0 Recursion vs. Iteration at Higher Orders

1

Recursion Versus Iteration at Higher-Orders

A. J. Kfoury

Boston University, Boston, MA 02215, USA
e-mail: kfoury@cs.bu.edu

Abstract. We extend the well-known analysis of recursion-removal in first-order program schemes to a higher-order language of *finitely typed and polymorphically typed functional programs*, the semantics of which is based on call-by-name parameter-passing. We introduce methods for recursion-removal, i.e. for translating higher-order *recursive* programs into higher-order *iterative* programs, and determine conditions under which this translation is possible. Just as finitely typed recursive programs are naturally classified by their orders, so are finitely typed iterative programs. This syntactic classification of recursive and iterative programs corresponds to a semantic (or computational) classification: the higher the order of programs, the more functions they can compute.

1 Background and Motivation

Although our analysis is entirely theoretical, as it combines methods from typed λ-calculi, from abstract recursion theory and from denotational semantics, the problems we consider have a strong practical motivation.

The translation of *recursive* procedures into *iterative* procedures, sometimes called *recursion-removal*, has been a standard feature of programming practice for a long time. A frequent and important use of recursion-removal is, for example, in compiler design; modern treatments of this topic are in [6, Chapter 8] and [23, Chapter 21]. Discussions about other benefits of recursion-removal can be found elsewhere in the literature, e.g. in the books [1, 13, 19], among several others. The usual implementation of a recursive procedure requires a stack, in addition to basic control mechanisms to change the order of execution in a computation, such as **goto** instructions. The stack in question is used to save information, the so-called *activation records* [2, Chapter 10], from successive recursive calls of the procedure. By contrast, the implementation of an iterative procedure does not require a stack, resulting in more efficient use of storage space.

By appropriate coding and decoding of arbitrarily long sequences of activation records, it is in principle always possible to avoid the use of a stack and directly implement recursive procedures as iterative ones. The problem is that

* Partly supported by NSF grant CCR–9417382. A shorter version of this paper appears in the conference proceedings of the *Foundations of Software Technology and Theoretical Computer Science*, held on December 18-20, 1997, at the *Indian Institute of Technology*, Kharagpur, India.

such a coding and decoding mechanism is not always available, and even if it is, it often leads to just as expensive implementations by trading storage space for computational complexity. In the absence of such a coding and decoding mechanism, there are recursive procedures that can be translated into iterative ones, while there are others that cannot. A procedure in the latter case is inherently recursive, in the sense that any other procedure defining the same function cannot be iterative and requires a stack for its implementation, if we also preclude the use of a coding and decoding mechanism to simulate a stack.

Comparative Schematology. The preceding facts are well-known among computer scientists. In the early 1970's people working in *comparative schematology* aimed at rigorously establishing these and other similar results. Classes of *program schemes* were defined, modelling different control mechanisms in conventional programming languages, such as recursion and iteration. An elegant and sometimes highly technical theory was developed to examine questions of inter-translatability between these classes. The first paper in this area, sparking an intensive research activity for many years, was by Paterson and Hewitt [22] who spelled out precise conditions under which recursion-removal is not possible. We summarize the Paterson-Hewitt result by writing ITER $<$ REC, meaning that iteration is a strictly weaker control structure than recursion.

Pebbling vs. Counting. Paterson and Hewitt gave two very different proofs for ITER $<$ REC. Their first proof was based on the *pebble game*. As it turned out in later years, the pebble game and many of its variants became a basic technique in other areas of theoretical computer science; in particular, the most numerous applications have been in complexity theory, e.g. in time-space trade-offs [24, 25], rather than in comparative schematology or in logics of programs. The second proof by Paterson and Hewitt was a *counting argument*. Freely speaking, counting (rather than pebbling) is used to show that a simulating iterative procedure runs out of time (rather than space). This dichotomy between pebbling and counting pervades nearly all studies of the original question recursion vs. iteration and later variants. For the early results for example, see the survey in [10, Chapters VII and VIII].

Other Theoretical Studies. Recursion pervades the whole field of computer science. Ways of removing or simplifying recursion abound. The background reviewed above by no means exhausts the topic and only covers that part of the field directly motivating our work. There is for example an extensive literature on boundedness of database logic programs, see [12] and the references therein; a database program is *bounded* if the depth of recursion is a constant independent of the input, in which case the program is equivalent to a recursion-free first-order formula. Another motivation for the study of recursion vs. iteration comes from inductive definability in abstract recursion theory; see for example [20] and [21], where a formal logic framework is defined to study the question. Another work still is [5], where the question is considered in relation to program synthesis. However interesting these and other studies are, none is directly related to this paper. Closer to our concerns is part of the literature on *continuations* that deals with issues of recursion removal, specifically, the use of CPS (continuation-passing style) to transform a recursive functional program into tail-recursive form. Although “iteration” and “tail-recursion” coincide in the case of first-order programs, the two notions are not exactly the same at higher-orders. In the full version [16] of this report, we adduce several reasons for this distinction; in any case, an attractive theory and many non-trivial results emerge out of the distinction, as shown in [16].

Some Past Results and Some Questions. This history of results, starting with the Paterson-Hewitt result in 1970, is entirely concerned with the first-order case of recursion vs. iteration. That is, recursive and iterative procedures are always activated by receiving input values of ground type, and they always return an output value, if any, of ground type.

Many modern programming languages allow their programs to use higher-order objects in their computations. The role of such objects is purely auxiliary in that, semantically, higher-order values are not mentioned in the input-output relation of the program. In our setting, only the main procedure in a program is first-order, but it can call intermediate procedures of arbitrary finite orders. It is now important to distinguish between a single procedure and a finite collection of procedures calling each other, of which only one is the main procedure. In this paper, we call such a collection a *functional program.* The *order* of a functional program is the highest order of any of its procedures. In general, a functional program is *recursive*, as no restriction is imposed on its syntax; under the restriction that all of its procedures are iterative, we say that the functional program is *iterative.* If ITER_n and REC_n are, respectively, the set of all iterative functional programs and the set of all recursive functional programs of order $\leqslant n$, where n is a positive integer, then:

(1) $\mathrm{ITER}_n \subset \mathrm{REC}_n$

i.e. every program in ITER_n, as a syntactic object, is a program in REC_n, but not vice-versa. We also have the following strict syntactic hierarchies:

(2) $\mathrm{ITER}_1 \subset \mathrm{ITER}_2 \subset \mathrm{ITER}_3 \subset \cdots$

(3) $\mathrm{REC}_1 \subset \mathrm{REC}_2 \subset \mathrm{REC}_3 \subset \cdots$

(ITER_1 and REC_1 here are ITER and REC mentioned earlier.) It is well-know there is a strict computational hierarchy (see, for example, [9] or [15] or references cited therein):

(4) $\mathrm{REC}_1 < \mathrm{REC}_2 < \mathrm{REC}_3 < \cdots$

In words, increasing the order of functional programs results in a gain of computational power. On the other hand, so far we can only assert, by (2):

(5) $\mathrm{ITER}_1 \leqslant \mathrm{ITER}_2 \leqslant \mathrm{ITER}_3 \leqslant \cdots$

and the obvious question is whether this computational hierarchy is strict. In the presence of higher-order procedures, the question of recursion vs. iteration splits into two parts. We first ask, for $n \geqslant 2$:

(6) Is $\mathrm{ITER}_n < \mathrm{REC}_n$?

If the answer to question (6) is *yes*, it will generalize to higher-orders the Paterson-Hewitt result in [22], namely: At every order, there are functional programs that are inherently recursive. We further ask, for $n \geqslant 1$:

(7) Is $\mathrm{REC}_n \leqslant \mathrm{ITER}_{n'}$ for some $n' > n$? If *yes*, what is the least such n' ?

In words, question (7) asks whether recursion-removal is always possible after all, but now at the price of increasing the order of the program. These and other related questions, to be formulated in due course, are the immediate motivation for the present work.

Organization of the Paper. The first task is to define precisely the framework of the investigation. Our choices are not the only possible ones: How we define higher-order procedures, and how we interpret and execute them, depend on choices inspired by a particular programming paradigm. This is the paradigm of strongly-typed pure functional programs, where there are no side-effects and where the only execution mechanism is call-by-name parameter-passing. This is all done in Section 2 and Section 3. Hence, *equivalence* between programs here means equivalence under an operational semantics based on call-by-name execution. In addition to the hierarchies $\{\mathrm{REC}_n\}$ and $\{\mathrm{ITER}_n\}$ mentioned earlier, we define the class p-REC of *polymorphically typed recursive* programs and the class p-ITER of *polymorphically typed iterative* programs.

Section 4 re-examines the recursive program used by Paterson and Hewitt [22] to show that $\mathrm{ITER}_1 < \mathrm{REC}_1$. The purpose of this examination is partly to gather evidence and intuition for the systematic translation developed in Section 5 from higher-order recursion to higher-order iteration.

Our key technical lemmas are presented in Section 5 and Section 6. From these results, we draw several consequences about the hierarchies $\{\mathrm{REC}_n\}$ and $\{\mathrm{ITER}_n\}$ in Section 7.

Further details for some of the proofs here, as well as additional results relating higher-order recursion, tail-recursion and iteration, are in our forthcoming report [16]. For typesetting convenience, all figures are displaced to the very end of the report.

2 Basic Definitions: Syntax and Semantics

What we set up is the syntax of a typed λ-calculus + conditionals + recursion. We introduce recursion by means of mutually recursive functional equations, and not by application of fixpoint operators.

Definition 1 (Types). Let TVar be an infinite set of *type variables*. Let $*$ and **bool** be two type constants, which we call *ground types*. The *atomic types* are $\{*, \mathbf{bool}\} \cup \mathrm{TVar}$. The set $\mathbb{T}$ of *types* is the least such that

$$\mathbb{T} \;\supseteq\; \{*, \mathbf{bool}\} \;\cup\; \mathrm{TVar} \;\cup\; \{\ (\sigma \to \tau) \mid \sigma, \tau \in \mathbb{T}\ \}$$

For $\sigma \in \mathbb{T}$, let $\mathrm{TVar}(\sigma)$ denote the finite set of type variables occurring in σ. We partition $\mathbb{T}$ into the set $\mathbb{T}_{\mathrm{fin}}$ of *finite types* and the set $\mathbb{T}_{\mathrm{gen}}$ of *generic types*:

$$\mathbb{T}_{\mathrm{fin}} \;=\; \{\ \sigma \in \mathbb{T} \mid \mathrm{TVar}(\sigma) = \varnothing\ \} \qquad \text{and} \qquad \mathbb{T}_{\mathrm{gen}} = \mathbb{T} - \mathbb{T}_{\mathrm{fin}}\ .$$

The *order* of a finite type $\sigma \in \mathbb{T}_{\mathrm{fin}}$ is:

$$order(\sigma) = \begin{cases} 0, & \text{if } \sigma = * \text{ or } \mathbf{bool}; \\ \max\{order(\tau_1) + 1, order(\tau_2)\}, & \text{if } \sigma = (\tau_1 \to \tau_2). \end{cases}$$

We do not define $order(\sigma)$ if σ is generic. In what follows, every term M has a type τ, which we indicate by writing $M : \tau$. If τ is a finite type and $M : \tau$, we say that M is *finitely typed*. If τ is a generic type and $M : \tau$, we say that M is *polymorphically typed*.

A *type substitution* is a map $S : \mathrm{TVar} \to \mathbb{T}$ such that $\{\alpha \in \mathrm{TVar} | S(\alpha) \neq \alpha\}$ is finite. Every type substitution extends in a natural way to a $\{*, \mathbf{bool}, \to\}$-homomorphism $S : \mathbb{T} \to \mathbb{T}$. For $\sigma, \tau \in \mathbb{T}$, we say that τ is an *instance* of σ, in symbols $\sigma \preccurlyeq \tau$, if there is a type substitution S such that $\tau = S(\sigma)$.

Definition 2 (Terms). A class of functional programs is defined relative to a fixed first-order *signature* $\Sigma_A = \Sigma \cup A$, where Σ is a finite set of *relation* and *function symbols* and A is a countable set of *individual symbols*. To avoid trivial and uninteresting situations, we require that both $\Sigma \neq \varnothing$ and $A \neq \varnothing$. Every relation (or function) symbol $f \in \Sigma$ has a fixed arity $k \geqslant 1$, in which case its type is

$$\underbrace{* \to \cdots \to *}_{k} \to \rho\ , \qquad \text{abbreviated as} \quad *^k \to \rho\ ,$$

where $\rho = \mathbf{bool}$ (or $*$, resp.), i.e. a finite type of order 1. Every constant in A is of type $*$. It is convenient to use two disjoint sets of variables: *object variables* and *function names*. For every type τ there is a countably infinite set of object variables of type τ, and for every non-atomic type τ a countably infinite set of function names of type τ. The set of *terms* is the smallest containing

$$\{\ \mathit{tt}, \mathit{ff}\ \} \;\cup\; A \;\cup \qquad\qquad \text{(ground constants)}$$

$$\Sigma \;\cup\; \{\ \mathbf{if}^{\tau} \mid \tau \in \mathbb{T}\ \} \;\cup \qquad\qquad \text{(other constants)}$$

$$\{\ \text{object variables}\ \} \cup \{\ \text{instantiated function names}\ \} \qquad \text{(variables)}$$

and closed under application and λ-abstraction. The details of the definition of well-typed terms are given in Figure 1.

