Higher type recursive program schemes and the nested pushdown automaton

Andreas Goerdt[*)], Technische Universität Chemnitz
Fakultät für Informatik, 09107 Chemnitz, Germany

On the occasion of the retirement of
Prof. Dr. Klaus Indermark, RWTH Aachen

November 2005

Abstract. When implementing recursive programs on a standard von Neumann random access machine (RAM) it is natural and well known how to use a pushdown store in order to keep track of the recursive procedure calls when the program runs. The more abstract world of program schemes abounds with examples of this pushdown – recursion relation. As a concrete example it is well known how to implement monadic recursive program schemes (de Bakker Scott schemes) on a pushdown automaton with data storage. In this case he reverse direction can be easily proved, too: The pushdown automaton considered has exactly the same computational power as de Bakker Scott schemes.

Higher type concepts are at the heart of modern object oriented and functional programming languages in particular. The aim of this paper is to develop a theoretically well founded framework for implementing higher type recursion. We show that a natural generalization of the classical pushdown store, the nested pushdown allows to extend the equivalence above to a natural hierarchy of "higher type de Bakker Scott schemes ".

[*)] e-mail: goerdt@informatik.tu-chemnitz.de

Introduction

Background and motivation. For code generation the compiler has to translate the control structures of the programming language considered into machine code. To this end the compiler looks at the control structure as being separated from the underlying data domain. Those features of modern programming languages known as “polymorphic ” (or generic) realize an explicit form of this separation: Procedures can be defined to be invoked with actual parameters of data types which are unspecified and unknown at the time of writing the procedure.

In this note we are concerned with what may well be considered as one of the technically most complex control structures occurring nowadays: recursion based on functions of higher types. Being popularized perhaps through the imperative programming language ALGOL, higher type recursion is a characteristic feature in particular of functional but also of object oriented languages.

Two similarly looking but conceptually different approaches to studying control structures are known: The “simple ” or “ monadic ”or “de Bakker Scott ” approach considers control structures as computing only with a single *atomic* element of data. This view studies the control structure in the purest form possible ensuring high generality and relevance of the results obtained from a practical point of view. The second approach is to allow computation with *several* data elements. Note that in this case it may even become possible to store an unbounded amount of data elements during computation (using for example local variables in a recursive procedure.)

It is well known that when computing over data domains containing classical arithmetic standard recursive function declarations are universal.That is for any computable function there exists an equivalent system of recursive function definitions. This is not at all true in the present framework of program schemes. Here two function definitons are equivalent iff they are semantically equal for *all* data domains possible. We use this notion of equivalence classical in the theory of program schemes throughout. It is reassuring to know, but not really needed in the sequel, that a generalization of Church’s thesis with respect to computability over arbitrary strcutures is known: Generalizing the classical Turing machine to compute over arbitrary structures yields a universal computing device with respect to the present notion of equivalence [Sh]. Standard recursive function definitions are well known not to be universal in this sense.

In order to approach higher type concepts some common notational background is seemingly unavoidable. The most general type structure with one base type is given by: ind (for “ individuals ”) and if τ, ρ are types then $\tau \ \rightarrow \rho$. We sometimes call τ argument type and ρ result type. The model programming language of finitely typed λ-terms with $if - then - else$ and recursion (technically known as fixpoint) is based on this type structure, cf. [LoSi 84]. This language may be considered an example of the second approach to control structures mentioned above.

To the best of the author’s knowledge the de Bakker Scott approach to higher types has not yet been considered for the preceding most general type structure. To exemplify the generality of this structure we consider an example type like

$$ind \rightarrow \Big(((ind \rightarrow ind) \rightarrow ind) \rightarrow ind\Big).$$

Here we have the argument type ind whereas the result type stands for a function even taking a function as argument. This may be disturbing from a programmer’s point of view. Consequently, restricted type structures closer to the programming practice have been considered.

A conceptually interesting restricted type structure is that of "(general) homogeneous types " (for a definition see the subsection: Homogeneous types: Notation and Results, below). From the literature on this type structure we only cite [Da 82] and [DaGo 86]. From the de Bakker Scott point of view standard context free grammars bear some similarity to systems of recursive function definitions (nonterminals correspond to function names, the nondeterminism inherent to grammars corresponds to the – being somewhat generous here – $if - then - else$, unavoidable in recursive definitions.) With this analogy in mind [Da 82] looks at grammers with nonterminals being of higher homogeneous type. [DaGo 86] gives an automata theoretical characterization of classes of languages induced by a natural syntactic hierarchy with respect to the higher types allowed. This hierarchy derives from the obvious classification of types according to their "(functional) level " (see the subsequent subsections.) The characterization uses the "nested pushdown automaton " as introduced in [Ma 76]. Disregarding the initial motivation, the results obtained remain firmly in the realm of formal language theory.

In this note we show that the techniques from [DaGo 86] can be modified to be of direct use for program schemes and therefore programming languages. We transfer the aforementioned automata theoretical characterization of formal languages to yield an automaton (with a nested pushdown store as characteristic feature) equivalent to the hierarchy of de Bakker Scott schemes. This extends the recursion – pushdown equivalence known for classical de Bakker Scott schemes to higher homogeneous types. Moreover, it yields a seemingly new way to implement higher type recursive procedures.

The technical starting point of our results is the aforementioned and well known

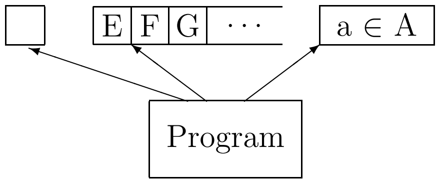
Theorem 1 (Cited from [In 79]) *(a) Each de Bakker Scott scheme can be implemented on a pushdown automaton with data store.*

(b) Each pushdown automaton with data store can be transformed into an equivalent de Bakker Scott scheme.

A de Bakker Scott scheme is a system of recursive equations like

$$F_0(y) = t_0, \ldots, F_m(y) = t_m$$

where the right hand sides are build from applicative terms like
$F(g(H(f(K(y)))))$ where g, f are operations on the underlying (unspecified) structure and F, G, K are among the F_i being defined on the left hand sides. Each right hand side either is an applicative term or it is a term like $if\ p(y)\ then\ s_1\ else\ s_2\ fi$, where p is a relation (predicate) on the underlying structure and the s_i are purely *applicative* terms. Given a data domain and an element a from this domain we start with $F_0(a)$ and iteratively reduce the current term by plugging in the right hand sides until the result appears. By convention we always reduce the innermost term possible (call-by-value). The pushdown automaton with data store is visualized as



The instructions and programs of the automaton involve some technicalities specified in due course. Only for expository reasons we cite the technically simple proof of Theorem 1 in section 1.

For motivation and readability we choose to present our results first for the possibly simplest structure of higher types, that of "monadic (homogeneous) types". The bulk ot the technical contents of this note is contained in this case. Subsequently we treat "(general) homogeneous types."

