Logic Column 9

Column Editor: Jon G. Riecke
Bell Laboratories, Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974
riecke@bell-labs.com

Programming languages capturing complexity classes*

Martin Hofmann
Laboratory for Foundations of Computer Science
University of Edinburgh
(mxh@dcs.ed.ac.uk)

February 3, 2000

1 Introduction

In recent years there has been a growing interest in programming languages with the property that all definable functions belong to some complexity class, often polynomial time or space. This research has established a link between computational complexity theory and the fields of type systems and programming language semantics with emerging applications to resource certification and programming under resource restrictions as arise in embedded systems. The purpose of this article is to survey this development beginning from Cobham's 1965 result which was the first in the area, up to the current research frontier.

The starting point is the question: how can we tame primitive recursion, i.e. restrict it so that it only defines polynomial time computable functions? The first obstacle is that ordinary primitive recursion generally makes an exponential number of recursive calls in terms of the binary length $|x| = \lceil \log_2(x+1) \rceil$, i.e., the number of bits needed to write down x. This may be overcome by replacing primitive recursion with the scheme of *recursion on notation*: define $f(x, \vec{y})$ from $g(\vec{y})$ and $h(x, \vec{y}, z)$ by

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}) \\ f(x, \vec{y}) &= h(x, \vec{y}, f(\lfloor x/2 \rfloor, \vec{y})), \text{ when } x > 0 \end{aligned} \tag{1}$$

An evaluation of $f(x, \vec{y})$ by rewriting its defining equations requires $|x|$ recursive calls.

Unfortunately, recursion on notation is too weak a restriction. Consider,

$$\begin{aligned} q(0, y) &= y \\ q(x, y) &= 4 \cdot q(\lfloor x/2 \rfloor, y), \text{ if } x > 0 \end{aligned} \tag{2}$$

We then have $q(x,y) = [x]^2 \cdot y$ where $[x] = 2^{|x|}$, hence $|q(x,y)| = 2(|x|+1)+|y|$ Now put

$$\begin{aligned} e(0) &= 1 \\ e(x) &= q(e(\lfloor x/2 \rfloor), 1), \text{ if } x > 0 \end{aligned} \tag{3}$$

and we obtain (by repeated squaring) a function of superpolynomial growth: $e(x) \geq 2^{[x]}$ thus leading outside polynomial time and in fact neutralising the benefit of recursion on notation.

2 Bounded recursion on notation

One can rule out the definition of $e(x)$ by requiring an *a priori* bound on the function to be defined. Formally: a function $f(x,\vec{y})$ is defined from $g(\vec{y}), h(x,z,\vec{y}), k(x,\vec{y})$ by *bounded recursion on notation* if f is defined from g,h by recursion on notation and moreover $f(x,\vec{y}) \leq k(x,\vec{y})$.

In order to get off the ground we need to provide a basic function giving us polynomial-sized bounds. This can be done in the form of the smash function $x \# y = 2^{|x|\cdot|y|}$. Notice that $|x|^k = |x \# \ldots \# x|$ (k factors).

Of course, we also need a few other basic functions, e.g., the set $\mathcal{B}$ consisting of the binary successor functions $\mathsf{S}_0(x) = 2x, \mathsf{S}_1(x) = 2x+1$, the constant 0, parity, integer division by two, and a ternary conditional.

Cobham's well-known result [8] states that the class of functions definable by bounded recursion on notation and composition from $\mathcal{B}$, projections, and the smash function equals the class of polynomial time computable functions.

The proof is simple: for one direction one defines concretely the one-step and I/O functions of a given Turing machine. Iterating this one-step function a polynomial number of times (using the smash function) and composing with the I/O functions gives the result. The other direction, i.e., that all definable functions are polynomial time computable, proceeds by showing that under the bounding condition $f(x,y) \leq k(x,y)$ the obvious for-loop computing f from g,h runs in polynomial time because the size of all intermediate values is polynomially bounded.

This result was the first machine-independent characterisation of a complexity class and "was the start for modern complexity theory, indicating a robust and mathematically interesting field. [7]"

Another approach to imposing *a priori* bounds is finite model theory: rather than requiring that a particular instance of recursion on notation be bounded we can interpret everything in a finite domain $\{0,1,\ldots,N\}$ for some fixed N. Increasing basic functions such as successor then have to cut off at size overflow. A result due to Gurevich (see [15]) asserts that ordinary primitive recursion modified in this way captures (in a certain natural sense) the complexity class "logarithmic space". See also [14].

3 Safe recursion and tiering

Cobham's characterisation of polynomial time and the finite model approach can be criticised on the grounds that although they do not mention a particular machine model they still involve resources in the form of size bounds or the domain size.

Moreover, Cobham's system suffers from the defect that it is undecidable to tell whether a given definition is legal because of the side condition that k bounds the function to be defined. As has been done in [10] this problem can be overcome by changing the semantics of bounded recursion so as to cut off at size overflow, see Eqn. 9 below. Then, however, like in the finite model approach,

we defer the burden of establishing resource bounds to the verification: in order that the usual recursion equations (Eqn. 1) hold, we have to prove that $f(x, \vec{y}) \leq k(x, \vec{y})$.