We omit the type subscript τ in $\mathbf{if}^\tau$ and in an instantiated function name $\mathcal{F}^\tau$ whenever possible, if no ambiguity is introduced. (**if** M **then** N **else** P) is a sugared version of (**if** $M\ N\ P$). For simplicity, we often omit the type of a λ-binding, and therefore write $(\lambda v\ M)$ instead of $(\lambda v : \sigma.\ M)$. If a *closed* term M and all its subterms are finitely typed, the order of M is:

$$\mathit{order}(M) = \max \{ \ \mathit{order}(\tau) \mid N \subseteq M \text{ and } N : \tau \ \}$$

If M is not closed and $N \equiv (\lambda \mathbf{v}\ M)$ is the closure of M, then $\mathit{order}(M) = \mathit{order}(N)$. We do not define the order of a polymorphically typed term. By a slight abuse of terminology, we often say "M is a n-th order term" to mean that $\mathit{order}(M) \leqslant n$ rather than $\mathit{order}(M) = n$.

Definition 3 (Functional Programs). Let $\mathcal{F}$ be a function name of type $\sigma \in \mathbb{T}$. A *function definition* for $\mathcal{F}$ is an equation of the form $\mathcal{F} = M$ where M is an arbitrary closed term of type σ. $\mathcal{F}$ on the left of "=" is not instantiated, while every occurrence of $\mathcal{F}$ on the right of "=" is instantiated as $\mathcal{F}^\tau$, for some τ such that *type*$(\mathcal{F}) \preccurlyeq \tau$, and corresponds to a recursive call to $\mathcal{F}$ at type τ.

A *functional program* $\mathcal{P}$ is a finite non-empty set of function definitions, together with a distinguished function name $\mathcal{F}$ such that:

- The type of $\mathcal{F}$ is $*^k \to \rho$ for some $k \geqslant 1$, where $\rho = *$ or **bool**.
- For every function name $\mathcal{G}$, $\mathcal{P}$ has at most one function definition for $\mathcal{G}$.
- For every function name $\mathcal{G}$, $\mathcal{P}$ has a definition for $\mathcal{G}$ if there is an instance $\mathcal{G}^\tau$ of $\mathcal{G}$ appearing in $\mathcal{P}$ (i.e. on the right of "=", a call to $\mathcal{G}$ at some type τ).

The restrictions we impose on the type of the distinguished $\mathcal{F}$ have to do with the fact that the inputs and output, if any, of a program are always ground values.

Let $\mathcal{P}$ be the functional program $\{\mathcal{F}_i = M_i\}_{1 \leqslant i \leqslant \ell}$. In a ML-like language that supports polymorphic recursion, $\mathcal{P}$ can be written as:

$$\lambda v_1 \cdots v_k.\ \mathbf{letrec}\ \ (\mathcal{F}_1, \ldots, \mathcal{F}_\ell) \ = \ (M_1, \ldots, M_\ell) \ \ \mathbf{in} \ \ \mathcal{F}_1 v_1 \cdots v_k$$

where $\mathcal{F}_1$ is the distinguished function of arity $k \geqslant 1$. If the types of all the terms $M_1, \ldots, M_\ell$ and their subterms are finite, then $\mathcal{P}$ is finitely typed; otherwise, $\mathcal{P}$ is polymorphically typed. If $\mathcal{P}$ is finitely typed, then its *order* is

$$\mathit{order}(\mathcal{P}) = \max\{\mathit{order}(M_1), \ldots, \mathit{order}(M_\ell)\}$$

If this order is $n \geqslant 1$, we also say this is an order-n functional program. We do not define the order of a polymorphically typed program. Several examples of finitely typed programs are in Sections 3, 4 and 5, and an example of a polymorphically typed program is in Section 6.

Although it does not restrict the later analysis in any way, let us assume that the right-hand side of every function definition $\mathcal{F}_i = M_i$ in $\mathcal{P}$ is in β-normal form. Under this assumption, it is easy to check that if $1 \leqslant \mathit{order}(\mathcal{F}_i) < \mathit{order}(M_i)$ then there is an instantiated function name $\mathcal{F}_j$ occurring in M_i such that $\mathit{order}(\mathcal{F}_j) = \mathit{order}(M_i)$. Hence, under this assumption, we can equivalently and more simply define $\mathit{order}(\mathcal{P})$ by: $\mathit{order}(\mathcal{P}) = \max\{\mathit{order}(\mathcal{F}_1), \ldots, \mathit{order}(\mathcal{F}_\ell)\}$. The syntactic hierarchy of functional programs is given by:

- REC_n = { finitely typed functional programs of order $\leqslant n$ }
- **REC** = $\bigcup_{n\geqslant 1} \mathrm{REC}_n$
- p-REC = { polymorphically typed functional programs }

Definition 4 (Call-by-Name Semantics). A functional program computes relative to a Σ_A-*structure* $\mathcal{A}$ which assigns a meaning to every symbol in the signature Σ. We take the universe of $\mathcal{A}$ to be precisely the set A of all individual constants of type $*$. The meaning in $\mathcal{A}$ of the symbol $f \in \Sigma$ of type $*^k \to \rho$, where $k \geqslant 1$ and $\rho = *$ (or **bool**), is a total function from A^k to A (or to $\{tt, ff\}$). For the functional program $\mathcal{P} = \{\mathcal{F}_i = M_i\}_{1\leqslant i\leqslant \ell}$ we define the *reduction relation* $\xrightarrow[\mathcal{A},\mathcal{P}]{}$ relative to $\mathcal{A}$ by the rewrite rules of Figure 2.

In the δ-reduction rule, $a_1, \ldots, a_k$ are arbitrary elements of A, $f \in \Sigma$ has arity $k \geqslant 1$, and f interpreted in $\mathcal{A}$ is a function that maps $a_1 \cdots a_k$ to $a_{k+1} \in A \cup \{tt, ff\}$. We often leave the structure $\mathcal{A}$ and the program $\mathcal{P}$ implicit, and write $\longrightarrow$ instead of $\xrightarrow[\mathcal{A},\mathcal{P}]{}$. We write $\twoheadrightarrow$ for the reflexive transitive closure of $\longrightarrow$.

A *redex* is one of the terms appearing on the left-hand side of $\longrightarrow$ in one of the first 4 rules of Figure 2. In addition to the usual β-redexes, we also have $\mathcal{P}$-redexes, **if**-redexes, and δ-redexes in our calculus. We often need to refer to the position of a redex occurrence Δ relative to other redex occurrences in the same term. If Δ is not a single symbol, i.e. it is not a $\mathcal{P}$-redex, we agree that the position of Δ is determined by its outermost left parenthesis. For example, in the term:

$$(\ f\ \underset{1}{(}\ \ \mathbf{if}\ \ tt\ \ \mathbf{then}\ \ \underset{2}{(}\ \ (\lambda v.\ \underset{3}{\mathcal{F}}\ v)\ a_1\ \ )\ \ \mathbf{else}\ \ a_2\ )\ \ \underset{4}{(}\ \ f\ a_1\ a_2\ )\ )$$

there are four redexes labelled with 1 (an **if**-redex), 2 (a β-redex), 3 (a $\mathcal{P}$-redex) and 4 (a δ-redex), in the order in which they occur from left to right.

The first term in a computation of program $\mathcal{P}$ is always $(\mathcal{F}\ a_1 \cdots a_k)$, where $\mathcal{F}$ is the distinguished function name of $\mathcal{P}$ with arity $k \geqslant 1$, for some $a_1, \ldots, a_k \in A$. We call $\mathbf{a} = a_1 \cdots a_k$ an *input* vector for $\mathcal{P}$. We call $\langle \mathcal{A}, \mathbf{a}\rangle$ an *interpretation* for $\mathcal{P}$. Note that $(\mathcal{F}\ a_1 \cdots a_k)$ is a closed term of a ground type, $*$ or **bool**.

The next proposition is proved in [15, Sections 2 and 3]. The Church-Rosser property, the standarization property, and other concepts of the λ-calculus are defined in [4]. A term is in *normal form* if it does not contain any redexes, i.e. it cannot be reduced. To reduce a term in *normal order* means to reduce its leftmost redex at every step.

Proposition 5. *Let $\mathcal{P}$ be a program over the signature Σ_A, whose distinguished function name is $\mathcal{F} : *^k \to \rho$ where $k \geqslant 1$ and $\rho \in \{*, \mathbf{bool}\}$. Let $\langle \mathcal{A}, \mathbf{a}\rangle$ be an interpretation for $\mathcal{P}$.*

1. *The above reduction system has the Church-Rosser property and the standarization property.*
2. *If applying the reduction rules in some arbitary order causes $(\mathcal{F}\ \mathbf{a})$ to terminate at a term M, i.e. M is in normal form, then M is a ground constant of type ρ.*

3. *If applying the reduction rules in some arbitrary order causes* $(\mathcal{F}\ \mathbf{a})$ *to terminate at ground constant* b*, then applying the reduction rules in normal order also causes* $(\mathcal{F}\ \mathbf{a})$ *to terminate at the same ground constant* b. $\square$

The *call-by-name semantics* of functional programs corresponds to carrying out their computations, viewed as reduction sequences, in normal order. Let $\mathcal{P}$ be a functional program over the signature Σ_A, whose distinguished function name is $\mathcal{F} : *^k \to \rho$ where $k \geqslant 1$ and $\rho = *$ or **bool**. Over a Σ_A-structure $\mathcal{A}$, the program $\mathcal{P}$ defines a partial function $\mathcal{P}^{\mathcal{A}} : A^k \to A$ or $\mathcal{P}^{\mathcal{A}} : A^k \to \{tt, f\!f\}$, given by: $\mathcal{P}^{\mathcal{A}} = \{\ \langle \mathbf{a}, b\rangle \mid (\mathcal{F}\ \mathbf{a}) \twoheadrightarrow b\ \}$ where $b \in A$ or $b \in \{tt, f\!f\}$, respectively. Implicit in this definition, by the preceding proposition, is that $\mathcal{P}^{\mathcal{A}}$ is the call-by-name semantics of $\mathcal{P}$ over the structure $\mathcal{A}$.

3 Iterative Form

A term M is *simple* if M does not mention instantiated function names. A term M is *left-linear* if either M is simple or $M \equiv (\mathcal{F}\ N_1 \cdots N_k)$ where $\mathcal{F}$ is an instantiated function name and $N_1, \ldots, N_k$ are simple terms.

Recall that the order of a finitely typed term M is the order of its closure. Hence, M is *first-order simple* iff every variable occurring in M is of ground type *and* M does not mention instantiated function names.

Definition 6 (Iterative Form). A function definition $\mathcal{F} = M$ is in *iterative form* (or *left-linear form*) if

— either $M \equiv \lambda\mathbf{v}.\ N$, where N is left-linear,
— or $M \equiv \lambda\mathbf{v}.\ \mathbf{if}\ N\ \mathbf{then}\ P\ \mathbf{else}\ Q$, where N is first-order simple, and P and Q are left-linear.

A functional program $\{\mathcal{F}_i = M_i\}$ is in *iterative form* if every definition in it is in iterative form. The syntactic hierarchy of iterative programs is given by:

- ITER_n = { finitely typed iterative programs of order $\leqslant n$ }
- **ITER** = $\bigcup_{n \geqslant 1} \mathrm{ITER}_n$
- p-ITER = { polymorphically typed iterative programs }

It is tempting to generalize the definition of iterative form above, by relaxing the restriction on the first argument N of an outermost **if** on the right-hand side of a function definition. This can be done in two ways, the second being itself a generalization of the first:

1. N is a simple term in general, not restricted to first-order.
2. N is a left-linear term, not restricted to simple.

The first generalization is in fact just syntactic sugar; every program according to this generalization can be translated into an equivalent iterative program in our sense and of the same order (we do not prove this result here, but consider

Definition 7 and the analysis in Section 5 for suggestions). The second generalization does introduce a powerful feature, in that the resulting class of programs of order $\leqslant n$, for any fixed n, cannot be translated into our class of iterative programs of order $\leqslant n$, as illustrated by the next example. In any case, our definition of iterative programs is not too broad and certainly within the criteria of other definitions of iterative form in the literature.