Monadic types: Notation and results. The type structure of "monadic types" is given by: The only base type is *ind*. Higher types are defined inductively: If τ is a type then $\tau \to \tau$ is a type. Thus argument and result type are always the same and we have only one argument. Throughout we will be concerned with the notion of level of a type. We say that level(ind) $=$ 0 and level($\tau \to \tau$) $=$ level(τ) $+$ 1. Note that we have exactly one type of each level: *ind* of level 0, $ind \to ind$ of level 1, and $(ind \to ind) \longrightarrow (ind \to ind)$ of level 2 .... A monadic type τ of level n can be uniquely decomposed as

$$\begin{aligned}\tau &= \tau_{n-1} \to \tau_{n-1} = \tau_{n-1} \to (\tau_{n-2} \to \tau_{n-2})\\ &= \tau_{n-1} \to (\tau_{n-2} \to \cdots \to (\tau_1 \to (\tau_0 \to \tau_0))\cdots),\end{aligned}$$

where level(τ_i) $= i$ and therefore $\tau_0 = ind$. Througout we use the convention that $\to$ in types associates to the right. This allows to simplify the bracketing as then

$$\begin{aligned}&\tau_{n-1} \to (\tau_{n-2} \to \cdots \to (\tau_1 \to (\tau_0 \to \tau_0))\cdots)\\ &= \tau_{n-1} \to \tau_{n-2} \to \cdots \to \tau_1 \to \tau_0 \to \tau_0.\end{aligned}$$

Thus an object (term) having a monadic homogeneous type of level n has n arguments of level $n-1$ through 0. We have a typed family of variables $Var = (Var^\tau \mid \tau \text{ a type })$, a set of operation symbols, *Opsym*. Our operation symbols are always of type $ind \to ind$. Additionally we have a set of relation symbols *Relsym*. The set of variables Var^τ is partitioned into formal parameters and function symbols. The family of applicative terms $Term = (Term^\tau \mid \tau \text{ a monadic type })$ is defined by

$$\begin{aligned} Var^\tau &\subset Term^\tau,\\ Opsym &\subset Term^{ind \to ind},\\ \text{and if } t \in Term^{\tau \to \tau} \quad &\text{and} \quad s \in Term^\tau \text{ then } t(s) \in Term^\tau.\end{aligned}$$

Sometimes the notation $t : \tau$ is used instead of $t \in Term^\tau$. We use it also for typed families different from $Term$. We set level(t) $=$ level(τ) where $t : \tau$.

Each non atomic applicative term $t : \tau$ can be uniqely decomposed as $t = t'(t_1)$ where $t_1 : \tau$. This inductively implies that $t : \tau$ contains no subterm of level $<$ level(t) $=$ level(τ). This is a characteristic feature of terms based on homogeneous type structures. Inductively we can continue to decompose t'

$$t = t'(t_1) = t''(t_2)(t_1) = \cdots = y(t_s)(t_{s-1})\cdots(t_2)(t_1)$$

where $y : \tau_s \to \tau_{s-1} \to \cdots \tau_1 \longrightarrow \tau_1$ is a variable (or operation symbol), $t_i : \tau_i$, and $\tau_1 = \tau$. Thus the type of a non-atomic term like t is equal to the type of its rightmost argument (again reflecting homogeneity.)

The semantics of our syntactic objects introduced here and further below is given with respect to an interpretation (or algebra) $\mathcal{A} = (A, \phi_{Ops}, \phi_{Rel})$. Here A is an arbitrary set, and ϕ_{Ops} assigns to each function symbol $f : ind \rightarrow ind$ a total function denoted by $\phi(f) : A \rightarrow A$. ϕ_{Rel} assigns to each relation symbol r a subset denoted by $\phi(r) \subseteq A$ of those individuals which make r true. Note that we consider only functions and relations with one argument.

We define de Bakker Scott schemes with higher monadic types as considered here analogously to de Bakker Scott schemes in Theorem 1. Section 1 has a more detailed definition. The nested pushdown automaton is formally introduced in Section 2. Theorem 2 to be proved in Section 3 is an extension of Theorem 1 (a) to higher types. .

Theorem 2 *Given a level n de Bakker Scott scheme with monadic higher types we can construct a level n pushdown automaton equivalent to it.*

At present it is not clear if the reverse direction of the theorem can be obtained. To this end (among others) we consider the classical type structure of (general) homogeneous types.

Homogeneous types: Notation and results. The type structure of " homogeneous types" is given by: The only base type is ind we have level$(ind) = 0$. The type $ind \rightarrow ind$ is the only type of level 1. If
$\tau, \tau_1, \ldots, \tau_m$, $m \geq 1$ are all types of the *same* level n then $(\tau_1 \times \cdots \times \tau_m) \rightarrow \tau$ is a type of level $n+1$. Thus for example *all* types of level 2 are
$(\tau \times \cdots \times \tau) \rightarrow \tau$, where $\tau = ind \rightarrow ind$. As we adhere to the de Bakker Scott principle we have only one individual and therefore no argument types like $ind \times \ldots \times ind$. We have analogous decomposition observations as in the monadic case where instead of a single argument type we now have a tuple $(\tau_1 \times \cdots \times \tau_m)$ of argument types of the same level. As in ths monadic case we have a family of variables Var now homogeneously typed, operation and relation symbols. Applicative terms now are built with the rule

$$\text{if } t_1 : \tau_1, \ldots, t_m : \tau_m, \; t : (\tau_1 \times \cdots \times \tau_m) \rightarrow \tau \text{ then } t(t_1, \ldots, t_m) : \tau.$$

Again we can decompose analogously to the monadic case substituting arguments by argument lists of the same level. Consequently a term t contains no subterm of level $<$ level(t). de Bakker Scott schemes for this case are formally defined in Section 4 and we can state our final

Theorem 3 *(a) Each level n pushsdown automaton can be transformed into an equivalent level n de Bakker Scott scheme with general homogeneous types.*
(b) Given a level n de Bakker Scott scheme with homogeneous types we can construct an equivalent level n pushdown automaton.

This characterization shows that Theorem 1 can be extended to higher types, item (b) gives us a novel way implementing higher type recursion as announced. We leave a comparison of this result to known ways to implement higher type recursion to future work.

1 Schemes with monadic types

After introducing the schemes with monadic higher types, we introduce the classical pushdown automaton computing over an arbitrary structure and present the proof of Theorem 1 in order motivate the subsequent more complex considerations.

Syntax. Given an applicative term $t : \tau_{n-1} \to \tau_{n-2} \to \tau_{n-3} \to \ldots \to \tau_0 \to \tau_0$ where $\tau_0 = ind$, we sometimes use the notation

$$t \downarrow = t(y_{n-1})(y_{n-2})(y_{n-3}) \ldots (y_0) \ : ind$$

where y_i is a formal parameter of type τ_i. We use this notation mostly when t is just a function symbol. Our program schemes are systems of recursive equations between terms of base type. The set of right hand sides RHS is larger than the set of applicative terms of base type in that we allow for the $if - then - else$ control structure. That is

$$\begin{array}{rcl} Term^{ind} & \subset & RHS, \\ \text{and } \ if \ r(y) & then & t \ else \ s \ fi \in RHS, \end{array}$$

where r is a relation symbol, $y : ind$ is a formal parameter, and $t, \ s \ \in Term^{ind}$. Note that our right hand sides can all be considered as of type ind. The definition of the condition of the $if - then - else - fi$ construct seems arbitrarily restrictive, one might expect the condition $r(t)$ where $t : ind$. We have not checked in how far our restriction can be removed, but even with this definition non-trivial programs become possible as the example further below shows. Now we come to the actual program schemes. A monadic recursive program scheme of higher type is a system of equations as

$$F_0(y_0) \ = \ t_0, \ \ F_1 \downarrow = \ t_1, \ \ldots, \ F_m \downarrow = \ t_m,$$

where the F_i are different function symbols (making the scheme deterministic), $F_0 \ : ind \to ind$ is the main function (symbol), y_0 is a formal parameter of type ind, $t_i \in RHS$, and the variables allowed in t_i are all the F_j and the formal parameters from $F_i \downarrow$. Thus each variable in t_i is bound, either as a formal parameter or as a function symbol defined by one of the equations. The level of a scheme is the *maximal* level of the F_j and thus the maximal level of all symbols occurring. In our case it is simply the maximal number of arguments an F_j can have. Note that the de Bakker Scott schemes from the Introduction correspond to schemes of level 1 according to our definition.