More "intrinsic" characterisations of polynomial time were given by Bellantoni-Cook [3] and Leivant-Marion [23, 22] under the name *safe*, resp. *tiered* recursion. The crucial insight behind both is the observation that the "legal" definition of $q(x, y)$ in Eqn. 2 and the "illegal" definition of $e(x, y)$ in Eqn. 3 differ qualitatively in the following aspect. In Eqn. 2 the value $q(x, y)$ is obtained by applying a composition of basic functions to the recursive call $q(\lfloor x/2 \rfloor, y)$. On the other hand, in the second equation of (3) we apply a recursively defined function, namely $q(-, 1)$ to the result of a recursive call, namely $e(\lfloor x/2 \rfloor)$.

Both safe and tiered recursion are aimed at preventing such "illegal" recursions. In safe recursion [3] the variables of every function are divided into two zones like so $f(\vec{x}; \vec{y})$. The variables before the semicolon are called *normal*; the ones after the semicolon are called *safe*. The idea is that $f(\vec{x}\,;\vec{y})$ will make arbitrary use of its normal arguments, in particular feed them as input to recursively defined operations; but will only apply basic functions to its safe arguments (which these are need not be determined statically). In order to prevent definitions like Eqn. 3, one then requires that results of recursive calls are further processed only via safe arguments. More formally, given $g(\vec{x}\,;\vec{y}), h(\vec{x}, x\,;\vec{y}, z)$ the function defined from g, h by *safe recursion on notation* is the function $f(\vec{x}, x\,;\vec{y})$ given by

$$\begin{aligned} f(\vec{x}, 0\,;\vec{y}) &= g(\vec{x}\,;\vec{y}) \\ f(\vec{x}, x\,;\vec{y}) &= h(\vec{x}, x\,;\vec{y}, f(\vec{x}, \lfloor x/2 \rfloor;\vec{y})), \text{ if } x > 0 \end{aligned} \tag{4}$$

In order to maintain the invariant that safe variables are not subjected to recursive unfolding we must also restrict composition: given $u_1(\vec{x};) \ldots u_m(\vec{x};)$ and $v_1(\vec{x}\,;\vec{y}) \ldots v_n(\vec{x}\,;\vec{y})$ and $g(\vec{a}\,;\vec{b})$ the function $f(\vec{x}\,;\vec{y})$ defined from $\vec{u}, \vec{v}, g$ by *safe composition* is given by

$$f(\vec{x}\,;\vec{y}) = g(\vec{u}(\vec{x};)\,;\vec{v}(\vec{x}\,;\vec{y})) \tag{5}$$

In other words, we may plug anything into a safe position, but nothing depending on a safe variable may be plugged into a normal position.

The basic functions in $\mathcal{B}$ are dubbed safe in all their arguments. Projections are available from both safe and normal positions. The availability of projections means that we can always "demote" a variable from the safe zone into the normal zone, i.e., from $f(\vec{x}\,;\vec{y}, y)$ we obtain $f(\vec{x}, y\,;\vec{y})$ by substituting $u(\vec{x}, y\,;\vec{y}) = y$ for y in f. Moving variables from the normal zone into the safe zone is not possible for this would require to substitute a term depending on safe variables into a normal position.

For example, the function $q(x, y)$ from Eqn. 2 above is definable by safe recursion on notation with x normal and y safe by

$$\begin{aligned} q(0\,;y) &= y \\ q(x\,;y) &= \mathsf{S}_0(;\mathsf{S}_0(;q(\lfloor x/2 \rfloor\,;y))) \end{aligned} \tag{6}$$

Here $g(x\,;y) = y$ is a projection from the safe zone and $h(x\,;y,z) = \mathsf{S}_0(;\mathsf{S}_0(;z))$ is obtained from basic functions by safe composition and another projection.

There is no way to turn the definition of $e(x)$ into a safe recursion: the step function in Eqn. 3 would be $h(z;) = q(z\,;1)$ but that is not allowed as the recursion variable z is in the normal zone, i.e., is being recursed on.

It is, however, perfectly legal to feed the result of a recursive call into the second, safe, argument of $q(x\,;y)$ like in

$$\begin{aligned} f(0, y;) &= y \\ f(x, y;) &= q(y\,;f(\lfloor x/2 \rfloor, y;)) \end{aligned} \tag{7}$$

We have $|f(x,y;)| = 2|x||y|$ or $f(x,y) = 2^{2|x||y|}$, hence we can define functions of polynomial growth rate from scratch and do not need the smash function to get started.

The main result of [3] asserts that a function $f(\vec{x})$ is polynomial time computable iff $f(\vec{x};)$ is definable by safe recursion and composition from the abovementioned basic functions.