Example 1. $\mathcal{P}$ is the following first-order functional program:

$$\begin{array}{rcl}
\mathcal{F}\ x & = & \text{SEARCH}\ x\ \varepsilon \\
\text{SEARCH}\ x\ y & = & \mathbf{if}\ \ (\mathbf{eq}\ x\ y)\ \ \mathbf{then}\ \ tt\ \ \mathbf{else}\ \ \text{NEXT}\ x\ y \\
\text{NEXT}\ x\ y & = & \mathbf{if}\ \ \text{LEFT}\ x\ y\ \ \mathbf{then}\ \ tt\ \ \mathbf{else}\ \ \text{RIGHT}\ x\ y \\
\text{LEFT}\ x\ y & = & \mathbf{if}\ \ (\mathbf{eq}\ (Ly)\ y)\ \ \mathbf{then}\ \ \mathit{ff}\ \ \mathbf{else}\ \ \text{SEARCH}\ x\ (Ly) \\
\text{RIGHT}\ x\ y & = & \mathbf{if}\ \ (\mathbf{eq}\ (Ry)\ y)\ \ \mathbf{then}\ \ \mathit{ff}\ \ \mathbf{else}\ \ \text{SEARCH}\ x\ (Ry)
\end{array}$$

The signature of the program is $\Sigma = \{\mathbf{eq}, L, R\}$ and ε is a ground constant. The types of the symbols in Σ, the variables x and y, and the constant ε are

$$\begin{array}{rcl}
\mathbf{eq} & : & * \to * \to \mathbf{bool} \\
x, y, \varepsilon & : & * \\
L, R & : & * \to *
\end{array}$$

The types of each of the function names in $\{\mathcal{F}, \text{SEARCH}, \text{NEXT}, \text{LEFT}, \text{RIGHT}\}$ is uniquely determined by the preceding types. $\mathcal{P}$ is not in iterative form because of the definition for NEXT, where the first argument of the **if** is left-linear rather than simple. It turns out that $\mathcal{P}$ is not equivalent to any iterative program of the same order (first-order). This is shown next.

When interpreted in an infinite treelike structure $\mathcal{A}$ – with ε taken as the root of the treelike universe of $\mathcal{A}$, L as the "left" operator, R as the "right" operator, and $\mathbf{eq}$ as the equality relation – $\mathcal{F}$ acquires the following meaning:

$$\mathcal{F}a = \begin{cases} tt, & \text{if } a \text{ occurs on, or to the left of,} \\ & \text{the leftmost infinite path in } \mathcal{A}; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

This can be verified by considering the intuitive meaning of the function SEARCH. The computation of (SEARCH x y) searches for x in the tree rooted at y, in left-to-right depth-first fashion. If the tree rooted at y is finite, (SEARCH x y) terminates whether or not x occurs in this finite tree. (LEFT x y) searches for x in the left subtree of node y, and (RIGHT x y) searches for x in the right subtree of node y.

There are infinite treelike structures where all first-order flowchart programs are said to "unwind", for example the structure T_ω in [28]. This implies that over such structures, any first-order iterative program is equivalent to one without recursion at all,[2] and cannot therefore be equivalent to $\mathcal{P}$ above. □

[2] Strictly speaking, a functional program without "non-trivial" recursion. A "trivial"

Definition 7 (Higher-Order Flowcharts). The translation of a recursive functional program into an iterative one, whether of the same order or of a higher order, is often easier by means of an intermediate translation to a flowchart. We only define finitely typed flowcharts; it is possible to define polymorphically typed flowcharts, but the definition is somewhat cumbersome and there is no real need for it later.

A *flowchart of order* $n \geqslant 1$ is a flowchart that may use variables of order $< n$ (these should perhaps be called *locations*). Assignment instructions in a flowchart are of the form

$$v := M$$

where v is a variable, M is a simple term, and v and M have the same finite type. The execution of the instruction "$v := M$" produces a *side-effect*, namely, the normal form of M is stored in the "location" v. Test instructions are of the form

$$(f \ v_1 \cdots v_k) \ ?$$

where $v_1, \ldots, v_k$ are variables of ground type $*$, and f is a relation symbol in the signature Σ of type $*^k \to \mathbf{bool}$.

The finitely many input variables of an order-n flowchart are of ground type $*$, while its single output variable is of ground type $*$ or **bool**. Free variables other than input variables appearing in test instructions or to the right of ":=" in assignment instructions must be each properly initialized, i.e. assigned a term whose only free variables are among the input variables or previously initialized variables. It would be tedious for us to define higher-order flowcharts with any greater formality; we instead refer the reader to [15, Section 1] for more details, as well as to Example 2 below and to later examples in the paper.

Proposition 8. *Every order-n flowchart is translatable into an equivalent order-n iterative functional program.*

Proof. Use the so-called McCarthy method to translate the order-n flowchart into an order-n iterative functional program. The McCarthy method was originally devised for the translation of ordinary (first-order) flowcharts. The generalization to higher orders is immediate; consult the original [18] or any of its textbook accounts, e.g. in [11, Section 5.3]. □

Another straightforward, though somewhat tedious, translation is the following: Every order-n iterative functional program is translatable into an order-n flowchart. This is done as in the first-order case [11, Section 5.4], by "uniformizing" the functional program (not done here). Thus, the class of order-n iterative functional programs is intertranslatable with the class of order-n flowcharts.

---

function definition is of the form $\mathcal{F} = \mathcal{F}$ or $\mathcal{F}\mathbf{v} = \mathcal{F}\mathbf{v}$, which makes the program diverge whenever the function $\mathcal{F}$ is called. Allowing the definition $\mathcal{F}\mathbf{v} = \mathcal{F}\mathbf{v}$ in a functional program, which is otherwise recursion-free, is very much like allowing self-loops in an otherwise loop-free flowchart.

Example 2. $\mathcal{P}$ is the following second-order functional program:

$$\begin{aligned} \mathcal{F}x &= \mathcal{G}fx \\ \mathcal{G}Fx &= \textbf{if} \quad px \quad \textbf{then} \quad gx\big(\mathcal{G}(\lambda y.F^{(2)}y)(Fx)\big) \quad \textbf{else} \quad hx \end{aligned}$$

where $F^{(2)}y$ is a short-hand notation for $(F\ (F\ y))$. The types of the symbols in the signature $\Sigma = \{p, f, h, g\}$ and the variables $\{x, y, F\}$ are

$$\begin{aligned} x, y &: * \\ p &: * \to \textbf{bool} \\ f, h, F &: * \to * \\ g &: * \to * \to * \end{aligned}$$

$\mathcal{P}$ is in linear but not iterative form. With a little work, we can show there is an order-2 iterative program that simulates $\mathcal{P}$; actually, we construct an order-2 flowchart for the simulation and then invoke Proposition 8 to show the existence of an equivalent order-2 iterative program. This is not a syntax-directed translation and relies on our understanding of what the program computes. Consider a converging computation of $\mathcal{F}$, where we leave some of the δ-redexes unevaluated:

$$\begin{aligned} \mathcal{F}x &\longrightarrow \mathcal{G}fx \\ &\longrightarrow gx\Big(\mathcal{G}(\lambda y.f^{(2)}y)(fx)\Big) \\ &\longrightarrow gx\Big(g(fx)\Big(\mathcal{G}(\lambda y.f^{(4)}y)(f^{(3)}x)\Big)\Big) \\ &\longrightarrow \cdots \\ &\ \vdots \\ &\longrightarrow gx\Big(g(fx)\Big(g(f^{(3)}x)\cdots\Big(g(f^{(2^{n-1}-1)}x)\Big(\mathcal{G}(\lambda y.f^{(2^n)}y)(f^{(2^n-1)}x)\Big)\Big)\cdots\Big)\Big) \\ &\longrightarrow gx\Big(g(fx)\Big(g(f^{(3)}x)\cdots\Big(g(f^{(2^{n-1}-1)}x)(h(f^{(2^n-1)}x))\Big)\cdots\Big)\Big) \end{aligned}$$

We have interrupted the reduction at the first term not containing a call to the function $\mathcal{G}$. The flowchart that simulates this reduction is not concerned about the evaluation of simple terms to their normal forms, but only about the sequence of calls to the function $\mathcal{G}$: How many times it is called and what are the arguments (or actual parameters) of every call. When an assignment instruction $v := M$ in the flowchart is executed, v is bound to the normal-form term obtained after carrying out all possible β-, **if**-, and δ-reductions in the simple term M.

Whereas the final term is constructed in "outside-in" fashion in the above reduction sequence, the equivalent flowchart (Figure 3) constructs the same term in "inside-out" fashion. For convenience in the flowchart, we allow simultaneous assignment instructions of the form $(v, w) := (M, N)$. This is a second-order flowchart.

We leave it to the reader to verify that the flowchart of Figure 3 simulates correctly the computation of the functional program. Observe that the flowchart has two distinct parts; the first starts with instruction labelled 1 and the second with instruction labelled 2. The first part construct the "inner" subterm $(h\ (f^{(2^n-1)}\ x))$ of the output term, and the second part constructs the rest of the output term in "inside-out" fashion. □

4 Paterson-Hewitt Revisited

The examples in this section are inspired by the Paterson-Hewitt paper [22]. They illustrate some of the evidence, as well as the difficulty, in translating a functional program into an iterative one of higher order.

Example 3. Consider the following first-order functional program:

$$\mathcal{F}xy \;=\; \textbf{if}\ px\ \textbf{then}\ y\ \textbf{else}\ g\ (\mathcal{F}(fx)(Ly))\ (\mathcal{F}(fx)(Ry))$$

where the signature is $\Sigma = \{p, L, R, f, g\}$, and the types are

$$\begin{aligned} x, y &: * \\ p &: * \to \mathbf{bool} \\ L, R, f &: * \to * \\ \mathcal{F}, g &: * \to * \to * \end{aligned}$$

In standard first-order notation, which makes clearer how to apply the "pebble game" (below), we can rewrite the above program as:

$$\mathcal{F}(x,y) \;=\; \textbf{if}\ p(x)\ \textbf{then}\ y\ \textbf{else}\ g(\mathcal{F}(fx, Ly), \mathcal{F}(fx, Ry))$$

$\mathcal{F}$ is not equivalent to any first-order iterative program. To explain why this is so, consider the sequence of terms:

$$M_0 \equiv y \qquad \text{and} \qquad M_{n+1} \equiv g(M_n[y := Ly], M_n[y := Ry])$$

for every $n \geqslant 0$. Thus, $M_1 \equiv g(Ly, Ry)$, $M_2 \equiv g(g(LLy, RLy), g(LRy, RRy))$, etc. A straightforward application of the pebble game shows that, for every $n \geqslant 1$, none of the terms in $\{M_k | k \geqslant n\}$ is accessible to the n-pebble game. Now, looking at the execution of $\mathcal{F}$, it calls itself recursively twice, k times for a total of 2^k recursive calls, where k is the least integer $\geqslant 0$ such that $p(f^{(k)}x) = tt$. If there is no such k, then $\mathcal{F}$ diverges, and if there is such k, then $\mathcal{F}$ converges and returns the term M_k. Thus, $\{M_k | k \geqslant 0\}$ is exactly the set of output terms of $\mathcal{F}$.

On the other hand, $\mathcal{F}$ can be translated into second-order iterative form, as follows. Define the sequence of terms:

$$F_0 \equiv \lambda z.z \qquad \text{and} \qquad F_{n+1} \equiv \lambda z.g(F_n(Lz))(F_n(Rz))$$

for every $n \geqslant 0$. It is easy to check that $M_n = F_n y$ for every $n \geqslant 0$. We can thus write the second-order flowchart shown in Figure 4, which is equivalent to $\mathcal{F}$ and where the type of the variable F is $* \to *$. In the form of a second-order iterative program, this flowchart becomes

$$\begin{aligned} \mathcal{F}_1 xy &= \mathcal{F}_2\ (\lambda z.z)\ x\ y \\ \mathcal{F}_2 Fxy &= \textbf{if}\ px\ \textbf{then}\ Fy\ \textbf{else}\ \mathcal{F}_2\ (\lambda z.g(F(Lz))(F(Rz)))\ (fx)\ y \end{aligned}$$

by a straightforward application of the McCarthy method (Proposition 8). □

Example 4. The following first-order functional program is from the Paterson-Hewitt paper [22]:

$$\mathcal{F}x \;=\; \textbf{if}\ px\ \textbf{then}\ x\ \textbf{else}\ g\ (\mathcal{F}(Lx))\ (\mathcal{F}(Rx))$$

where the signature is $\Sigma = \{p, L, R, g\}$, and the types of the symbols in Σ are those given in the preceding example. Using the pebble game, Paterson and Hewitt show that this functional program cannot be simulated by a first-order iterative program. We examine some of the difficulties of simulation. Consider the stopping condition for this program: The execution of $\mathcal{F}x$ converges iff every sufficiently long sequence $\alpha \in \{L, R\}^*$ has a suffix β such that $p(\beta x) = tt$ iff, more formally,

$$(\exists n \geqslant 0)\ (\forall \alpha \in \{L, R\}^n)\ (\exists \beta, \gamma \in \{L, R\}^*)\ [\ \alpha = \gamma\beta \ \text{ and } \ p(\beta x) = tt\ ]$$

If $\beta = d_k \cdots d_2 d_1 \in \{L, R\}^k$, our notation βx abbreviates $(d_k(\cdots(d_2(d_1 x))\cdots))$. Given a structure $\mathcal{A} = (A, p, L, R, g)$ define the function $\varphi(p, L, R)$ by:

$$\begin{aligned}&\varphi(p, L, R) \;=\\ &\mu n \geqslant 0\ [\ (\forall \alpha \in \{L, R\}^n)\ (\exists \beta, \gamma \in \{L, R\}^*)\ \ \alpha = \gamma\beta \ \text{ and } \ p(\beta x) = tt\ \ ]\end{aligned}$$

It is not difficult to write a first-order flowchart *augmented with counters*, call it $\mathcal{P}$, which can compute the value of $\varphi(p, L, R)$ — if it exists, otherwise $\mathcal{P}$ diverges. The signature of $\mathcal{P}$ is also $\{p, L, R\}$ (the function g is not used by $\mathcal{P}$). We omit the details of how to write $\mathcal{P}$. In Figure 5 we give the second-order flowchart, augmented with counters, which is equivalent to $\mathcal{F}$. The type of the variable F is $* \to *$. The instruction labeled $\mathcal{P}$ in Figure 5 is a macro, whose expansion is a first-order flowchart augmented with counters. □

Example 5. Consider again the Paterson-Hewitt program in Example 4. A CPS (continuation-passing-style) transformation "by hand" produces the following program:

$$\begin{aligned}\mathcal{F}'x \;&=\; \mathcal{G}'\ x\ (\lambda v.v)\\ \mathcal{G}'x\kappa \;&=\; \textbf{if}\ px\ \textbf{then}\ \kappa x\ \textbf{else}\ \mathcal{G}'\ (Lx)\ (\lambda v.\mathcal{G}'\ (Rx)\ (\lambda w.\kappa(gvw)))\end{aligned}$$

The resulting program is in so-called "tail form", but not iterative in our sense. The order of the entire program is 2, as the type of $\mathcal{G}'$ is $* \to (* \to *) \to *$. □

None of the techniques used in Examples 3, 4 and 5 can transform the Paterson-Hewitt program into an equivalent order-2 iterative program over the same signature $\Sigma = \{p, L, R, g\}$. The transformation developed in the next section, partly inspired by the translations in Examples 3 and 4, can.

