A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion

Martin Hofmann

TU Darmstadt, FB 4, Schloßgartenstr. 7, 64289 Darmstadt, Germany
mh@mathematik.tu-darmstadt.de

Abstract. This paper introduces a simply-typed lambda calculus with both modal and linear function types. Through the use of subtyping extra term formers associated with modality and linearity are avoided. We study the basic metatheory of this system including existence and inference of principal types.
The system serves as a platform for certain higher-order generalisations of Bellantoni-Cook's function algebra capturing polynomial time using a separation of the variables into "safe" and "normal" ones.
The distinction between and the syntactic restrictions involved with the safe and normal variables in the Bellantoni-Cook framework are captured by the modal function space and the associated typing rules.
The linear function spaces on the other hand are used to enable a certain form of primitive recursion with *functional* result type which is conservative over polynomial time.
The proofs associated with these applications are based on an interpretation of the lambda calculus in a category-theoretic model in which all functions are polynomial time computable by construction. The details of this interpretation are not the main subject of this paper and will appear elsewhere.

1 Introduction and summary

We introduce a simply-typed lambda calculus SLR with linear function spaces and an S4 modality $\square$. It combines the $\to, \multimap$ fragment of intuitionistic linear logic with Pfenning's modal lambda calculus [16, 9].

The system serves as a platform for certain systems of restricted primitive recursion (due originally to Bellantoni-Cook [3]) which ensure that all definable first-order functions belong to a certain complexity class, e.g., polynomial time on a Turing machine. The S4 modality provides a type-theoretic formulation of Bellantoni-Cook's first-order notion of safe and normal variables: The type $(\square \mathsf{N})^m \to \mathsf{N}^n \to \mathsf{N}$ corresponds to functions in m normal variables and n safe variables. Semantically, a term of this type denotes a *PTIME*-function whose size is polynomial in the size of its normal arguments and constant in the size of its safe arguments. The modality allows us to formulate Bellantoni's operation of safe recursion as a single constant of second-order type.

Linearity on the other hand, provides a restricted version of safe recursion with *functional* result type which (unlike the unrestricted version) does not lead beyond polynomial time.

In this paper we will only give outlines of the proofs involved with these applications; the technical details will appear elsewhere [11]. The main aim of this paper is to describe the type system and its syntax which as we feel are rather interesting in their own right and perhaps can be applied in other situations where modality and/or linearity are required. Examples might be the use of modality to delineate run time from compile time in [16] and the use of linear types to optimise compilation described in Wadler's papers [20, 19].

1.1 The type system

The most important novel feature of the system is the presence of a subtyping which includes in particular the judgements $A \to B <: \Box A \to B$ and $A \multimap B <: A \to B$. This subtyping allows us to do without any explicit term forming operations referring to linearity and modality. In particular, there is only one kind of abstraction $\lambda x{:}A.e$, which according to the context can be given any of the four types $A \multimap B$, $\Box A \multimap B$, $A \to B$, $\Box A \to B$. Moreover, since the modality $\Box$ is only allowed on the left of an arrow there is no need for explicit introduction and elimination constructs for $\Box$ such as the `box` and `let box` constructs used in the modal calculi developed by Pfenning's group [9, 16].

A similar situation arises with the linear types. As is well known, we can define the intuitionistic function space $A \to B$ as $!A \multimap B$ where ! is a certain modal operator. If one gives ! the status of a first class type former then introduction and elimination rules are needed; if we restrict its occurrences to left sides of function spaces then we can avoid them.

Our type system is such that every closed term has a smallest type which can be automatically inferred. As an illustration we give here a few terms with their principal types indicated (A, B are arbitrary types)

$$\begin{array}{l}
\lambda x{:}A.x : A \multimap A \\
\lambda f{:}A \to B.\lambda x{:}A.f\,x : (A \to B) \multimap A \to B \\
\lambda f{:}\Box A \to B.\lambda x{:}A.f\,x : (\Box A \to B) \multimap \Box A \to B \\
\lambda f{:}\Box A \to B.\lambda g{:}A \to A.\lambda x{:}A.f(g\,x) : \\
\qquad (\Box A \to B) \multimap \Box(A \to A) \to \Box A \to B
\end{array}$$

The intuitive meaning of a linear function is that it uses its argument only once. The intuitive meaning of a function of type $\Box A \to B$ is that it may recur on its argument or on terms containing the argument, whereas a function of type $A \to B$ is not allowed to do so. Notice that in the fourth typing the argument g is "modalized" because it appears as a subterm of an argument to f.

We consider both a strictly linear and an affine linear system. In the latter system $\lambda x{:}A.\lambda y{:}A.x$ has principal type $A \multimap A \multimap A$, whereas in the strictly linear system it has principal type $A \multimap A \to A$. Thus, in the strictly linear system every linear variable must be used exactly once whereas in the affine system it must not be used more than once.

There is a base type N of natural numbers which has a property that $\mathsf{N} \to A$ and $\mathsf{N} \multimap A$ are equal (and printed as $\mathsf{N} \to A$). It comes with two constants $\mathsf{S}_0, \mathsf{S}_1 : \mathsf{N} \to \mathsf{N}$ with intended meaning $\mathsf{S}_0(x) = 2x$ and $\mathsf{S}_1(x) = 2x+1$ and a case construct which introduces a further source for linearity. If $e_1 : \mathsf{N}$ and $e_2 : A$ and $e_3, e_4 : \mathsf{N} \to A$ then we have $\mathsf{case}_A\, e_1$ zero e_2 even e_3 odd $e_4 : A$ with the intended meaning that if e_1 is zero then the result is e_2, if e_1 is $2n$ then the result is $e_3\, n$ and if e_1 is $2n+1$ then the result is $e_4\, n$. The important point is that if a variable appears linearly in each branch of a case construct then it appears linearly in the whole case expression although it makes up to three literal occurrences. We illustrate this by giving more terms with their principal types.

$$
\begin{aligned}
&\mathit{divtwo} = \lambda x{:}\, \mathsf{N}.\, \mathsf{case}_{\mathsf{N}}\; x\\
&\quad \mathsf{zero}\ 0\\
&\quad \mathsf{even}\ \lambda y{:}\, \mathsf{N}.y\\
&\quad \mathsf{odd}\ \lambda y{:}\, \mathsf{N}.y : \mathsf{N} \to \mathsf{N}
\end{aligned}
$$

$$
\begin{aligned}
&\lambda x{:}\, \mathsf{N}.\lambda f{:}\, \mathsf{N} \to \mathsf{N}.\, \mathsf{case}_{\mathsf{N}}\; x\\
&\quad \mathsf{zero}\ f(x)\\
&\quad \mathsf{even}\ f\\
&\quad \mathsf{odd}\ f : (\mathsf{N} \to \mathsf{N}) \multimap \mathsf{N} \to \mathsf{N}
\end{aligned}
$$

$$
\begin{aligned}
&\lambda f{:}\, \mathsf{N} \to \mathsf{N}.\lambda g{:}\, \mathsf{N} \to \mathsf{N}.\lambda x{:}\, \mathsf{N}.f(g\; x)\\
&\quad : (\mathsf{N} \to \mathsf{N}) \multimap (\mathsf{N} \to \mathsf{N}) \multimap \mathsf{N} \to \mathsf{N}
\end{aligned}
$$

In the third of these examples g occurs linearly because by the above convention on N the function f has in fact the type $\mathsf{N} \multimap \mathsf{N}$. With N replaced by A this function would have principal type $(A \to A) \multimap (A \to A) \to A \to A$.

1.2 Safe recursion

The $\Box$-modality is put to use in the type of the "safe recursor"

$$\mathsf{saferec}_A : \Box\mathsf{N} \to A \to (\Box\mathsf{N} \to A \to A) \to A$$

with the following intended meaning

$$
\begin{aligned}
&\mathsf{saferec}_A\; 0\; g\; h = g\\
&\mathsf{saferec}_A\; n\; g\; h = h\, n(\mathsf{saferec}_A\; \lfloor n/2 \rfloor\; g\; h), \text{when } n > 0.
\end{aligned}
$$

Thus $\mathsf{saferec}_A$ allows us to define a function $f : \Box\mathsf{N} \to A$ from a constant $g : A$ and function $h : \Box\mathsf{N} \to A \to A$ describing the passage from n and $f(\lfloor n/2 \rfloor)$ to $f(n)$. We will mostly be interested in the fragment where $A = \mathsf{N}$.

The important point about safe recursion is that the function h is required to be of type $\Box\mathsf{N} \to A \to A$ rather than $\Box\mathsf{N} \to \Box A \to A$ so that h cannot recur on its (recursive) second argument (if $A = \mathsf{N}$.)

