=_\Lambda$ and such that

(*) if $D = \langle\{A_1, \ldots, A_1, P_1, \ldots, P_n\}, \{g_1, \ldots, g_k\}\rangle$ is a data system and v is a Λ-variable of Λ-type $\underline{P_i}$, then

$$v\underline{P_1} \cdots \underline{P_n}\,\underline{g_1} \cdots \underline{g_k} \simeq v.$$

The last equation is similar to the fixed point equation (Corollary 5.7) except that there is a free variable v instead of a closed Λ-term $\underline{x}$. Note that since $\simeq$ extends $=_\Lambda$, the variable v may actually be replaced by any Λ-term t of Λ-type P_i.

Lemma 7.2. *Let $P_1, \ldots, P_m$ be proper data structures, and $v_1, \ldots, v_m$ be Λ-variables of Λ-type $\underline{P_1}, \ldots, \underline{P_m}$ respectively.*

Let $a[v_1, \ldots, v_m]$ and $b[v_1, \ldots, v_m]$ be Λ-terms containing at most $v_1, \ldots, v_m$ as free variables. We claim that if $a[v_1, \ldots, v_m] \simeq b[v_1, \ldots, v_m]$, then, for all $x_1 \in P_1, \ldots, x_m \in P_m$, we have $a[\underline{x}_1, \ldots, \underline{x}_m] =_\Lambda b[\underline{x}_1, \ldots, \underline{x}_m]$.

Proof. The proof follows by induction on the derivation of $a[v_1, \ldots, v_m] \simeq b[v_1, \ldots, v_m]$; the only critical point is rule (*) of Definition 7.1, which is settled by the fixed point equation (Corollary 5.7). □

Theorem 7.3 (Λ-program equivalence). *If a, b are closed Λ-terms of Λ-type* $\underline{P_1} \to \cdots \to \underline{P_m} \to \underline{P}$ *(with* $P_1, \ldots, P_m$, P *proper data structures) and* $a \simeq b$, *then a, b are equivalent Λ-programs.*

Proof. Choose Λ-variables $v_1, \ldots, v_m$ of Λ-type $\underline{P_1}, \ldots, \underline{P_m}$ respectively. Since $a \simeq b$, and "$\simeq$" is a congruence, $(av_1 \cdots v_m) \simeq (bv_1 \cdots v_m)$; thus, by Lemma 7.2, for all $x_1 \in P_1, \ldots, x_m \in P_m$,

$$(a\underline{x_1} \cdots \underline{x_m}) =_\Lambda (b\underline{x_1} \cdots \underline{x_m}),$$

since both members have type $\underline{P}$, by the Completeness Theorem there is an $x \in P$ such that both $(a\underline{x_1} \cdots \underline{x_m})$ and $(b\underline{x_1} \cdots \underline{x_m})$ reduce to $\underline{x}$. Thus, a, b are equivalent Λ-programs. □

7.1. Applications

The following examples show how the equivalence "$\simeq$" can be used to transform Λ-programs, obtained by the program synthesis paradigm, into equivalent Λ-programs which satisfy some extra properties.

(a) With reference to example (a) of Section 5.3 we have

$$\begin{aligned}
\underline{\text{cat}} &\equiv \lambda u^{\underline{L}} \lambda v^{\underline{L}}.u\underline{L}\ \underline{\text{cons}}\ v \\
&\simeq \lambda u^{\underline{L}} \lambda v^{vL}.u\underline{L}\ \underline{\text{cons}}(v\underline{L}\ \underline{\text{cons}}\ \underline{\text{nil}}) \\
&=_\Lambda \lambda u^{\underline{L}} \lambda v^{\underline{L}}.((\Lambda \boldsymbol{L} \lambda \boldsymbol{c}^{\boldsymbol{A}\to\boldsymbol{L}\to\boldsymbol{L}} \lambda \boldsymbol{n}^{\boldsymbol{L}}.u\boldsymbol{L}\boldsymbol{c}(v\boldsymbol{L}\boldsymbol{c}\boldsymbol{n}))\underline{L}\ \underline{\text{cons}}\ \underline{\text{nil}}) \\
&\simeq \lambda u^{\underline{L}} \lambda v^{\underline{L}} \Lambda \boldsymbol{L} \lambda \boldsymbol{c}^{\boldsymbol{A}\to\boldsymbol{L}\to\boldsymbol{L}} \lambda \boldsymbol{n}^{\boldsymbol{L}}.u\boldsymbol{L}\boldsymbol{c}(v\boldsymbol{L}\boldsymbol{c}\boldsymbol{n}) \\
&\equiv^{\text{def}} \underline{\text{cat}}^*.
\end{aligned}$$