5 From Recursion to Iteration

We restrict our analysis in this section to finitely typed programs, as we do not know how to extend it to polymorphically typed programs. Unless stated otherwise, all programs in this section are finitely typed. We deliberately omit details in several proofs in this section, as some of the ideas and arguments are lifted with only minor adjustments from denotational semantics and strictness analysis.

Let $\mathcal{P} = \{\mathcal{F}_i = M_i\}_{1 \leqslant i \leqslant \ell}$ be a functional program where $\mathcal{F}_1$ is the distinguished function name. The ℓ function definitions in $\mathcal{P}$ are mutually recursive. Suppose $\mathcal{F}_1 : \sigma_1, \ldots, \mathcal{F}_\ell : \sigma_\ell$ for some finite types $\sigma_1, \ldots, \sigma_\ell \in \mathbb{T}_{\text{fin}}$. As $\mathcal{F}_1$ is the distinguished function name, let $\sigma_1 \equiv *^k \to \rho$ for some $k \geqslant 1$ and some ground type ρ. We temporarily introduce a special symbol $\bot_\sigma : \sigma$ with the exhibited type for every $\sigma \in \mathbb{T}_{\text{fin}}$. Intuitively, $\bot_\sigma$ stands for "a diverging computation at type σ". We define a "flowchart" $\overline{\mathcal{P}}$ whose call-by-name semantics coincide with the call-by-name semantics of $\mathcal{P}$. $\overline{\mathcal{P}}$ is shown in Figure 6, where $x_1, \ldots, x_k$ are k input variables all of type $*$ and z is a fresh variable of type ρ

The construction of $\overline{\mathcal{P}}$ from $\mathcal{P}$ is suggested by standard methods in denotational semantics. $\overline{\mathcal{P}}$ basically evaluates the fixed point of the equation

$$(\mathcal{F}_1, \ldots, \mathcal{F}_\ell) = (M_1(\mathcal{F}_1, \ldots, \mathcal{F}_\ell), \ldots, M_\ell(\mathcal{F}_1, \ldots, \mathcal{F}_\ell))$$

by evaluating successive approximations that are increasingly finer. We write $M_i(\mathcal{F}_1, \ldots, \mathcal{F}_\ell)$ rather than just M_i to make explicit that M_i is a term that may mention some of the function names in $\{\mathcal{F}_1, \ldots, \mathcal{F}_\ell\}$. If on input $\mathbf{a} = a_1 \cdots a_k$, the call-by-name evaluation of $(\mathcal{F}_1\mathbf{a})$ terminates and returns a ground value b, then $\overline{\mathcal{P}}$ evaluates enough of this fixed point in order to return the same output b. Note that $\overline{\mathcal{P}}$ is not a legal flowchart in the sense of Definition 7, because $\overline{\mathcal{P}}$ mentions: (1) the artificial symbols $\bot_\sigma$, (2) function names $\mathcal{F}_i$ in assignment instructions, and (3) equality with $\bot_\rho$ in a test instruction — all features that are not allowed in flowcharts. But $\overline{\mathcal{P}}$ is a convenient description of a "parallel evaluation" of $\mathcal{P}$, which we explain further below, and suggests the later transition to iterative form.

Let the signature of $\mathcal{P}$ and $\overline{\mathcal{P}}$ be Σ_A, and assume there is a fixed Σ_A-structure $\mathcal{A}$ over which $\mathcal{P}$ and $\overline{\mathcal{P}}$ are interpreted. It is clear how to carry out the computation of $\overline{\mathcal{P}}$ relative to an interpretation $\langle \mathcal{A}, \mathbf{a} \rangle$, once we introduce a reduction rule ($\bot$-reduction) and modify the other rules to account for the presence of the special symbols $\bot_\sigma$. These are shown in Figure 8.

To explain more carefully the relationship between $\mathcal{P}$ and $\overline{\mathcal{P}}$, we first define an operation φ on terms relative to $\mathcal{P} = \{\mathcal{F}_i = M_i\}_{1 \leqslant i \leqslant \ell}$. (We should perhaps identify the dependence of φ on $\mathcal{P}$ by writing "$\varphi^{\mathcal{P}}$", for example, but "φ" is more economical.) We only consider terms N where all function names are from the set $\{\mathcal{F}_1, \ldots, \mathcal{F}_\ell\}$. For every such term N, we define $\varphi(0, N) = N$. Proceeding inductively, for arbitrary $n \geqslant 0$, we define

$$\varphi(n+1, N) \;=\; N[\mathcal{F}_1 := \varphi(n, M_1), \ldots, \mathcal{F}_\ell := \varphi(n, M_\ell)]$$

Intuitively, $\varphi(n, N)$ is obtained from N by "macro expanding" recursive calls to the functions $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ in parallel and repeatedly down to depth n.[3] For every

[3] This can be called the n-th *iterative* (or *inside-out*) *expansion* of N and corresponds

$n \geqslant 0$, we define the n-th *expansion of* $\mathcal{P}$ to be the following functional program:

$$\varphi(n,\mathcal{P}) \;=\; \{\mathcal{F}_1 = \varphi(n, M_1),\; \dots ,\mathcal{F}_\ell = \varphi(n, M_\ell)\}$$

The 0-th expansion $\varphi(0,\mathcal{P})$ is none other than $\mathcal{P}$. We define another operation ψ on terms N containing only function names in $\{\mathcal{F}_1, \dots ,\mathcal{F}_\ell\}$, given by:

$$\psi(n, N) \;= \varphi(n, N)[\mathcal{F}_1 := \bot_{\sigma_1}, \dots ,\mathcal{F}_\ell := \bot_{\sigma_\ell}]$$

Note $\psi(n, N)$ is a simple term (it does not contain any function names). Returning to the flowchart $\overline{\mathcal{P}}$, the vector

$$\big(\psi(n, M_1), \dots ,\psi(n, M_\ell)\big)$$

can be viewed as the *state* of $\overline{\mathcal{P}}$ after its n-th iteration is completed. The following lemmas explain the connection between $\mathcal{P}$ and $\overline{\mathcal{P}}$.

Lemma 9. *Let $\mathcal{P}$ be a functional program as defined above, and $\langle\mathcal{A}, \mathbf{a}\rangle$ an interpretation for $\mathcal{P}$.*

1. *For every $n \geqslant 0$, the n-th expansion $\varphi(n,\mathcal{P})$ is equivalent to $\mathcal{P}$.*
2. *If the evaluation of $(\mathcal{F}_1\mathbf{a})$ converges, i.e. $(\mathcal{F}_1\mathbf{a}) \twoheadrightarrow b$ for some ground constant b of type ρ, then there is $n \geqslant 0$ such that $\delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, M_1\mathbf{a})\big) = b$.*
3. *If the evaluation of $(\mathcal{F}_1\mathbf{a})$ diverges, then $\delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, M_1\mathbf{a})\big)$ is not a simple term for every $n \geqslant 0$.*

Proof. A rigorous proof of part 1 uses standard techniques of denotational semantics, omitted here. See [14, Chapter 5] or [30, Chapters 9 and 11], among other texts. Parts 2 and 3 follow from general principles on the relationship between $\mathcal{P}$ and its successive expansions: If the evaluation of $(\mathcal{F}_1\mathbf{a})$ converges to b, then for a sufficiently large expansion $\mathcal{P}'$ of $\mathcal{P}$, say

$$\mathcal{P}' \;=\; \varphi(n,\mathcal{P}) \;=\; \{\mathcal{F}'_1 = \varphi(n, M_1),\; \dots ,\mathcal{F}'_\ell = \varphi(n, M_\ell)\}$$

where n is very large and $\mathcal{F}_i$ is replaced by $\mathcal{F}'_i$ throughout, the evaluation of $(\mathcal{F}'_1\mathbf{a})$ or, equivalently, $\varphi(n, M_1\mathbf{a})$ to b does not involve any $\mathcal{P}'$-reduction step, in which case it suffices to use δ, **if** and β reduction steps only. Details omitted. ◻

Lemma 10. *If N is a closed term of ground type ρ, then for every $n \geqslant 0$:*

to the way $\overline{\mathcal{P}}$ works. We can denote it $\varphi_{\mathrm{io}}(n, N)$ to distinguish it from the n-th *recursive* (or *outside-in*) *expansion* of N, which is defined by: $\varphi_{\mathrm{oi}}(0, N) = N$ and $\varphi_{\mathrm{oi}}(n + 1, N) = \varphi_{\mathrm{oi}}(n, N)[\mathcal{F}_1 := M_1, \dots ,\mathcal{F}_\ell := M_\ell]$ for every $n \geqslant 0$. A recursive expansion of N corresponds to the way reductions are carried out: Going from $\varphi_{\mathrm{oi}}(n, N)$ to $\varphi_{\mathrm{oi}}(n+1, N)$ can be expressed as a one-step parallel reduction, because $\varphi_{\mathrm{oi}}(n+1, N)$ is obtained by simultaneous reduction of all $\mathcal{P}$-redexes in $\varphi_{\mathrm{oi}}(n, N)$. It is a non-trivial exercise, omitted here, to show that $\varphi_{\mathrm{oi}}(n, N) = \varphi_{\mathrm{io}}(n, N)$ for every $n \geqslant 0$.

1. *If $\delta, \mathbf{if}, \beta$-nf$\big(\varphi(n,N)\big)$ is a simple term, then*

$$\bot, \delta_\bot, \mathbf{if}_\bot, \beta\text{-nf}\big(\psi(n,N)\big) = \delta, \mathbf{if}, \beta\text{-nf}\big(\psi(n,N)\big) = \delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n,N)\big) = b$$

for some ground constant b of type ρ.

2. *If $\delta, \mathbf{if}, \beta$-nf$\big(\varphi(n,N)\big)$ is not a simple term, then*

$$\bot, \delta_\bot, \mathbf{if}_\bot, \beta\text{-nf}\big(\psi(n,N)\big) = \bot_\rho$$

Proof. As $\bot$-reduction, $\delta_\bot$-reduction and $\mathbf{if}_\bot$-reduction do not introduce β-redexes, carry out all β-reduction steps first in both $\varphi(n,N)$ and $\psi(n,N)$. The resulting terms P and Q in β-normal form are such that

$$Q \;\equiv\; P[\mathcal{F}_1 := \bot_{\sigma_1}, \ldots, \mathcal{F}_\ell := \bot_{\sigma_\ell}]$$

Then carry out all δ-reduction and $\mathbf{if}$-reduction steps (not $\delta_\bot$-reduction and $\mathbf{if}_\bot$-reduction steps) in both P and Q. The resulting P' and Q' in $\delta, \mathbf{if}, \beta$-normal form are such that:

$$Q' \;\equiv\; P'[\mathcal{F}_1 := \bot_{\sigma_1}, \ldots, \mathcal{F}_\ell := \bot_{\sigma_\ell}]$$

Q' is a simple term, possibly containing the special symbols $\bot_\sigma$. The argument so far does not use the hypothesis that N is closed of ground type. With this hypothesis, Q' can be $\bot, \delta_\bot, \mathbf{if}_\bot$-reduced to $\bot_\rho$ or else to a ground constant b of type ρ. In the first case, P' and therefore $\varphi(n,N)$ are not simple terms. In the second case, it must be that $P' \equiv Q' \equiv b$. □

Lemma 11. *Consider the computation of $\mathcal{P}$ over an interpretation $\langle \mathcal{A}, a_1, \ldots, a_k\rangle$.*

1. *If $(\mathcal{F}_1 a_1 \cdots a_k) \twoheadrightarrow b$ where b is a ground constant of type ρ, then there is $n \geqslant 0$ such that $\bot, \delta_\bot, \mathbf{if}_\bot, \beta$-nf$\big(\psi(n, M_1 a_1 \cdots a_k)\big) = b$.*
2. *If $(\mathcal{F}_1 a_1 \cdots a_k)$ diverges, then $\bot, \delta_\bot, \mathbf{if}_\bot, \beta$-nf$\big(\psi(n, M_1 a_1 \cdots a_k)\big) = \bot_\rho$ for every $n \geqslant 0$.*

This establishes that $\overline{\mathcal{P}}$ simulates $\mathcal{P}$ correctly, because if $\overline{\mathcal{P}}$ is executed over the interpretation $\langle \mathcal{A}, a_1, \ldots, a_k\rangle$, then $\bot, \delta_\bot, \mathbf{if}_\bot, \beta$-*nf*$\big(\psi(n, M_1 a_1 \cdots a_k)\big)$ is exactly the value assigned to variable z at the $(n+1)$-st iteration of the loop, for $n \geqslant 0$.