For example, the following function computes a value in the order of x^2.

$$
\begin{aligned}
&sq = \lambda x{:}\, \mathsf{N}.\\
&\quad \mathsf{saferec}_{\mathsf{N}}\; x\; 1\; (\lambda y{:}\, \mathsf{N}.\lambda q{:}\, \mathsf{N}.\mathsf{S}_0(\mathsf{S}_0 q)) : \Box\mathsf{N} \to \mathsf{N}
\end{aligned}
$$

The above equations then specialise to

$$
\begin{aligned}
sq\ 0 &= 1 \\
sq\ x &= \mathsf{S}_0(\mathsf{S}_0(sq\lfloor x/2 \rfloor))
\end{aligned}
$$

hence $sq(2^i) = 2^{2i}$. Now our type system allows us to iterate sq:

$$\lambda x{:}\,\mathsf{N}.sq(sq\ x) : \Box\mathsf{N} \to \mathsf{N}$$

Intuitively, this is so because sq can be "lifted" to the (non-existing) type $\Box\mathsf{N} \to \Box\mathsf{N}$ using the S4 rules. We will give a precise explanation below in Section 2.3.

However, the following term which would compute an exponentially growing function is ill-typed

$$\lambda x{:}\,\mathsf{N}.\mathsf{saferec}_{\mathsf{N}}\ x\ 1\ (\lambda y{:}\,\mathsf{N}.\lambda x{:}\,\mathsf{N}.sq\ x)$$

because $\lambda y{:}\,\mathsf{N}.\lambda x{:}\,\mathsf{N}.sq\,x$ has type $\mathsf{N} \to \Box\mathsf{N} \to \mathsf{N}$ which is not a subtype of the required type $\Box\mathsf{N} \to \Box\mathsf{N} \to \mathsf{N}$.

By generalizing Bellantoni's proof (which was for a first-order system) one can show that all functions definable using $\mathsf{saferec}_{\mathsf{N}}$ are computable in polynomial time. In a nutshell, the idea is to prove by induction on terms that if $t : (\Box\mathsf{N})^m \to \mathsf{N}^n \to \mathsf{N}$ then t is an $(m+n$-ary) *PTIME*-function and moreover the length[1] $|t(x,y)|$ of $t(\boldsymbol{x},\boldsymbol{y})$ is bounded by $p(|\boldsymbol{x}|)+\max(|\boldsymbol{y}|)$ for some m-variate polynomial p. In order to get the induction through in the presence of higher typed functions we need to prove a somewhat stronger statement, see Section 4 below.

On the other hand, Bellantoni's first-order system embeds into the present one so all *PTIME*-functions can be defined in our calculus. More precisely, for each Bellantoni-Cook definable function $f(\boldsymbol{x};\boldsymbol{y})$ in m normal variables and n safe variables we can construct a closed term $\hat{f} : (\Box\mathsf{N})^m \to \mathsf{N}^n \to \mathsf{N}$ whose denotation agrees with the one of f. This construction is by direct induction on the definition of f in Bellantoni-Cook's system.

We remark that the linear types are *not* required for this higher-order extension of Bellantoni-Cook's system and for the definition of all *PTIME*-functions. Their role is to provide a certain form of safe recursion with *functional* result type thus allowing for more direct and shorter definitions.

1.3 Linear recursion

Unfortunately, the explicit definition even of simple functions in Bellantoni's original system can become rather complicated.

For example, if we want to define an addition function on natural numbers using digit-wise addition with carry then we run into the problem that in the recursive call we want to change the carry bit, in other words we want to perform a substitution of parameters. In a system with higher types primitive recursion

[1] Here and in the sequel $|x| \stackrel{\text{def}}{=} \lceil \log_2(x+1) \rceil$ denotes the length of the binary expansion of x. If $\boldsymbol{x} = (x_1, \ldots, x_n)$ then $|\boldsymbol{x}| = (|x_1|, \ldots, |x_n|)$.

with substitution of parameters can be elegantly expressed using higher result type, for instance using $\mathsf{saferec}_{\mathsf{N}\to\mathsf{N}}$. However, primitive recursion with higher result type leads beyond *PTIME*. The reason is that in $\mathsf{saferec}_{\mathsf{N}\to\mathsf{N}} x\ g\ h$ the *function* $\mathsf{saferec}_{\mathsf{N}\to\mathsf{N}} \lfloor x/2 \rfloor\ g\ h$ may be called more than once and even nested. For example,

$$f := \lambda x{:}\,\mathsf{N}.\mathsf{saferec}_{\mathsf{N}\to\mathsf{N}}\ x\ \mathsf{S}_0\ (\lambda y{:}\,\mathsf{N}.\lambda f{:}\,\mathsf{N}\to\mathsf{N}.\lambda x{:}\,\mathsf{N}{:}\,f.(f\ x)) : \Box\mathsf{N}\to\mathsf{N}\to\mathsf{N}$$

satisfies $f(0,y) = 2y$ and $f(x,y) = f(\lfloor x/2\rfloor, f(\lfloor x/2\rfloor, y))$ and thus $f(x,y) = 2^{2^{|x|}}y$. In $\mathsf{saferec}_{\Box\mathsf{N}\to\mathsf{N}}$ such nesting is ruled out by the typing, but we can still define PSPACE complete functions by calling $f(\lfloor x/2\rfloor, -)$ more than once in the computation of $f(x,y)$. For example, given an appropriate encoding of quantified boolean formulas we can define a function f such that $f(x,y) = 1$ if $y = \lceil\varphi\rceil$ is the code of a true quantified boolean formula φ and $x > y$. The definition of $f(x, \lceil\varphi\rceil)$ proceeds by case distinction on the outermost constructor of φ. If e.g. $\varphi = \forall x.\psi(x)$ then we call $f(\lfloor x/2\rfloor, \lceil\psi(tt)\rceil)$ and $f(\lfloor x/2\rfloor, \lceil\psi(\mathit{ff})\rceil)$.

The task of linearity is to rule out the blow up in complexity caused by higher result type. If we allow $f(\lfloor x/2\rfloor, -)$ to be called at most once in the computation of $f(x,y)$ then we remain in *PTIME*. To do this formally, we introduce constants

$$\mathsf{linrec}_A : \Box\mathsf{N}\to A\multimap(\Box\mathsf{N}\to A\multimap A)\to A$$

with the same intended meaning as $\mathsf{saferec}_A$. Their type ensures that exactly one (at most one in the affine system) recursive call is made at each unfolding step.

We will demonstrate below that linrec_A for A of the form $\mathsf{N}\to\ldots\to\mathsf{N}\to\mathsf{N}$, does not lead beyond *PTIME*. For example, using $\mathsf{linrec}_{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ we can define an addition function

$$\mathit{add} : \Box\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathsf{N}$$

where

$$\mathit{add}\ l\ x\ y\ c = \hat{x} + \hat{y} + (c \bmod 2).$$

where $\hat{x} = x \bmod 2^{|l|-1}$. In other words, $\hat{x}$ consists of the $|l|-1$ least significant bits of x.

$$\begin{array}{l}
\mathit{add} = \lambda l{:}\,\mathsf{N}.\\
\quad \mathsf{linrec}_{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathsf{N}}\ l\\
\quad\quad (\lambda x{:}\,\mathsf{N}.\lambda y{:}\,\mathsf{N}.\lambda c{:}\,\mathsf{N}.c \bmod 2)\\
\quad\quad (\lambda u{:}\,\mathsf{N}.\lambda a{:}\,\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathsf{N}.\lambda x{:}\,\mathsf{N}.\lambda y{:}\,\mathsf{N}.\lambda c{:}\,\mathsf{N}.\\
\quad\quad\quad \mathsf{let}\ \mathit{carry} = (x\wedge(y\vee c))\vee((\neg x)\wedge(y\wedge c))\ \mathsf{in}\\
\quad\quad\quad\quad \mathsf{case}_{\mathsf{N}}\ x\oplus y\oplus c\\
\quad\quad\quad\quad\quad \mathsf{zero}\ \mathsf{S}_0(a\ \lfloor x/2\rfloor\ \lfloor y/2\rfloor\ \mathit{carry})\\
\quad\quad\quad\quad\quad \mathsf{even}\ \lambda u{:}\,\mathsf{N}.(\mathsf{S}_0(a\ \lfloor x/2\rfloor\ \lfloor y/2\rfloor\ \mathit{carry})\\
\quad\quad\quad\quad\quad \mathsf{odd}\ \lambda u{:}\,\mathsf{N}.(\mathsf{S}_0(a\ \lfloor x/2\rfloor\ \lfloor y/2\rfloor\ \mathit{carry}))
\end{array}$$

Here $\mathsf{let}\ldots\mathsf{in}$ is syntactic sugar for a β-expansion and the auxiliary functions $\vee, \wedge, \ldots$ are the indicated boolean functions of type $N \to N$ and $N \to N \to N$ defined using case_N. They only look at the last bit of a number. Finally, $\lfloor -/2 \rfloor$ and $- \bmod 2$ are quotient and remainder with respect to division by two also defined using case_N.