We notice that the Λ-program $\underline{\text{cat}}^*$ is associative with respect to every Λ-variables u, v, w of type $\underline{L}$ (i.e., $\underline{\text{cat}}^*\ u\ (\underline{\text{cat}}^*\ vw) = \underline{\text{cat}}^*(\underline{\text{cat}}^*\ uv)w$), while the Λ-program $\underline{\text{cat}}$ is associative with respect to closed Λ-terms of Λ-type $\underline{L}$ only.

Moreover, $\underline{\text{cat}}^*$ does not use the 'subprograms' $\underline{\text{cons}}$, $\underline{\text{nil}}$.

(b) We give, without proof, two equivalent Λ-programs for multiplication of natural numbers:

$$\underline{\text{Mult}} \equiv \lambda u^{\underline{N}} \lambda v^{\underline{N}}.u\underline{N}(v\underline{N}\underline{s})\boldsymbol{o},$$

$$\underline{\text{Mult}}^* \equiv \lambda u^{\underline{N}} \lambda v^{\underline{N}} \Lambda \boldsymbol{N} \lambda \boldsymbol{s}^{\boldsymbol{N}\to\boldsymbol{N}}.u\boldsymbol{N}(v\boldsymbol{N}\boldsymbol{s}).$$

It can easily be checked that $\underline{\text{Mult}}^*$ is associative with respect to free Λ-variables while $\underline{\text{Mult}}$ is associative with respect to closed Λ-terms only.

Open problem

The congruence relation $\simeq$ introduced above proved itself stronger than Λ-convertibility. It seems doubtful that a corresponding notion of normal form, stronger than β-η normal form may easily be developed: for example (see example (a) of Section 7.1), if we think cat* to be the normal form of cat, then the reduction cannot be defined, as usual, asymmetrizing the congruence since cat* is obtained using the relation on both sides.

It is an open problem (as suggested to us by M. O'Donnell) to characterize the congruence $\simeq$ inside a precise and natural context.

8. Extension to higher functional types

We have given in Definition 5.5 a program synthesis paradigm for the Λ-representation of any iterative function.

We shall illustrate by an example how the paradigm could be extended to functions of higher type which we call 'iterative functionals'.

The definition of iterative functionals parallels the definition of iterative functions (Section 2.3), except that their 'types' are arbitrary high finite types based on the types of the data structures. The only proviso is obviously that the domain of a functional given by iterative definition (parallel to Section 2.1) is a data structure.

Iterative functionals may be thought of as a generalization of the Gödel recursive functionals to any data structure (i.e., not necessarily natural numbers).

In the following example we assume any question of convergence to be settled.

Example 8.1. We define a functional ack: $N \to N \to N$ and we use the (extended) program synthesis paradigm to represent ack in Λ.

For each $f: N \to N$ define $\sigma_f: N \to N$ iteratively as follows:

$$\sigma_f(o) = f(1), \qquad \sigma_f(s(m)) = f(\sigma_f(m)).$$

Define $\sigma: (N \to N) \to N \to N$ explicitly by $\sigma(f) = \sigma_f$.

Notice that the previous definition transforms the iterative definition of the unary function σ_f into an iterative definition of the binary (curried) functional σ.

Define ack: $N \to N \to N$ iteratively by

$$\mathrm{ack}(o) = s, \qquad \mathrm{ack}(s(m)) = \sigma(\mathrm{ack}(m)).$$

Notice that, by definition, ack becomes an iterative functional.

