TYPE SYSTEMS FOR POLYNOMIAL-TIME COMPUTATION

Vom Fachbereich Mathematik
der Technischen Universität Darmstadt
angenommene

HABILITATIONSSCHRIFT

von

Martin Hofmann, PhD

aus Erlangen

Referenten:
Prof. Dr. Th. Streicher (Darmstadt)
Prof. Dr. K. Keimel (Darmstadt)
Prof. Dr. D. Basin (Freiburg)
Prof. Ph. Scott, PhD (Ottawa)

Tag der Einreichung: 30. Oktober 1998
Tag des wissenschaftlichen Gesprächs: 12. Februar 1999

Abstract

This thesis introduces and studies a typed lambda calculus with higher-order primitive recursion over inductive datatypes which has the property that all definable number-theoretic functions are polynomial time computable. This is achieved by imposing type-theoretic restrictions on the way results of recursive calls can be used.

The main technical result is the proof of the characteristic property of this system. It proceeds by exhibiting a category-theoretic model in which all morphisms are polynomial time computable by construction.

The second more subtle goal of the thesis is to illustrate the usefulness of this semantic technique as a means for guiding the development of syntactic systems, in particular typed lambda calculi, and to study their meta-theoretic properties.

Minor results are a type checking algorithm for the developed typed lambda calculus and the construction of combinatory algebras consisting of polynomial time algorithms in the style of the first Kleene algebra.

Contents

1 Introduction **2**

1.1 Primitive recursion with higher-order functions 3

1.1.1 Safe recursion 4

1.1.2 Linearity 4

1.1.3 Modal and linear function spaces 6

1.2 Statement of soundness results 7

1.3 Primitive recursion with inductive datatypes 9

1.4 Main result and proof idea 11

1.5 Expressivity 13

1.6 Previous and related work 14

2 Background **16**

2.1 Complexity theory 16

2.2 Bounded recursion 18

2.3 Safe recursion 21

2.4 Tiered recursion 23

2.5 Higher-order functions and the typed lambda calculus 24

2.5.1 Set-theoretic interpretation 26

2.5.2 The system PV^ω 26

2.6 Background on categories 27

2.6.1 Categories 28

2.6.2 Terminal object and cartesian products 29

2.6.3 Functors and natural transformations 30

2.6.4 Canonical isomorphisms and object equality 31

2.6.5 Subcategories 32

2.6.6 Global elements 32

2.6.7 Function spaces 32

2.6.8 Interpretation of the typed lambda calculus 33

2.6.9 Logical relations 35

2.6.10 Presheaves 36

2.6.11 Adjoint functors 43

2.6.12 Comonads 45

2.7 Affine linear categories 49
2.7.1 Examples 51
2.7.2 Wiring maps 52
2.7.3 Indexed tensor product 53
2.7.4 Linear function spaces 53
2.7.5 Comonads in ALC 55
2.7.6 ALC and the Yoneda embedding 55

3 The type system SLR **61**
3.1 Types and subtyping 61
3.2 Terms and typing rules 64
3.2.1 Contexts 65
3.2.2 Typing rules 65
3.2.3 Examples 67
3.3 Syntactic metatheory 72
3.4 Comparison with other systems 77
3.5 Set-theoretic semantics 78
3.6 Reduction 79
3.7 Alternative syntax with modal types 80
3.8 Other variations of SLR 84

4 Semantics of SLR **85**
4.1 A *BCK*-algebra of *PTIME*-functions 86
4.1.1 A new length measure 87
4.1.2 Construction of H_p 88
4.1.3 Truth values and numerals 92
4.2 Realisability sets 93
4.2.1 Natural numbers and other datatypes 95
4.3 Interpreting recursion 97
4.3.1 Leivant trees 103
4.4 Recursion operators as higher-typed constants 104
4.4.1 Polynomial-time functions via a comonad 106
4.4.2 Interpretation of SLR 109
4.4.3 Main result 112
4.4.4 Interpretation of $\lambda^{\Box\,!}$ 114
4.5 Duplicable numerals 114
4.6 Computational interpretation 121
4.7 More *BCK*-algebras 122
4.7.1 Linear lambda terms 123
4.7.2 A *BCK*-algebra of all *PTIME*-functions 123

5 Conclusions and further work **125**

Acknowledgements

This thesis would not have come into existence without the help of several people to all of which I should like to express my thanks here.

My advisors Klaus Keimel and Thomas Streicher and my referees David Basin and Philip Scott read and examined this thesis and provided helpful comments.

Invitations to seminars or workshops by Carsten Butz, Neil Jones, Ulrich Kohlenbach, Helmut Schwichtenberg and Daniel Leivant allowed me to present preliminary stages of this work and to obtain valuable feedback.

Benjamin Pierce introduced me to implementing type checkers in ML and allowed me to use his $F^{\omega}_{\leq}$-implementation as a basis for the typechecker for the systems described in this thesis.

Peter Lietz deserves special thanks for proofreading large parts of this thesis and spotting several typos and ambiguities.

Finally, the following people provided stimulating input to my research through discussion: Samson Abramsky, Thorsten Altenkirch, Stephen Bellantoni, Peter Clote, John Longley, Karl-Heinz Niggl, Gordon Plotkin, John Power, Helmut Schwichtenberg.

Chapter 1

Introduction

The aim of this thesis is twofold. Firstly, we develop type-theoretic syntactic restrictions of primitive recursion on inductively defined data structures which ensure polynomial time complexity.

The second more subtle goal is to illustrate the usefulness of category-theoretic semantics as a technique for guiding the precise formulation of syntactic systems and to establish metatheoretic properties.

The main technical result of the thesis is the proof that all first-order functions definable in a certain typed lambda calculus are polynomial time computable. This proof proceeds by exhibiting a category-theoretic model of the calculus in which all denotations of first-order functions are polynomial time computable by construction.

In retrospect, the semantic method may seem complicated as it involves quite a bit of technical machinery and it might well be that sooner or later a more direct syntactic proof will appear.

To appreciate the semantic approach the reader should bear in mind the following points.

- The apparent overhead of the semantic approach is largely due to the need of developing several general concepts such as affine linear categories, comonads, functor categories, logical relations, realisability sets. Once these concepts are available the actual proofs are quite short and perspicuous as they emphasize the invariants needed to establish the desired property. The abovementioned general concepts we develop in Chapter 2 and Section 4.2 of Chapter 4 are with very few exceptions independent of the particular application and can be used in other situations.

- The syntax has been developed hand in hand with the semantics and has been guided by the semantic intuition. In the author's opinion the type system described in Chapter 3 is very clear, almost free from ad-hoc constructions, and easily extensible. These (obviously subjective) properties are largely due to the semantic intuition for the syntactic concepts such as modal and linear function spaces.

1.1 Primitive recursion with higher-order functions

The starting point of this thesis is an analysis of higher-order primitive recursion on natural numbers and other inductive datatypes, in particular of the situations which lead to superpolynomial growth and/or runtime.

First, we recall that the size of natural numbers is defined as the length of their binary representation, i.e., $|x| = \lceil \log_2(x+1) \rceil$[1].

This means that ordinary primitive recursion which expresses a value $f(x)$ in terms of $f(x-1)$ will almost always lead to exponential (in $|x|$) runtime of f.

In order to define polynomial time functions on natural numbers one should use *recursion on notation* as given by the following schema:

Definition 1.1.1 *Let A be a set, $g \in A$, and $h \in \mathbb{N} \times A \longrightarrow A$. The function $f : \mathbb{N} \longrightarrow A$ is defined from g, h by* recursion on notation, *if*

$$\begin{aligned} &f(0) = g \\ &f(x) = h(x, f(\left\lfloor \tfrac{x}{2} \right\rfloor)), \textit{ when } x > 0 \end{aligned}$$

The set A is called the result type, *the function h is called the* step function *of the recursive definition*

The number of recursive unfoldings needed to evaluate $f(x)$ is precisely $|x|$, thus linear in $|x|$. Nevertheless, as we will now show, recursion on notation leads beyond polynomial time. Consider the following definition of a function $\mathrm{sq} : \mathbb{N} \longrightarrow \mathbb{N}$.

$$\begin{aligned} &\mathrm{sq}(0) = 1 \\ &\mathrm{sq}(x) = 4 \cdot \mathrm{sq}(\left\lfloor \tfrac{x}{2} \right\rfloor), \text{ when } x > 0 \end{aligned}$$

This defines a function of quadratic growth, more precisely $\mathrm{sq}(x) = [x]^2$ where $[x] =_{\mathrm{def}} 2^{|x|}$ is the least power of two greater than x. From sq we can then define an exponentially growing function $\exp : \mathbb{N} \longrightarrow \mathbb{N}$ by

$$\begin{aligned} &\exp(0) = 2 \\ &\exp(x) = \mathrm{sq}(\exp(\left\lfloor \tfrac{x}{2} \right\rfloor)), \text{ when } x > 0 \end{aligned}$$

Indeed, $\exp(x) = 2^{[x]}$.