The proof that all polynomial-time functions are definable proceeds by induction on definitions in Cobham's system. One shows that for each polynomial time computable function $f(\vec{y})$ there exists a definition $g(x\,;\vec{y})$ and a polynomial p so that $f(\vec{y}) = g(x\,;\vec{y})$ provided $|x| > p(|\vec{y}|)$.

The other direction, i.e., that all definable functions are polynomial time computable is a consequence of the following lemma.

Lemma if $f(\vec{x}\,;\vec{y})$ is definable in Bellantoni-Cook's system then $f(\vec{x}\,;\vec{y})$ is polynomial time computable and, moreover,

$$|f(\vec{x}\,;\vec{y})| \leq p(|\vec{x}|) + \max(|y_1|, \ldots, |y_n|) \tag{8}$$

for some polynomial p.

This allows us in particular to get an *a priori* bound for a function defined by safe recursion on notation from the polynomials for the functions g and h.

We emphasise that in safe recursion neither explicit bounds are mentioned nor explicit reference to polynomials is made (as is done in Cobham's system via the smash function).

The above lemma does not characterise the definable functions: these satisfy the stronger property that the value $f(\vec{x}\,;\vec{y})$ only depends upon the last $p(|\vec{x}|)$ bits of the $\vec{y}$ variables. This points to an intrinsic limitation of safe recursion to which we return in Section 7.

Tiering In Leivant-Marion's systems of tiered recursion every variable and every function comes equipped with a level or *tier*; basic functions are available at all tiers, composition must preserve tiers, i.e., we can substitute a tier i term for a tier i variable but not for a variable of tier lower or greater than i. An instance of recursion on notation is legal if the tier of the recursion variable is higher than the tier of the recursively defined function. For example, we can define $q(x,y)$ of tier 0 with x of tier 1 and y of tier 0. More generally, we can define $q(x,y)$ as of tier i with x of tier greater than i and y of tier i. What is important is that for a recursive definition to be legal the tier of the variable z in the step function $h(x,\vec{y},z)$ from Eqn. 1 must have the same tier as h and this is for example not the case in Eqn. 3. It does, however, hold for Eqn. 7. This is not accidental: it can be shown by induction that whenever $f(\vec{x}\,;\vec{y})$ admits a definition in Bellantoni-Cook's system then for each result tier i there exists a definition of $f(\vec{x},\vec{y})$ in Leivant-Marion's system with the safe variables $\vec{y}$ of tier i and the normal variables $\vec{x}$ of tier greater than i. The converse also holds in the sense that the variables of tier equal to the tier of the function become safe and the ones of higher tier become normal. It is easy to see that the value of a function cannot depend on arguments of lower tier.

Other complexity classes We remark that a number of similar systems have been proposed which capture other complexity classes both below and above polynomial time. Typically, these rely on other recursion schemes and—in the case of bounded recursion also on replacing the smash function with functions of higher or lower growth rates. See [7] for a survey.

4 Higher-order functions

So far we have dealt with first-order functions on integers. Let us now see what happens if we include higher-order functions.

The simplest thing we can do is to conservatively extend the systems with function types, i.e., "build a lambda calculus around them". For bounded recursion on notation this has been done by Cook and Urquhart [10]. Their system PV^ω is a simply-typed lambda calculus with a basic type N for integers, constants for the basic functions, e.g., $\mathsf{S}_0, \mathsf{S}_1 : \mathsf{N}\to\mathsf{N}$ or $\# : \mathsf{N}\to\mathsf{N}\to\mathsf{N}$ and a second order constant (recursor) for bounded recursion on notation:

$$\mathcal{R} : \mathsf{N}\to(\mathsf{N}\to\mathsf{N}\to\mathsf{N})\to(\mathsf{N}\to\mathsf{N})\to(\mathsf{N}\to\mathsf{N})$$

One can interpret PV^ω in the full type hierarchy over the integers, i.e., interpret N as $\mathbb{N}$ and $\to$ as the set of functions. The recursor is interpreted by $\mathcal{R}(g,h,k) = f$ if f is defined by

$$\begin{aligned} f(0) &= g \\ f(x) &= \min(k(x), h(x, f(\lfloor x/2 \rfloor))), \text{ if } x > 0 \end{aligned} \tag{9}$$

Notice that no undecidable side condition on applicability of $\mathcal{R}$ is needed anymore. Notice also that parameters are not explicitly mentioned as in lambda calculus terms may contain free variables.

One can show by normalisation or using a semantical argument that if $t : \mathsf{N}\to\mathsf{N}$ is a *closed* term of PV^ω then (the interpretation of) t is polynomial time computable. Cook and Urquhart use this result to show that whenever a $\forall\exists$ statement is provable in Buss' *bounded arithmetic* [5] then this statement can be witnessed by a polynomial time computable function.