Proof. This is an immediate consequence of the two preceding lemmas. □

In the form of a functional program, again not legal in our formalism, $\overline{\mathcal{P}}$ can be written as

$$\begin{array}{rcl} \mathcal{G}\,\mathbf{x} & = & \mathcal{H}\,\mathbf{x}\;\bot_\rho\;\bot_{\sigma_1}\cdots\bot_{\sigma_\ell} \\ \mathcal{H}\,\mathbf{x}\;z\;F_1\cdots F_\ell & = & \mathbf{if}\quad (z \neq \bot_\rho)\quad \mathbf{then}\quad (F_1\mathbf{x})\quad \mathbf{else}\quad \mathcal{H}\,\mathbf{x}\;(F_1\mathbf{x})\;M_1^\bullet\cdots\;M_\ell^\bullet \end{array}$$

by a straightforward application of the McCarthy method (Proposition 8), where $\mathbf{x} = x_1 \cdots x_k$ and $M_i^\bullet$ is M_i with $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ replaced by $F_1, \ldots, F_\ell$, respectively, and $\mathcal{G}$ and $\mathcal{H}$ are fresh function names with the following types:

$$\begin{array}{rcl} \mathcal{G} & : & *^k \to \rho \\ \mathcal{H} & : & *^k \to \rho \to \sigma_1 \to \cdots \to \sigma_\ell \to \rho \end{array}$$

The challenge here is how to turn $\overline{\mathcal{P}}$ into a legal functional program, in particular how to simulate the artificial symbols $\bot_\sigma$. For this, we first transform $\mathcal{P}$ into $(\!(\mathcal{P})\!)$, which incorporates an appropriate simulation of $\bot_\sigma$. In a second stage, we transform $(\!(\mathcal{P})\!)$ into $iter(\mathcal{P})$, the final iterative form of $\mathcal{P}$.

The transformation $(\!(\ )\!)$ is defined in Figure 9. If $\mathcal{P}$ mentions object variable v (resp. function name $\mathcal{F}$), then $(\!(\mathcal{P})\!)$ mentions both v (resp. $\mathcal{F}$) and a fresh object variable $\underline{v}$ (resp. fresh function name $\underline{\mathcal{F}}$). If $\mathcal{P} = \{\mathcal{F}_i = M_i\}_{1\leqslant i\leqslant \ell}$ then $(\!(\mathcal{P})\!)$ is the program:

$$(\!(\mathcal{P})\!) \;=\; \{\ \mathcal{F}_i = M_i\ \}_{1\leqslant i\leqslant \ell}\ \cup\ \{\ \underline{\mathcal{F}_i} = (\!(M_i)\!)\ \}_{1\leqslant i\leqslant \ell}$$

Note that $\mathcal{P}$ and $(\!(\mathcal{P})\!)$ are over the same signature. It is clear what $\varphi(n, (\!(\mathcal{N})\!))$ means for an arbitrary $n \geqslant 0$ and arbitrary term N containing function names only in $\{\mathcal{F}_1, \underline{\mathcal{F}_1}, \ldots, \mathcal{F}_\ell, \underline{\mathcal{F}_\ell}\}$. We define one more operation ψ' on such a term N:

$$\psi'(n, (\!(\mathcal{N})\!)) \;=\; \varphi(n, (\!(\mathcal{N})\!))[\mathcal{F}_1 := c_{\sigma_1}, \underline{\mathcal{F}_1} := \Omega_{\sigma_1}, \ldots, \mathcal{F}_\ell := c_{\sigma_\ell}, \underline{\mathcal{F}_\ell} := \Omega_{\sigma_\ell}]$$

where Ω_σ is defined Figure 9, and if $\sigma \equiv \tau_1 \to \cdots \to \tau_k \to \rho$ with $k \geqslant 1$ and $\rho \in \{*, \mathbf{bool}\}$, then c_σ is defined as

$$c_\sigma \;\equiv\; (\lambda v_1 : \tau_1.\ \cdots .\lambda v_k : \tau_k.\ a)$$

where a is an arbitrary, but otherwise fixed throughout the rest of this section, ground constant of type ρ. The following lemmas relate $\mathcal{P}$ and $(\!(\mathcal{P})\!)$.

Lemma 12. *If N is a closed term of ground type ρ, then for every $n \geqslant 0$:*

1. *If $\delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, N)\big)$ is a simple term, then $\delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, (\!(N)\!))\big) = tt$.*
2. *If $\delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, N)\big)$ is not a simple term, then neither is $\delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, (\!(N)\!))\big)$.*

Proof. First prove that if $P \longrightarrow Q$ then $(\!(P)\!) \twoheadrightarrow (\!(Q)\!)$, by considering each kind of reduction step separately: δ, **if** and β. Details omitted. This implies that

$$\delta, \mathbf{if}, \beta\text{-}nf\big(\varphi(n, (\!(N)\!))\big) \;=\; (\!(\delta, \mathbf{if}, \beta\text{-}nf\big(\varphi(n, N)\big))\!)$$

The two parts of the lemma now follow from the fact that if $\delta, \mathbf{if}, \beta\text{-}nf\big(\varphi(n, N)\big)$ is simple, then it must be a ground constant b of type ρ. □

Lemma 13. *If N is a closed term of ground type ρ, then for every $n \geqslant 0$:*

1. *If $\delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, (\!(N)\!))\big)$ is simple, then* $\delta, \mathbf{if}, \beta\text{-nf}\big(\psi'(n, (\!(N)\!))\big) = \delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, (\!(N)\!))\big) = tt$.
2. *If $\delta, \mathbf{if}, \beta\text{-nf}\big(\varphi(n, (\!(N)\!))\big)$ is not simple, then $\delta, \mathbf{if}, \beta\text{-nf}\big(\psi'(n, (\!(N)\!))\big) = ff$.*

Proof. This is a consequence of the definition of $\psi'(n, (\!(N)\!))$ from $\varphi(n, (\!(N)\!))$. The details are similar to the details in the proof of Lemma 10. We only give an outline. As δ-reduction and **if**-reduction do not introduce β-redexes, carry out all β-reduction steps first in both $\varphi(n, (\!(N)\!))$ and $\psi'(n, (\!(N)\!))$. The resulting terms P and Q in β-normal form are such that

$$Q \equiv P[\mathcal{F}_1 := c_{\sigma_1}, \underline{\mathcal{F}_1} := \Omega_{\sigma_1}, \ldots, \mathcal{F}_\ell := c_{\sigma_\ell}, \underline{\mathcal{F}_\ell} := \Omega_{\sigma_\ell}]$$

treating every c_σ, resp. Ω_σ, as a formal constant of type σ, resp. $(\!(\sigma)\!)$. Then carry out all δ-reduction and **if**-reduction steps in both P and Q. The resulting P' and Q' in $\delta, \mathbf{if}, \beta$-normal form are such that:

$$Q' \equiv P'[\mathcal{F}_1 := c_{\sigma_1}, \underline{\mathcal{F}_1} := \Omega_{\sigma_1}, \ldots, \mathcal{F}_\ell := c_{\sigma_\ell}, \underline{\mathcal{F}_\ell} := \Omega_{\sigma_\ell}]$$

Q' is a simple term, possibly containing the special symbols c_σ and Ω_σ. The argument so far does not use the hypothesis that N is closed of ground type. With this hypothesis, and after macro expanding each c_σ and each Ω_σ, Q' can be $\delta, \mathbf{if}, \beta$-reduced to *tt* or *ff*. In the first case, it must be that $P' \equiv Q' \equiv tt$. In the second case, P' and therefore $\varphi(n, (\!(N)\!))$ are not simple terms. □

Lemma 14. *Consider the computation of $\mathcal{P}$ over an interpretation $\langle \mathcal{A}, a_1, \ldots, a_k \rangle$.*

1. *If $(\mathcal{F}_1 a_1 \cdots a_k) \twoheadrightarrow b$ where b is a ground value of type ρ, then there is $n \geqslant 0$ such that $\delta, \mathbf{if}, \beta\text{-nf}\big(\psi'(n, (\!(M_1)\!)\ tt\ a_1 \cdots tt\ a_k)\big) = tt$.*
2. *If $(\mathcal{F}_1 a_1 \cdots a_k)$ diverges, then $\delta, \mathbf{if}, \beta\text{-nf}\big(\psi'(n, (\!(M_1)\!)\ tt\ a_1 \cdots tt\ a_k)\big) = ff$ for every $n \geqslant 0$.*

Proof. Immediate consequence of Lemma 9, 12 and 13. □

The earlier construction of $\overline{\mathcal{P}}$ from $\mathcal{P}$ suggests a similar construction of "flowchart" $\overline{(\!(\mathcal{P})\!)}$ from $(\!(\mathcal{P})\!)$, shown in Figure 7, the correctness of which follows from Lemma 14. Note that if $\overline{(\!(\mathcal{P})\!)}$ is executed relative to the interpretation $\langle \mathcal{A}, a_1, \ldots, a_k \rangle$, then

$$\delta, \mathbf{if}, \beta\text{-}nf\big(\psi'(n, (\!(M_1)\!)\ tt\ a_1 \cdots tt\ a_k)\big)$$

is precisely the value assigned to variable z at the $(n+1)$-st iteration of the loop of $\overline{(\!(\mathcal{P})\!)}$, for every $n \geqslant 0$.

In the form of a functional program, $\overline{(\!(\mathcal{P})\!)}$ is basically the desired iterative form of $\mathcal{P}$, denoted *iter*($\mathcal{P}$). The formal definition of *iter*($\mathcal{P}$) is given next.

Definition 15 (Transformation to Iterative Form). Let $\mathcal{P}$ be the finitely typed functional program $\{\mathcal{F}_i = M_i\}_{1 \leqslant i \leqslant \ell}$ where:

— $\mathcal{F}_1 : \sigma_1, \ldots, \mathcal{F}_\ell : \sigma_\ell$ for some $\sigma_1, \ldots, \sigma_\ell \in \mathbb{T}_{\text{fin}}$,
— $\mathcal{F}_1$ is the distinguished function name and $\sigma_1 \equiv *^k \to \rho$ for some $k \geqslant 1$ and $\rho \in \{*, \mathbf{bool}\}$.