Once we have exponentiation we can define ordinary primitive recursion in terms of recursion on notation so that we are back to square one, as it were.

One can rule out the definition of such fast-growing functions by requiring an a priori bound on the function to be defined. This leads to Cobham's recursion-theoretic characterisation of the polynomial time computable functions which we discuss later in Section 2.2. Our aim in this thesis is, however, not to require explicit bounds be it on size or on runtime, but to ensure polynomial time by syntactic type-theoretic restrictions. For first-order functions such syntactic restrictions exist in the form of tiered and safe recursion.

[1] More generally, we write $|\vec{x}|$ for $(|x_1|, \ldots, |x_n|)$ when $\vec{x} = (x_1, \ldots, x_n)$.

1.1.1 Safe recursion

Leivant [25] and Bellantoni-Cook [3] have carefully analysed recursion on notation schemes and concluded that the deeper reason for the definability of exponentiation is that in the recursive equation for $\exp(x)$ we apply a recursively defined function, namely sq to the result of a recursive call, namely $\exp(\left\lfloor\frac{x}{2}\right\rfloor)$. Their systems of tiered [25], respectively safe [3] recursion consist of formal syntactic restrictions which effectively rule out such application of recursively defined functions thereby ensuring that the definable functions are exactly those computable in polynomial time. For reasons that become clear later we call this a *modality restriction.*

In a nutshell the idea behind safe recursion is that variables are grouped into two zones usually separated by a semicolon like so: $f(\vec{x};\vec{y})$. The variables before the semicolon are called *normal* whereas those after the semicolon are called *safe*. The intended invariant is that such a function is polynomial-time computable and moreover satisfies a growth restriction of the form

$$|f(\vec{x};\vec{y})| \leq p(|\vec{x}|) + \max(|\vec{y}|)$$

for some polynomial p. The syntactic restrictions are now set up in such a way that this invariant is maintained. In particular one imposes that recursion on notation is always over a normal variable, but that results of recursive calls must be accessed via safe variables only; in other words from $g(\vec{u};\vec{v})$ and $h(x,\vec{u};y,\vec{v})$ we may form $f(x,\vec{u};\vec{v})$ by $f(0,\vec{u};\vec{v}) = g(\vec{u};\vec{v})$ and $f(x,\vec{u};\vec{v}) = h(x,\vec{u};f(\left\lfloor\frac{x}{2}\right\rfloor,\vec{u};\vec{v}),\vec{v})$, but such recursion is not allowed if y also is a normal variable.

The function $\mathrm{sq}(x) = [x]^2$ can now be defined from $g(;) = 1$ and $h(x;y) = 4y$ (Multiplication by two, hence by four is considered as a basic function of a safe argument). For $\exp(x)$ we would have to use $h(x;y) = \mathrm{sq}(y;)$ which is not allowed as y is a safe variable in the left-hand side, but normal in the right-hand side. More details will be given in Section 2.3. Tiered recursion will be described in Section 2.4.

1.1.2 Linearity

These systems were designed for a first-order result type $A = \mathbb{N}$. With a functional result type, e.g., $A = \mathbb{N}{\rightarrow}\mathbb{N}$ (the set of functions from $\mathbb{N}$ to $\mathbb{N}$), new phenomena occur which again allow for the definition of non-polynomial time computable functions. Consider the following recursion on notation with result type $A = \mathbb{N}{\rightarrow}\mathbb{N}$.

$$\begin{aligned} f(0) &= \lambda y{\in}\,\mathbb{N}.2y \\ f(x) &= f(\left\lfloor\tfrac{x}{2}\right\rfloor) \circ f(\left\lfloor\tfrac{x}{2}\right\rfloor) \end{aligned}$$

Here $\lambda y{\in}\,\mathbb{N}.2y$ denotes the function $\mathbb{N}{\ni}x \mapsto 2x$ and $\circ$ is composition of functions in the applicative order. In other words, we have

$$\begin{aligned} f(0,y) &= 2y \\ f(x,y) &= f(\left\lfloor\tfrac{x}{2}\right\rfloor, f(\left\lfloor\tfrac{x}{2}\right\rfloor, y)) \end{aligned}$$

Notice that this recursion must be considered "safe" as no previously recursively defined function is applied to results of recursive calls.

Yet, induction on x shows that $f(x,y) = \exp(x) \cdot y$.

We claim that this is due to the fact that $f(\left\lfloor \frac{x}{2} \right\rfloor)$ is called *twice* in the course of the computation of $f(x)$ and we will show that if we allow at most one "use" of the result of a recursive call then higher result types do not lead beyond polynomial time. This restriction will be called a *linearity restriction*.

One may think that in the above example the *nested* application of $f(\left\lfloor \frac{x}{2} \right\rfloor)$ in the step function is the culprit and that several independent usages of $f(\left\lfloor \frac{x}{2} \right\rfloor)$ might be harmless. However, by adapting an example from [28] one obtains a definition of an evaluation function for quantified boolean formulas which is complete for polynomial space and thus (commonly believed not to be) polynomial time computable.

Let us now give an example of a useful recursive definition which does obey the linearity restriction.

Consider an addition function add : $\mathbb{N} \longrightarrow A$ where $A = \mathbb{N}^3 \to \mathbb{N}$ specified by

$$\mathrm{add}(l, x, y, c) = x + y + (c \bmod 2) \text{ provided } |l| \geq \max(|x|, |y|)$$

Such a function admits a definition by recursion on notation with result type $A = \mathbb{N}^3 \to \mathbb{N}$ using only case distinction on parity and binary successor functions $\mathsf{S}_0(x) = 2x, \mathsf{S}_1(x) = 2x + 1$ and some self-explanatory abbreviations:

```
$\mathrm{add}(0) = \lambda(x,y,c)\in\mathbb{N}^3.c \bmod 2$
$\mathrm{add}(x) = \lambda(x,y,c)\in\mathbb{N}^3.$let carry = $(x \wedge (y \vee c))((\neg x) \wedge (y \wedge c))$ in
  if $x \oplus y \oplus c = 0$
    then
      $\mathsf{S}_0(\mathrm{add}(\left\lfloor \frac{x}{2} \right\rfloor)(\left\lfloor \frac{x}{2} \right\rfloor, \left\lfloor \frac{y}{2} \right\rfloor, \mathrm{carry}))$
    else
      $\mathsf{S}_1(\mathrm{add}(\left\lfloor \frac{x}{2} \right\rfloor)(\left\lfloor \frac{x}{2} \right\rfloor, \left\lfloor \frac{y}{2} \right\rfloor, \mathrm{carry}))$
```

This definition obeys the following two restrictions. For one thing, no recursively defined function is applied to the result of a recursive call; for another the step function $h : \mathbb{N} \times A \longrightarrow A$ uses its second argument only once. Notice that the argument $\mathrm{add}(\left\lfloor \frac{x}{2} \right\rfloor)$ literally appears twice and even gets applied to different values each time. However this does not count, since these branches are within different branches of a case distinction. We shall see below how this is expressed syntactically.

The above definition can be seen as a recursion on notation with *first-order* result type $A = \mathbb{N}$ and *substitution of parameters*. Indeed, "linear" recursion with functional result type $A = \mathbb{N}^k \to \mathbb{N}$ can always be reduced to recursion with parameter substitution using a semantic technique described in [18, 20]. However, as we shall describe, linearity is more subtle in the presence of other inductive datatypes such as binary trees.