Example. To introduce the semantics of our schemes we give an example (of a level 2 scheme). We consider the following interpretation: A is the set of all words over the symbols a and b, that is $A \ = \ \{a, \ b\}^*$. We have the operation symbols $a, b : ind \to ind$ with the interpretation $\phi(a)(w) \ = \ aw \ \in A$ and similarly for b. Thus our interpretation is a so called "free " interpretation. r is interpreted as $\phi(r) \ = \ \{a^n b^n \,|\, n \geq 0\} \subset A$. We use the function symbols

$$\begin{array}{rcl} F & : & ind \to ind, \quad \text{and} \\ C, \ G & : & (ind \to ind) \to (ind \to ind). \end{array}$$

Our scheme is given by

$$\begin{array}{rcl} F(y_0) & = & G(a)(y_0) \qquad\qquad \text{(Here } a \text{ is the operation symbol.)} \\ G(y_1)(y_0) & = & if \ r(y_0) \ then \ y_1(y_0) \ else \ G(C(y_1))(b(y_0)) \ fi \\ & & \qquad\qquad\qquad \text{(Generation of the same number of } C\text{'s and } b\text{'s.)} \\ C(y_1)(y_0) & = & y_1(y_1(y_0)) \\ & & \qquad\qquad \text{(Iterated copying depending on the number of } C\text{'s.)} \end{array}$$

Semantics. The semantics of our schemes is given by a deterministic reduction or evaluation strategy. The evaluation strategy transforms *computation terms* of base type. Computation terms are defined as the family of applicative terms, but with two modifications: Each individual of the interpretation considered is a computation term of type *ind*. Such a term is what we call an *evaluated* computation term . Moreover, we allow no formal parameters in computation terms (reflecting the call-by-value character of our reduction strategy below.) Only function symbols occurring in the program scheme under consideration and operation symbols are admitted in computation terms.

Our evaluation strategy is a call-by-value strategy defined as follows: Only computation terms of type *ind* can be reduced. Always the innermost computation term of type *ind* which is not yet evaluated is reduced. Computation terms like $f(t)$ where f is an operation symbol therefore are only reduced when $t \in A$ is an individual. In this case $\phi(f)$ is applied to t giving another individual. Computation terms like $F(t_{n-1})\cdots(t_0)$ where F is a function symbol are only reduced when the term t_0 is evaluated, that is an individual. In this case we reduce the term $F(t_{n-1})\cdots(t_0)$ in the natural way: plugging in the right hand side of the equation for $F\downarrow$ substituting the formal paramters with the actual parameters t_i. In case that the right hand side of $F\downarrow$ is an $if-then-else$ we first evaluate the condition of the $if-then-else$. This is possible without further reduction steps as t_0 is already evaluated, the condition reads $r(y_0)$ where r is a relation symbol, and y_0 corresponds to t_0. Then our reduction strategy branches directly. Note that this allows us to consider only purely applicative terms as computation terms. No further cases can arise and our reduction strategy is defined at this point. We denote the execution of one reduction step of our reduction strategy by $\Rightarrow$, and i reduction steps are denoted by $\Rightarrow^i$.

Example (continued). We evaluate our scheme above starting with $F(a)$ where a is the individual $a \in A$.

$$\begin{aligned} F(a) &\Rightarrow G(a)(a) \text{ (The first } a \text{ is the operation symbol, the second} \\ &\qquad\qquad a \text{ the individual, } if-then-else \text{ is evaluated directly.)} \\ &\Rightarrow G(C(a))\,(ba) \text{ (Note } ba \in \phi(r), \\ &\qquad\qquad\quad \text{the } if-then-else \text{ is evaluated directly.)} \\ &\Rightarrow C(a)(ba) \\ &\Rightarrow a(a(ba)) \\ &\Rightarrow a(aba) \Rightarrow aaba = aaba. \end{aligned}$$

As a second example we evaluate

$$\begin{aligned} F(aa) &\Rightarrow G(a)(aa) \\ &\Rightarrow^4 G(C(C(a)))(bbaa) \quad \text{(Two times G and two times b.)} \\ &\Rightarrow C(C(a))(bbaa) \\ &\Rightarrow C(a)(C(a)(bbaa)) \\ &\Rightarrow C(a)(a(a(bbaa))) \quad \text{(Two function evaluations of } a.) \\ &\Rightarrow^2 C(a)(aabbaa)) \\ &\Rightarrow a(a(aabbaa)) \\ &\Rightarrow^2 aaaabbaa = a^4bbaa. \end{aligned}$$

The function computed is $a^n \mapsto a^{2^n}\, b^n\, a^n$, it is undefined for arguments $\neq a^n$. The scheme exhibits what we think is one characteristic feature of higher type schemes, an iterated copying process leading to exponential growth.

The (classical) pushdown automaton. Conceptually all the automata considered here are heavily restricted versions of Friedman's generalized Turing machine [Sh]. We recall the visualization from the introduction.

The data store always stores one individual. The pushdown store is as usual; it stores elements from a given set of storage symbols Γ (the storage alphabet). Moreover, we need a finite set of states Q. Given an interpretation, the automaton can evaluate the operations and relations with the individual stored as argument in one step. The configurations of the automaton are triples (q, W, a), where $q \in Q$, $W \in \Gamma^*$, and a is an individual of the interpretation considered. To program the automaton we have a specified set of instructions.

Delete instruction:	$(q,\ A,\ pop,\ p)$, here $A \in \Gamma,\ p, q \in Q$.
Push instruction:	$(q, A, push(W), p)$, here $W \in \Gamma^+ = \Gamma^* \setminus \{e\}$.
Function evaluation:	$(q,\ A,\ exec(f),\ p)$, f is an operation symbol.
Relation evaluation:	$(q,\ A,\ exec(r),\ p,\ p')$, r is a relation symbol.

Finally given operation and relation symbols a program is a finite sequence (or set) of instructions as above with a given initial state $q_0 \in Q$ and a given initial symbol $A_0 \in \Gamma$ for the store. As we want our programs deterministic we require that for *each pair* q, A we have *exactly one* instruction $(q,\ A, \ldots)$ in our program.

Given an interpretation we give the natural semantics to our instructions. The instruction $(a,\ A \ldots)$ induces one computation step on the configurations $(q, AV, -)$, $V \in \Gamma^*$. We denote it by $\vdash$.

Delete instruction:	$(q,\ AV,\ -) \vdash (p, V, -)$.
Push instruction:	$(q, AV, -), \vdash\ (p, WV, -)$.
Function evaluation:	$(q,\ AV,\ a))\ \vdash (p, AV,\ \phi(f)(a)\ )$.
Relation evaluation:	$(q,\ AV,\ a) \vdash (p, AV, a)$ or (p', AV, a) depending on $\phi(r)(a)$.

Given an individual a the automaton starts in the configuration $(q_0,\ A_0,\ a)$ and transforms the current configuration in a deterministic way until the store is empty. Then and only then the automaton stops and the stored individual is the output. Clearly the automaton need not always stop.

Proof of Theorem 1 (a). The proof relies on the observation that computation terms of a given program scheme can be naturally represented as configurations,

$$F_1(F_2(\cdots(F_m(a))\cdots)) \ \rightsquigarrow\ (q,\ F_m \ldots F_2 F_1,\ a),$$

where q will be treated by the program to be defined as a designated " normal state ". Given a level 1 scheme

$$F_0(y)\ =\ t_0,\quad \ldots,\quad F_n(y)\ =\ t_n$$

the automaton is programmed to simulate the execution of the equations. This is greatly simplified when we assume that te scheme is in (Chomsky) normalform. That is as right hand sides t of $F(y)\ =\ t$ we only have five possibilities.

$$G(y),\ \ H(G(y)),\ \ y,\ \ f(y),\ \ if\, r(y)\, then\ G(y)\ else\ H(y)\ fi,$$

where G, H are variables, f is an operation symbol and r is a relation symbol. A simple inductive process transforms a given scheme into normalform by decomposing the given right hand sides. For example the equation
$F(y) = G(H(f(K(y))))$ is decomposed as

$$\begin{aligned} F(y) &= L_1(K(y)), & L_1(y) &= L_2(L_f(y)), \\ L_f(y) &= f(y), & L_2(y) &= G(H(y)), \end{aligned}$$

where L_i and L_f are new variables not yet occurring in the program scheme.