Remark 1. In an earlier draft version of this paper it was stated that all functions definable using $\mathsf{saferec}_{\Box\mathsf{N}\to\mathsf{N}}$ were in PSPACE. Unfortunately, this is not the case, as we can apply $f(\lfloor x/2 \rfloor, -)$ to values of size polynomial in the size of y when computing $f(x, y)$ because y is normal in this case. For example, we can make the following tail recursive definition of exponentiation:

$$f := \lambda x\colon \mathsf{N}.\mathsf{saferec}_{\Box\mathsf{N}\to\mathsf{N}}\ x\ (\lambda y\colon \mathsf{N}.y)(\lambda w\colon \mathsf{N}.\lambda u\colon \Box\mathsf{N} \to \mathsf{N}.sq)$$

i.e., $f(0, y) = y$ and $f(x, y) = f(\lfloor x/2 \rfloor, sq(y))$. In order to rule this out, we might introduce a constant

$$\mathsf{saferec}'_{\Box\mathsf{N}\to\mathsf{N}} : \Box\mathsf{N} \to (\Box\mathsf{N} \to \mathsf{N}) \to (\Box\mathsf{N} \to (\Box\mathsf{N} \to \mathsf{N}) \to (\mathsf{N} \to \mathsf{N})) \to (\Box\mathsf{N} \to \mathsf{N})$$

and in addition an application functional

$$app : (\Box\mathsf{N} \to \mathsf{N}) \to \Box\mathsf{N} \to \mathsf{N} \to \mathsf{N}$$

with intended meaning $app(f, x, y) = f(\min(x, y))$. This application functional is needed so that we can define a PSPACE-complete function. We conjecture that in this way we do indeed capture PSPACE computation, but concede that this solution is rather ad hoc.

1.4 Related work

The modal part of our system builds upon the modal lambda calculi by Pfenning *et. al.* [9]. As already mentioned the inference of modalities and the use of subtyping is new.

The idea of restricting modalities to the argument position of function types arises in Pfenning and Cervesato's *Linear Logical Framework* [7] and also in Plotkin's formulation of linear system F [17]. Again, none of these systems allows for automatic inference of modality.

Wadler *et. al.* [20, 19] describe type systems for linearity inference which employ universal quantification over so-called *use variables* instead of subtyping. These systems assign possibly linear types to *untyped* lambda terms and thus solve a more general problem than the (linear part of) the present system. The price is a very high complexity of the type system which makes a detailed comparison difficult.

Leivant and Marion also have studied lambda calculus characterisations of polynomial time [13] and other complexity classes [14]. Their approach is based on a countable number of copies of the base type ("tiers"). Since application and abstraction are to respect these tiers ordinary simply-typed lambda calculus suffices. It might, however, be interesting to design a type system which tries to assign tiers to terms of simply-typed lambda calculus with a single base type.

Another difference to Leivant-Marion's work is that they do not use linearity to restrict the complexity of primitve recursion with functional result type. Indeed, in [15] they show that a certain form of tiered recursion with functional result type captures the class *PSPACE*.

After the presentation of this work at the CSL '97 conference Bellantoni, Niggl, and Schwichtenberg [2] have independently come up with a very similar system in which linear recursion for arbirtary result types built up from N and $\multimap$ (not just the first-order ones) is sound w.r.t. *PTIME*. We believe that this result also holds for the present system (see [11] for some discussion in this direction), but a proof must await further research. The work by Bellantoni et al. does not treat the issues of subtyping and type checking studied in this paper.

1.5 Overview

The next section gives the formal definition of SLR. It contains the definitions of types and subtyping, of bindings and contexts, and finally the typing rules. Section 3 contains our main result. We show that SLR admits reconstruction of principal types, i.e. that the omission of linearity and modality annotations in abstractions and applications does not represent a loss of information. Unlike in other systems with subtyping like Cardelli's $F_{<:}$ [5] we first need to prove a stronger result: *existence of principal solutions to typechecking*, of which existence of principal types is a simple corollary.

In Section 4 we give some applications of SLR to Computational Complexity. We give two characterisations of *PTIME* as the functions definable in certain subsystems of SLR. The proofs of these results are only briefly sketched in this paper; the full version will appear elsewhere [11]. The technique to obtain such results is an interpretation of SLR in various models based on presheaves and Chu-spaces [18, 12]. These models can serve as a justification of the system in their own right.

2 Syntax

In this section we define types, contexts, and expressions and give the rules for typing and subtyping.

2.1 Types and subtyping

The type expressions of the calculus SLR are given by the following grammar.

$$A, B ::= \mathsf{N} \mid A \to B \mid A \multimap B \mid \Box A \to B \mid \Box A \multimap B$$

An *aspect* is a pair (l, m) where $l \in \{\multimap, \to\}$ and $m \in \{\not\Box, \Box\}$. We order the aspects componentwise where $\to \leq \multimap$ and $\Box \leq \not\Box$. An aspect $(\multimap, m)$ is called linear; an aspect $(\rightarrow, m)$ is called nonlinear; an aspect $(l, \not\Box)$ is called nonmodal; an aspect $(l, \Box)$ is called modal.

Now we use the following generic notations for the four function spaces where a is an aspect.

$$
\begin{array}{l}
A \xrightarrow{a} B \text{ is } \quad A \multimap B \text{ when } a = (\multimap, \not\Box) \\
A \xrightarrow{a} B \text{ is } \Box A \multimap B \text{ when } a = (\multimap, \Box) \\
A \xrightarrow{a} B \text{ is } \quad A \rightarrow B \text{ when } a = (\rightarrow, \not\Box) \\
A \xrightarrow{a} B \text{ is } \Box A \rightarrow B \text{ when } a = (\rightarrow, \Box)
\end{array}
$$

The subtyping relation $<:$ between types is defined by the rules in Figure 1. The subtyping rule for function types contains (by reflexivity) the special cases

$$
\begin{array}{ll}
\text{S-REFL} & \dfrac{}{A <: A} \\[2ex]
\text{S-AX1} & \dfrac{}{\mathsf{N} \rightarrow A <: \mathsf{N} \multimap A} \\[2ex]
\text{S-AX2} & \dfrac{}{\Box N \rightarrow A <: \Box \mathsf{N} \multimap A} \\[2ex]
\text{S-TRANS} & \dfrac{A <: B \quad B <: C}{A <: C} \\[2ex]
\text{S-ARR} & \dfrac{B_1 <: A_1 \quad A_2 <: B_2 \quad a' \leq a}{A_1 \xrightarrow{a} A_2 <: B_1 \xrightarrow{a'} B_2}
\end{array}
$$

Fig. 1. Subtyping rules

$A \multimap B <: A \rightarrow B <: \Box A \rightarrow B$ and $A \multimap B <: \Box A \multimap B <: \Box A \rightarrow B$. Note that $\Box A \multimap B$ and $A \rightarrow B$ are incomparable, but have $A \multimap B$ and $\Box A \rightarrow B$ as greatest lower bound and least upper bound, respectively.

The subtyping axioms S-AX1,2 mean that linearity is a higher-order concept; first-order functions are always linear by definition.

Proposition 1. *Subtyping is decidable by a syntax-directed procedure.*

Proof. Combining the subtyping axioms with appropriate instances of S-ARR yields a transitivity-free presentation of subtyping which immediately gives rise to a syntax-directed procedure by backward application of these rules.

2.2 Typing

The expressions of our language are given as follows.

$$
\begin{array}{lll}
e ::= & x & \text{(variable)} \\
& \mid (e_1\, e_2) & \text{(application)} \\
& \mid \lambda x{:}\, A.e & \text{(abstraction)} \\
& \mid c & \text{(constants)} \\
& \mid \mathsf{case}_A\ e_1\ \mathsf{zero}\ e_2\ \mathsf{even}\ e_3\ \mathsf{odd}\ e_4 & \text{(case distinction)}
\end{array}
$$

Here x ranges over a countable set of variables and c ranges over the set $\mathbb{N} \cup \{\mathsf{S}_0, \mathsf{S}_1, \mathsf{saferec}_A, \mathsf{linrec}_A\}$ where A is a type. The expressions are identified up to renaming of bound variables. The type $\tau(c)$ of a constant c is defined by

$$
\begin{array}{l}
\tau(c) = \mathsf{N}, \text{when } c \text{ is an integer constant} \\
\tau(\mathsf{S}_0) = \tau(\mathsf{S}_1) = \mathsf{N} \to \mathsf{N} \\
\tau(\mathsf{saferec}_A) = \Box N \to A \to (\Box N \to A \to A) \to A \\
\tau(\mathsf{linrec}_A) = \Box N \to A \multimap (\Box N \to A \multimap A) \to A
\end{array}
$$

In order to assign types to the other expressions we need to introduce contexts (also known as type assignments).