The iterative form *iter*($\mathcal{P}$) of $\mathcal{P}$ is the following functional program:

$$\begin{aligned} \mathcal{G}\,\mathbf{x} &= \mathcal{H}\,\mathbf{x}\,ff\,c_{\sigma_1}\ \Omega_{\sigma_1} \cdots c_{\sigma_\ell}\ \Omega_{\sigma_\ell} \\ \mathcal{H}\,\mathbf{x}\ z\ F_1\ \underline{F_1} \cdots F_\ell\ \underline{F_\ell} &= \mathbf{if}\ \ z\ \ \mathbf{then}\ \ (F_1\ x_1 \cdots x_k)\ \ \mathbf{else} \\ &\quad\ \mathcal{H}\,\mathbf{x}\ (\underline{F_1}\ tt\,x_1 \cdots tt\,x_k)\ M_1^\bullet\ (\!(M_1)\!)^\bullet \cdots\ M_\ell^\bullet\ (\!(M_\ell)\!)^\bullet \end{aligned}$$

where

1. $\mathbf{x} = x_1 \cdots x_k$.
2. $F_1 : \sigma_1,\ \underline{F_1} : (\!(\sigma_1)\!), \ldots,\ F_\ell : \sigma_\ell,\ \underline{F_\ell} : (\!(\sigma_\ell)\!)$ are fresh variables.
3. $M_i^\bullet$ is M_i with $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ replaced by $F_1, , \ldots, F_\ell$, respectively.
4. $(\!(M_i)\!)^\bullet$ is $(\!(M_i)\!)$ with $\mathcal{F}_1, \underline{\mathcal{F}_1}, \ldots, \mathcal{F}_\ell, \underline{\mathcal{F}_\ell}$ replaced by $F_1, \underline{F_1}, \ldots, F_\ell, \underline{F_\ell}$.
5. $\mathcal{G}$ and $\mathcal{H}$ are fresh function names with the following types:

$$\begin{aligned} \mathcal{G} &: *^k \to \rho \\ \mathcal{H} &: *^k \to \mathbf{bool} \to \sigma_1 \to (\!(\sigma_1)\!) \to \cdots \to \sigma_\ell \to (\!(\sigma_\ell)\!) \to \rho \end{aligned}$$

Theorem 16. *Every finitely typed functional program $\mathcal{P}$ of order $n \geqslant 1$ can be translated into an equivalent iterative program* iter$(\mathcal{P})$ *of order $n+1$.*

Proof. Let $\mathcal{P} = \{\mathcal{F}_i = M_i\}_{1 \leqslant i \leqslant \ell}$ as in Definition 15. The equivalence of $\mathcal{P}$ and $iter(\mathcal{P})$ follows from the preceding discussion. To check that $iter(\mathcal{P})$ is in iterative form is a straightforward inspection of its definition. Finally, if $order(\mathcal{P}) = n$, we can take:

$$n \;=\; \max\{order(\mathcal{F}_1), \ldots, order(\mathcal{F}_\ell)\} \;=\; \max\{order(\sigma_1), \ldots, order(\sigma_\ell)\}$$

See Definition 3. This implies:

$$\begin{aligned} order(iter(\mathcal{P})) &= order(\mathcal{H}) \\ &= order(*^k \to \mathbf{bool} \to \sigma_1 \to (\!(\sigma_1)\!) \to \cdots \to \sigma_\ell \to (\!(\sigma_\ell)\!) \to \rho) \\ &= n+1 \end{aligned}$$

The last equality (to $n+1$) is immediate from the definition of *order*. □

Example 6. Consider again the Paterson-Hewitt program $\mathcal{P}$ in Example 4. $\mathcal{P}$ consists of a single function definition $\{\mathcal{F} = M\}$, where

$$M \equiv \big(\lambda x.\mathbf{if}\ \ px\ \ \mathbf{then}\ \ x\ \ \mathbf{else}\ \ g(\mathcal{F}(Lx))(\mathcal{F}(Rx))\big)$$

The functional program $iter(\mathcal{P})$ is the following second-order iterative program:

$$\begin{aligned} \mathcal{G}x &= \mathcal{H}\ x\ \mathit{ff}\ (\lambda v.x)\ (\lambda \underline{w}.\lambda w.\mathit{ff}) \\ \mathcal{H}xzF\underline{F} &= \mathbf{if}\ \ z\ \ \mathbf{then}\ \ Fx\ \ \mathbf{else}\ \ \mathcal{H}\ x\ (\underline{F}\ tt\ x)\ N\ P \end{aligned}$$

where $\mathcal{G}$ and $\mathcal{H}$ are fresh function names with the following types:

$$\begin{aligned} \mathcal{G} &: * \to * \\ \mathcal{H} &: * \to \mathbf{bool} \to (* \to *) \to (\mathbf{bool} \to * \to \mathbf{bool}) \to * \end{aligned}$$

and N and P are the following terms:

$$\begin{aligned} N &\equiv \lambda x.\ \mathbf{if}\ px\ \mathbf{then}\ x\ \mathbf{else}\ g(F(Lx))(F(Rx)) \\ P &\equiv \lambda \underline{x}.\lambda x.\ \mathbf{if}\ \underline{x}\ \mathbf{then}\ \Big(\mathbf{if}\ px\ \mathbf{then}\ \underline{x}\ \mathbf{else}\ \big(\mathbf{and}\ (\underline{F}\ \underline{x}\ (Lx))\ (\underline{F}\ \underline{x}\ (Rx))\big)\Big)\ \mathbf{else}\ \mathit{ff} \end{aligned}$$

N is simply M with function name $\mathcal{F}$ replaced by variable F, and P is $(\!(N)\!)$. □

Open Problem. Extend the transformation $(\!(\ )\!)$ to polymorphically typed functional programs. Define the iterative form $iter(\mathcal{P})$ of a polymorphically typed $\mathcal{P}$ based on this extension of $(\!(\ )\!)$.

6 Polymorphic Iteration

We prove that there are polymorphically typed iterative programs that cannot be translated into equivalent finitely typed functional programs. We start with an example of a polymorphically typed iterative program $\mathcal{P}$, which we use in the proof of Theorem 17.

Example 7. $\mathcal{P}$ is the following polymorphically typed functional program:

$$\begin{aligned} \mathcal{F}x &= \mathcal{G}xfa \\ \mathcal{G}xFZ &= \textbf{if} \ \ rx \ \ \textbf{then} \ \ FZ \ \ \textbf{else} \ \ \mathcal{G}(fx)(\lambda v.\lambda w.v(vw))FZ \end{aligned}$$

The types of the symbols in the signature $\Sigma = \{f, r\}$, the ground constant a, the variables $\{x, v, w, F, Z\}$, and the function names $\{\mathcal{F}, \mathcal{G}\}$ are

$$\begin{aligned} a, x &: * \\ f &: * \to * \\ r &: * \to \textbf{bool} \\ v, F &: \alpha \to \alpha \\ w, Z &: \alpha \\ \mathcal{F} &: * \to * \\ \mathcal{G} &: * \to (\alpha \to \alpha) \to \alpha \to \alpha \end{aligned}$$

The instance of $\mathcal{G}$ in the definition for $\mathcal{F}$ has type

$$* \to (* \to *) \to * \to *$$

The instance of $\mathcal{G}$ in the definition for $\mathcal{G}$ has type

$$* \to ((\alpha \to \alpha) \to (\alpha \to \alpha)) \to (\alpha \to \alpha) \to (\alpha \to \alpha)$$

Noting that $\mathcal{P}$ is in iterative form, what we have in this example is a case of *polymorphic iteration.* If n is the least natural number such that $r(f^{(n)}x)$ is *tt*, we have the following converging computation:

$$\begin{aligned} \mathcal{F}x &\twoheadrightarrow \mathcal{G}\ x\ f\ a \\ &\twoheadrightarrow \mathcal{G}\ (fx)\ \mathbf{2}\ f\ a \\ &\twoheadrightarrow \mathcal{G}\ (f^{(2)}x)\ \mathbf{2}\ \mathbf{2}\ f\ a \\ &\twoheadrightarrow \cdots \\ &\ \ \vdots \\ &\twoheadrightarrow \mathcal{G}\ (f^{(n)}x)\ \underbrace{\mathbf{2}\ \cdots\ \mathbf{2}}_{n}\ f\ a \\ &\twoheadrightarrow \underbrace{\mathbf{2}\ \cdots\ \mathbf{2}}_{n}\ f\ a \equiv \underbrace{f(f\cdots(f\ a)\cdots)}_{e(n)} \equiv (f^{(e(n))}\ a) \end{aligned}$$

where $\mathbf{2}$ denote the term $(\lambda v.\lambda w.v(vw))$, and the function e is given by $e(0) = 1$ and $e(n+1) = 2^{e(n)}$ for all $n \geqslant 0$. Each call to $\mathcal{G}$ in the computation is at a different type. An explicitly typed intermediate term in this computation is (types inserted as superscripts):

$$\mathcal{G}^{*\to*[k+1]\to*[k+1]}\ (f^{(k)}x)^{*}\ \mathbf{2}^{*[k+1]}\ \mathbf{2}^{*[k]}\ \cdots\ \mathbf{2}^{*[2]}\ f^{*[1]}\ a^{*}$$

where $0 \leqslant k \leqslant n$ and we use the following type abbreviations: $*[0] = *$ and $*[k+1] = (*[k] \to *[k])$. Note that because the program is polymorphic, $\mathcal{G}$ is repeatedly applied to finitely typed arguments with increasingly complex types. This example is adapted from [15, Example 5.3]. □

Theorem 17. *There is a polymorphically typed iterative program $\mathcal{P}$ which is not equivalent to any finitely typed functional program.*

Proof. [4] The desired $\mathcal{P}$ is the program given in Example 7. It suffices to show for every finitely typed functional program $\mathcal{Q}$ of order $n \geqslant 1$, over the signature Σ of $\mathcal{P}$, there is a Σ_A-structure $\mathcal{A}$ such that $\mathcal{P}^{\mathcal{A}} \neq \mathcal{Q}^{\mathcal{A}}$. We choose $\mathcal{A}$ to be of the form $\langle\{a_1,\dots,a_u\} \cup \mathbb{N}, r, f\rangle$ where u is a positive integer (to be appropriately selected depending on $\mathcal{Q}$), $\mathbb{N}$ is the set of natural numbers, r is the predicate such that $(r\ x) = tt$ iff $x = a_u$, and f is the function: $(f\ a_i) = a_{i+1}$ for $1 \leqslant i < u$, $(f\ a_u) = a_u$, and $(f\ i) = i+1$ for every $i \in \mathbb{N}$.

$\mathcal{P}$ mentions only one ground constant, namely a, while $\mathcal{Q}$ may mention a as well as several other ground constants. We choose a to be 0 in $\mathbb{N}$, and we can always define $\mathcal{A}$ so that all the other ground constants mentioned by $\mathcal{Q}$ are in $\{a_1,\dots,a_u\}$. No matter what the value of u is, the computation of $\mathcal{P}$ over the interpretation $\langle\mathcal{A}, a_1\rangle$ converges and returns the value:

$$(f^{(e(u))}\ a) \;=\; (f^{(e(u))}\ 0) \;=\; e(u)$$

by the analysis in Example 7. We need to select u large enough so that the computation of $\mathcal{Q}$, over the same interpretation $\langle\mathcal{A}, a_1\rangle$, either diverges or converges and returns a value $\neq e(u)$. By Theorem 16, we can restrict $\mathcal{Q}$ to be in iterative form. We can therefore write $\mathcal{Q}$ in the form of an order-n flowchart consisting of, say, $k \geqslant 1$ instructions and $\ell \geqslant 1$ variables, all of order $< n$. As there is no "communication" between the elements in $\{a_1,\dots,a_u\}$ and the elements in $\mathbb{N}$, the behavior of $\mathcal{Q}$ is entirely determined by the substructure $\mathcal{B} = \langle\{a_1,\dots,a_u\}, r, f\rangle$, i.e. it is not difficult to check that the computation of $\mathcal{Q}$ over the interpretation $\langle\mathcal{A}, a_1\rangle$ converges iff the computation of $\mathcal{Q}$ over the interpretation $\langle\mathcal{B}, a_1\rangle$ converges.

Define the function exp such that $exp(0,n) = n$ and $exp(m+1,n) = 2^{exp(m,n)}$, for all $m, n \in \mathbb{N}$. It is not difficult to show there is a fixed polynomial $\zeta : \mathbb{N} \to \mathbb{N}$ such that if $v : \sigma$ is a variable in $\mathcal{Q}$ and $order(\sigma) = m \geqslant 0$, then v will be assigned at most $exp(m, \zeta(u))$ distinct values, i.e. functions of type σ over the universe $\{a_1,\dots,a_u\}$, in the course of the computation of $\mathcal{Q}$ over the interpretation $\langle\mathcal{B}, a_1\rangle$. If a *state* of this computation consists of an instruction label (k of them) in $\mathcal{Q}$, together with the values assigned to the ℓ variables in $\mathcal{Q}$, then $k \cdot \big(exp(n, \zeta(u))\big)^{\ell}$ is an upper bound on the number of distinct states

[4] Joint with Paweł Urzyczyn.

that $\mathcal{Q}$ can visit in the course of its computation. If a state is repeated, the computation is doomed to diverge. Noting that k, ℓ and n depend on $\mathcal{Q}$ and are therefore fixed in this proof, let $\xi(u) = k \cdot \big(exp(n, \zeta(u))\big)^{\ell}$. Suppose $z : *$ is a variable in $\mathcal{Q}$ which is assigned the final output, if any. The value assigned to z can be changed at most $\xi(u)$ times in the course of a converging computation of $\mathcal{Q}$, now over $\langle \mathcal{A}, a_1 \rangle$. Using the fact that a finitely-typed simple term M reduces to normal form in at most $exp(|M|, |M|)$ steps, see [7, 8, 26], the final value assigned to z cannot exceed $exp(p \cdot \xi(u), p \cdot \xi(u))$ for some $p \in \mathbb{N}$ depending on $\mathcal{Q}$. The desired conclusion now follows, because $e(u) > exp(p \cdot \xi(u), p \cdot \xi(u))$ for sufficiently large u. □

7 Hierarchies

Let $\mathbf{C}_1$ and $\mathbf{C}_2$ be classes of functional programs over the same signature. Recall that the *equivalence of programs* is relative to their call-by-name semantics (Definition 4). We write $\mathbf{C}_1 \leqslant \mathbf{C}_2$ if:

$$(\forall \mathcal{P}_1 \in \mathbf{C}_1)\ (\exists \mathcal{P}_2 \in \mathbf{C}_2)\ [\ \mathcal{P}_1 \text{ and } \mathcal{P}_2 \text{ are equivalent }\ ]$$

We write $\mathbf{C}_1 < \mathbf{C}_2$ iff $\mathbf{C}_1 \leqslant \mathbf{C}_2$ but $\mathbf{C}_2 \nleqslant \mathbf{C}_1$.