To translate a scheme in normalform into a program we translate each equation separately. Given the equation $F(y) = t$, we distinguish the five cases for t and translate them as

$$\begin{array}{rcl} G(y) & \rightsquigarrow & (q,\, F,\, push(G),\, q), \\ G(H(y)) & \rightsquigarrow & (q,\, F,\, push(HG),\, q), \\ y & \rightsquigarrow & (q,\, F,\, pop,\, q), \\ f(y) & \rightsquigarrow & (q,\, F,\, exec(f),\, q_F), \text{ and } (q_F,\, F,\, pop,\, q), \\ if\, r(y)\, then\, G(y) & else & H(y)\, fi \\ & \rightsquigarrow & (q,\, exec(r),\, q_G,\, q_H), \text{ and } (q_G,\, F,\, push(G), q) \text{ and} \\ & & (q_H,\, F,\, push(H),\, q). \end{array}$$

Here the q_G, q_H, q_F are new states occurring only at this place. To be syntactically fully correct, we have to add some "dummy instructions " $(q_H, A, \ldots), \ldots$ which however never are used in our simulation. Clearly Γ consists of all variables of the program, Q contains the normal state q and some additional states like the q_G, q_H, q_F above, used to control the simulation of equations. The initial symbol of the store is F_0 from the scheme. Clearly the initial state has to be the normal state q.

As each reduction step is simulated by one or two computation steps of the automaton, the equivalence of automaton and scheme follows easily.

Proof of Theorem 1 (b) The proof relies on the observation that configurations can be naturally represented as computation terms

$$(p,\, A_1\, A_2 \ldots\, A_m,\, a) \rightsquigarrow A_m(\cdots A_2(pA_1(a))\cdots).$$

The next computation step of the automaton depends on the state p and the top symbol A_1 of the store . The next reduction step induced by the scheme to be defined will depend on the variable called pA_1. Problems arise when the automaton executes the delete instruction $(p,\, A_1, pop, p')$. To simulate this, the scheme should have the equation $pA_1(y) = y$. This would yield the computation term $A_m(\cdots A_2(a))\cdots))$, unfortunately we have lost the state p'. We cannot add p' to A_2 because we cannot syntactically construct variables depending on information obtained during the reduction process. Moreover, we cannot store p' as $p'A_2$ when A_2 is stored because we do not really know the right p' at this moment of the computation. Therefore we impose an *additional syntactic restriction* on our programs.

Definition 1.1 *For any program of the pushdown automaton there is a state d such that all delete instructions are like* $(-,\, -,\, pop,\, d)$.

That is, d is a *fixed* "(post-)delete state ". Fortunately, the automaton constructed above satisfies this restriction, the delete state is q, the normal state of the simulation. With this restriction it is clear how configurations are represented, as

$$(p, A_1\, A_2 \ldots\, A_m,\, a) \rightsquigarrow dA_m(\cdots dA_2(pA_1(a))\cdots),$$

where d is the delete state of the program.

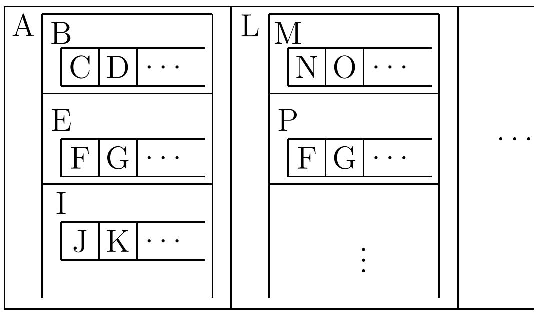
Given a program with delete state d we construct a program scheme with exactly one equation $pA(y) = t$ for each instruction $(p, A, \ldots)$ of the automaton.

$$\begin{aligned}
(p,\, A,\, pop,\, d) &\rightsquigarrow t = y,\\
(p,\, A,\, push(A_1\, A_2 \ldots A_m),\, p') &\rightsquigarrow t = dA_m(\cdots(dA_2(p'A_1(y)))\cdots),\\
(p,\, A,\, exec(f),\, p') &\rightsquigarrow t = p'A(f(y)),\\
(p,\, A,\, exec(r),\, p',\, p'') &\rightsquigarrow t = if\, r(y)\, then\, p'A(y)\, else\, p''A(y)\;\; fi.
\end{aligned}$$

The main equation of the program scheme is the unique equation $q_0A_0(y) = \ldots$ where A_0 is the start symbol of the store and q_0 is the initial state of the automaton. Starting with the computation term $q_0A_0(a)$ the scheme induces a reduction sequence directly corresponding to the computation of the automaton.

2 The nested pushdown automaton

The nested pushdown store. The nested pushdown storage structure has been orignally introduced by Maslov [Ma 76]. The storage structure of the level 2 pushdown store is given by a classical level 1 pushdown store where each element stored points to another level 1 pushdown store. In a level 3 store each element of a classical pushdown store ponts to a level 2 store. This is best visualized nesting pushdown stores:



One can also write instead:

$$\begin{array}{ccccccccccccccccc}
 & & C & D & & F & G & & J & K & & & N & O & & Q & R\\
 & B & & & E & & & I & & & & M & & & P & & \\
A & & & & & & & & & & L & & & & & &
\end{array}$$

In the nested pushdown store we allow for an arbitrary nesting depth indicated by the nesting depth of brackets in

Definition 2.1 *Given a set of storage symbols Γ the set configurations of the nested pushdown store over Γ is given by the set NPU defined as*

$$\begin{array}{rcl}
 \Gamma^* & \subset & NPU,\\
if\; A \in \Gamma\; and\; W \in NPU,\; W\; non\text{-}empty & then & A[W] \in NPU,\\
if\; V,\, W \in NPU, & then & V\, W \in \Gamma.
\end{array}$$

For example $A_1\Big[A_2[A_3]\,C[D[E]]\Big]\,BE$ stands for the store

$$\begin{array}{ccccccc}
 & & & & E & & \\
 & & A_3 & D & & & \\
 & A_2 & & C & & & \\
A_1 & & & & & B & E
\end{array}$$

Each occurrence of a symbol occurs in a specified nesting depth. A_1 in nesting depth 1, A_2 in nesting depth 2, A_3 in nesting depth 3. A symbol can occur in several nesting depths, like E above first in nesting depth 2 and then in 1. Note that each non-empty configuration X from NPU can be uniquely decomposed as

$$X \;=\; A_1[A_2[A_3[\cdots[A_{m-1}[A_m X_m]X_{m-1}]\cdots]X_3]X_2]X_1$$

where $A_i \in \Gamma$ and $X_i \in NPU$. The NPU " belonging " to A_1 is $A_2 \dots X_2$ the one belonging to A_{m-1} is $A_m X_m$, to A_m belongs only the empty store. The nested pushdown automaton to be defined will have access to the whole top of the store, consisting of the *string* $A_1 \dots A_m$. For $X \in NPU$ the level of X is defined to be level(X) $=$ the maximal nesting depth of any occurrence of a symbol in X. So classical pushdown store configurations are of level 1, the store in the preceding picture is of level 3.

Operations defined on NPU. Let X be a non-empty nested pushdown store, specified as

$$X \;=\; A_1[A_2[A_3[\cdots[A_{m-1}[A_m X_m]X_{m-1}]\cdots]X_3]X_2]X_1 \in NPU.$$

For $n \leq m$ the pushdown store $pop_n(X) \in NPU$ is obtained by deleting A_n with the pushdown belonging to A_n, that is $A_n[A_{n+1} \dots X_{n+1}]$. It is defined by

Definition 2.2

$$pop_n(X) = \begin{cases} A_1[A_2[\cdots[A_{n-1}[X_n]X_{n-1}]\cdots]X_2]X_1, & \textit{if } X_n \textit{ non empty} \\ A_1[A_2[\cdots[A_{n-1}X_{n-1}]\cdots]X_2]X_1, & \textit{if } X_n \textit{ is empty.} \end{cases}$$

For example we have

$$pop_2(A[B]C) \;=\; AC \qquad pop_1(A[B]C) = C$$

$$pop_1(A[B[CD]E]F) \;=\; F, \qquad pop_2(A[B[CD]E]F) \;=\; A[E]F,$$

in the third example above the NPU belonging to A is $B[CD]E$ and is deleted together with A, in the fourth example the NPU of B is CD and consequently gets deleted.