If x is a variable and A is a type then a *binding* with variable x and type A is one of the following.

$$
\begin{array}{lll}
b ::= & x{:}\,A & \text{(ordinary binding)} \\
 & \mid \Box x{:}\,A & \text{(modal binding)} \\
 & \mid x \hat{:}\, A & \text{(linear binding)} \\
 & \mid \Box x \hat{:}\, A & \text{(modal linear binding)}
\end{array}
$$

We also use the notation $x \overset{a}{:} A$ where $x \overset{a}{:} A$ means

$$
\begin{array}{ll}
x{:}\,A & \text{when } a = (\to, \not\Box) \\
\Box x{:}\,A & \text{when } a = (\to, \Box) \\
x \hat{:}\, A & \text{when } a = (\multimap, \not\Box) \\
\Box x \hat{:}\, A & \text{when } a = (\multimap, \Box)
\end{array}
$$

A *context* is a set of bindings with pairwise distinct variables. If Γ is a context we write $\mathrm{dom}(\Gamma)$ for the set of variables bound in Γ. If $x \overset{a}{:} A \in \Gamma$ then we write $\Gamma(x)$ for A and $\Gamma((x))$ for the aspect a.

The bindings $x{:}\,A$ and $\Box x{:}\,A$ are called *nonlinear*; the bindings $\Box x{:}\,A$ and $\Box x \hat{:}\, A$ are called *modal.* A context Γ is nonlinear (modal) if all its bindings are nonlinear (modal).

Two contexts Γ, Δ are disjoint if the sets $\mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Delta)$ are disjoint. If Γ and Δ are disjoint we write Γ, Δ for the union of Γ and Δ.

For example, if $\Gamma = \{x{:}\,\mathsf{N}, y \hat{:}\, \mathsf{N} \to \mathsf{N}, \Box z{:}\,\mathsf{N}, \Box w \hat{:}\, \mathsf{N} \to \mathsf{N}\}$ then $\mathrm{dom}(\Gamma) = \{x, y, z, w\}$. Furthermore, $\Gamma(x) = \mathsf{N}$, $\Gamma(w) = \mathsf{N} \to \mathsf{N}$, $\Gamma((y)) = (\multimap, \not\Box)$, etc.

The typing relation $\Gamma \vdash e : A$ between contexts, expressions, and types is defined inductively by the rules below. We suppose that all contexts, types, and terms occurring in such a rule are well-formed; in particular, if Γ, Δ or similar appears as a premise or conclusion of a rule then Γ and Δ must be disjoint for the rule to be applicable.

$$\text{T-VAR}\quad \frac{x \in \mathrm{dom}(\Gamma) \qquad y \in \mathrm{dom}(\Gamma), y \neq x \text{ implies } \Gamma((y)) \text{ nonlinear}}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{T-SUB}\quad \frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}$$

$$\text{T-ARR-I}\quad \frac{\Gamma, x \overset{a}{:} A \vdash e : B}{\Gamma \vdash \lambda x{:}A.e : A \xrightarrow{a} B}$$

$$\text{T-ARR-E}\quad \frac{\Gamma, \Delta_1 \vdash e_1 : A \xrightarrow{a} B \quad \Gamma, \Delta_2 \vdash e_2 : A \qquad \Gamma \text{ nonlinear} \qquad x \overset{a'}{:} X \in \Gamma, \Delta_2 \text{ implies } a' \leq a}{\Gamma, \Delta_1, \Delta_2 \vdash (e_1\, e_2) : B}$$

$$\text{T-CASE}\quad \frac{\Gamma, \Delta_1 \vdash e_1 : \mathsf{N} \qquad \Gamma, \Delta_2 \vdash e_2 : A \qquad \Gamma, \Delta_2 \vdash e_3 : \mathsf{N} \to A \qquad \Gamma, \Delta_2 \vdash e_4 : \mathsf{N} \to A \qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{case}_A\ e_1\ \mathsf{zero}\ e_2\ \mathsf{even}\ e_3\ \mathsf{odd}\ e_4 : A}$$

$$\text{T-CONST}\quad \frac{\Gamma \text{ nonlinear}}{\Gamma \vdash c : \tau(c)}$$

Remark 2. Alternatively, we can define an affine linear system in which weakening is allowed, and only duplication of variables is forbidden. Then the variable rule would be replaced by

$$\text{T-VAR-AFF}\quad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

Furthermore, the nonlinearity premise to (T-Con) would be dropped.

Remark 3. The application rule T-ARR-E specialises to the following four cases.

$$\frac{\Gamma, \Delta \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A \qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta \vdash (e_1\, e_2) : B} \qquad \frac{\Gamma_1, \Gamma_2, \Delta_1 \vdash e_1 : \Box A \multimap B \quad \Gamma_1, \Delta_2 \vdash e_2 : A \qquad \Gamma_{1,2} \text{ nonlinear} \quad \Gamma_1, \Delta_2 \text{ modal}}{\Gamma_1, \Gamma_2, \Delta_1, \Delta_2 \vdash (e_1\, e_2) : B}$$

$$\frac{\Gamma_1, \Gamma_2, \Delta \vdash e_1 : \Box A \to B \quad \Gamma_1 \vdash e_2 : A \qquad \Gamma_{1,2} \text{ nonlinear} \quad \Gamma_1 \text{ modal}}{\Gamma_1, \Gamma_2, \Delta \vdash (e_1\, e_2) : B} \qquad \frac{\Gamma, \Delta_1 \vdash e_1 : A \multimap B \quad \Gamma, \Delta_2 \vdash e_2 : A \qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash (e_1\, e_2) : B}$$

2.3 Examples

Let us see how the type system assigns the correct types to the examples from Section 1.2. We have $y{:}\,\mathsf{N}, q{:}\,\mathsf{N} \vdash \mathsf{S}_0 q : \mathsf{N}$ by rules T-Var, T-Const, and T-Arr-E with $a = (\not\square, \rightarrow)$, $\Gamma = \Delta_1 = \emptyset$, $\Delta_2 = y{:}\,\mathsf{N}, q{:}\,\mathsf{N}$. Similarly, we get $y{:}\,\mathsf{N}, q{:}\,\mathsf{N} \vdash \mathsf{S}_0(\mathsf{S}_0\, q) : \mathsf{N}$. Therefore, $t \stackrel{\text{def}}{=} \lambda y{:}\,\mathsf{N}.\lambda q{:}\,\mathsf{N}.\mathsf{S}_0(\mathsf{S}_0\, q) : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$, and $t : \square\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ by T-Sub. Now $x{:}\square\mathsf{N} \vdash \mathsf{saferec}_{\mathsf{N}}\, x\, 1\, t : \mathsf{N}$ by T-Const and three instances of T-Arr-E; the first with $a = (\square, \rightarrow)$. Finally, T-Arr-I gives $sq \stackrel{\text{def}}{=} \lambda x{:}\,\mathsf{N}.\mathsf{saferec}\, x\, 1\, t : \square\mathsf{N} \to \mathsf{N}$.

Now $x{:}\square\mathsf{N} \vdash sq\, x : \mathsf{N}$ and thus $x{:}\square\mathsf{N} \vdash sq\,(sq\, x)\mathsf{N}$ by rule T-Arr-E with $a = (\square, \rightarrow)$.

3 Syntactic metatheory

The expression $e_1[e_2/x]$ denotes the capture-free substitution of e_2 for x in e_1. The proofs of the following two propositions are straightforward inductions on derivations.

Proposition 2. *The rules* Weak *and* Subst *below are admissible. The rule* Weak-Aff *below is admissible in the affine system.*

$$\textsc{Weak}\quad \frac{\Gamma \vdash e : A \qquad \Delta \textit{ nonlinear}}{\Gamma, \Delta \vdash e : A}$$

$$\textsc{Weak-Aff}\quad \frac{\Gamma \vdash e : A}{\Gamma, \Delta \vdash e : A}$$

$$\textsc{Subst}\quad \frac{\begin{array}{c}\Gamma, \Delta_1, x \stackrel{a}{:} A \vdash e_1 : B \qquad \Gamma, \Delta_2 \vdash e_2 : B \\ \Gamma \textit{ nonlinear} \\ y \stackrel{a'}{:} Y \in \Gamma, \Delta_2 \textit{ implies } a' \leq a\end{array}}{\Gamma, \Delta_1, \Delta_2 \vdash e_1[e_2/x] : B}$$

Proposition 3. *1. If $\Gamma \vdash x : A$ then $x \in \mathrm{dom}(\Gamma)$ and $\Gamma(x) <: A$ and either $\Gamma(x)$ is nonlinear or $\Gamma = \Gamma', x \stackrel{a}{:} X$ where Γ' is nonlinear and a is linear. In the affine system we only know $x \in \mathrm{dom}(\Gamma)$ and $\Gamma(x) <: A$.*

2. If $\Gamma \vdash \lambda x{:}A.e : C$ then $\Gamma, x \stackrel{a}{:} A \vdash e : B$ for some a and B such that $A \xrightarrow{a} B <: C$.