1.1.3 Modal and linear function spaces

One of the main contributions of this work will be the definition and analysis of a formal system in which these criteria, viz. linearity and modality, have a well-defined syntactical status. This system takes the form of a typed lambda calculus (a formal system for defining higher-order functions) with three kinds of function spaces:

- Modal function space, written $\Box A \to B$: A function of this type is allowed to use its argument as input to a recursively defined function and also an arbitrary number of times. In particular the type of functions defined by recursion on notation with result type A will be $\Box \mathsf{N} \to A$[2].

- Linear function space, written $A \multimap B$: A function of this type is neither allowed to recurse on its argument nor to use it more than once. The application functional $h(f) = f(0)$ is in $(\mathsf{N} \to A) \multimap A$ or $(\Box \mathsf{N} \to A) \multimap A$. Also basic functions such as $\mathsf{S}_0(x) = 2x$ and $\mathsf{S}_1(x) = 2x + 1$ are in $\mathsf{N} \multimap \mathsf{N}$.

- Nonlinear function space, written $A \to B$: A function of this type is allowed to use its argument an arbitrary number of times without, however, recursing on it. The ordinary function space is included in the modal one and contains the linear function space. A typical inhabitant of this function space which is not linear is the composition functional $h(f) = f \circ f$ which is in $A \to A$ when A is of the form $B \multimap B$, $B \to B$, or $\Box B \to B$.

  We will sometimes adopt the convention that a function on natural numbers is always linear, even if it uses its argument more than once, i.e., we may identify $\mathsf{N} \to A$ and $\mathsf{N} \multimap A$.

The scheme of recursion on notation is then subject to the following three restrictions:

- The step function must be of type $\Box \mathsf{N} \to A \multimap A$, i.e., $f(\lfloor \frac{x}{2} \rfloor)$ must be used at most once in the definition of $f(x)$ and in particular must not appear as a subterm of a term to be recursed on; the recursion variable x itself may be used more than once and even recursed on.

- The type of a function defined by recursion on notation is $\Box \mathsf{N} \to A$.

- The result type must be built up from N and $\multimap$ (and a few more constructors to be introduced later); in particular it must neither contain modal nor ordinary function space. Such types will be called *safe types*.

The first restriction rules out both nonlinear and recursive use of results of recursive calls.

The second restriction is a consequence of the first one; recursively defined functions get modal type thus (by the first restriction) cannot be used within step functions.

[2] We will write N for the type expression denoting the set $\mathbb{N}$ of natural numbers. This distinction between type expression and denoted set becomes important when we consider other interpretations.

The third restriction is necessary since otherwise the first restriction could be overcome by a kind of tail recursion as in

$$\begin{aligned} f(0) &= \lambda y{\in}\,\mathsf{N}.y \\ f(x) &= \lambda y{\in}\,\mathsf{N}.f(\left\lfloor \tfrac{x}{2} \right\rfloor, \mathrm{sq}(y)) \end{aligned}$$

Here, the step function

$$h(x, u) = \lambda y{\in}\,\mathsf{N}.u(\mathrm{sq}(y)))$$

has the type $\Box\mathsf{N}{\to}(\Box\mathsf{N}{\to}\mathsf{N}){\multimap}(\underline{\Box}\mathsf{N}{\to}\mathsf{N})$ where the underlined $\Box$ comes from the fact that y is used as argument to a recursively defined function, namely sq.

If result type $A = \Box\mathsf{N}{\to}\mathsf{N}$ would be allowed then this would be a legal recursion on notation; however, $f(x, y) = y^{[x]}$.

Syntactically, these restrictions are enforced by representing recursion on notation as a family of higher-order constants

$$\mathsf{rec}^{\mathsf{N}}[A] : \mathsf{N}{\multimap}(\Box\mathsf{N}{\to}A{\multimap}A){\to}\Box\mathsf{N}{\to}A$$

for each allowed result type A. In an application $\mathsf{rec}^{\mathsf{N}}[A]\,g\,h$ the first argument g is the base case and the second one h is the step function. Notice that the passage from step function to the recursively defined function is nonlinear. Were it linear one could circumvent the syntactic restrictions by some encoding.

The main contribution of this thesis is an appropriate formalisation and semantic interpretation of these three function space which ensures that the ensuing recursion patterns allow the definition of polynomial time computable functions only. The proof of this soundness property proceeds by interpretation in a model which at the same time provides semantic intuition for the three function spaces.

1.1.3.1 Role of the nonlinear function space

Several people have asked why we need the intermediate nonlinear function space $A{\to}B$ and whether it wouldn't be simpler to have just $A{\multimap}B$ and $\Box A{\to}B$. The answer is that the nonlinear function space allows for finer typings and higher expressivity, but can be left out if the user does not feel comfortable with it.

The main purpose of the nonlinear function space is that it allows one to postulate the type equality $\mathsf{N}{\to}A = \mathsf{N}{\multimap}A$ which permits multiple use of ground type variables. Without nonlinear function space this would have to be formulated as an ad-hoc condition. Nonlinear function space also allows for a slightly more generous typing of recursion constants, cf. Remark 3.2.1.

1.2 Statement of soundness results

Due to the presence of higher-order functions and different function spaces it might not be immediately clear how to express formally that all definable function are polynomial time computable and indeed there are different inequivalent such formalisations. The approach we take is based on a set-theoretic interpretation of our systems which interprets all three function spaces as set-theoretic function space and expressions as elements of the denotation of their type. In particular, recursion on notation will be interpreted as in Def. 1.1.1. In other words, this set-theoretic interpretation simply consists of forgetting about linearity and modality and treating functional expressions as the functions they intuitively denote. The soundness property is then stated as follows

Principle 1.2.1 (Soundness, first version) *The set-theoretic interpretation of a term of type* $\Box\mathsf{N}\to\mathsf{N}$ *is always a polynomial time computable function.*

In favour of this definition we put forward the following reasons.

- Eventually we are only interested in evaluating programs sending integers to integers; higher-order functions only appear as subterms of first-order programs.
- If in our system it was possible to define a non-polynomial time computable (in some sense) higher-order function then in many cases it would be possible to construct from it a non-polynomial time computable function of type $\Box\mathsf{N}\to\mathsf{N}$ in contradiction to the soundness principle.

Crucial to this approach is that higher-order functions are viewed as auxiliary concepts needed to structure, modularise, and simplify eventually first-order programs. If we allow for programs with functional input as might occur e.g., in exact real number computation then a more fine-grained approach could be necessary. It is known, see e.g. [8] that there are several non-equivalent notions of second-order polynomial-time computability which are equivalent as far as the effect on first-order functions is concerned.

Again in support of our view, one could say that even programs computing with exact real numbers will eventually accept environment requests such as keyboard inputs and produce answers such as screen outputs so that in essence a first-order function is computed the complexity of which can be captured by our approach.

Next, we should explain the prominent status of polynomial time in our approach. For one thing, polynomial time is generally considered as the appropriate mathematical formalisation of "feasible computation". As any formalisation it has its limits and indeed, as is well-known, there are several obviously feasible algorithms which exhibit superpolynomial runtime in the worst case (simplex algorithm, ML type inference) and there are even undecidable problems which can nevertheless be decided quickly in all interesting cases (subtyping in $F_{\leq}$ [36]). On the other hand, in some cases even quadratic runtime must be considered unfeasible. But it seems difficult to give a better formalisation of feasible computation without getting lost in details.[3]

Secondly, the methods we use are to a large extent independent of the particular complexity class and could well be applied to higher-order extensions of recursion schemata

[3] Recently, some people have proposed to identify "feasibility" with the probabilistic complexity class BPP. We will not consider this.

capturing other classes as can be found e.g. in [7]. The advantage of polynomial time as opposed to smaller classes is that it is relatively robust with respect to choice of encoding, machine model, etc. and thus allows one to largely abstract away from these issues.

Finally, one might object against the platonic existence statement contained in the soundness principle. What does it help if we know that the function computed by our program is in principle polynomial time computable when our very program computes it in, say, exponential time.

At this point a finer look at the proofs of the soundness property is needed. As will turn out, these proofs are entirely constructive, so that it is possible to extract an algorithm, viz., a compiler, which effectively transforms a program of type $\Box\mathsf{N}\to\mathsf{N}$ into an extensionally equal polynomial time algorithm. Due to the fact that already the expansion of higher-order definitions takes superpolynomial time [40] we cannot expect that this compilation process itself runs in polynomial time.