For X as above $1 \leq n \leq m{+}1$ and $W \in \Gamma^+ \;=\; \Gamma \backslash \{e\}$ the NPU $push_n(W,\ X) \in NPU$ is obtained by to pushing $W \;=\; B_1 \cdots B_k$ in place of A_n or on top of A_m if $n = m+1$. It is defined by

Definition 2.3 $push_n(W,\, X) =$

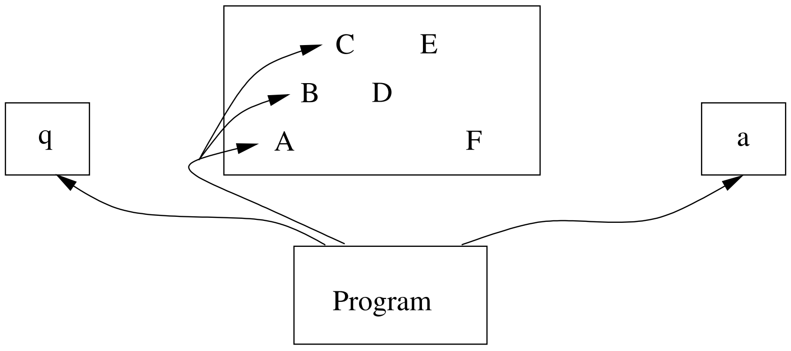
$$\begin{cases} A_1[A_2[\cdots[A_{m-1}[A_m[W]X_m]X_{m-1}]\cdots]X_2]X_1 \in NPU \;\; \textit{if } n = m+1 \\ A_1[A_2[\cdots[A_{m-1}[WX_m]X_{m-1}]\cdots]X_2]X_1 \in NPU \;\; \textit{if } m = n \\ A_1[A_2\cdots[A_{n-1}[B_1[\mathbf{Z}]B_2[\mathbf{Z}]\dots B_k[\mathbf{Z}]]X_{n-1}]\cdots]X_2]X_1 \in NPU \\ \textit{if } \;n < m \;\textit{ and }\; \mathbf{Z} \;=\; A_{n+1}[\dots[A_m X_m]\cdots]X_{n+1} \\ \textit{is the } NPU \textit{ belonging to } A_n. \end{cases}$$

Thus $push_n(W, X)$ substitutes A_n with the word W copying the NPU belonging to A_n such that it belongs to *each* symbol of W. This copying ability seems to be a characteristic feature of the nested pushdown store. Thus for example we have

$$\begin{aligned} push_1(DE, A[B]C) &= D[B]E[B]C & push_1(DE, A[X]C) &= D[X]E[X]C, \\ & & &\text{for } X \in NPU \\ push_2(DE, A[B]C) &= A[DE]C & push_3(DE, A[B]C) &= A[B[DE]]C, \end{aligned}$$

note how the nesting depth increases in the fourth example and how the X as a whole is copied in the second example.

The nested pushdown automaton. The level n pushdown automaton to be defined next is the natural extension of the level 1 pushdown automaton considered above, where the pushdown store configurations are of level $\leq n$ as opposed to 1. An example configuration of the automaton is



Thus the configurations are triples (q, X, a) where level$(X) \leq n$, a is an individual of the interpretation considered, and q is a state from a given finite set of states. Next we give syntax and semantics of the instructions. Throughout we assume $m \leq n$. The delete instructions are

$$(q,\ A_1 \ldots A_m,\ pop_l,\ p), \text{ where } l \leq m.$$

This instruction is only applicable to configurations $(q,\ X,\ -)$ where

$$X = A_1[A_2[A_3[\cdots[A_{m-1}[A_m X_m]X_{m-1}]\cdots]X_3]X_2]X_1,$$

that is, in order to apply this (and any other) instruction the automato checks the *whole sequence of symbols at the top* of the NPU. The computation step then is the obvious one,

$$(q, X, -) \vdash (p, pop_l(X),\ -).$$

The push instructions are treated totally analogously. They are

$$(q,\ A_1 \ldots A_m,\ push_l(W),\ p)$$

where $l \leq m+1$ if $m < n$ and $l \leq m$ if $m = n$ and W is a non empty word over the storage alphabet. Semantically we get

$$(q, X, -) \vdash (p, push_l(X),\ -).$$

As for operation evaluation we have the instructions $(q, A_1 \ldots A_m, exec(f), p)$ with the obvious semantics. Relation evaluation is introduced totally analogously and we omit the formal definition at this point.

For reasons as stated in the proof ot Theorem 1 (b) we restrict the delete instructions.

Definition 2.4 *We require that for any level n automaton there is a fixed state d such that all instructions deleting in level* 1 *are like* $(-,\, -,\, pop_1,\, d)$.

That is, d is a *fixed* "(post-)deletion in level 1 state ". We do not need such a restriction for $pop_l, l > 1$. It becomes only necessary for the proof of Theorem 3(a).

Finally, a program is a finite sequence (or set) of instructions as above together with a given initial state q_0 and a given initial symbol A_0 for the pushdown store. As we want our programs deterministic and do not want that they abort irregularly, we require that for *each pair* $q, A_1 \ldots A_m$, where $1 \leq m \leq n$ we have exactly one instruction $(q, A_1 \ldots A_m,\, \ldots)$. Given an interpretation and an individual a the automaton starts in the configuration (q_0, A_0, a) and transforms the current configuration in a deterministic way until the store is empty which is the only case in which it stops. The current element in the data storage is the output.

3 Proof of Theorem 2

The proof follows the principles of the proof of Theorem 1 in section 2 above. However it is more complex as already the representation of the computation terms is not so obvious. Our representation derives from that used in [DaGo 86].

Representing computation terms. For a term $r : ind \to ind$ let $Rep(r) \in NPU$ denote the representation to be defined in Definition 3.1. Any computation term r can be decomposed as $r = r_1(r_2(\cdots(r_m(a))\cdots))$ where $r_i : ind \to ind$ as follows from the type structure. We will present r by the following configuration (q will be the normal state of the simulation),

$$r \rightsquigarrow (q,\, Rep(r_m) \ldots Rep(r_1),\, a). \tag{1}$$

Recall that the top of the pushdown store is on the left. The ordering corresponds to our call-by-value semantics; the scheme reduces r_m first, the automaton has access to r_m. Interestingly, subterms of level > 1 will be stored the other way round, in the same ordering in which they occur in the term. We are left with the problem to define $Rep(r)$ for $r : ind \to ind$. If $r = F$ is atomic clearly $Rep(r) = F$. Otherwise $r = s_1(\cdots(s_m(F))\cdots)$, with

$$s_i : (ind \to ind) \to (ind \to ind)$$

and $F : ind \to ind$. We let

$$Rep(r) = F[Rep(s_1) \ldots Rep(s_m)] \text{ (note the ordering).}$$

We are left with the problem to define $Rep(s)$ for

$$s : (ind \to ind) \to (ind \to ind).$$

This goes in the same way. We decompose $s = t_1(\cdots(t_m(G))\cdots)$ where $t_i : \tau \to \tau$ and

$$\tau = (ind \to ind) \to (ind \to ind).$$

Then

$$Rep(s) = G[Rep(t_1) \ldots Rep(t_m)].$$

Clearly this yields a (surprsingly simple) inductive definition of $Rep(r)$ where level$(r) \geq 1$. It is subsumed as

Definition 3.1

$$\begin{aligned} Rep(F) &= F \ \textit{(the atomic case)}, \\ Rep(s_1(\cdots(s_m(G))\cdots)) &= G[Rep(s_1)\dots Rep(s_m)] \ \textit{(non atomic case.)} \end{aligned}$$

Note that a term containing symbols of maximum level n is represented by a pushdown store of level n. A symbol of level l always occurs in nesting depth l of the store. For example for $F : ind \to ind$

$$\begin{aligned} G(H(K))(F) &\rightsquigarrow F[K[GH]], \\ G'(H'(K'))\Big(G(H(L(K)))\Big)(F) &\rightsquigarrow F\Big[K'[G'H']K[GHL]\Big]. \end{aligned}$$

Note that the level of F is 1, that of K, K' is 2 and that of G, H, L, G', H' is 3. The sequence of top symbols in the first example is FKG and in the second example $FK'G'$.