The results of Sections 5 and 6 do not presume that the signature Σ of programs contains a symbol $\mathbf{eq} : *^2 \to \mathbf{bool}$ which is always interpreted as the equality relation on the universe A of a Σ_A-structure. To compare the hierarchies $\{\text{ITER}_n\}$ and $\{\text{REC}_n\}$, and use the results of [15], we now assume the existence of such a symbol $\mathbf{eq}$. It is easy to see that Theorems 16 and 17 imply the following computational hierarchy:

$$\text{ITER}_1 < \text{REC}_1 \leqslant \text{ITER}_2 \leqslant \text{REC}_2 \leqslant \text{ITER}_3 \leqslant \text{REC}_3 \leqslant \cdots < \text{p-ITER} \leqslant \text{p-REC}$$

The strictness of the first level, $\text{ITER}_1 < \text{REC}_1$, is the classical result of Paterson and Hewitt [22]. Using the already known fact that $\text{REC}_n < \text{REC}_{n+1}$ for every $n \geqslant 1$, see [15, Theorem 3.9], we can conclude:

Theorem 18. $\text{ITER}_1 < \text{ITER}_3 < \text{ITER}_5 < \cdots < \mathbf{ITER} < \text{p-ITER}$ (odd orders)

Theorem 19. $\text{ITER}_2 < \text{ITER}_4 < \text{ITER}_6 < \cdots < \mathbf{ITER} < \text{p-ITER}$ (even orders)

Although we are not yet able to separate two consecutive levels in this hierarchy, we have thus shown that increasing the order of finitely-typed iterative programs results in a net gain of computational power, and adding polymorphic iteration results in a further gain of computational power. Based on the preceding, it is only natural to state the following.

Conjecture. $\text{ITER}_1 < \text{ITER}_2 < \text{ITER}_3 < \text{ITER}_4 < \cdots$ (all orders)

To settle this conjecture, we can proceed in one of two ways to prove that consecutive levels of the hierarchy $\{\text{ITER}_n\}$ can indeed be separated. This is similar to the situation in the first-order case, where separation results can be established in two different ways, depending on whether "counting" or "pebbling" is used. The first approach is to sharpen the counting arguments we have used so far, with a view to separate consecutive levels in the hierarchy. The second approach, more problematic at this point, is to try some kind of pebbling argument.

This clearly raises the question of how to define a *higher-order pebble game*. We can define it with the aim to directly show that:

(1) $\mathrm{ITER}_n < \mathrm{ITER}_{n+1}$ for every $n \geqslant 1$,

or, using the already established reduction $\mathrm{REC}_n \leqslant \mathrm{ITER}_{n+1}$ of Theorem 16, the stronger result that:

(2) $\mathrm{ITER}_n < \mathrm{REC}_n$ for every $n \geqslant 1$.

In both cases there are several technical issues to be sorted out. Any of these two results, (1) or (2), will settle the strictness of the hierarchy $\{\mathrm{ITER}_n\}$ for all orders. Moreover, any of these two results will also imply that the hierarchy $\{\mathrm{REC}_n\}$ is infinite, already shown in [15] by different techniques: If (1) is true, then the even orders and odd orders of $\{\mathrm{REC}_n\}$ form two infinite strict hierarchies; if (2) is true, then all orders of $\{\mathrm{REC}_n\}$ form a single infinite strict hierarchy.

References

1. Abelson, H., and Sussman, G., *Structure And Interpretation Of Computer Progams*, MIT Press/McGraw-Hill, NY, 1984.
2. Aho, A.V. and Ullman, J.D., *Principles of Compiler Design*, Addison-Wesley, 1979.
3. Auslander, M.A., and Strong, H.R., "Systematic recursion removal", *Communications ACM*, **21**, no. 2, pp 127-134, Feb 1978.
4. Barendregt, H.P., *The Lambda Calculus, Its Syntax and Semantics*, revised edition, North-Holland, Amsterdam, 1984.
5. Böhm, C., and Berarducci, A., "Automatic synthesis of typed lambda-programs on term algebras", *Theoretical Computer Science*, **39**, pp 135–154, 1985.
6. Friedman, D.P., Wand, M., and Haynes, C.T., *Essentials of Programming Languages*, MIT Press/McGraw-Hill, NY, 1992.
7. Gandy, R.O., "An early proof of normalization by A.M. Turing", in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, eds. Seldin and Hindley, pp 453-455, Academic Press, 1980.
8. Gandy, R.O., "Proofs of strong normalization", in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, eds. Seldin and Hindley, pp 457-477, Academic Press, 1980.
9. Goerdt, A., "On the computational power of the finitely typed lambda-terms", in *Proceedings of 13th MFCS*, LNCS 324, pp 318-328, 1988.
10. Greibach, S.A., *Theory of Program Structures: Schemes, Semantics, Verification*, LNCS **36**, Springer-Verlag, 1975.
11. Henderson, P., *Functional Programming, Application and Implementation*, Prentice-Hall, 1980.

12. Hillebrand, G.G., Kanellakis, P.C., Mairson, H.G., and Vardi, M.Y., “Undecidable Boundedness Problems for Datalog Programs”, *Journal of Logic Programming* **25**:2, pp 163-190, 1995.
13. Kamin, S.N., *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990.
14. Kfoury, A.J., Moll, R.N., and Arbib, M.A., *A Programming Approach to Computability*, Springer-Verlag, 1982.
15. Kfoury, A.J., Tiuryn, J., and Urzyczyn, P., “On the expressive power of finitely typed and universally polymorphic recursive procedures”, *Theoretical Computer Science*, **93**, pp 1-41, 1992.
16. Kfoury, A.J., “Recursion, Tail-Recursion, and Iteration at Higher-Orders”. In preparation.
17. Kozen, D., and Tiuryn, J., “Logics of Programs”, in *Handbook of Theoretical Computer Science, Vol. B, Formal Methods and Semantics*, ed. J. van Leeuven, Elsevier Science Publ. and The MIT Press, pp 789–840, 1990.
18. McCarthy, J. , “Recursive Functions of Symbolic Expressions and Their Computation by Machine”, *Comm. ACM*, **3** (4), pp 184-195, 1960.
19. Mitchell, J.C., *Foundations for Programming Languages*, MIT Press, Cambridge, Mass, 1996.
20. Moschovakis, Y.N., *Elementary Induction on Abstract Structures*, North-Holland, 1974.
21. Moschovakis, Y.N., “The Formal Language of Recursion”, *Journal of Symbolic Logic*, **54**, pp 1216-1252, 1989.
22. Paterson, M.S., and Hewitt, C., “Comparative schematology”, *MIT A.I. Lab Technical Memo No. 201* (also in *Proc. of Project MAC Conference on Concurrent Systems and Parallel Computation*), 1970.
23. Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.
24. Pippenger, N., “Pebbling”, *Fifth Symposium on Mathematical Foundations of Computer Science*, IBM Japan, 1980.
25. Pippenger, N., “Advances in Pebbling”, *Proc. of 9th ICALP*, LNCS no. 140, Springer-Verlag, 1982.
26. Statman, R., “The typed λ-calculus is not elementary recursive”, *Theoretical Computer Science*, **9**, pp 73-81, 1979.
27. Strong, H.R., “Translating recursion equations into flowcharts”, *J. Computer and System Sciences*, **5**, pp 254-285, 1971.
28. Urzyczyn, P., “Non-trivial definability by flowchart programs”, *Information and Control*, **58**, nos. 1-3, pp 59-87, Sept 1983.
29. Walker, S.A. and Strong, H.R., “Characterizations of flowchartable recursions”, *J. Computer and System Sciences*, **7**, pp 404-447, 1973.
30. Winskel, G, *The Formal Semantics of Programming Languages: An Introduction*, The MIT Press, 1993.

Ground constants	$c : \rho$	$\mathit{type}(c) = \rho$
δ-*terms*	$\dfrac{N_1 : * \quad \cdots \quad N_k : *}{(f N_1 \cdots N_k) : \rho}$	$\mathit{type}(f) = *^k \to \rho$
if-*terms*	$\dfrac{N_1 : \mathbf{bool} \qquad N_2 : \tau \qquad N_3 : \tau}{(\mathbf{if}\ N_1\ N_2\ N_3) : \tau}$	
Variables	$v : \tau$	$\mathit{type}(v) = \tau$
Function names	$\mathcal{F}^{\tau} : \tau$	$\mathit{type}(\mathcal{F}) = \sigma \preccurlyeq \tau$
Applications	$\dfrac{M : \sigma \to \tau \qquad N : \sigma}{(M\ N) : \tau}$	
Abstractions	$\dfrac{M : \tau}{(\lambda v : \sigma.\ M) : (\sigma \to \tau)}$	$\mathit{type}(v) = \sigma$
Programs	$\dfrac{M_1 : \sigma_1 \qquad M_2 : \sigma_2 \qquad \cdots \qquad M_\ell : \sigma_\ell}{\{\mathcal{F}_1 = M_1,\ \mathcal{F}_2 = M_2,\ \ldots, \mathcal{F}_\ell = M_\ell\}}$	
	M_i is a closed term and $\mathit{type}(\mathcal{F}_i) = \sigma_i$ for $i \in \{1, \ldots, \ell\}$	

Fig. 1. Rules for well-typed terms and programs. Throughout, $\rho \in \{*, \mathbf{bool}\}$.

$$((\lambda v\ N)\ P) \xrightarrow[\mathcal{A},\mathcal{P}]{} N[v := P] \qquad (\beta\text{-reduction})$$

$$\mathcal{F}_i \xrightarrow[\mathcal{A},\mathcal{P}]{} M_i \qquad (\mathcal{P}\text{-reduction})$$

$$(\mathbf{if}\ \ b\ \ \mathbf{then}\ \ N\ \ \mathbf{else}\ \ P) \xrightarrow[\mathcal{A},\mathcal{P}]{} \begin{cases} N, \text{ if } b = tt, \\ P, \text{ if } b = \mathit{ff}. \end{cases} \qquad (\mathbf{if}\text{-reduction})$$

$$(f\ a_1 \cdots a_k) \xrightarrow[\mathcal{A},\mathcal{P}]{} a_{k+1} \qquad (\delta\text{-reduction})$$

$$\frac{N \xrightarrow[\mathcal{A},\mathcal{P}]{} P}{(NQ) \xrightarrow[\mathcal{A},\mathcal{P}]{} (PQ)}$$

$$\frac{N \xrightarrow[\mathcal{A},\mathcal{P}]{} P}{(QN) \xrightarrow[\mathcal{A},\mathcal{P}]{} (QP)}$$

$$\frac{N \xrightarrow[\mathcal{A},\mathcal{P}]{} P}{(\lambda v\ N) \xrightarrow[\mathcal{A},\mathcal{P}]{} (\lambda v\ P)}$$

Fig. 2. Reduction rules for program $\mathcal{P} = \{\mathcal{F}_i = M_i\}_{1 \leqslant i \leqslant \ell}$.