Something similar can be done for Bellantoni-Cook's system, see [16]; the syntactic restrictions are then conveniently formulated using a unary operator $\Box$ on types which obeys typing rules reminiscent of S4 modal logic: the safe recursor then takes the form of a constant

$$\mathcal{R} : \mathsf{N}\to(\Box(\mathsf{N})\to\mathsf{N}\to\mathsf{N})\to\Box(\mathsf{N})\to\mathsf{N}$$

The rules for the $\Box(-)$ type former are[1]

- if $e : \Box(\tau)$ then $e : \tau$,
- if $e : \tau$ and all free variables in e have types of the form $\Box(-)$ then $e : \Box\tau$ ("necessitation").

A function with the first argument normal and the second one safe corresponds in this scheme to a term of type $\Box(\mathsf{N})\to\mathsf{N}\to\mathsf{N}$.

Like the separation of variables into safe and normal this new type former is merely an annotation aimed at restricting the definable functions; it has no effect on the semantics.

Let us see how we typecheck the definition of $q(x\,;y)$ in this system:

$$h = \lambda x{:}\Box(\mathsf{N}).\lambda z{:}\mathsf{N}.\mathsf{S}_0(\mathsf{S}_0(z)) : \Box(\mathsf{N})\to\mathsf{N}\to\mathsf{N}$$

So,

$$q = \lambda x{:}\Box(\mathsf{N}).\lambda y{:}\mathsf{N}.\mathcal{R}(y,h,x) : \Box(\mathsf{N})\to\mathsf{N}\to\mathsf{N}$$

Let us try to define $e(x)$. We would have to put

$$h = \lambda x{:}\Box\mathsf{N}.\lambda z{:}\Box(\mathsf{N}).q(z,\mathsf{S}_1(0)) : \Box(\mathsf{N})\to\Box(\mathsf{N})\to\Box(\mathsf{N})$$

[1] These rules are not closed under β-reduction. Formulations for which this is the case have been given in [16] and [26, 25] in the context of partial evaluation.

But this does not have the type of the second argument to $\mathcal{R}$. The definition in Eqn. 7 can, however, be done: we put

$$h = \lambda y{:}\Box(\mathsf{N}).\lambda n{:}\Box(\mathsf{N}).\lambda z{:}\mathsf{N}.q(y,z) : \Box(\mathsf{N}){\to}\Box(\mathsf{N}){\to}\mathsf{N}{\to}\mathsf{N}$$
$$f = \lambda x{:}\Box(\mathsf{N}).\lambda y{:}\Box(\mathsf{N}).\mathcal{R}(y,h(y),x)$$

The necessitation rule is used for safe composition. For example, in Bellantoni-Cook's system we can form $u(x\,;y) = q(q(x\,;1)\,;y)$ and $v(x,y;) = q(q(x\,;y)\,;y)$. In the higher-order system we put

$$u = \lambda x{:}\Box(\mathsf{N}).\lambda y{:}\mathsf{N}.q(q(x,\mathsf{S}_1(0)),y)$$
$$v = \lambda x{:}\Box(\mathsf{N}).\lambda y{:}\Box(\mathsf{N}).q(q(x,y),y)$$

where for example in the second equation we use the fact that $q(x,y) : \Box(\mathsf{N})$ provided that $x, y : \Box(\mathsf{N})$.

Again, one can show that all definable first-order functions are polynomial time computable and, moreover, similar to Eqn. 8.

5 Higher-type recursion

In the previous section higher-order functions played an auxiliary role of a framework for handling variables and substitution. More far-reaching are the following two extensions:

1) Study the definable higher-order functions as opposed to the first-order fragment,

2) Introduce higher-type recursors, i.e., allow recursive definition of function valued functions.

The first question has been investigated by Cook and Kapron [9] where a machine-characterisation of the definable functionals in PV^ω is given. See also [1] for similar investigations concerning safe recursion.

In the rest of this paper we will survey recent works towards the second goal.

A first attempt would consist of introducing a recursor

$$\mathcal{R}^A : A{\to}(\Box(\mathsf{N}){\to}A{\to}A){\to}\Box(\mathsf{N}){\to}A \tag{10}$$

for any type A, not only $A = \mathsf{N}$. Alas, already with $A = \mathsf{N}{\to}\mathsf{N}$ we can define exponentially growing functions:

$$\begin{aligned} f(0) &= \lambda y{:}\mathsf{N}.\mathsf{S}_0(y) \\ f(x) &= \lambda y{:}\mathsf{N}.f(\lfloor x/2\rfloor, f(\lfloor x/2\rfloor, y)) \end{aligned} \tag{11}$$

or,

$$f = \mathcal{R}^{\mathsf{N}\to\mathsf{N}}(\mathsf{S}_0, \lambda n{:}\Box(\mathsf{N}).\lambda z{:}\mathsf{N}{\to}\mathsf{N}.\lambda u{:}\mathsf{N}.z(z(u))) : \Box(\mathsf{N}){\to}\mathsf{N}{\to}\mathsf{N} \tag{12}$$

We have $f(x,y) = 2^{|x|}\cdot y$, leading outside polynomial time.