3. If $\Gamma \vdash (e_1\, e_2) : B$ then we can find a decomposition $\Gamma = \Gamma', \Delta_1, \Delta_2$, a type A and an aspect a such that $\Gamma', \Delta_1 \vdash e_1 : A \xrightarrow{a} B$ and $\Gamma', \Delta_2 \vdash e_2 : A$ and Γ' is nonlinear and whenever $x \stackrel{a'}{:} X$ in Γ', Δ_1 then $a' < a$.

4. If $\Gamma \vdash \mathsf{case}_A\, e_1\, \mathsf{zero}\, e_2\, \mathsf{even}\, e_3\, \mathsf{odd}\, e_4 : B$ then $A <: B$ and we can find a decomposition $\Gamma = \Gamma', \Delta_1, \Delta_2$ where Γ' is nonlinear and $\Gamma', \Delta_1 \vdash e_1 : \mathsf{N}$ and $\Gamma, \Delta_2 \vdash e_2 : A$ and $\Gamma, \Delta_2 \vdash e_{3,4} : \mathsf{N} \to A$.

5. *If $\Gamma \vdash c : A$ then $\tau(c) <: A$ and Γ is nonlinear. In the affine system Γ can be arbitrary.*

In the following development we assume that we are *not* in the affine system. We discuss below what changes (simplifications) need to be made in order to account for the latter.

Let Γ, Δ be contexts. We say that Δ is a subcontext of Γ if $\Delta(x) = \Gamma(x)$ for each $x \in \mathrm{dom}(\Delta)$. Notice that we do not require $\Gamma((x)) = \Delta((x))$.

Definition 1. *Let Δ, Δ' be contexts. We write $\Delta \ll \Delta'$ to mean that $\Delta((x)) \leq \Delta'((x))$ for each $x \in \mathrm{dom}(\Delta')$ and that $\Delta((x))$ is nonlinear if x not in $\mathrm{dom}(\Delta')$.*

Proposition 4. *If $\Delta \ll \Delta'$ and $\Delta' \vdash e : A$ then $\Delta \vdash e : A$.*

Definition 2. *Let Γ be a context and e be an expression. Say that* typechecking e under Γ fails *if $\Delta \nvdash e : A$ for all subcontexts Δ of Γ and types A. Say that (Δ, A) is a* principal solution *for the typechecking problem (Γ, e) if*

- *Δ is a subcontext of Γ*
- *$\Delta \vdash e : A$*
- *whenever $\Delta' \vdash e : A'$ and Δ' is a subcontext of Γ then $\Delta' \ll \Delta$.*

In other words, typechecking (Γ, e) fails if we cannot assign a type to e even if we allow to omit variables from Γ and to assign arbitrary aspects to the leftover bindings. A principal solution (Δ, A) is obtained by omitting from Γ as many variables as possible and to relax the required bindings as little as possible so as to obtain a type for e. One might think that the thus obtained type isn't necessarily the best possible or in other words that one could perhaps obtain a smaller type by further relaxing the context. Fortunately, our type system is such that this cannot happen. The reason is that relaxing the context (by omitting variables or relaxing aspects) can only help to typecheck a term at all, but not to decrease the type of an already typable term.

Notice that the aspects in Γ do not affect the solvability of a typechecking problem (Γ, e). In fact, we only use Γ to ascribe types to the variables.

We are now ready to formulate our main result.

Theorem 1. *Let Γ be a context and e an expression. Either typechecking e under Γ fails or there exists a principal solution for the typechecking problem (Γ, e). Moreover, there exists a syntax-directed procedure which decides whether a principal solution exists and computes it in the affirmative case.*

Proof. By induction on the structure of e.

Case $e = x$. If $x \notin \mathrm{dom}(\Gamma)$ then typechecking x under Γ obviously fails. Otherwise, we claim that $(\{x \hat{:} \Gamma(x)\}, \Gamma(x))$ is a principal solution. To see this, assume $\Delta \vdash x : A$ for some subcontext Δ of Γ. Then by generation of typing we must have $\Delta(x) <: A$ and either Δ is nonlinear or $\Delta = \Delta', x \overset{a}{:} X$ where Δ' is nonlinear and a is linear. In each case $\Delta' \ll x \hat{:} \Delta(x)$. The claim follows as $\Delta(x) = \Gamma(x)$ by assumption on Δ.

Case $e = \lambda x{:}A.e'$. We may assume w.l.o.g. that x is not in $\mathrm{dom}(\Gamma)$. If $\Delta \vdash e : C$ for some subcontext Δ of Γ then by generation of typing we have $\Delta, x\overset{a}{:}A \vdash e' : B$ for some B,a and $A \xrightarrow{a} B <: C$.Therefore, typechecking (Γ, e) fails if typechecking $(\Gamma, x{:}A\ ,\ e')$ fails. Assume that this is not the case and let (Δ, B) be a principal solution of $(\Gamma, x{:}A, e')$. We have two cases to distinguish.

1. $x \notin \mathrm{dom}(\Delta)$. Then $(\Delta, A \to B)$ is a principal solution.
2. $\Delta = \Delta_1, x\overset{a}{:}A$ where $x \notin \mathrm{dom}(\Delta_1)$. Then $(\Delta_1, A \xrightarrow{a} B)$ is a principal solution.

In case 1 we have $\Delta, x{:}A \vdash e' : B$ by Weak and thus $\Delta \vdash e : A \to B$. In case 2 $\Delta_1 \vdash e : A \xrightarrow{a} B$ is immediate from T-Arr-I. For principality assume that $\Delta' \vdash e : C$ for some subcontext Δ' of Γ. By the above analysis we get $\Delta', x\overset{a'}{:}A \vdash e' : B'$ for some a',B' with $A \xrightarrow{a'} B' <: U$. The induction hypothesis gives us $\Delta', x\overset{a'}{:}A \ll \Delta$ and $B <: B'$. If x is not in $\mathrm{dom}(\Delta)$ then we also have $\Delta' \ll \Delta$ and by definition of $\ll$ we know that a' is nonlinear. Thus, $A \to B <: A \xrightarrow{a'} B' <: C$. If $\Delta = \Delta_1, x\overset{a}{:}A$ then we get $\Delta_1 \ll \Delta$ and $a' \leq a$, hence $A \xrightarrow{a} B <: A \xrightarrow{a'} B' <: C$.

Case $e = (e_1\, e_2)$. By generation of typing we know that if either (Γ, e_1) or (Γ, e_2) fails then the problem $(\Gamma, e_1\, e_2)$ fails. So assume that (Δ_1, A_1) and (Δ_2, A_2) are principal solutions to these latter problems. Generation of typing gives us that $A_1 = B \xrightarrow{a} C$ for some B, C, a_0. If A_2 is not a subtype of B then typechecking $e_1\, e_2$ obviously fails. Otherwise, we define an improved aspect $a \geq a_0$ by $a = (\multimap, m)$ if $B = \mathsf{N}$, $a_0 = (l, m)$ and $a = a_0$ otherwise. Notice that $\Delta_1 \vdash e_1 : B \xrightarrow{a} C$ by S-Ax1,2. We claim that the principal solution is (Δ, C) where Δ is the largest (w.r.t. $\ll$) context satisfying the following requirements.

1. $\mathrm{dom}(\Delta) = \mathrm{dom}(\Delta_1) \cup \mathrm{dom}(\Delta_2)$
2. $x \in \mathrm{dom}(\Delta_1)$ implies $\Delta((x)) \leq \Delta_1((x))$
3. $x \in \mathrm{dom}(\Delta_2)$ implies $\Delta((x)) \leq \Delta_2((x))$
4. $x \in \mathrm{dom}(\Delta_2)$ implies $\Delta((x)) \leq a$
5. $x \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$ implies that $\Delta((x))$ is nonlinear

To see that $\Delta \vdash e : C$ we decompose Δ as $\Theta, \Lambda_1, \Lambda_2$ where Λ_1 is Δ restricted to $\mathrm{dom}(\Delta_1) \setminus \mathrm{dom}(\Delta_2)$ and Λ_2 is Δ restricted to $\mathrm{dom}(\Delta_2) \setminus \mathrm{dom}(\Delta_1)$ and Θ is the rest, i.e., Δ restricted to $\mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$. From (5) we know that Θ is nonlinear. Furthermore, if $x\overset{a'}{:}X$ in Θ, Λ_2 then $a' \leq a$ by (4) so rule T-Arr-E yields $\Delta \vdash e : C$.