It can easily be proved that $\mathrm{ack}(m)(n) = \mathrm{Ack}(m, n)$ where Ack: $N \times N \to N$ is the Ackermann function usually defined as follows:

$$\mathrm{Ack}(o, n) = s(n),$$

$$\mathrm{Ack}(s(m), o) = \mathrm{Ack}(m, 1),$$

$$\mathrm{Ack}(s(m), s(n)) = \mathrm{Ack}(m, \mathrm{Ack}(s(m), n)).$$

By the (extended) program synthesis paradigm we have, in Λ,

$$\underline{\sigma} \equiv \lambda f^{\underline{N}\to\underline{N}} \lambda v^{\underline{N}}.v\underline{N}f(f\underline{1}),$$

$$\underline{\mathrm{ack}} \equiv \lambda v^{\underline{N}}.v(\underline{N}\to\underline{N})\underline{\sigma}\underline{s},$$

that is, $\underline{\mathrm{ack}(m)(n)} = \underline{m}(\underline{N}\to\underline{N})\underline{\sigma}\underline{s}\underline{n}$.

This representation of the Ackermann function is almost the same as the one in [8].

9. Λ-types as notations for data structures

We notice that any data structure S is completely determined by the Λ-type $\underline{S}$ representing S. (This is evident if we look at Examples 4.2.)

Thus, we can use Λ-types as a system of notation to describe data structures.

Below we give a list of (Λ-types for) several interesting data structures (in addition to those we have already considered):

(a) *heretogenous pairs*: $\Delta\beta((\alpha_1\to\alpha_2\to\beta)\to\beta)$ (α_1, α_2 are parametric types).

(b) *n-elements sets*: $\Delta\beta(\underbrace{\beta\to\cdots\to\beta}_{n \text{ times}}\to\beta)$.

(c) *binary strings*: $\Delta\beta((\beta\to\beta)\to(\beta\to\beta)\to\beta\to\beta)$.

(d) *binary labelled trees*: $\Delta\beta((\alpha\to\beta\to\beta\to\beta)\to\beta\to\beta)$ (α is a parametric type).

Remark 9.1. It can easily be proved that if a Λ-type γ is equal to $\underline{S}$ for some data structure S, then γ is a closed Λ-type of the shape $\Delta\beta_1\cdots\Delta\beta_n(\theta_1\to\cdots\to\theta_k\to\beta_i)$, where $\theta_1, \ldots, \theta_k$ have degree 1. The converse is not true in general; for example, the Λ-type $\Delta\beta((\beta\to\beta)\to\beta)$ has no corresponding data structure (this is related to Remark 1.1(c)).

Acknowledgment

We owe our thanks to Mariangiola Dezani for the suggestion to work with second-order polymorphic lambda calculus.

We are grateful to John Backus, Donald Knuth, Luis Sanchis, John Reynolds, Alan Robinson, Mitchell Wand, Mario Coppo, and Wolf Gross for helpful and encouraging discussions.

Special thanks are due to Michael O'Donnell for his useful criticism and an unknown referee whose stimulating remarks were literally absorbed in our introduction.

References

[1] C. Böhm and D. Kozen, Eliminating recursion over acyclic data structures in functional programs, in: *4th Internat. Workshop on the Semantics of Programming Languages*, Bad Honnef, 1983; Summary in: *Bull. EATCS* **20** (1983) 205.

[2] R.L. Constable, Programs as proofs: A synopsis, *Inform. Process. Lett.* **16** (3) (1983) 105–112.

[3] S. Fortune, D. Leivant and M. O'Donnell, The expressiveness of simple and second-order type structures, *J. ACM* **30**(1) (1983) 151–185.

[4] J.Y. Girard, Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur, Thèse de Doctorat d'Etat, Univ. de Paris, 1972.

[5] C. Goad, Computational uses of the manipulation of formal proofs, Dissertation, Dept. of Computer Science, Standford Univ. (1980) 122.

[6] K. Gödel, Über eine bisher noch nicht benützte Erweiterung des finites Standpunktes, *Dialectica* **12** (1958) 280–287.

[7] D. Leivant, Reasoning about functional programs and complexity classes associated with type disciplines, *24th Ann. Symp. on Foundation of Computer Science* (1983) 460–469.

[8] J.C. Reynolds, Toward a theory of type structure, *Programming Symposium* (*Colloque sur la Programmation*), Lecture Notes in Computer Science **19** (Springer, Berlin, 1974) 408–425.

[9] S. Takasu, Proofs and programs, in: *Proc. 3rd IBM Symp. on Mathematical Foundations of Computer Science*, Japan, 1978.