One way to prevent this from happening is to require that in defining $f(x)$ the result of the recursive call $f(\lfloor x/2\rfloor)$ should be used at most once. This would mean that we move to an *affine linear lambda calculus* in which a variable may be used at most once, unless it has a type $\Box(A)$ or is of type N, i.e., values of ground type may be freely duplicated. In a recursive definition the step function h is also "used" several times. In order to prevent this from introducing unwanted

duplications, we therefore have to require that the step function be duplicable, i.e., insert another $\Box$. We thus arrive at the following improved typing for the recursor

$$\mathcal{R}^A : A \multimap (\Box(\mathsf{N}) \multimap A \multimap A) \multimap \Box(\mathsf{N}) \multimap A$$

where we use $\multimap$ instead of $\to$ to emphasise linear usage of variables. For example, we have

$$\lambda u{:}\mathsf{N} \multimap \mathsf{N}.\lambda x{:}\mathsf{N}.ux : (\mathsf{N} \multimap \mathsf{N}) \multimap (\mathsf{N} \multimap \mathsf{N})$$

but $\lambda u{:}\mathsf{N} \multimap \mathsf{N}.\lambda x{:}\mathsf{N}.u(ux)$ is not typeable. However, we have

$$\lambda u{:}\Box(\mathsf{N} \multimap \mathsf{N}).\lambda x{:}\mathsf{N}.u(u(x)) : \Box(\mathsf{N} \multimap \mathsf{N}) \multimap \mathsf{N} \multimap \mathsf{N}$$

since variables of $\Box$-type may be duplicated.

It also turns out that if the result type A is allowed to contain $\Box$ then the exponential function becomes definable so we impose the further restriction that A must be built up from N and $\multimap$.

The resulting system has been studied in [18] and independently (with minor variations) in [2]. It is shown in these papers that all definable first-order functions are polynomial time computable. The proof is fairly technical; the underlying idea, however, is simply that linear lambda terms can be normalised in linear time without size increase. The complications stem from the presence of the duplicable types N and $\Box(-)$ and require the use of semantical techniques [18] or a sophisticated normalisation argument [2].

We are not aware of any attempt of extending bounded recursion on notation to higher types; naive attempts at doing so fail; see, however, the remark on *bounded linear logic* below in Section 8. In the realm of finite model theory we have Goerdt's result [14] stating that ordinary primitive recursion with first-order result type ($A = \mathsf{N}^n \to \mathsf{N}$) captures polynomial space.

We also mention [21] where it is shown that tiered recursion with first-order result type without linearity restriction characterises polynomial space. The idea is that the tiering prevents the composition of the recursively defined function with itself as in Eqn. 11, yet still allows for multiple evaluation on different (independent) arguments as is needed in typical polynomial space complete problems, e.g. evaluation of quantified boolean formulas.

6 Inductive datatypes

Another extension of safe recursion is with inductive datatypes. We discuss here a type of unlabelled binary trees. Other datatypes are analogous; we briefly mention lists below in Section 8 below.

We augment the higher-order version of safe recursion with a type T of trees and constructors $\mathsf{leaf} : \mathsf{T}$ and $\mathsf{node} : \mathsf{T} \multimap \mathsf{T} \multimap \mathsf{T}$. In addition to that we introduce a recursor

$$\mathcal{R}^A_{\mathsf{T}} : A \multimap \Box(\Box(T) \multimap \Box(T) \multimap A \multimap A \multimap A) \multimap T \multimap A \tag{13}$$

where $f = \mathcal{R}^A_{\mathsf{T}}(g, h)$ means $f(\mathsf{leaf}) = g$ and $f(\mathsf{node}(l, r)) = h(l, r, f(l), f(r))$. We also need a construct for case distinction and destruction, see [18] for details.

We notice that we cannot simply encode trees as natural numbers because the (definable) pairing function will have at least one normal argument (or be of type $\Box\mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$), but constructor functions should be safe in all their arguments.

The following example shows that trees should not be duplicable: if they were, e.g., a variable of type T could be used more than once and then we could define the following function $f : \Box(\mathsf{N}) \multimap \mathsf{T}$

$$\begin{aligned} f(0) &= \mathsf{leaf} \\ f(x) &= \mathsf{node}(f(\lfloor x/2 \rfloor), f(\lfloor x/2 \rfloor)) \end{aligned} \tag{14}$$

where $f(x)$ is the full binary tree of depth $|x|$. Without duplication but with higher-type recursion, i.e., $\mathcal{R}_{\mathsf{T}}^{\mathsf{N}\multimap\mathsf{N}}$ we get

$$\begin{aligned} g(\mathsf{leaf}) &= \lambda y{:}\mathsf{N}.\mathsf{S}_0(y) \\ g(\mathsf{node}(l,r)) &= \lambda y{:}\mathsf{N}.g(l,g(r,y)) \end{aligned} \tag{15}$$

and we have $g(f(x),y) = 2^{|x|}\cdot y$ which typechecks by necessitation.