When we consider the configuration encoding the computation term in (1) we see that the head symbol of r_m being responsible for the next reduction step is A_k when $B_1B_2\dots B_{k-1}A_k$ is the sequence of top symbols of the pushdown store $Rep(r_m)\dots Rep(r_1)$.

(Chomsky) normalform. To simulate the reduction steps of a monadic scheme on our pushdown automaton, matters are simplified considerably when we restrict the right hand sides of equations. A level n scheme is in (Chomsky) normalform iff it only has equations $F(y_m)(y_{m-1})\cdots(y_1)(y_0) = t$ where $m \leq n-1$ and the right hand side t is as in one of the following cases (capital letter $F, G, H, \dots$ are function symbols).

Case 1. $G(y_m)(y_{m-1})\cdots(y_1)(y_0)$ (Renaming the head.)

Case 2. $y_m(y_{m-1})\cdots(y_1)(y_0)$, $m \geq 0$
(Deleting the head (projection). The level of the head may decrease.)

Case 3. $G(y_{m-1})\cdots(y_1)(y_0)$, $m > 0$
(The level of the head decreases.)

Case 4. $f(y_0)$ only if $m = 0$
(Operation evaluation. The whole equation reads $F(y_0) = f(y_0)$)

Case 5a. $G\Big(H\Big)(y_m)(y_{m-1})\cdots(y_1)(y_0)$, $m \leq n-2$
(Level of the head increases , level(G) $= m+2$.)

Case 5b. $G\Big(H(y_m)\Big)(y_{m-1})(y_{m-2})\cdots(y_1)(y_0)$

Case 5c. $G(y_m)\Big(H(y_m)(y_{m-1})\Big)(y_{m-2})(y_{m-3})\cdots(y_1)(y_0)$

Case 5d. $G(y_m)(y_{m-1})\Big(H(y_m)(y_{m-1})(y_{m-2})\Big)(y_{m-3})\cdots(y_1)(y_0)$

...

...

Case 5e. $G(y_m)(y_{m-1})\cdots(y_2)\Big(H(y_m)(y_{m-1})\cdots(y_1)\Big)(y_0)$

Case 5f. $G(y_m)(y_{m-1})\cdots(y_1)\Big(H(y_m)(y_{m-1})\cdots(y_1)(y_0)\Big)$

(5c through 5f exhibit the seemingly characteristic copying feature – 5f only when $m > 0$ but $m = 0$ is possible then it is as 5b.)

Case 6. $\quad$ *if* $r(y_0)$ *then* $H(y_m) \cdots (y_0)$ *else* $G(y_m) \cdots (y_0)$ *fi*

An inductive argument some details of which are presented in Section 5 shows

Lemma 3.2 *Each monadic level n scheme can be transformed into an equivalent level n scheme in normalform.*

Construction of the automaton. Given the level n scheme

$$F_0(y_0) = t_0,\ F_1 \downarrow =\ t_1,\ \ldots,\ F_m \downarrow =\ t_m$$

in normalform we construct an equivalent level n pushdown automaton. The construction is based on the representation of computation terms as configurations as given above. Each equation

$$\mathbf{F}(y_m)(y_{m-1}) \cdots (y_1)(y_0)\ =\ t$$

where $m \le n-1$ of the scheme is simulated by a dedicated set of instructions, depending on which case of the normalform applies to t. The state q is our normal state Γ is the storage alphabet, it simply consists of the function symbols and operations of the scheme.

Case 1.

$$(q,\ K_1 \ldots K_m\mathbf{F},\ \mathit{push}_{m+1}(G),\ q),\ \text{ for } \mathit{all}\ K_i.$$

The K_i will be universally quantified throughout. This reflects that the reduction step depends only on the head **F**, the rest of the term is not "read", but only formally plugged into the equation. The automaton on the other hand is required to read *all* top symbols in order to determine which instruction to execute.

Case 2.

$$(q,\ K_1 \ldots K_m\mathbf{F},\ \mathit{pop}_{m+1},\ q)\ \text{ again for } \mathit{all}\ K_i.\ \text{Recall Definition 2.4 .}$$

Only the **F** is deleted.

Case 3.

The restriction in Definition 2.4 causes a minor problem. First we treat the easy case ,$m > 1$. An example reduction step with $m = 2$,

$$\mathbf{F}(\mathbf{A})\Big(D(E)(H)\Big)(a)\ \Rightarrow\ \mathbf{I}\Big(D(E)(H)\Big)(a),$$

the $\mathbf{F}(\mathbf{A})$ is substituted with the **I**. We have

$$\mathit{Rep}\Big(\mathbf{F}(\mathbf{A})\Big(D(E)(H)\Big)\Big)\ =\ H[\mathbf{A}][\mathbf{F}]E[D]],\ \mathit{Rep}\Big(\mathbf{I}\Big(D(E)(H)\Big)\Big)\ =\ H[\mathbf{I}E[D]]$$

The first (set of) instructions is

$$(q,\ K_1 \dots K_m\mathbf{F},\ pop_m, qF1),\ \ qF1 \text{ a new state },$$

and as $m > 1$ Definition 2.4 is respected.

In our example we get

$$H[\mathbf{A}[\mathbf{F}]E[D]] \vdash H[E[D]].$$

The next instruction (set) is

$$(qF1,\ K_1 \dots K_{m-1}L_1 \dots L_k,\ push_m(GL_1),\ qF2) \ \text{ all } K_i, L_i, k,\ m-1+k \leq n.$$

In our example

$$H[E[D]] \vdash H[\mathbf{I}[D]E[D]] \ \ (D \text{ is copied.})$$

Finally we have

$$(qF2,\ K_1 \dots K_{m-1}GL_2 \dots L_k,\ pop_{m+1},\ q) \text{ all } K_i,\ L_i, k.$$

In the example

$$H[\mathbf{I}[D]E[D]] \vdash H[\mathbf{I}E[D]],$$

and the instructions fulfill their task.

Now we come to the case where $m = 1$. The equation reads

$$F(y_1)(y_0) \ = \ G(y_0),$$

here the first of our instructions above would violate Definition 2.4. Instead we first push the G and then pop iteratively until level 2 is empty. The instructions are (for all K)

$$(q,\ KF,\ push_1(G),\ qF1),\ \ (qF1,\ GK,\ pop_2,\ qF1),\ (qF1,\ G,\ push_1(G),\ q),$$

and we end up in the normal state. In the case $m > 1$ we could avoid this iterative deleting in level $m+1$ because the same effect could be obtained by deleting once in level m, but with a different state.

Case 4.

$$(q,\ F,\ exec(f),\ q_F),\ \ (q_F,\ F,\ pop_1,\ q) \text{ (Recall Definition 2.4).}$$

Case 5a.

$$(q,\ K_1 \dots K_m\mathbf{F},\ push_{m+1}(\mathbf{H}),\ qF1),\ (qF1,\ K_1 \dots K_m\mathbf{H},\ push_{m+2}(G),\ q)$$

Case 5b.

We simply get the instruction

$$(q,\ K_1 \dots K_m\mathbf{F},\ push_{m+1}(GH),\ q).$$

An example with $m \ = \ 2$ is

$$\mathbf{F}\Big(B(C)(D)(A)\Big)(E(I))(a) \ \Rightarrow\ \mathbf{G}\Big(\mathbf{H}\,(B(C)(D)(A))\ \Big)(E(I))(a)$$

and the instruction in fact yields the correct transformation of the representation

$$I[A[\mathbf{F}[D[C[B]]]E \;\vdash\; I[A[\mathbf{GH}D[C[B]]]E.$$

Case 5c.