In current work, which is, however, not reported in this thesis we implement this translation and explore to what extent it gives efficiency gains compared to usual compiler techniques which do not take into account the extra information given by the linear/modal typing.

So, summing up, we will actually establish the following refined version of the soundness principle.

Principle 1.2.2 (Soundness, final version) *The set-theoretic interpretation of a term of type* $\Box\mathsf{N}\to\mathsf{N}$ *is always a polynomial time computable function and a polynomial-time algorithm witnessing this can be effectively obtained from such term.*

1.3 Primitive recursion with inductive datatypes

The formulation of the soundness principle is sufficiently flexible to encompass data structures other than the natural numbers such as lists or trees. All we ask is that their addition should not affect the complexity of definable functions of type $\Box\mathsf{N}\to\mathsf{N}$.

Inductive datatypes can be used in two different ways. For one thing, they can be part of result types, for another we can define functions on inductive datatypes by recursion along their inductive definition.

As prototypical examples we will in this thesis use lists and binary labelled trees, both over some parameter type which undergoes the same restriction as the result type of recursion on notation. Other inductive types can be added by following this pattern provided their constructors do not contain functional arguments. Inductive types which do not satisfy this restriction and more generally mixed-variance recursive types such as $D \cong \mathsf{N}\multimap D$ can also be added, but no recursor will be available for those. The details of these extensions are not contained in this thesis and will be presented elsewhere.

Lists over type A come with two primitive constructor functions $\mathsf{nil} : \mathsf{L}(A)$, denoting the empty list and $\mathsf{cons} : A\multimap\mathsf{L}(A)\multimap\mathsf{L}(A)$, where $\mathsf{cons}(a,l)$ is obtained from l by prefixing a.

Trees $\mathsf{T}(A)$ over type A are constructed by way of the functions $\mathsf{leaf} : A \multimap \mathsf{T}(A)$ and $\mathsf{node} : A \multimap \mathsf{T}(A) \multimap \mathsf{T}(A)$.

Notice the linear typing of these operations.

If we were to encode lists or trees as natural numbers using some (recursively defined) pairing function then we would obtain the weaker typings

$$\begin{aligned}&\mathsf{cons} : A \multimap \Box\mathsf{L}(A) \to \mathsf{L}(A)\\ &\mathsf{node} : A \multimap \Box\mathsf{T}(A) \to \Box\mathsf{T}(A) \to \mathsf{T}(A)\end{aligned}$$

which prevents the use of cons and node in step functions and thus the definition of almost any reasonable function involving lists or trees.

This also suggests that (unlike e.g., in Gödel's system T) lists and trees can *not* be obtained as a definitional extension from the subsystem with natural numbers and functions between them.

Definition 1.3.1 *A function f on lists yielding values in B is defined from h_{nil} and h_{cons} by list recursion with result type B and step function h_{cons} if*

$$\begin{aligned}&f(\mathsf{nil}) = h_{\mathsf{nil}}\\ &f(\mathsf{cons}(a, l)) = h_{\mathsf{cons}}(a, l, f(l))\end{aligned}$$

In our system the step function is required to be of type $\Box(A) \to \Box(\mathsf{L}(A)) \to B \multimap B$ and the resulting function f will be of type $\Box\mathsf{L}(A) \to B$.

Definition 1.3.2 *A function f on trees yielding values in B is defined from h_{leaf} and h_{node} by tree recursion with result type B and step function h_{node} if*

$$\begin{aligned}&f(\mathsf{leaf}(a)) = h_{\mathsf{leaf}}(a)\\ &f(\mathsf{node}(a, l, r)) = h_{\mathsf{node}}(a, l, r, f(l), f(r))\end{aligned}$$

In our system the step function is required to be of type

$$\Box(A) \to \Box(\mathsf{T}(A)) \to \Box(\mathsf{T}(A)) \to B \multimap B \multimap B$$

and the resulting function f will be of type $\Box\mathsf{T}(A) \to B$.

Here is an example of a function involving recursion on notation with result type $\mathsf{L}(A)$. Let $a : A$ be a parameter and define $f_a : \Box\mathsf{N} \to \mathsf{L}(A)$ by

$$\begin{aligned}&f_a(0) = \mathsf{nil}\\ &f_a(x) = \mathsf{cons}(a, f_a(\left\lfloor \tfrac{x}{2} \right\rfloor))\end{aligned}$$

We get $f_a : \Box\mathsf{N} \to \mathsf{L}(A)$ and $\lambda a{:}A.f_a : A \to \Box\mathsf{N} \to \mathsf{L}(A)$. Giving the stronger type $A \multimap \Box\mathsf{N} \to \mathsf{L}(A)$ would be incorrect as a is effectively used more than once.

Indeed, together with constructions for decomposing lists which we introduce later we are able to define a function $g : \mathsf{L}(\mathsf{N} \multimap \mathsf{N}) \multimap \mathsf{N} \multimap \mathsf{N}$ which composes the first two entries of a list, i.e., $g(\mathsf{cons}(u, \mathsf{cons}(v, l))) = u \circ v$, which together with the (illegal) typing of f above would allow us to define $\exp : \Box\mathsf{N}{\to}\mathsf{N}$ using recursion on notation with result type $\mathsf{N}{\multimap}\mathsf{N}$.

As an example of list recursion, we can define concatenation as a function of type $\Box\mathsf{L}(A){\to}\ \mathsf{L}(A){\multimap}\mathsf{L}(A)$ by

$$\begin{aligned}&\mathrm{append}(\mathsf{nil}, l) = l\\&\mathrm{append}(\mathsf{cons}(a, l'), l) = \mathsf{cons}(a, \mathrm{append}(l', l))\end{aligned}$$

Here l is a parameter of type $\mathsf{L}(A)$, the result type is $\mathsf{L}(A)$ and the step function is $h_{\mathsf{cons}}(a, l', k) = \mathsf{cons}(a, k)$ which admits the type $\Box\mathsf{L}(A){\to}\Box A{\to}\mathsf{L}(A){\multimap}\mathsf{L}(A)$. In fact, it even has the stronger type $\mathsf{L}(A){\multimap}A{\multimap}\mathsf{L}(A){\multimap}\mathsf{L}(A)$. We remark at this point that through the use of *subtyping* we do not need to commit ourselves to a particular type when defining a function. Within certain limits the appropriate one can be inferred whenever a term is used.

Binary trees as result type provide another justification for the linearity restriction in step functions. The function $\mathrm{dup}(x) : \mathsf{leaf}(0, x, x)$ takes a tree $x{:}\,\mathsf{T}(\mathsf{N})$ and constructs a new tree labelled with 0 and having two copies of x as immediate successors. This function dup gets the type $\mathsf{T}(\mathsf{N}){\to}\mathsf{T}(\mathsf{N})$ rather than $\mathsf{T}(\mathsf{N}){\multimap}\mathsf{T}(\mathsf{N})$ because its argument is used twice. This thwarts the following tentative definition of a function $f : \Box\mathsf{N}{\to}\mathsf{T}(\mathsf{N})$:

$$\begin{aligned}&f(0) = \mathsf{leaf}(0)\\&f(x) = \mathrm{dup}(f(\left\lfloor \tfrac{x}{2} \right\rfloor))\end{aligned}$$

This function constructs a tree of size $2^{|n|}$ thus exhibits exponential growth. In Leivant's system [25], which does not provide linearity, such function is nevertheless allowed, which he justifies by measuring trees by their depth rather than by their number of nodes. Caseiro [5], from where this example is taken, develops semantic criteria which rule out such functions. Loc. cit. does not contain linear types, however.

Using tree recursion we can define a function $z : \Box\mathsf{T}(\mathsf{N}){\to}\mathsf{N}{\multimap}\mathsf{N}$ such that $z(t, x) = 2^{\#(t)} \cdot x$ where $\#(t)$ is the number of leaves in t:

$$\begin{aligned}&z(\mathsf{leaf}(a)) = \lambda x.2x\\&z(\mathsf{node}(a, l, r)) = \lambda x.z(l)(z(r)(x))\end{aligned}$$

We have $z(f(x), 1) = 2^{[x]}$ which is not polynomial time computable and shows why f must be rejected. In Leivant's system this is not possible as it does not allow for functional result types.

1.4 Main result and proof idea

The main result of this thesis is the proof that the above-described modal linear lambda calculus (or rather a suitable formalisation thereof) is sound in the sense spelled out in Section 1.2 above.