This shows that duplication of trees together with higher-type recursion leads outside polynomial time. In [18] we show that linear trees can be added to higher-type safe recursion without affecting the definable first-order functions. On the other hand, Jean-Yves Marion has reported[2] that Bellantoni-Cook's original first-order system can be extended with binary trees without linearity restrictions. In the proof of polynomial time computability these trees are represented as DAGs, so that in particular the "full binary tree" returned by the function f in Eqn. 14 which is legal in Marion's proposed extension only requires linear space.

A system similar in spirit to higher-type safe recursion with inductive datatypes is Girard's "light linear logic" (LLL), [12]. In LLL inductive datatypes are definable via variants of the familiar Church-style encodings [4]. The derived recursion schemes look similar to the ones discussed above; for example, to define in LLL a function from a type of trees T to some other type A we need to provide an element $g : A$ and a closed function $h : A\multimap A\multimap A$. From these we obtain in LLL by iteration a function $f : \mathsf{T}\multimap\S(A)$ where just like the $\Box$ in safe recursion the $\S$ (albeit on the other side of the arrow) prevents a thus obtained function from being used as a step function in a subsequent recursive definition. Despite this similarity it does not seem possible to directly translate LLL into higher-type safe recursion or vice versa. A detailed comparison, going beyond the obvious fact that the first-order functions representable in either system agree, must be left to future research.

7 Limitations of safe recursion

Safe recursion (as well as tiered recursion and LLL) has the disadvantage that many naturally occurring polynomial time algorithms do not fit into the rather rigid discipline it imposes. Consider, for example, the familiar insertion sort algorithm on lists (where [] denotes "nil", the empty list, and $a :: l$ is the list with head a and tail l ("cons")).

$$\begin{aligned} &\mathsf{insert}(a,[]) = [a] \\ &\mathsf{insert}(a,b::l) = \mathsf{if}\ a \le b \\ &\qquad \mathsf{then}\ a :: b :: l \\ &\qquad \mathsf{else}\ b :: \mathsf{insert}(a,l) \end{aligned} \tag{16}$$

$$\begin{aligned} &\mathsf{sort}([]) = [] \\ &\mathsf{sort}(a::l) = \mathsf{insert}(a,\mathsf{sort}(l)) \end{aligned}$$

We may assume that a, b are of ground type so that the double usage of a and b does not violate linearity. Nevertheless, insertion sort as such is not "safe recursive". The reason is that the insertion function, being recursively defined, would receive the typing

$$\mathsf{insert} : \mathsf{N}\multimap\Box(\mathsf{L}(\mathsf{N}))\multimap\mathsf{L}(\mathsf{N}) \tag{17}$$

Here $\mathsf{L}(\mathsf{N})$ is the type of lists over N with constructors [] : $\mathsf{L}(\mathsf{N})$ and ::: $\mathsf{N}\multimap\mathsf{L}(\mathsf{N})\multimap\mathsf{L}(\mathsf{N})$ and an iteration principle analogous to the one for trees above (13). Similarly, in LLL we would have $\mathsf{insert} : \mathsf{N}\multimap\mathsf{L}(\mathsf{N})\multimap\S(\mathsf{L}(\mathsf{N}))$.

[2] Personal communication about work in progress.

The definition of sort is now not possible because the purported step function, namely insert, has a modal type. Indeed, the pattern of recursion here is quite similar to the example of the functions q and e above in Eqns. 2 & 3: a recursively defined function is iterated a second time. The key difference is of course that unlike q which doubles the size, the function insert merely increases it by one. Accordingly, its iteration in the definition of sort does not produce an exponential-sized list, but a list of the same length as the argument.

The problem is that under the strict regime of safe recursion there is no way to detect this situation. It is based on the worst case assumption that every recursively defined function might polynomially increase the size of its argument.

Caseiro [6] has noticed this and developed partly semantic criteria on first-order recursive programs which allow one to detect this situation and which in particular apply to the insertion sort example. Unfortunately her criteria are not obviously decidable and do not generalise to higher-order functions.

In [17] the author has presented a type-theoretic alternative and generalisation of Caseiro's idea. Before presenting this system let us remark that by the completeness result a sorting *function* is of course representable with safe recursion and even better something fairly close to insertion sort is representable; namely we can with safe recursion define a function

$$\mathsf{insert}' : \Box(\mathsf{N}) \multimap \mathsf{N} \multimap \mathsf{L}(\mathsf{N}) \multimap \mathsf{L}(\mathsf{N}) \tag{18}$$

obeying the specification that $\mathsf{insert}'(n, x, l) = \mathsf{insert}(x, l)$ provided the length of l is bounded by $|n|$.

Now we can iterate $\mathsf{insert}'(n, x, l)$ keeping n as a parameter. It is then clear that the correct sorting function is obtained by instantiating n with the length of the list to be sorted. This can be compared to the simulation of general recursion by primitive recursion along a previously computed measure. Again, this makes the correctness of the program dependent upon a correct analysis of its growth rate.