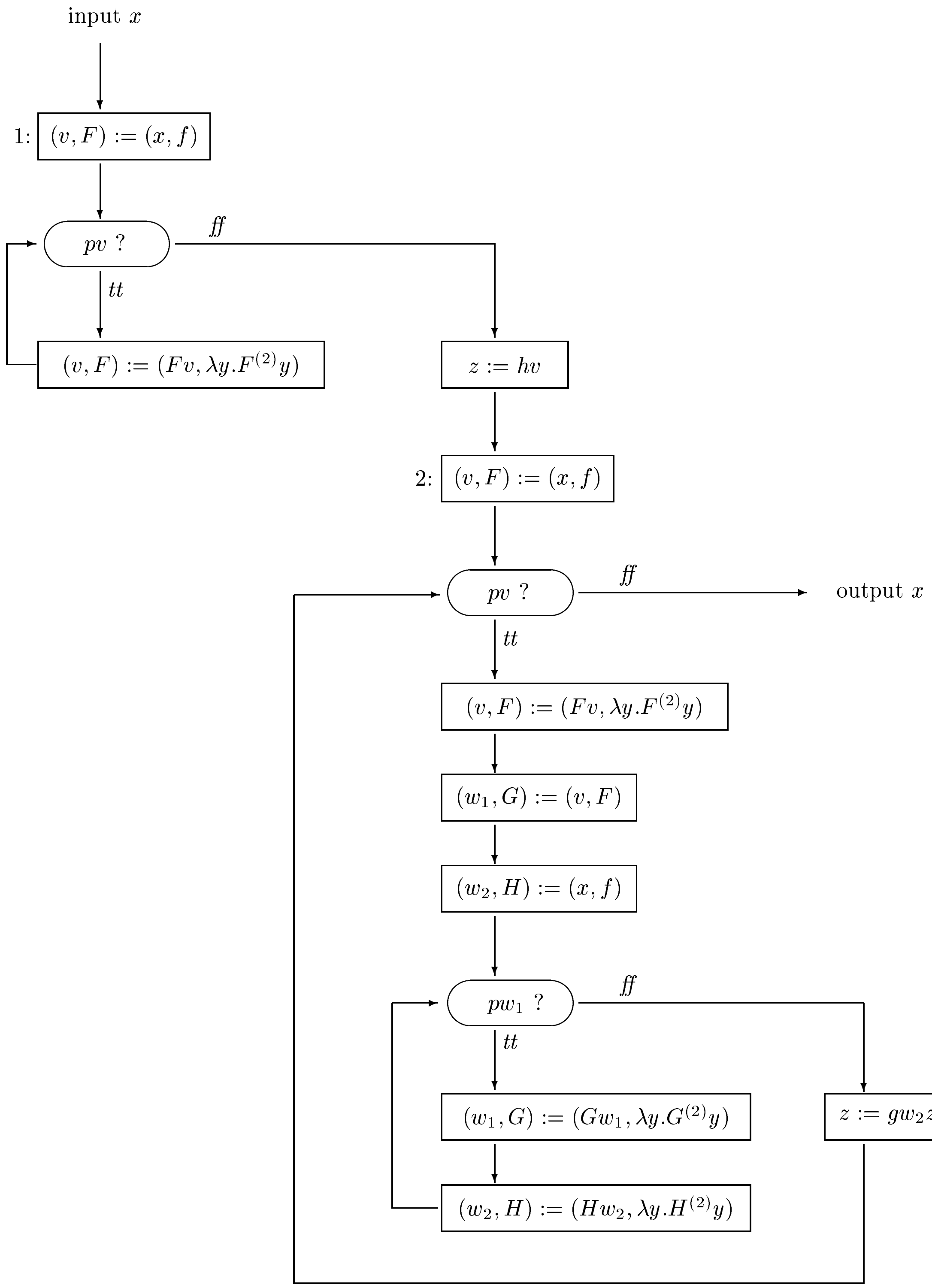

Fig. 3. Flowchart for Example 2.

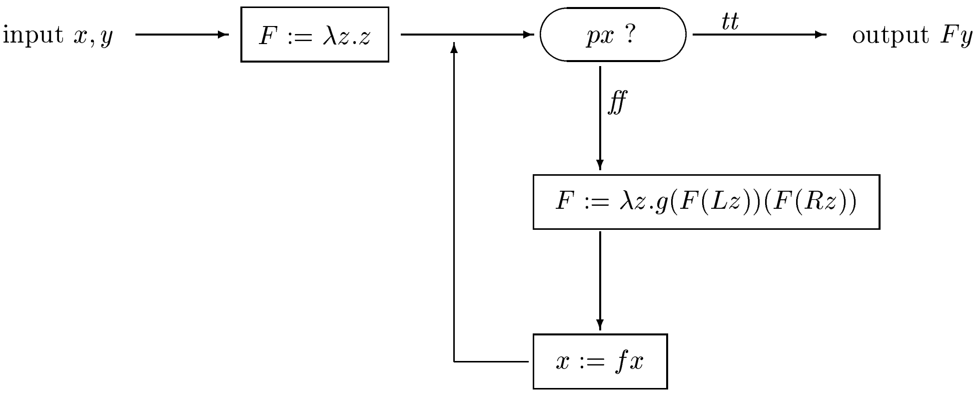

Fig. 4. Flowchart for Example 3.

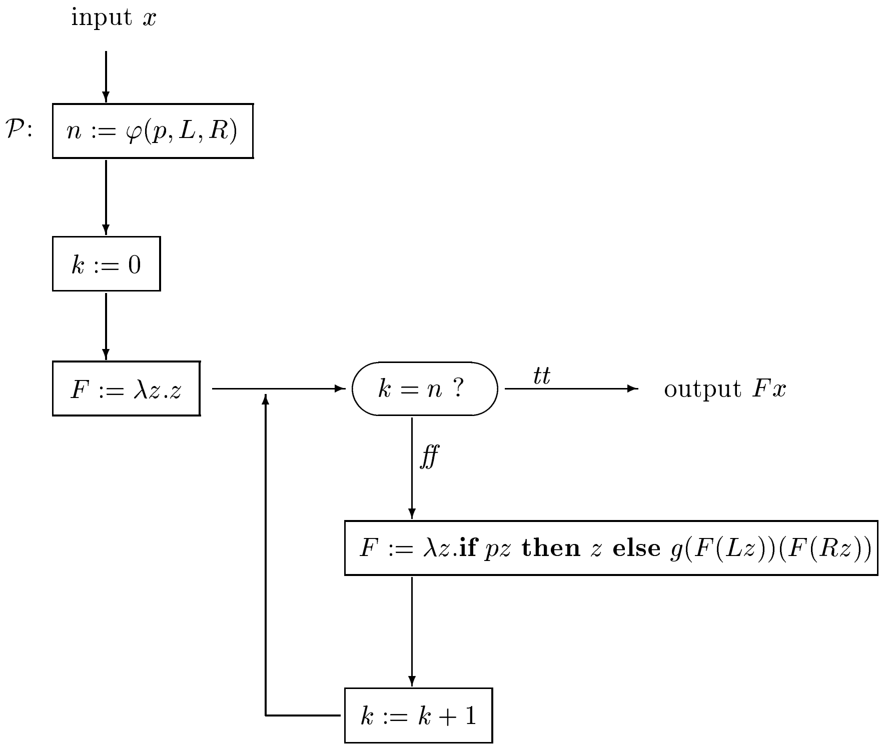

Fig. 5. Flowchart for Example 4.

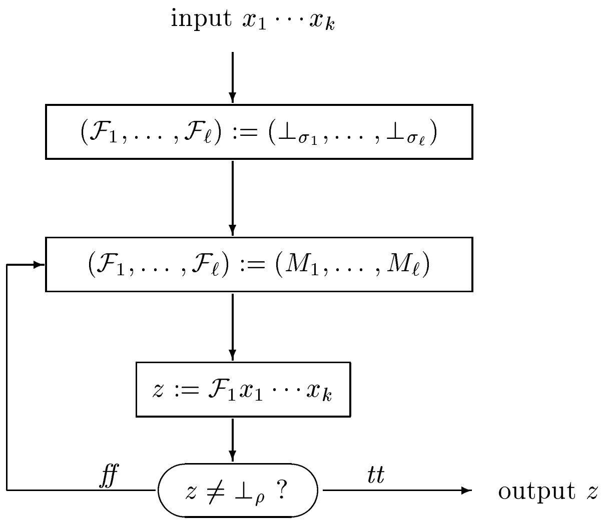

Fig. 6. Flowchart $\overline{\mathcal{P}}$.

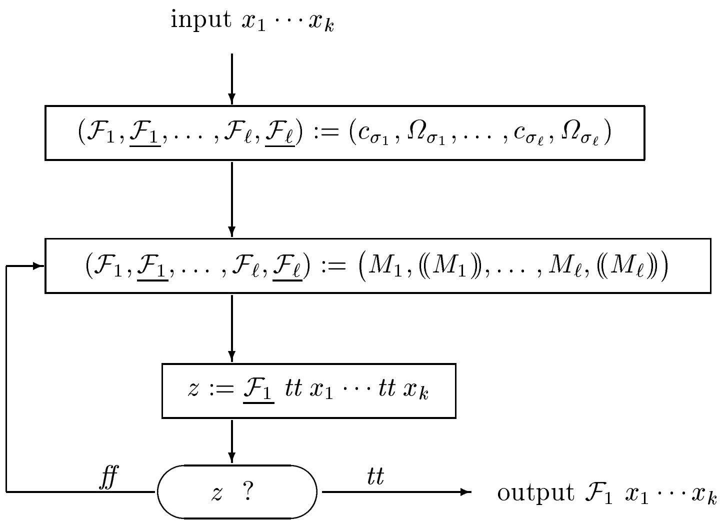

Fig. 7. Flowchart $\overline{(\!(\mathcal{P})\!)}$.

$$(\bot_{\sigma\to\tau}\ N) \underset{\mathcal{A},\mathcal{P}}{\longrightarrow} \bot_\tau \qquad (\bot\text{-reduction})$$

$$(\mathbf{if}^\tau\ \ b\ \ \mathbf{then}\ \ N\ \ \mathbf{else}\ \ P) \underset{\mathcal{A},\mathcal{P}}{\longrightarrow} \begin{cases} N, & \text{if } b = \mathit{tt}, \\ P, & \text{if } b = \mathit{ff}. \\ \bot_\tau, & \text{if } b = \bot_{\mathbf{bool}}. \end{cases} \qquad (\mathbf{if}_\bot\text{-reduction})$$

$$(f\ a_1 \cdots a_k) \underset{\mathcal{A},\mathcal{P}}{\longrightarrow} \begin{cases} a_{k+1}, & \text{if } \bot_* \notin \{a_1, \ldots, a_k\}, \\ \bot_\rho, & \text{if } \bot_* \in \{a_1, \ldots, a_k\}. \end{cases} \qquad (\delta_\bot\text{-reduction})$$

Fig. 8. Reduction rules in the presence of the special symbols $\bot_\sigma$:
Add rule ($\bot$-reduction), and adjust rules (**if**-reduction) and (δ-reduction).
Read with Figure 2.

Types	$\langle\!\langle * \rangle\!\rangle = \mathbf{bool}$
	$\langle\!\langle \mathbf{bool} \rangle\!\rangle = \mathbf{bool}$
	$\langle\!\langle \sigma \to \tau \rangle\!\rangle = \langle\!\langle \sigma \rangle\!\rangle \to \sigma \to \langle\!\langle \tau \rangle\!\rangle$
Special symbols	$\Omega_* \equiv \mathit{ff} : \mathbf{bool}$
	$\Omega_{\mathbf{bool}} \equiv \mathit{ff} : \mathbf{bool}$
	$\Omega_{\sigma\to\tau} \equiv \big(\lambda \underline{v} : \langle\!\langle \sigma \rangle\!\rangle . \lambda v : \sigma .\ \Omega_\tau\big) : \langle\!\langle \sigma \to \tau \rangle\!\rangle$
Terms	$\langle\!\langle c : \rho \rangle\!\rangle = tt : \langle\!\langle \rho \rangle\!\rangle$
	$\langle\!\langle f N_1 \cdots N_k : \rho \rangle\!\rangle = \mathbf{and}\ \langle\!\langle N_1 \rangle\!\rangle\ \cdots\ \langle\!\langle N_k \rangle\!\rangle : \langle\!\langle \rho \rangle\!\rangle$
	$\langle\!\langle \mathbf{if}\ N_1\ N_2\ N_3 : \tau \rangle\!\rangle = \big(\mathbf{if}\ \langle\!\langle N_1 \rangle\!\rangle\ \big(\ \mathbf{if}\ N_1\ \langle\!\langle N_2 \rangle\!\rangle\ \langle\!\langle N_3 \rangle\!\rangle\ \big)\ \Omega_\tau\big) : \langle\!\langle \tau \rangle\!\rangle$
	$\langle\!\langle v : \tau \rangle\!\rangle = \underline{v} : \langle\!\langle \tau \rangle\!\rangle$
	$\langle\!\langle \mathcal{F} : \tau \rangle\!\rangle = \underline{\mathcal{F}} : \langle\!\langle \tau \rangle\!\rangle$
	$\langle\!\langle MN : \tau \rangle\!\rangle = \big(\langle\!\langle M \rangle\!\rangle \langle\!\langle N \rangle\!\rangle N\big) : \langle\!\langle \tau \rangle\!\rangle$
	$\langle\!\langle \lambda v : \sigma . M : \sigma \to \tau \rangle\!\rangle = \big(\lambda \underline{v} : \langle\!\langle \sigma \rangle\!\rangle . \lambda v : \sigma .\ \langle\!\langle M \rangle\!\rangle\big) : \langle\!\langle \sigma \to \tau \rangle\!\rangle$
Programs	$\{\ \mathcal{F}_1 = M_1,\ \ldots, \mathcal{F}_\ell = M_\ell,\ \underline{\mathcal{F}_1} = \langle\!\langle M_1 \rangle\!\rangle,\ \ldots, \underline{\mathcal{F}_\ell} = \langle\!\langle M_\ell \rangle\!\rangle\ \}$

Fig. 9. The transformation $\langle\!\langle\ \rangle\!\rangle$ of finitely typed functional programs.

Read with Figure 1. Throughout, $\rho \in \{*, \mathbf{bool}\}$.
The boolean operation ($\mathbf{and}\ P\ Q$) is shorthand for
($\mathbf{if}\ P\ (\mathbf{if}\ Q\ tt\ \mathit{ff})\ (\mathbf{if}\ Q\ \mathit{ff}\ \mathit{ff})$). With several arguments, ($\mathbf{and}\ P\ Q\ R$) is shorthand for ($\mathbf{and}\ P\ (\mathbf{and}\ Q\ R)$).