We get the instructions

$$\begin{aligned}(q,\, K_1 \ldots K_m\mathbf{F},\, \mathit{push}_{m+1}(\mathbf{H}),\, qF1),\\ (qF1,\, K_1 \ldots K_m\mathbf{H},\, \mathit{push}_m(K_mK_m),\, qF2)\ \text{(copying } y_m\text{)},\\ (qF2,\, K_1 \ldots K_m\mathbf{H},\, \mathit{push}_{m+1}(G),\, q).\end{aligned}$$

This calls for an example.

$$\mathbf{F}\,(A(B))\,(E(L))\,(I(J))\,(a) \;\Rightarrow\; \mathbf{G}\,(A(B))\,\Big(\mathbf{H}(A(B))(E(L))\Big)(I(J))(a)$$

Our instructions, here we have the case $m = 3$, yield

$$\begin{aligned} & J\Big[L\,[B\,[\mathbf{F}\,A\,]\,E\,]\,I\,\Big]\\ \vdash\;\; & J\Big[L\,[B\,[\mathbf{H}\,A\,]\,E\,]\,I\,\Big]\\ \vdash\;\; & J\Big[L\,[\mathbf{B}\,[\mathbf{H}\,A\,]\,\mathbf{B}\,[\mathbf{H}\,A\,]E\,]\,I\,\Big]\\ \vdash\;\; & J\Big[L\,[\mathbf{B}\,[\mathbf{G}\,A\,]\,\mathbf{B}\,[\mathbf{H}\,A\,]E\,]\,I\,\Big],\end{aligned}$$

and the representation of the first computation term is transformed into the representation of the second computation term.

Cases 5d. through f.

These cases are treated analogously, only the m of the second instruction above needs to be adapted. We get

$$\begin{aligned}(qF1,\, K_1 \ldots K_m\mathbf{H},\, \mathit{push}_{m-1}(K_{m-1}K_{m-1}),\, qF2)\ \ \text{for 5d and}\\ (qF1,\, K_1 \ldots K_m\mathbf{H},\, \mathit{push}_{m-2}(K_{m-2}K_{m-2}),\, qF2)\end{aligned}$$

for the next (omitted) the case. The cases indicated with the dots are treated alike, adapting the level of the *push* instruction as before. In case 5e the aforementioned *push* operation is push_2.

$$\mathbf{F}(A)(B)(C)(D)(a) \;\Rightarrow\; \mathbf{G}(A)(B)(C)\,\Big(\mathbf{H}(A)(B)(C)(D)\Big)(a).$$

the simulation runs as

$$\begin{aligned} & D\Big[\,C\,[\,B\,[\,A\,[\,\mathbf{F}\,]\,]\,]\,\Big]\\ \vdash\;\; & D\Big[\,C\,[\,B\,[\,A\,[\,\mathbf{H}\,]\,]\,]\,\Big]\\ \vdash\;\; & D\Big[\,C\,[\,B\,[\,A\,[\,\mathbf{H}\,]\,]\,]\;C\,[\,B\,[\,A\,[\,\mathbf{H}\,]\,]\,]\,\Big]\\ \vdash\;\; & D\Big[\,C\,[\,B\,[\,A\,[\,\mathbf{G}\,]\,]\,]\;C\,[\,B\,[\,A\,[\,\mathbf{H}\,]\,]\,]\,\Big]\end{aligned}$$

Case 5f.

Due to the call-by-value reduction strategy of the semantics, this case is treated differently from the preceding cases. We have to push *first* the G and then the H. For $m > 0$ we get the instructions

$$(q,\, K_1 \dots K_m\mathbf{F},\, \mathit{push}_{m+1}(\mathbf{G}),\, qF1),$$
$$(qF1,\, K_1 \dots K_m\mathbf{G},\, \mathit{push}_1(K_1K_1),\, qF2),$$
$$(qF2,\, K_1 \dots K_m\mathbf{G},\, \mathit{push}_{m+1}(\mathbf{H}),\, q).$$

For $m = 0$ the second push is omitted and we use $qF1$ instead of $qF2$.

Another example

$$\mathbf{F}(A)(a) \;\Rightarrow\; \mathbf{G}(A)(\mathbf{H}(A)(a))$$

leads to

$$A[\mathbf{F}] \;\vdash\; A[\mathbf{G}] \;\vdash\; A[\mathbf{G}]\,A[\mathbf{G}] \;\vdash\; A[\mathbf{H}]\,A[\mathbf{G}]$$

and the head symbol **H** is accessible to the automaton, see also (1).

Case 6.

This case is easy we state it for completeness. The instructions are

$$(q,\, K_1 \dots K_m\mathbf{F},\, \mathit{exec}(r),\, pH,\, pG),$$
$$(pH,\, K_1 \dots K_m\mathbf{F},\, \mathit{push}_{m+1}(H),\, q),$$
$$(pG,\, K_1 \dots K_m\mathbf{F},\, \mathit{push}_{m+1}(G),\, q).$$

This brings the construction of the automaton to the end. To be fully in line with the definition some dummy instrutions – never occurring in simulations – must be added. As the simulation of each single equation is correct the automaton is equivalent to the initial scheme.

4 Proof of Theorem 3 (sketch)

The syntax and semantics of schemes with general homogeneous types is as for the monadic case we give a characteristic

Example (continued from page 6). We let

$$G : (\mathit{ind} \to \mathit{ind}) \times (\mathit{ind} \to \mathit{ind}) \to (\mathit{ind} \to \mathit{ind})$$

and a, b, C as before. Our scheme is

$$\begin{aligned} F(y_0) &= G(a,\, b)(y_0) && \text{(Here } a, b \text{ are the operation symbolsd.)}\\ G(y_1, y_2)(y_0) &= \mathit{if}\ r(y_0)\ \mathit{then}\ y_2(y_1(y_0))\ \mathit{else}\ G(C(y_1), C(y_2))(b(y_0))\ \mathit{fi} \\ & \qquad \text{(Generation twice the same number of } C\text{'s as } b\text{'s.)} \\ C(y_1)(y_0) &= y_1(y_1(y_0)) \\ & \qquad \text{(Iterated copying depending on the number of } C\text{'s.)} \end{aligned}$$

The semantics is given by the call-by-value reduction strategy. The semantics of the scheme over the interpretation as on page 6 is $a^n \mapsto b^{2^n}a^{2n}b^na^n$ and undefined for the remaining arguments.

We do not see how to get an equivalent monadic scheme. However, a level 2 automaton can easily be obtained: Generate n C's on level 2. Then execute $push_1(AB)$ copying the C's. Finally delete the C's. With each deletion execute $push_1(AA)$ first , then for each A append the a on the data store. This yields the a^{2^n}. After this our store reads $B[C \ldots C]$ and we proceed in the same way with the B's and b's.

Proof of Theorem 3 (a). We are given a level n pushdown automaton. We need to represent configurations from NPU as computation terms. Our representation almost the same as the one used in [DaGo 86]. The head symbol of the term must consist of *all* top symbols of the store plus the state of the automaton. The function symbols of level 1 , that is of type $ind \to ind$ are all symbols like qA, q any state and $A \in \Gamma$. Those of level 2 are qA_1A_2 and $A_i \in \Gamma$. Their type is $\tau = (ind \to ind)^{\times m} \to (ind \to ind)$ where $m = \sharp$ function symbols of level 1, that is $\sharp qA$'s. This can naturally be extended to higher levels. Thus the symbols of level 3 are all $qA_1A_2A_3$. Their type is throughout $\tau^l \to \tau$ where $l = \sharp$ symbols of level 2.