8 Impredicative recursion

Let us now take a look at the system in [17]. It is based on the following observation. While iterating a function which doubles its argument size leads to exponential growth, the iteration of a function which does not increase the size at all leads to a function with the same property.

The system accordingly enforces the invariant that all definable functions do not increase the size of their input and therefore may contain iteration principles without any modality restriction. Two questions remain to be solved:

1) What does "non-size-increasing" mean for higher-order functions?

2) How do we deal with obviously size increasing functions such as successor or "cons"?

The answer to 1) is that we define the size of a function to be the amount by which it increases the size of its input. For example, we can define the function $@ : \mathsf{L}(\mathsf{N}) \multimap \mathsf{L}(\mathsf{N}) \multimap \mathsf{L}(\mathsf{N})$ which appends two lists. It takes a list l_1 of some length x_1 as argument and returns a function of size x_1, namely the function which takes a list l_2 and returns $l_1 @ l_2$—a list of length $x_1 + x_2$. With these definitions it can be seen that non-size increasing functions form a model of affine linear lambda calculus (a *symmetric monoidal closed category*). The answer to the second question is the introduction of a special "resource type" $\diamond$ which has one element of size 1. Accordingly, the (binary) successor

functions, "cons", and "node" are non-size increasing as functions of type

$$\begin{array}{l}\mathsf{S}_0, \mathsf{S}_1 : \Diamond \multimap \mathsf{N} \multimap \mathsf{N} \\ \mathsf{cons} : \Diamond \multimap \mathsf{N} \multimap \mathsf{L(N)} \multimap \mathsf{L(N)} \\ \mathsf{node} : \Diamond \multimap \mathsf{T} \multimap \mathsf{T} \multimap \mathsf{T}\end{array} \tag{19}$$

The values of type $\Diamond$ necessary to apply these constructor functions are made available in step functions of recursive definitions. Namely, we have the following typing for list iteration ("fold-left" in the terminology of functional programming)

$$\mathcal{I}^{A}_{\mathsf{L(N)}} : A \multimap (\Diamond \multimap \mathsf{N} \multimap A \multimap A) \multimap \mathsf{L(N)} \multimap A \tag{20}$$

and similar ones for integers and trees. Here is a definition of tail recursive list reversal using $\mathcal{I}^{\mathsf{L(N)}}_{\mathsf{L(N)}}$.

$$\begin{array}{l}\mathsf{rev\_aux} = \mathcal{I}^{\mathsf{L(N)}}_{\mathsf{L(N)}}(\lambda t{:}\mathsf{L(N)}.t, \lambda d{:}\Diamond.\lambda a{:}\mathsf{N}.\lambda z{:}\mathsf{L(N)} \multimap \mathsf{L(N)}.\lambda u{:}\mathsf{L(N)}.z(\mathsf{cons}(d, a, u))) \\ \mathsf{rev} = \lambda l{:}\mathsf{L(N)}.\mathsf{rev\_aux}(l, []) : \mathsf{L(N)} \multimap \mathsf{L(N)}\end{array} \tag{21}$$

Similarly, insert can be defined[3] as of type $\mathsf{insert} : \Diamond \multimap \mathsf{N} \multimap \mathsf{L(N)} \multimap \mathsf{L(N)}$ so that we obtain insertion sort by $\mathsf{sort} = \mathsf{It}^{\mathsf{L(N)}}_{\mathsf{L(N)}}([], \mathsf{insert})$.

The resulting system is surprisingly expressive and while, of course, not every polynomial time function can be representable (as these in general increase the size) one can define all functions which are computable in polynomial time and simultaneously in linear space; [17] also contains a characterisation result for a fragment of the system. It is also shown that a natural generalisation of the system captures polynomial space. The correctness proof makes use of a realisability interpretation of the system by untyped resource bounded algorithms.

A new correctness proof for the system based on normalisation has recently been announced by Schwichtenberg and Aehlig.

The systems of safe and tiered recursion have been dubbed *predicative* in [1] because a recursive definition (like quantification in predicative set theory) raises the level. A system which allows arbitrary nesting of recursive definitions may therefore be called *impredicative*. Apart from [17] and Cobham's system, another impredicative system is *bounded linear logic* (BLL), [13].

This system allows for the definition of all polynomial time computable functions and, like LLL, can define datatypes via their Church encodings. Like in Cobham's system explicit size bounds need to be maintained; the difference is that enforcement of these bounds is intrinsically guaranteed by the type system and does neither rely on external reasoning nor an *ad hoc* solution such as cutting off at size overflow (Eqn. 9). Unfortunately, BLL has received very little attention since its publication; a further elaboration might prove worthwhile.

9 Conclusion

We have given a guided tour through the host of systems proposed in order to capture complexity classes via syntactic restrictions. Although all serve a similar purpose, they differ largely in naturality of use and in their expressivity w.r.t. particular algorithms and programming language features.