The idea behind this proof is to show that a function

$$f : \Box A_1 \to \dots \Box A_m \to B_1 \multimap \dots B_n \multimap C$$

where the types A_i, B_j, C are "safe" in the sense of Section 1.1.3 is polynomial time computable and satisfies a growth restriction of the form

$$\ell(f(\vec{x};\vec{y})) \leq p(\ell(|\vec{x}|)) + \sum_{i=0}^{n} \ell(y_i)$$

where p is a polynomial depending on f and ℓ is an appropriate length measure which on integers agrees with the usual length in binary. There is an obvious formal similarity to the Bellantoni-Cook invariant used to establish soundness of safe recursion mentioned in Section 1.1.1. The main difference is the replacement of maximum by summation. This is what accounts for linearity on the semantic side. For one thing the constructor for trees will in general satisfy $\ell(\mathsf{node}(a, l, r)) = \ell(a) + \ell(l) + \ell(r) + O(1)$, but not $\ell(\mathsf{node}(a, l, r)) = \max(\ell(a), \ell(l), \ell(r)) + O(1)$. For another, we can adopt the view that the length of a function $f : A \longrightarrow B$ is the amount by which it extends the length of its input, i.e.,

$$\ell(f) = \max_{a \in A}(\ell(f(a)) \dot{-} \ell(a))$$

where a ranges over the possible inputs to f. (We may put $\ell(f) = \infty$ if the maximum doesn't exist.) Then the length of a composition $g \circ f$ will be less or equal to $\ell(f)+\ell(g)$, but in general not be bounded by $\max(\ell(f), \ell(g))$. Thus, if we want to define step functions involving composition and, more generally, linear lambda calculus, then we are lead to replace maximisation by summation in the Bellantoni-Cook invariant.

This invariant also shows the necessity of linearity: Suppose that $\ell(f(x, y)) \leq \ell(x) + \ell(y) + c$ for some constant c. If we form $g(x) = f(x, x)$ then we can in general not find a constant d so that $\ell(g(x)) \leq \ell(x) + d$. Thus, the above invariant is not maintained under identification of arguments.

Most of the technical work described in this thesis can be seen as an appropriate formalisation of this rather intuitive idea. The necessary steps are the following.

- We must find an appropriate notion of polynomial time computability for higher-order linear functions and also an appropriate length measure for those. This will be done using an applicative structure of certain length bounded polynomial time algorithms. The main result in this part is that these algorithms can be organised into a so-called *BCK*-algebra which via the "modest-set construction" provides an appropriate algorithmic interpretation for the linear lambda calculus fragment of our system. This interpretation fleshes out the above-mentioned "polysum" invariant.

- We must give meaning to types other than those of the form $\Box\vec{A}\to\vec{B}\multimap C$, in particular the nonlinear function spaces and iterated modal function space like in e.g., $\Box(\Box\mathsf{N}\multimap\mathsf{N})\multimap\mathsf{N}$. This is done by a general category-theoretic technique derived from the *Yoneda embedding*. The results we develop here are formulated and proved in a sufficiently abstract way so that they become independent of the intended application and could be used in other situations as well.

– We must formally define the syntax and typing rules for modal linear type lambda calculus, define its interpretation in the category-theoretic semantics and relate it to its set-theoretic standard interpretation to deduce the main result. Here our rationale was that the terms should not contain any term-formers or constants referring to linearity or modality. The user should rather write down terms which look like ordinary terms in simply-typed lambda calculus—the "aspect", i.e., modal or linear function space will be inferred automatically according to the usages of abstracted variables.

  In our opinion such inference mechanism simplifies the use of the system and indeed experiments with a prototype implementation are promising in this respect.

  The category-theoretic semantics which gives the desired soundness proof is, however, within a wide range, independent of the particular syntax chosen. We will exploit this fact by defining another calculus which has only a single function space: $A \multimap B$ and two unary type formers $\Box(-)$ and $!(-)$ which allow the definition of modal function space as $\Box(A) \multimap B$ and of nonlinear function space as $!(A) \multimap B$. In this calculus we do need extra constants referring, e.g., to necessitation.

  The category-theoretic semantics can be used without any changes to interpret this alternative calculus thus giving a soundness result for it as well.

1.5 Expressivity

Another possible criticism against the soundness principle is that it does not say anything about the expressivity of our systems; indeed, it could be trivially met by a programming language which does not define any function at all or only constant functions.

Of course, soundness is only a minimal sanity condition and we have to argue for expressivity separately, notably by giving examples and by providing as many accepted programming language constructs as possible.

The systems presented in this thesis will enjoy the property that all polynomial time computable functions are in fact definable by terms of type $\Box\mathsf{N} \to \mathsf{N}$. However, such completeness property does not say very much about the expressivity of a system from a programming language point of view as the proof of such property can be (and usually is) conducted by an encoding of Turing machine computations. Indeed, the implementation guaranteed by such completeness theorems usually proceed by showing that functions of arbitrary polynomial growth can be represented and that a function $\mathrm{step} : \Box\mathsf{N} \to \mathsf{N} \multimap \mathsf{N}$ with the property that $\mathrm{step}(n, c)$ is the next configuration of a Turing machine configuration c provided $|c| \leq |n|$. Given these ingredients any polynomial time computable function can be defined by iterating step sufficiently many times.

Clearly, this is practically unsatisfactory; not only because it involves rather heavy encoding; this in fact could be overcome using appropriate abbreviations; but more seriously because it places the burden of finding an appropriate polynomial runtime bound on the programmer rather than on the type system.

Admittedly, one cannot expect much more from a completeness theorem which should encompass all polynomial time computable functions even those which are polynomial time computable for some deep mathematical reason rather than something superficial that could be detected automatically by a type system. However, in order that our systems can claim to be of relevance for practical programming we must provide constructs which allow the programmer to code in an uncontrived way as close as possible to what he or she is used to.

The addition of higher-order functions as well as inductive datatypes are a step in this direction on the theoretical side. On the practical side, we have developed an implementation which allows one to carry out somewhat larger examples.

1.6 Previous and related work

The system presented here is closely related to the one developed independently by Bellantoni-Niggl-Schwichtenberg [2]. Their system also uses modal and linear types and boasts safe recursion with higher result type. At present, it is based entirely on integers and the only type formers are the function spaces $\Box A \to B$ (in their notation $!A \multimap B$) and $A \multimap B$.

Another important difference is the setup of the type system as a whole. Whereas the present system enjoys linear time decidability of all judgements, well-typedness in [2] is intertwined with untyped reduction and consequently as complex as the latter. We also believe that due to subtyping and modality inference the present system is somewhat easier to use in practice than the other one.

The main difference between the two approaches lies, however, in the soundness proof. Whereas op. cit. is based on a syntactical analysis of a normalisation procedure the present proof is based on an interpretation of the calculus in a semantic model. It is at this point not clear whether this syntactic method can be extended to the more general syntax studied here, e.g., general inductive types, cartesian products, and polymorphism.

Another related system is Girard-Asperti's [13, 1] Light Linear Logic (LLL). Like [2] this system is a linearly typed lambda calculus admitting a polynomial time normalisation procedure. Although it can be shown that all polynomial time functions are expressible in LLL, the pragmatics, i.e., expressibility of particular algorithms, is unexplored, and superficial evidence suggests that the system would need to be improved in this direction so as to compete with our calculus. A more detailed comparison between the available programming patterns in either system would be very desirable, but must at present await further research.

The systems of tiered recursion studied by Leivant and Marion [26, 27, 28] also use restrictions of primitive recursion in order to achieve complexity effects. One difference is that the use of modality is replaced by the use of several copies of the base types ("tiers"). Another difference is that linearity and the ensuing recursion patterns with higher-result type have not been studied in the context of the Leivant-Marion work. Accordingly, functional result types in these systems lead to higher complexity classes [28]. We show in Section 2.4 that safe recursion can be seen as a "high-level language" for tiered recursion, i.e., that there is a procedure which annotates an SLR-term (without higher-order result types) with tiers so that it becomes typable in Leivant-Marion's system. It therefore seems plausible that by adding linearity to their system one could achieve higher-order result types and conversely that the characterisations in [28] could be transported to the realm of safe recursion.