For principality assume that $\Delta' \vdash e : C'$. Generation of typing yields a decomposition $\Delta' = \Theta, \Lambda_1, \Lambda_2$ where Θ is nonlinear and $\Theta, \Lambda_1 \vdash e_1 : B' \xrightarrow{a'} C'$ and $\Theta, \Lambda_2 \vdash e_2 : B'$ and $y\overset{a''}{:}Y \in \Theta, \Lambda_2$ implies $a'' \leq a'$. The induction hypothesis gives $\Theta, \Lambda_1 \ll \Delta_1$ and $\Theta, \Lambda_2 \ll \Delta_2$ and $B \xrightarrow{a} C <: B' \xrightarrow{a'} C'$ and $B <: B'$. By definition of subtyping we get $a' \leq a$ and $C <: C'$.

The definition of context relaxation yields in particular that if Θ, Λ_1 and Θ, Λ_2 contain variables not mentioned in $\Delta_{1,2}$ hence Δ then those are bound nonlinearly. So to show $\Delta' \ll \Delta$ it is enough to show that Δ' restricted to $\mathrm{dom}(\Delta)$ meets requirements (2–5) above.

From the induction hypothesis we get $\Delta'((x)) \leq \Delta_1((x))$ if $x \in \mathrm{dom}(\Delta_1)$ and $\Delta'((x)) \leq \Delta_2((x))$ if $x \in \mathrm{dom}(\Delta_2)$. Furthermore, if $x \in \mathrm{dom}(\Delta_2)$ then $\Delta'((x)) = \Theta, \Lambda_2((x)) \leq a' \leq a$. Finally, $x \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$ implies $x : \mathrm{dom}(\Theta)$ as Λ_1, Λ_2 are disjoint. Hence $\Delta'((x)) = \Theta((x))$ is nonlinear. Thus $\Delta' \ll \Delta$ by definition of Δ.

Case $e = \mathsf{case}_A\, e_1\, \mathsf{zero}\, e_2\, \mathsf{even}\, e_3\, \mathsf{odd}\, e_4$. Let (Δ_i, A_i) be the principal solutions of (Γ, e_i) for $i = 1 \ldots 4$. If those do not exist then we fail. We also can assume that $A_1 = \mathsf{N}$, $A_2 <: A$, $A_{3,4} <: \mathsf{N} \to A$ for otherwise typechecking obviously fails.

We claim that a principal solution is (Δ, A) where Δ is the greatest (with respect to $\ll$) context such that the following hold

1. $\mathrm{dom}(\Delta) = \mathrm{dom}(\Delta_1) \cup \mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3) \cup \mathrm{dom}(\Delta_4)$.
2. If $x \in \mathrm{dom}(\Delta_i)$ then $\Delta((x)) \leq \Delta_i((x))$.
3. If $x \in \mathrm{dom}(\Delta_1) \cap (\mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3) \cup \mathrm{dom}(\Delta_4))$ then $\Delta((x))$ is nonlinear.
4. If $x \in (\mathrm{dom}(\Delta_1) \cup \mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3)) \setminus (\mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) \cap \mathrm{dom}(\Delta_3))$ then $\Delta((x))$ is nonlinear.

We can decompose Δ as $\Theta, \Lambda_1, \Lambda_2$ where Λ_1 is Δ restricted to $\mathrm{dom}(\Delta_1) \setminus (\mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3) \cup \mathrm{dom}(\Delta_4))$ and Λ_2 is Δ restricted to $(\mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3) \cup \mathrm{dom}(\Delta_4)) \setminus \mathrm{dom}(\Delta_1)$ and Θ is the rest, i.e., Δ restricted to $\mathrm{dom}(\Delta_1) \cap (\mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3) \cup \mathrm{dom}(\Delta_4))$; therefore Θ is nonlinear by (3). Now $\mathrm{dom}(\Theta, \Lambda_1) = \mathrm{dom}(\Delta_1)$ so $\Theta, \Lambda_1 \ll \Delta_1$ by (2). Similarly, $\mathrm{dom}(\Theta, \Lambda_2)$ is $\mathrm{dom}(\Delta_1) \cup \mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3)$ and we have $\Theta, \Lambda_2 \ll \Delta_{2,3,4}$ from (2) and 4. So $\Delta \vdash e : A$ follows with T-Case.

For principality suppose that $\Delta' \vdash e : A'$. By generation of typing we know that $A <: A'$ and $\Delta' = \Theta, \Lambda_1, \Lambda_2$ where Θ is nonlinear and $\Theta, \Lambda_1 \vdash e_1 : \mathsf{N}$ and $\Theta, \Lambda_2 \vdash e_2 : A$ and $\Theta, \Lambda_2 \vdash e_{3,4} : \mathsf{N} \to A$. The induction hypothesis gives us $\Theta, \Lambda_1 \ll \Delta_1$ and $\Theta, \Lambda_2 \ll \Delta_{2,3,4}$. As in the case of application we know that the variables in Δ' which are not bound in Δ are bound nonlinearly in Δ' so to show $\Delta' \ll \Delta$ it is enough to show that Δ' restricted to $\mathrm{dom}(\Delta)$ meets the requirements (2,3,4). Part (2) is immediate from the induction hypothesis. If $x \in \mathrm{dom}(\Delta')$ and either $x \in \mathrm{dom}(\Delta_1) \cap (\mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3) \cup \mathrm{dom}(\Delta_4))$ or $x \in \mathrm{dom}(\Delta_2) \cup \mathrm{dom}(\Delta_3) \cup \mathrm{dom}(\Delta_4) \setminus (\mathrm{dom}(\Delta_2) \cap \mathrm{dom}(\Delta_3) \cap \mathrm{dom}(\Delta_4))$ then x must have been declared in Θ, thus is nonlinear in Δ'.

Case $e = c$. The principal solution is $(\emptyset, \tau(c))$. Clearly, $\emptyset \vdash c : \tau(c)$. If $\Gamma \vdash c : A$ then generation of typing yields $\tau(c) <: A$ and Γ nonlinear. Thus $\Gamma \ll \emptyset$.

Corollary 1. *Given Γ, e it is decidable whether there exists a type A such that $\Gamma \vdash e : A$.*

Proof. Let (Δ, A) be the principal solution of the problem (Γ, e). If none exists then clearly no such A can exist. Otherwise check whether $\Gamma \ll \Delta$. If yes then by context subsumption we also have $\Gamma \vdash e : A$; otherwise $\Gamma \vdash e : B$ is impossible by principality.

Corollary 2. *Let Γ be a context and e be a term. Either e is not typable in Γ or there exists a typing $\Gamma \vdash e : A$ such that whenever $\Gamma \vdash e : A'$ then $A <: A'$.*

Proof. Let (Δ, A) be the principal solution of (Γ, e). If $\Gamma \ll \Delta$ then A has the required property, otherwise e is not typable.

Remark 4. In order to account for the affine system in which arbitrary weakening is allowed we need to make the following changes to the proof of Theorem 1

- The definition of $\Gamma \ll \Delta$ is changed to $\Gamma((x)) \leq \Delta((x))$ for all $x \in \mathrm{dom}(\Delta)$.
- In subcase (1) of case $e = \lambda x{:}A.e'$ we get $A \multimap B$ instead of $A \to B$ as principal type.
- Clause (4) in case $e = \mathsf{case}_A\, e_1$ zero e_2 even e_3 odd e_4 can be dropped
- Clauses (1–3) in case $e = (e_1\, e_2)$ are in the affine case equivalent to $\Delta \ll \Delta_1$ and $\Delta \ll \Delta_2$. Clauses (1,2) in case $e = \mathsf{case}_{e1}$ zero e_2 even e_3 odd e_4 are in the affine case equivalent to $\Delta \ll \Delta_i$.

Remark 5. The type checking algorithm proceeds by direct structural induction on terms and does not maintain large intermediate results. It therefore is of low-degree polynomial complexity. We hope that future extensions of the system presented in this paper allow us to infer such estimates automatically from the *typing* of the checking algorithm in a functional programming language with modal and linear types.

4 Denotational semantics and application to complexity theory

The results in this section are only briefly sketched and serve mainly as a motivation for the system SLR. The details will appear elsewhere [11].

We can associate a set $[\![A]\!]$ to each type A by interpreting N as the set $\mathbb{N}$ of natural numbers and $A \xrightarrow{a} B$ as the full function space $[\![B]\!]^{[\![A]\!]} = [\![A]\!] \to [\![B]\!]$ thus ignoring aspects. Note that all the subtypings become identities.