[3] We do not give the definition here because one has to overcome the slight complication that the entries a, b appear twice, i.e., nonlinearly, in the body of $\mathsf{insert}(a, b :: l)$ in Eqn. 16. Even ground type variables may not be duplicated in the system, see [17] for details.

The reader may ask what this is all good for. As already mentioned, the original motivation behind Cobham's result and the Cook-Bellantoni work was to provide a resource-free, intrinsic, characterisation of complexity classes thus providing further evidence for their naturality.

A later motivation was to use these systems to interpret via realisability logical systems which have the property that proofs give rise to functions belonging to a certain complexity class. We have already mentioned Buss' bounded arithmetic and the proofs of its polynomial time property by functional interpretation. Similar studies by various authors are currently underway for higher type safe recursion. In [12] Girard announces a naive set theory based on LLL which avoids Russell's paradox in a new way.

From the field of programming language theory there has recently been an interest in formally and semi-automatically certifying properties such as memory usage and time consumption [24, 11, 20]. It appears that the systems surveyed in this article will be of use towards this goal. For example, in [19] a variant of the impredicative system [17] has been used to guarantee static bounds on the heap usage of functional programs involving inductive datatypes. The type system for embedded functional programming proposed by Hughes and Pareto [20] bears some superficial similarity with BLL. Most importantly, the proofs establishing polynomial runtime bounds for all programs can be turned into a method for *deriving and certifying* such bounds, possibly within the formalism proposed in [11]. It may be hoped that this article encourages further interaction between the hitherto rather distinct fields of logical complexity theory and programming languages.

References

[1] S. Bellantoni. *Predicative recursion and computational complexity.* PhD thesis, University of Toronto, 192. Technical Report 264/92.

[2] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Ramification, Modality, and Linearity in Higher Type Recursion. *Annals of Pure and Applied Logic*, 2000. to appear.

[3] Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[4] Corrado Boehm and Alessandro Berarducci. Automatic Synthesis of typed λ-programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] Samuel R. Buss. *Bounded Arithmetic.* Bibliopolis, 1986.

[6] Vuokko-Helena Caseiro. *Equations for Defining Poly-time Functions.* PhD thesis, University of Oslo, 1997. Available by ftp from `ftp.ifi.uio.no/pub/vuokko/0adm.ps`.

[7] Peter Clote. Computation models and function algebras. available electronically under `http://thelonius.tcs.informatik.uni-muenchen.de/~clote/Survey.ps.gz`, 1996.

[8] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Logic, Methodology, and Philosophy of Science II*, pages 24–30. Springer Verlag, 1965.

[9] S. Cook and B. Kapron. Characterisations of the basic feasible functionals at all finite types. In S. Buss and P. Scott, editors, *Feasible Mathematics*, pages 154–159. Birkhäuser, 1990.

[10] S. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63:103–200, 1993.

[11] K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Symp. Principles of Prog. Lang. (POPL)*. ACM, 2000. to appear.

[12] J.-Y. Girard. Light Linear Logic. *Information and Computation*, 143, 1998.

[13] J.-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic. *Theoretical Computer Science*, 97(1):1–66, 1992.

[14] Andreas Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100:45–66, 1992.

[15] Yuri Gurevich. Algebras of feasible functions. In *Proc. Symp. Found. of Comp. Sci. (FOCS)*, pages 210–214, 1983.

[16] Martin Hofmann. An application of category-theoretic semantics to the characterisation of complexity classes using higher-order function algebras. *Bulletin of Symbolic Logic*, 3(4):469–485, 1997.

[17] Martin Hofmann. Linear types and non size-increasing polynomial time computation. Submitted for publication. See `www.dcs.ed.ac.uk/home/papers/icc.ps.gz` for a draft. An extended abstract has appeared under the same title in Proc. Symp. Logic in Comp. Sci. (LICS) 1999, Trento, 2000.

[18] Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 2000. to appear.

[19] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP)*. Springer, 2000. to appear.

[20] J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ML programming. In *Proc. International Conference on Functional Programming (ACM). Paris, September '99.*, pages 70–81, 1999.

[21] D. Leivant and J.-Y. Marion. Predicative Functional Recurrence and Poly-Space. In *Springer LNCS 1214: Proc. CAAP*, 1997.

[22] Daniel Leivant. Stratified Functional Programs and Computational Complexity. In *Proc. 20th IEEE Symp. on Principles of Programming Languages*, 1993.

[23] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterisations of polytime. *Fundamentae Informaticae*, 19:167–184, 1993.

[24] George Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Prog. Lang. (POPL)*. ACM, 1997.

[25] Frank Pfenning and Rowan Davies. A modal analysis of staged computation. In *Proc. POPL '96, Florida*. ACM, 1996.

[26] Frank Pfenning and Hao-Chi Wong. On a modal lambda calculus for S4. In *Proceedings of the 11th Conference on Mathematical Foundations of Programming Semantics (MFPS), New Orleans, Louisiana*. Electronic Notes in Theoretical Computer Science, Volume 1, Elsevier, 1995.