The representation of a given NPU is defined decreasing with respect to the nesting level in which the NPU "starts". Rep_l is the representation of an NPU which is considered as being nested in level l, its nesting level itself in this case is $\leq n - l + 1$. It will turn out that

$$Rep_l(X) = \begin{cases} \text{a (argument) list of level } l-1 \text{ terms for } l \geq 2, \\ \\ \text{a single level 1 term for } l = 1. \end{cases}$$

For $W = \mathbf{A}V$ a non empty word over Γ we have

$$Rep_n(W) = (F_1\mathbf{A}Rep_n(V), \ldots, F_k\mathbf{A}Rep_n(V)),$$

where the F_i are *all* level $n-1$ symbols and $F_i\mathbf{A}$ is to be understood as the appropriate level n symbol. If W is empty we simply get $Rep_n(W) = (F_1, \ldots, F_k)$. And for $2 \leq l < n$ we get in the same way for a non-empty NPU decomposed as

$$X = A_1[A_2[A_3[\cdots[A_{m-1}[A_mX_m]X_{m-1}]\cdots]X_3]X_2]X_1 \in NPU,\ m \leq n-l+1$$

$$\begin{aligned} Rep_l(X) \quad = \quad & \Big(G_1\mathbf{A_1}\ldots\mathbf{A_m}Rep_{l+m-1}(X_m)\, Rep_{l+m-2}(X_{m-1})\ldots Rep_l(X_1), \ldots \\ & \ldots \\ & G_k\mathbf{A_1}\ldots\mathbf{A_m}Rep_{l+m-1}(X_m)\, Rep_{l+m-2}(X_{m-1})\ldots Rep_l(X_1)\Big), \end{aligned}$$

where the G_i now are all symbols of level $l-1$ and $G_i\mathbf{A_1}\ldots\mathbf{A_m}$ is to be understood as the respective symbol of level $l+m-1$. If X is empty we just get

$$Rep_l(X) = (G_1, \ldots, G_k).$$

For $l = 1$ and $X = A_1[Y]$ that is, referring to our decomposition above we have X_1 empty, we let

$$Rep_1(X) = \mathbf{d}A_1\ldots A_m(Rep_m(X_m))(Rep_{m-1}(X_{m-1}))\cdots(Rep_2(X_2))$$

where $\mathbf{d}$ is our unique fixed deletion state, cf. Definition 2.4.

Finally we consider a general configuration $(q,\ X,\ a)$ with X now decomposed as above and $X_1 = Z_1 \ldots Z_k$ where the Z_i (necessarily of the form $B[Y]$) are all NPU's of X_1 starting in level 1. We represent this configuration as

$$Rep_1(Z_k)\Big(Rep_1(Z_{k-1})\Big(\cdots\Big(Rep_1(Z_1)$$
$$\Big(qA_1 \ldots A_m(Rep_m(X_m))\cdots(Rep_2(X_2))(a)\Big)\Big)\cdots\Big)\Big).$$

Note that $Rep_1(Z_i) = \mathbf{d}B_i\ldots$ reflecting our call-by-value strategy on the one hand and Definition 2.4 on the other hand. The instruction
$(q, A_1 \ldots A_m, pop_1,\ \mathbf{d})$ is simulated by the equation

$$qA_1 \ldots A_m(\overline{y_m})\cdots(\overline{y_1})(y_0) \ = \ y_0,$$

which results in a correct simulation. Higher level deletion is simulated by a projection onto one of the arguments. Note that our (long) lists of arguments allow to pick the right projection, such that the head symbol after the projection corresponds to the top symbols and state of the automaton. To simulate the push operation causes no principal difficulties any more, we need just translate it into the representation. Note that $push_1$ is to be treated differently from $push_l$, $l > 1$.

Proof of Theorem 3(b). We can essentially reduce the proof to the monadic case. First observe that we can restrict attention to right hand sides in normalform. The right hand side of an equation $F(\overline{y_m})\cdots(\overline{y_1})(y_0) \ = \ t$ in normalform looks as follows (analogously to the monadic case.) We list only a few not totally obvious cases.

Case 2. $y_{m,k}(\overline{y_{m-1}})\cdots(\overline{y_1})(y_0), m \geq 0$ where
$y_{m,k}$ the k'th parameter of y_m (Deleting the head (projection). The level of the head may decrease.)

Case 5a. $G\Big(\overline{H}\Big)(\overline{y_m})(\overline{y_{m-1}})\cdots(\overline{y_1})(y_0),\ m \leq n-2$
(Level of the head increases , level$(G) = m+2$.)

Case 5b. $G\Big(H_1(\overline{y_m}), \cdots, H_k(\overline{y_m})\ \Big)(\overline{y_{m-1}})(\overline{y_{m-2}})\cdots(\overline{y_1})(y_0)$
$\ldots$
$\ldots$

Inductively we have the following

Lemma 4.1 *Let $(t_1, t_2, \ldots, t_{m-1},\ t_m)$ be a list of arguments occurring somewhere in a computation term of a scheme in normalform. Then we have symbols H_i all of the same argument type and one list of terms $\overline{s} = (s_1,\ \ldots,\ s_k)$ such that $t_i = H_i(\overline{s})$ for all i.*

Note that this holds trivially for monadic schemes. Now writing $H_1 \ldots H_k(\overline{s})$ for the list in the lemma and proceeding in this way inductively we can translate such a computation term into a monadic term. We consider $H_1 \ldots H_k$ as *one* symbol. We are somewhat generous here because the H_i have no common result type. But this causes no problem because we never hit on a term like $H_1 \dots H_k(\overline{s})(t)$ in which case we would need the result type. Instead we have a projection before leading to something like $H_i(\overline{s})(t)$.

With the preceding reduction in mind ourn representation of terms now is as in the monadic case. Concerning the simulation we only consider

Case 2.

Here we have to distinguish 2 subcases. If $m = 0$ we only have $k = 1$ simulate with $(q,\ F, pop_1,\ q)$ recall Definition 2.4. For $m > 0$ we have

$$(q,\ K_1 \cdots K_m F, pop_{m+1},\ qF), (qF, K_1 \dots K_m H_1 \dots H_l, push_{m+1}(H_k), q)$$

for all $H_1 \dots H_l$ considered as one symbol of the storage alphabet.

5 The (Chomsky) normalform proof (short sketch)

The normalform proof relies on the following observation: If we have a general equation $F(\overline{y_m}) \cdots (\overline{y_1})(y_0) \ = \ t$ which is not an $if - then - else$ then we can decompose

$$t \ = \ t_1(\overline{t_2})(\overline{t_3}) \cdots (\overline{t_m})(t_0)$$

with level(t_i) decreasing from left to right. This implies due to homogeneity that t_0 can contain all parameters, $\overline{t_1}$ all except of y_0, $\overline{t_2}$ all except y_0 and $\overline{y_1}$ .... Using the cases numbered 5a to 5f of the Chomsky normalform we can inductively unwind the right hand sides. The remaining cases are essentially for the end of this process when no unwinding is possible any more. We leave it at these remarks.

Conclusion

Guided by the formal language theoretic result of [DaGo 86] we have extended the classical recursion –pushdown relation to higher type concepts. Some questions may be worth of further study.

- Can we extend the results to non homogeneous types? Or are de Bakker Scott schemes with non homogeneous types provably more powerful?
- With real programming languages in mind, one may ask how this can be transferred to programming schemes with several variables of base type *ind*. What about most general language here the typed λ-terms?
- Concerning formal language theory one may ask, if ideas used here may lead to an automata theoretical characterization of languages from the "IO-hierarchy" [Da 82]. Here the derivation mode is call-by-value, therefore the question at this place. Perhaps the difference between IO and OI only is the fixed deletion state.
- How efficient is it to use the nested pushdown for an actual implementation of higher type recursion with appropriate type concept?
- Semantic hierarchy results for the de Bakker Scott schemes considered here should be provable.

References

[Da 82] Werner Damm. The IO- and OI- hierarchies. Theoretical Computer Science 20, 1982, 95 – 207.

[DaGo 86] Werner Damm, Andreas Goerdt. An automata theoretic characterization of the OI-hierarchy. Information and Control 71, 1986, 1 – 32.

[In 79] Klaus Indermark. Vorlesung Kontrollstrukturen WS 1979/80 (Course on control structures, winter term 1979/80) RWTH Aachen.

[LoSi 84] Jacques Loeckx, Kurt Sieber. The foundations of program verification. Wiley-Teubner 1984.

[Ma 76] A. N. Maslov. Multilevel stack automata. Problemy Peredachi Informatsii 12 (1) , 1976, 55 – 62.

[Sh] J. C. Shepherdson. Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory. In Harvey Friedman's Research on the Foundations of Mathematics, Studies in Logic and the Foundations of Mathematics 117, edited by L. A. Harrington et al., North Holland.