Finally, we mention Caseiro's systems [5]. She studies first-order extensions of safe recursion with inductive datatypes and develops criteria which apply to recursion patterns presently not allowed in SLR like the one used in the direct definition of insertion sort. Unfortunately, these criteria are rather complicated, partly semantical, and do not readily apply to higher-order functions. We hope that by further elaborating the techniques presented in this paper it will be possible to give a type-theoretic account of Caseiro's work which would constitute a further step towards a polynomial time functional programming language. In [21] we have carried out a step in this direction.

Superficially related are the works by Otto [31] and Goerdt [14]. Otto uses category-theoretic methods to account for the syntax of safe recursion, whereas we use these tools to provide a semantical proof of soundness.

Goerdt's approach guarantees polynomial time complexity of functions definable in a system like Gödel's T without any syntactic restrictions by interpreting it in a finite model. In other words, there is an a priori maximum number N for intermediate results. The successor functions are interpreted as $\mathsf{S}_i(x) = \min(N, 2x + i)$ so that the bound N is never exceeded. The price to be paid is that when reasoning about programs written in this system one always has to take into account the possibility of cut-off due to overflow.

Chapter 2

Background

In this chapter we develop the necessary technical background in the fields of computability and complexity theory, lambda calculus and type systems, and category theory. Most of the material is well-established or part of the folklore in the field. To the author's knowledge, are the following parts original:

- The use of presheaf semantics to establish conservativity of a higher-order extension of a first order theory with binding operators such as $\mathcal{F}$, see also [15],
- The comonad semantics of modal lambda calculus and the relationship between S4 modal operators and safe recursion, see also [18],
- The notion of well-pointed affine linear category, the use of extensional presheaves in this context, in particular the definition of the !-operator in $\mathrm{Ext}(\mathbb{C})$.
- The back-and-forth translation between safe recursion and tiered recursion.

The description is meant to be understandable for readers without prior knowledge of category theory and type systems. The reader who is familiar with these subjects might nevertheless want to at least skim the material in order to understand the notation and the new results mentioned above.

2.1 Complexity theory

For integer $x \in \mathbb{N}$ we write $|x| = \lceil \log_2(x+1) \rceil$ for the length of x in binary notation. For example, we have $|0| = 0, |3| = 2, |2^t| = t+1$. If $\vec{x} = (x_1, \dots, x_n)$ then $|\vec{x}| = (|x_1|, \dots, |x_n|)$.

We have $|x| + |y| - 1 \leq |xy| \leq |x| + |y|$ and $\max(|x|, |y|) \leq |x+y| \leq \max(|x|, |y|) + 1$.

We write $||x||$ for the length of the length of x, i.e., $||x|| = |a|$ when $a = |x|$.

By "polynomial" we will always mean "polynomial with positive integer coefficients".

We sometimes use Knuth's O-notation. If $f : X \longrightarrow \mathbb{N}$ is an $\mathbb{N}$-valued function (typically $X = \mathbb{N}^k$) then $O(f)$ is the set of all those functions $g : X \longrightarrow \mathbb{N}$ for which there exist constants c, d such that $g(a) \leq cf(a) + d$. It is customary to write $g = O(f)$ rather than $g \in O(f)$ in this case.

We must fix some universal machine model allowing us to formalise runtimes and to encode algorithms as natural numbers. Since we are only interested in polynomial time we need not be very precise about this machine model. For the sake of definiteness we take ordinary one-tape Turing machines. In order to write down particular algorithms we use a simple imperative language on integer variables which in addition to the usual control structures has the following two statements: $\mathbf{read}\,\vec{x}$ reads the input (presented e.g. as the content of the tape upon initialisation) and stores it in the variables $\vec{x}$. The statement $\mathbf{write}\,e$ outputs the value of expression e and terminates the whole program. All intermediate computations are carried out by way of evaluating expressions and assigning to integer variables. These statements occur each exactly once in an algorithm.

Let $f, T : \mathbb{N}^n \rightarrow \mathbb{N}$ be functions. We say that f has time complexity T if there exists a Turing machine $\mathcal{A}$ such that for each $\vec{x} = (x_1, \dots, x_n)$ the machine $\mathcal{A}$ started on input $\vec{x}$ (coded in binary with blanks as separators) terminates after not more than $T(|\vec{x}|)$ steps with output $f(\vec{x})$ (also coded in binary).

Since we need at least some time to write the output the time complexity also bounds the size of the result:

$$|f(\vec{x})| \leq T(|\vec{x}|)$$

A function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is polynomial time computable ($f \in \mathit{PTIME}$) if f has time complexity p for some n-variate polynomial p.

Lemma 2.1.1 (Pairing function) *There exist injections* $\mathsf{num} : \mathbb{N} \longrightarrow \mathbb{N}$, $\langle -, - \rangle : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ *with disjoint images such that* $\mathsf{num}(x)$, $\langle x, y \rangle$ *as well as the functions* $.1$ *and* $.2$, getnum, isnum, *and* ispair *defined by*

$$\begin{array}{l}
\langle x, y \rangle.1 = x \\
\langle x, y \rangle.2 = y \\
z.1 = z.2 = 0, \textit{ otherwise} \\
\mathsf{ispair}(\langle x, y \rangle) = 1 \\
\mathsf{ispair}(z) = 0, \textit{ otherwise} \\
\mathsf{getnum}(\mathsf{num}(x)) = x \\
\mathsf{getnum}(z) = 0, \textit{ otherwise} \\
\mathsf{isnum}(\mathsf{num}(x)) = 1 \\
\mathsf{isnum}(z) = 0
\end{array}$$

are computable in linear time and such that moreover we have

$$\begin{array}{l}
|\langle x, y \rangle| \leq |x| + |y| + 2||y|| + 3 \\
|\mathsf{num}(x)| \leq |x| + 1
\end{array}$$

Proof. Let $F(x)$ be the function which writes out the binary representation of x using 00 for 0 and 01 for 1, i.e., formally, $F(x) = \sum_{i=0}^{n} 4^i c_i$ when $x = \sum_{i=0}^{n} 2^i c_i$.

We now define $\langle x, y\rangle$ as $x \,\hat{}\, y \,\hat{}\, 1 \,\hat{}\, 1 \,\hat{}\, F(|y|) \,\hat{}\, 0$ where $\hat{}$ is juxtaposition of bit sequences, i.e., $x \,\hat{}\, y = x \cdot 2^{|y|} + y$. We define $\mathsf{num}(x)$ as $x \,\hat{}\, 1 = 2x + 1$.

In order to decode $z = \langle x, y\rangle$ we strip off the least significant bit (which indicates that we have a pair), then continue reading the binary representation until we encounter the first two consecutive ones. What we've read so far is interpreted as the length of y. Reading this far further ahead gives us the value of y. The remaining bits correspond to x.

□

Assume some reasonable coding of Turing machines and configurations as natural numbers using the above pairing function. For Turing machine e and input $\vec{x}$ we let $\mathsf{init}(e, \vec{x})$ denote the initial configuration of Turing machine e applied to input $\vec{x}$. For configuration $c \in \mathbb{N}$ (which includes the contents of the tapes as well as the machine itself) we let $\mathsf{step}(c)$ denote the configuration obtained from c by performing a single computation step. We let $\mathsf{term}(c) = 0$ if c is a terminated configuration and $\mathsf{term}(c) = 1$ otherwise. We may assume that $\mathsf{term}(c) = 0$ implies $\mathsf{step}(c) = c$. Finally, we let $\mathsf{out}(c)$ be the output contained in a terminated configuration c. We may assume that $\mathsf{term}(c) = 1$ implies $\mathsf{out}(c) = 0$. It is intuitively clear that these basic functions are computable in linear time as they only involve case distinctions and simple manipulations of bitstrings; see [7] for a formal proof.

Suppose that $f : \mathbb{N}^n \to \mathbb{N}$ has time complexity T and that e is a program (Turing machine) for f. We write

$$F(i, \vec{x}) = \mathsf{step}^i(\mathsf{init}(e, \vec{x}))$$

for the configuration reached after i steps on input $\vec{x}$. We have

$$f(\vec{x}) = F(T(|\vec{x}|), \vec{x})$$

and furthermore, if T is a polynomial then

$$\forall i.|F(i, \vec{x})| \leq p(|\vec{x}|)$$

for some polynomial $p \geq T$. "Morally", we should have $|F(i, \vec{x})| \leq T(|\vec{x}|)$ because the Turing machine cannot use more than $T(|\vec{x}|)$ storage bits. However, since the program is part of the configuration and due to the slight overhead involved with the use of pairing functions the configuration may get slightly larger.