To every term $\Gamma \vdash e : A$ we can then associate a function $[\![\Gamma \vdash e]\!] : (\prod_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]) \to [\![A]\!]$ by interpreting variables as projections, function abstraction and application as set-theoretic function abstraction and application, the case construct by set-theoretic case distinction and the constants $\mathsf{saferec}_A$ and linrec_A as explained in the introduction.

We can also define an operational semantics which effectively computes the set-theoretic meaning of closed terms of type N.

Such operational semantics can be formalised either using an inductive definition of a big-step evaluation relation or by compiling terms into ordinary functional programs which do not use the modalities. This is what we have done in our prototype implementation. We will not go into details here. The important point is that any reasonable operational semantics or even equational theory will be faithful w.r.t. the above set-theoretic semantics so that our soundness theorem below (Theorem 2) encompasses all these operational semantics.

Of course, this does not automatically imply that an operational semantics allows for *PTIME* execution of SLR-programs. However, the constructive nature of proof of Thm. 2 allows us to extract a (not necessarily *PTIME*) algorithm which *compiles* any SLR-term e of type $\Box\mathsf{N} \to \mathsf{N}$ into a *PTIME*-algorithm computing the function $[\![e]\!]$.

4.1 Discussion of subject reduction

We remark that this semantic definition of what it means for a function to be definable in SLR relieves us from having to stick to a particular notion of reduction. Indeed, the issue of reduction semantics for SLR is not unproblematic because the subtyping rule $\mathsf{N} \to A <: \mathsf{N} \multimap A$ destroys subject reduction for β-reduction:

If $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ then the function $h \equiv \lambda x{:}\mathsf{N}{:}(.f\,x\,x)$ gets type $\mathsf{N} \to \mathsf{N}$ hence $\mathsf{N} \multimap \mathsf{N}$. Thus $\lambda g{:}\mathsf{N} \to \mathsf{N}{:}\,h.(g\,0)$ gets type $(\mathsf{N} \to \mathsf{N}) \multimap \mathsf{N}$. However, our type system gives the weaker type $(\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$ to the β-reduced term $\lambda g{:}\,\mathsf{N} \to \mathsf{N}{:}\,f.\,(g\,0)(g\,0)$.

It is possible that a more refined type system could give the stronger type $(\mathsf{N} \to \mathsf{N}) \multimap \mathsf{N}$ here. Notice, however that such stronger type system would have to give type $(\mathsf{N} \to \mathsf{N}) \multimap \mathsf{N}$ to $\lambda g{:}\,\mathsf{N} \to \mathsf{N}{:}\,f.\,(g\,e_1)\,(g\,e_2)$ if there exists a common ancestor of e_1 and e_2, in particular, if e_1 reduces to e_2.

Thus the type system would need to examine the run time behaviour of terms which is usually not what one wants to do in order to enable fast type checking. We thus prefer to sacrifice subject reduction and to impose on the programmer the task of inserting β-expansions (or the syntactically more appealing `let`-expansions) if the stronger typing in examples like the above is desired. Of course, this means that we cannot rely on subject reduction to establish soundness properties like Theorem 2 below. But as said above all we need to know from an operational semantics is that it all its reductions are equalities in the set-theoretic interpretation.

However, it is easy to see that if we remove the special treatment of linearity at ground type then SLR does enjoy the subject reduction property w.r.t. β-reduction. If $\Theta \vdash (\lambda x{:}\,A.e_1)e_2 : C$ then there exists $B <: C$ and a decomposition $\Theta = \Gamma, \Delta_1, \Delta_2$ with Γ nonlinear and $\Gamma, \Delta_1 \vdash \lambda x{:}\,A.e_1 : A \xrightarrow{a} B$ and $\Gamma, \Delta_2 \vdash e_2 : A$ and $(\Gamma, \Delta_2)((y)) \leq a$ for all $y \in \mathrm{dom}(\Gamma, \Delta_2)$. Now, without the axioms S-Ax1, S-Ax2 we can conclude $\Gamma, \Delta_1, x \overset{a'}{:} A' \vdash e_1 : B$ for some $a \leq a'$ and $A <: A'$. The result then follows from the derived rule Subst.

4.2 Characterisations of polynomial time

We will now consider subsystems of SLR by restricting $\mathsf{saferec}_A$ and linrec_A to certain instances A. We will refer to such a restriction by the recursion operators available in that system. Let us say that a system *defines* a function $f : \mathbb{N}^n \to \mathbb{N}$ if there exists a closed term $g : \Box\mathsf{N} \to \Box\mathsf{N} \to \ldots \to \Box\mathsf{N} \to \mathsf{N}$ such that f arises as the set-theoretic meaning of g. We say that a system *captures* a complexity class (for example *PTIME*) if the functions definable in the system agree with the functions belonging to this complexity class.

We have the following result.

Theorem 2. *1.* $\mathsf{saferec}_{\mathsf{N}}$ *captures PTIME.*
2. $\mathsf{saferec}_{\mathsf{N}} + \mathsf{linrec}_{\mathsf{N}\to\mathsf{N}\to\ldots\to\mathsf{N}}$ *captures PTIME.*

Outline of proof. Part (1) is a higher-order extension of Bellantoni-Cook's result [3]. In order to prove it we interpret SLR in the category of presheaves over the category $\mathcal{C}$ of *polymax-bounded PTIME-functions* which is defined as follows: Objects of $\mathcal{C}$ are pairs (m, n) where $m, n \in \mathbb{N}$. A morphism from (m, n) to $(1, 0)$ is an m-ary *PTIME*-function. A morphism from (m, n) to $(0, 1)$ is a $m + n$-ary *PTIME*-function $f(\boldsymbol{x}; \boldsymbol{y})$ for which there exists a polynomial p such that $f(\boldsymbol{x}; \boldsymbol{y}) \leq p(|\boldsymbol{x}|) + \max|\boldsymbol{y}|$. A morphism from (m, n) to (m', n') consists of m' morphisms from (m, n) to $(1, 0)$ and n' morphisms from (m, n) to $(0, 1)$. Composition in $\mathcal{C}$ is set-theoretic composition. The presheaf category $\widehat{\mathcal{C}} = \mathcal{S}et^{\mathcal{C}^{op}}$ furnishes a model of SLR with the following settings.

$$
\begin{aligned}
&[\![\mathsf{N}]\!](m, n) = \mathcal{C}((m, n)\,,\,(0, 1))\\
&[\![A \to B]\!] = [\![A \multimap B]\!] = [\![A]\!] \Rightarrow [\![B]\!]\\
&[\![\Box A \to B]\!] = [\![\Box A \multimap B]\!] = \Box[\![A]\!] \Rightarrow [\![B]\!]
\end{aligned}
$$

Here $\Rightarrow$ denotes the function space of presheaves and for presheaf $F \in \widehat{\mathcal{C}}$ the presheaf $\Box F$ is defined by $(\Box F)(m, n) = F(m, 0)$. One can show that the natural transformations from $\Box[\![\mathsf{N}]\!]^m \times [\![\mathsf{N}]\!]^n$ to $[\![N]\!]$ are in 1-1 correspondence with the set $\mathcal{C}((m, n)\,,\,(0, 1))$ from which the result follows using an appropriately defined logical relation, see [11, 10] for a detailed description.

For Part (2) we only consider the case $\mathsf{linrec}_{\mathsf{N}\to\mathsf{N}}$. Suppose that $g : \mathsf{N}$ and $u : \Box\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ and $h : \Box\mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}$. Say that $f : \Box\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ is defined from g, u, h by safe recursion with substitution of parameters if (semantically)

$$
\begin{aligned}
&f(0, x) = g(x)\\
&f(n, x) = h(n, x, f(\lfloor n/2 \rfloor, u(n, x)))
\end{aligned}
$$

One can show directly that in this case f admits a definition from g, h, u using $\mathsf{saferec}_{\mathsf{N}}$. (Leivant and Marion [14] report that in their systems of tiered recursion substitution of parameters allows one to define a PSPACE complete function. This does not contradict the above, as they consider a more general notion of parameter substitution under which $f(\lfloor n/2 \rfloor, -)$ may be applied to more than one argument in the computation of $f(n, x)$.)

Now in order to reduce $\mathsf{linrec}_{N\to N}$ to safe recursion with substitution of parameters we interpret SLR in the category of Chu-spaces over the presheaf category $\widehat{\mathcal{C}}$ from above. In order to convey the gist of the argument it is enough to consider Chu-spaces over the category of sets.

A *Chu space* is a triple $A = (|A|, A^*, \langle -|-\rangle)$ where $|A|$, A^* are sets and $\langle -|-\rangle : |A| \times A^* \to \mathbb{N}$.