2.2 Bounded recursion

The above algebraisation of Turing machine computations allows one to characterise polynomial time computability in terms of a restricted pattern of recursion.

Definition 2.2.1 *Let $f : \mathbb{N}^n \to \mathbb{N}$ and $g_i : \mathbb{N}^k \to \mathbb{N}$ (for $i = 1 \dots n$) be functions. We say that $h : \mathbb{N}^k \to \mathbb{N}$ is defined from f and $\vec{g}$* by composition *if*

$$h(\vec{x}) = f(g_1(\vec{x}), g_2(\vec{x}), \dots, g_k(\vec{x}))$$

Definition 2.2.2 *Let $g : \mathbb{N}^n \to \mathbb{N}$, $h_0, h_1 : \mathbb{N}^{n+2} \to \mathbb{N}$, $k : \mathbb{N}^{n+1} \to \mathbb{N}$ be functions. We say that $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is defined from g, h_0, h_1, k by* bounded recursion on notation *if*

$$\begin{array}{l} f(0, \vec{y}) = g(\vec{y}) \\ f(2x, \vec{y}) = h_0(x, f(x, \vec{y}), \vec{y}), \textit{ if } x > 0 \\ f(2x+1, \vec{y}) = h_1(x, f(x, \vec{y}), \vec{y}) \end{array}$$

and, moreover, $f(x, \vec{y}) \leq k(x, \vec{y})$ for all $x, \vec{y}$.

Definition 2.2.3 (Cobham's function algebra) *The functions*

$$\begin{array}{l} \mathsf{S}_0, \mathsf{S}_1 : \mathbb{N} \to \mathbb{N} \textit{ (successor functions)} \\ \pi_i^n : \mathbb{N}^n \to \mathbb{N} \textit{ (projections)} \\ \# : \mathbb{N}^2 \to \mathbb{N} \textit{ (smash)} \end{array}$$

are defined by

$$\begin{array}{l} \mathsf{S}_0(x) = 2x \\ \mathsf{S}_1(x) = 2x + 1 \\ \pi_i^n(x_1, \dots, x_n) = x_i \\ x \# y = 2^{|x| \cdot |y|} \end{array}$$

The class $\mathcal{F}$ is the smallest class of functions containing the above basic functions and closed under composition and bounded recursion on notation.

Here is a definition of the function $\mathrm{conc}(x, y) = y \frown x = y \cdot 2^{|x|} + x$:

$$\begin{array}{l} \mathrm{conc}(0, y) = y \\ \mathrm{conc}(\mathsf{S}_0(x), y) = \mathsf{S}_0(\mathrm{conc}(x, y)) \\ \mathrm{conc}(\mathsf{S}_1(x), y) = \mathsf{S}_1(\mathrm{conc}(x, y)) \end{array}$$

Now, $textrmconc(x, y) \leq 2^{|x|} \cdot 2^{|y|} + 2^{|x|} \leq 2^{|x|}(2^{|y|} + 1) \leq 2^{|x|+|y|+1} \leq \mathsf{S}_0(x \# y)$. Hence, $\mathrm{conc} \in \mathcal{F}$.

More generally, notice that if $|f(x)| \leq |x|^k$ then $f(x) \leq x \# \dots \# x$ (k times) so that every polynomial time computable function is bounded by a function in $\mathcal{F}$.

We remark that we can replace bounded recursion on notation by a scheme like in Def. 1.1.1:

$$\begin{array}{l} f(0, \vec{y}) = g(\vec{y}) \\ f(x, \vec{y}) = h(x, f(\left\lfloor \frac{x}{2} \right\rfloor, \vec{y}), \vec{y}), \text{ if } x > 0 \\ \text{provided } f(x, \vec{y}) \leq k(x, \vec{y}) \end{array}$$

if we add to the basic functions a conditional cond : $\mathbb{N}^3 \to \mathbb{N}$ defined by

$$\begin{aligned}&\mathrm{cond}(0, y, z) = y\\&\mathrm{cond}(x+1, y, z) = z\end{aligned}$$

and functions for quotient and remainder of division by two: $\left\lfloor \frac{x}{2} \right\rfloor$, parity$(x)$.

Furthermore, we can "algebraicise" bounded recursion on notation, i.e., get rid of the proof obligation $f(x, \vec{y}) \leq k(x, \vec{y})$ by using the following scheme:

$$\begin{aligned}&f(x, \vec{y}) = g(\vec{y})\\&f(x, \vec{y}) = \min(h(x, f(\left\lfloor \tfrac{x}{2} \right\rfloor, \vec{y}), \vec{y})\,,\ k(x, \vec{x})), \text{ if } x > 0\end{aligned}$$

Theorem 2.2.4 (Cobham) *A function $f : \mathbb{N}^n \to \mathbb{N}$ is polynomial time computable iff $f \in \mathcal{F}$.*

Proof. The direction $\mathcal{F} \subseteq \mathit{PTIME}$ goes by induction on the definition of f. The basic functions are clearly in *PTIME* and the class *PTIME* is closed under composition. It remains to verify closure of *PTIME* under the scheme of bounded recursion on notation.

Let $\mathrm{bit}(i, x)$ be the ith most significant bit of x, i.e. formally, $\mathrm{bit}(i, x) = \lfloor \frac{x}{2^{|x|-i}} \rfloor \bmod 2$.

Suppose that g, h_0, h_1, k are polynomial time computable witnessed by polynomials p, q_0, q_1, r. The following algorithm obviously computes the function f defined from g, h_0, h_1, k by bounded recursion on notation.

```
read $x, \vec{y}$
$f := g(\vec{y}); x' := 0$
for $i = 1$ to $|x|$ do begin
if $\mathrm{bit}(i, x) = 0$
   then $f := h_0(x', f, \vec{y}); x' := \mathsf{S}_0(x')$
   else $f := h_1(x', f, \vec{y}); x' := \mathsf{S}_1(x')$
end
write $f$
```

The result now follows using the invariant $f = f(x', \vec{y})$ and $x' = \lfloor x/2^{|x|-i} \rfloor$.

It remains to estimate the running time. The loop is executed $|x|$-times and the runtime of each round can be bounded by

$$q(|x|, r(|x|, |\vec{y}|), |\vec{y}|) + l(|x|, r(|\vec{y}|), |\vec{y}|)$$

where $q = \max(q_0, q_1)$ and l is a linear term accounting for shuffling around intermediate results. Notice that we use in an essential way that all the intermediate results f are bounded by $k(x, \vec{y})$.

The proof of the converse direction $\mathit{PTIME} \subseteq \mathcal{F}$ is based on the representation of *PTIME*-functions using $\mathsf{init}, \mathsf{step}, \mathsf{out}$. We first have to show that these functions are in $\mathcal{F}$. Intuitively, this follows from the fact that they basically involve case distinctions and bit manipulations; for the (technical details) we refer to [7]. Next, recall that for every polynomial p there exists a function k built up from # and composition such that

$$|k(\vec{x})| \geq p(|\vec{x}|)$$

For example $|x\# \dots \#x| \geq |x|^n$.

Therefore, if $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is in *PTIME* by program e then we can find $k \in \mathcal{F}$ such that

$$\begin{array}{l} f(\vec{x}) = \mathrm{out}(F(|k(\vec{x})|, \vec{x})) \\ F(i, \vec{x}) \leq k(\vec{x}) \end{array}$$

where

$$F(i, \vec{x}) = \mathrm{step}^i(\mathrm{init}(e, \vec{x}))$$

The result follows since $G(z, \vec{x}) = F(|z|, \vec{x})$ admits a definition by bounded recursion on notation from $\mathrm{init}, \mathrm{step}$, and k. □

We have thus obtained a recursion-theoretic characterisation of the polynomial time computation. The disadvantage of $\mathcal{F}$ is that resource bounds are built into the definition by way of the bounding function k and that functions of arbitrary polynomial growth rate are "hardwired" by way of the basic #-function. Indeed, if we add exponentiation as basic function to $\mathcal{F}$ then we obtain the Kalmar elementary functions. So, one could say that the scheme of bounded recursion on notation is not "intrinsically polynomial time" and does not really provide any insight into the nature of feasible computation.