A linear map from Chu space A to Chu space B is given by a function $|f| : |A| \to |B|$ and a function $f^* : B^* \to A^*$ with $\langle f(a)|\beta\rangle = \langle a|f^*(\beta)\rangle$ for all

$a \in |A|$ and $\beta \in B^*$. A nonlinear map from A to B is simply a function from A to B

The nonlinear maps form a Chu space $A \to B$ where $|A \to B| = B^A$ and $(A \to B)^*$ is $|A| \times B^*$. Furthermore, $\langle (f|(a, \beta)\rangle \stackrel{\text{def}}{=} \langle f(a)|\beta\rangle$. Similarly, we can define a Chu space $A \multimap B$ of linear maps.

The Chu-space $[\![\mathsf{N}]\!]$ is given by $|[\![\mathsf{N}]\!]| = \mathbb{N}$, $[\![\mathsf{N}]\!] = \mathbb{N}^{\mathbb{N}}$, and $\langle n|\nu\rangle = \nu(n)$. Notice that $[\![\mathsf{N}]\!] \to A \cong [\![\mathsf{N}]\!] \multimap A$.

Now a linear map from $[\![\mathsf{N}]\!] \to [\![\mathsf{N}]\!]$ to $[\![\mathsf{N}]\!]$ consists of a functional $F : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ together with a function $F^* : \mathbb{N}^{\mathbb{N}} \to \mathbb{N} \times \mathbb{N}^{\mathbb{N}}$ such that $\langle F(f)|\nu\rangle = \langle f|F^*(\nu)\rangle$. Let $a \in \mathbb{N}$ and $r \in \mathbb{N}^{\mathbb{N}}$ be the two components of $F^*(id)$. Unfolding the definitions yields $F(f) = r(f(u))$. So a linear functional is indeed "linear" in the sense that it uses its argument exactly once. We can now set up a semantics which interprets types as Chu spaces and open terms as linear or nonlinear maps between them (according to the aspect of the binding). The above analysis then allows us to show that semantically an instance of $\mathsf{linrec}_{\mathsf{N}\to\mathsf{N}}$ is equivalent to an instance of safe recursion with substitution of parameters. Now, since the latter recursion pattern is available in $\hat{\mathcal{C}}$ we can interpret linear recursion in the category of Chu spaces. $\square$

There are further characterisations which have not yet been fully verified. One is the already mentioned characterisation of polynomial space using the ad-hoc modification of $\mathsf{saferec}_{\square\mathsf{N}\to\mathsf{N}}$ from Remark 1. Another one is that $\mathsf{saferec}_{\mathsf{N}\to\mathsf{N}}$ and in fact $\mathsf{saferec}_A$ for arbitrary $\square$-free A captures the Kalmar elementary functions. Similarly, in view of [2] we expect that linrec_A for arbitrary $\square, \to$-free A remains within polynomial time.

A rather pedestrian observation is that we can lift the other first-order characterisation of complexity classes using safe recursion contained in [1] and [8]; in particular the one for the parallel complexity class NC to SLR using appropriately typed recursors and basic functions. The soundness proof would then employ a category constructed like $\mathcal{C}$ above but with $PTIME$ replaced by another complexity class.

5 Conclusions and further work

We have presented a simply-typed lambda calculus with linear types and modality in which various higher-order extensions of Bellantoni-Cook's notion of safe recursion can be formulated and tested. The presence of subtyping allows for a particularly simple formulation of the syntax; notably there are no additional term formers having to do with linearity and modality. This is not so in existing formulations of modal and linear lambda calculus [16], [4] with the exception of [19]. We hope that our formulation might be of further use for more complicated systems of linear or modal types which go beyond our application to safe recursion.

A contribution of our work to the field of subsystems of primitive recursion is (apart from the results in Section 4) that SLR provides a pragmatically more satisfying formulation of safe recursion than the original one. The advantages of

SLR are on the one hand that it allows for higher-order functions and on the other hand that the aspects (safe or normal) of variables need not be explicitly stated but can be inferred automatically in the most general way.

Our rationale in introducing the linear recursor was that it provides a more flexible syntax for safe recursion with substitution of parameters. In particular, the substitution function u (in the notation of the proof of Theorem 2, Part 2 above) need not be isolated and put in front of the definition thus yielding more readable programs. However, the simple-minded evaluation of $\mathsf{linrec}_{\mathsf{N}\to\mathsf{N}}$ using higher-order primitive recursive definitions is rather inefficient. We hope that it might be possible to derive an automatic compilation of $\mathsf{linrec}_{\mathsf{N}\to\mathsf{N}}$ into first-order primitive recursive definitions with substitution of parameters using a non-standard operational semantics derived from the Chu-space semantics.

More speculatively, we hope that it might also be possible to make use of the $\Box$-modality for optimising evaluation, for example based on the fact that a function of type $\mathsf{N} \to \mathsf{N}$ cannot recur on its argument and thus could be evaluated symbolically.

The long-term goal behind this research is the development of a fully-fledged functional programming language based on linear and modal types. Of course, such a language will have to contain arbitrary primitive recursion and even general recursion; however, the type system could issue a warning if a recursion pattern cannot be formulated using $\mathsf{saferec}$ and thus encourage the programmer to use safe recursion wherever possible and hence to remain within polynomial time. Such a programming language would also have to encompass structural recursion over other datatypes such as lists or trees. An important observation in this context is that in the presence of datatypes with binary constructors such as trees linearity becomes important already at ground type; for example the constructor for trees must be given the type $\mathsf{Tree} \multimap \mathsf{Tree} \multimap \mathsf{Tree}$ rather than $\mathsf{Tree} \to \mathsf{Tree} \to \mathsf{Tree}$ for otherwise we could define exponentially large trees by iterating the function $\lambda x{:}\mathsf{Tree}.\mathsf{node}\, x\, x$. Caseiro's thesis [6] contains a semantic account of this phenomenon for first-order functions. Preliminary work has shown that her results can at least partly be expressed type-theroetically in an appropriate extension of SLR.

We also remark in this context that a prototype implementation of SLR exists and is made available on the author's homepage (see the references).

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. in preparation, 1998.
3. Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
4. Nick Benton. A mixed linear and nonlinear logic: proofs, terms, and models. In Jerzy Tiuryn and Leszek Pacholski, editors, *Proc. CSL '94, Kazimierz, Poland, Springer LNCS, Vol. 933*, pages 121–135, 1995.

5. Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1/2):4–56, 15 February/March 1994.
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. Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. In *Symposium on Logic in Computer Science (LICS'96)*, pages 264–275. IEEE, 1996.
8. Peter Clote. Computation models and function algebras. available electronically under http://thelonius.tcs.informatik.uni-muenchen.de/ clote/Survey.ps.gz, 1996.
9. J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive Recursion for Higher-Order Abstract Syntax. Technical Report CMU-CS-96-172, Carnegie Mellon University, 1996.
10. Martin Hofmann. An application of category-theoretic semantics to the characterisation of complexity classes using higher-order function algebras. To appear in *Bulletin of Symbolic Logic*, 1997.
11. Martin Hofmann. More results on modal/linear lambda calculus. Submitted. Available under www.mathematik.tu-darmstadt.de/∼mh, 1998.
12. Yves Lafont and Thomas Streicher. Game semantics for linear logic. In Proc. 6th Annual IEEE Symp. on Logic in Computer Science, editor, *Logic in Computer Science (LICS)*, pages 43–49, 1991.
13. Daniel Leivant and Jean-Yves Marion. Lambda calculus characterisations of poly-time. *Fundamentae Informaticae*, 19:167–184, 1993.
14. Daniel Leivant and Jean-Yves Marion. Ramified Recurrence and Computational Complexity II: Substitution and Poly-space. In Jerzy Tiuryn and Leszek Pacholski, editors, *Proc. CSL '94, Kazimierz, Poland, Springer LNCS, Vol. 933*, pages 4486–500, 1995.
15. Daniel Leivant and Jean-Yves Marion. Ramified Recurrence and Computational Complexity IV: Predicative functionals and Poly-space. Manuscript, 1997.
16. Frank Pfenning and Rowan Davies. A modal analysis of staged computation. In *Proc. POPL '96, Florida.* IEEE, 1996.
17. Gordon Plotkin. Type theory and recursion. Invited talk at the 8th Annual IEEE Symposium on Logic in Computer Science, Montreal, Canada, 1993.
18. V.R. Pratt. Chu spaces and their interpretation as concurrent objects. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Springer LNCS*, pages 392–405. Springer-Verlag, 1995.
19. David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture*, San Diego, California, June 1995.
20. Philip Wadler. Is there a use for linear logic? In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991.