2.3 Safe recursion

Let us now study Bellantoni and Cook's system of safe recursion in some more detail. As said in the Introduction the main idea behind safe recursion is to forbid recursion on results of recursive calls.

This is achieved by dividing the variables of a function f into two zones separated by a semicolon: $f(\vec{x}; \vec{y})$. The $\vec{x}$-variables are called *normal*; the $\vec{y}$-variables are called *safe*. Both range over natural numbers as before. The intended invariant is that f recurses on its normal variables and applies only basic functions to its safe arguments.

Formally, one should think of such functions as triples (m, n, f) where $m, n \in \mathbb{N}$ and $f : \mathbb{N}^m \times \mathbb{N}^n \rightarrow \mathbb{N}$.

In order that this invariant be maintained we must never substitute a term depending on a safe variable for a normal variable. Formally, this is done by restricting the scheme of composition

Definition 2.3.1 *Let $f(\vec{x}; \vec{y})$, $g_i(\vec{u};)$, and $h_j(\vec{u}; \vec{v})$ be functions of appropriate arity. We say that $k(\vec{u}; \vec{v})$ is defined from $f, \vec{g}, \vec{h}$ by* safe composition *if*

$$k(\vec{u}; \vec{v}) = f(\vec{g}(\vec{u};); \vec{h}(\vec{u}; \vec{v}))$$

Now we can restrict recursion on notation in such a way that results of recursive calls may be used via safe variables only.

Definition 2.3.2 *A function $f(x, \vec{y}; \vec{z})$ is defined from $g(\vec{x}; \vec{y})$ and $h(x, \vec{y}; u, \vec{z})$ by* safe recursion on notation *if*

$$\begin{aligned}
&f(0, \vec{y}; \vec{z}) = g(\vec{y}; \vec{z})\\
&f(x, \vec{y}; \vec{z}) = h(x, \vec{y}; f(\left\lfloor \tfrac{x}{2} \right\rfloor, \vec{y}; \vec{z}), \vec{z}),\ \textit{if}\ x > 0
\end{aligned}$$

Definition 2.3.3 (Bellantoni-Cook) *The class $\mathcal{B}$ is the least set of functions (or triples (m, n, f) for pedants) closed under safe composition and safe recursion on notation and containing the following basic functions:*

- *projections (of both safe and normal variables),*
- *the constant* 0*,*
- *the successor functions* $\mathsf{S}_0(; y) = 2y$, $\mathsf{S}_1(; y) = 2y + 1$*,*
- *division by two* $\mathrm{div}(; y) = \left\lfloor \frac{y}{2} \right\rfloor$*,*
- *the parity function* $\mathrm{parity}(; y) = y \pmod 2$*,*
- *the conditional* $\mathrm{cond}(; x, y, z)$ *defined by*

$$\begin{aligned}
&\mathrm{cond}(; 0, y, z) = y\\
&\mathrm{cond}(; x + 1, y, z) = z
\end{aligned}$$

all in safe arguments as indicated.

Here is a definition of the function $\mathrm{conc}(x; y) = 2^{|x|} \cdot y + x$ with x normal and y safe as indicated:

$$\begin{aligned}
&\mathrm{conc}(0; y) = y\\
&\mathrm{conc}(x; y) = \mathrm{cond}(; \mathrm{parity}(; x),\\
&\quad \mathsf{S}_0(; \mathrm{conc}(\left\lfloor \tfrac{x}{2} \right\rfloor; y))\\
&\quad \mathsf{S}_1(; \mathrm{conc}(\left\lfloor \tfrac{x}{2} \right\rfloor; y)))
\end{aligned}$$

We can now form $\mathrm{sq}(x;) = \mathrm{conc}(x; x)$, but the definition of exp

$$\begin{aligned}
&\exp(0;) = 1\\
&\exp(x;) = \mathrm{sq}(\exp(\left\lfloor \tfrac{x}{2} \right\rfloor;);)
\end{aligned}$$

is illegal because the result of the recursive call $\exp(\left\lfloor \frac{x}{2} \right\rfloor;)$ is placed into a normal position rather than a safe one as prescribed. However, finite iterations of sq such as $\mathrm{sq}(\mathrm{sq}(\mathrm{sq}(x;););)$ are allowed by safe composition.

Since the second argument of conc is safe, we can use it to further process the result of a recursive call:

$$\begin{aligned} f(0;) &= \\ f(x, y;) &= \mathrm{conc}(y; f(x, y;)) \end{aligned}$$

This function has a similar growth rate as the smash function #.

Similarly, we can define the smash function itself using an auxiliary function $\mathrm{pad}(x; y) = 2^{|x|} \cdot y$.

The following soundness theorem for $\mathcal{B}$ contains its proof in its statement.

Theorem 2.3.4 *If $f(\vec{x}; \vec{y})$ is definable in $\mathcal{B}$ then f is in PTIME and, moreover,*

$$|f(\vec{x}; \vec{y})| \le p(|\vec{x}|) + \max(|\vec{y}|)$$

for some polynomial p.

Proof. Direct induction over derivations. □

The converse also holds, i.e., all *PTIME*-functions admit a definition in $\mathcal{B}$. We refer to [7] for the (surprisingly technical) proof. See, however, Section 3.2.3 where we prove this result for the system with higher result types.

2.4 Tiered recursion

Leivant and Marion [25, 27] have proposed another resource-free characterisation of polynomial time based on a countable number of copies of the natural numbers called *tiers*. A term of tier k may be substituted for a variable of tier l provided that $k \ge l$.

In a recursion on notation we require that the (result) tier of the function to be defined be lower than the tier of the variable recursed on.

The restriction on recursions does not apply if $f(x)$ is not actually used in the course of the computation of $f(\mathsf{S}_0(x))$ or $f(\mathsf{S}_1(x))$, in other words if recursion on notation is used merely for a generalised case distinction. The main result is that all polynomial time functions are definable from the functions 0, $\mathsf{S}_0(x)$, $\mathsf{S}_1(x)$, and projections available at all tiers.

The reader may wish to check that the definition of $\mathrm{sq}(x) = [x]^2$ by recursion on notation in the introduction is legal in tiered recursion when x is of tier 1 and $\mathrm{sq}(x)$ is of tier 0. It is also easy to see that the definition of $\exp(x)$ admits no annotation with tiers.

Leivant and Marion prove the main result directly using a machine model; we will give here an alternative proof by relating tiered recursion to safe recursion. The aim of this exercise is to pinpoint the difference between the two systems and in particular to falsify the somewhat common belief that the two levels *safe* and *normal* correspond exactly to the tiers. This (wrong) belief appears all the more plausible as Leivant and Marion actually show that two tiers suffice to define all polynomial time computable functions.

Let us temporarily use the letter $\mathcal{L}$ for Leivant and Marion's system.

Proposition 2.4.1 *If $f(\vec{x};\vec{y};\vec{z})$ is definable in $\mathcal{L}$ of tier k and with variables $\vec{x}$ of tier greater than k, with variables $\vec{y}$ of tier equal to k, and with variables $\vec{z}$ of tier lower than k, then $f(\vec{x};\vec{y};\vec{z})$ does not depend on the $\vec{z}$-variables and admits a definition in $\mathcal{B}$ with variables $\vec{x}$ normal and variables $\vec{y}$ safe. Moreover, this definition of f in $\mathcal{B}$ is obtained in a compositional way following the definition of f in $\mathcal{L}$.*

Proof. Direct by induction on the definition of f in $\mathcal{L}$. Flat recursion can be simulated in $\mathcal{B}$ using conditional, division by two, and parity. □

The following version of the converse is also proved directly by induction.

Proposition 2.4.2 *If $f(\vec{x};\vec{y})$ is definable in $\mathcal{B}$ then for every tier k there exists a tier $l \geq k$ such that $f(\vec{x};\vec{y})$ of tier k is definable in $\mathcal{L}$ of tier k with variables $\vec{x}$ of tier l and variables $\vec{y}$ of tier k.*

We see that safe recursion can be viewed as a high-level language for $\mathcal{L}$ or a "tier-inference system".

2.5 Higher-order functions and the typed lambda calculus

A higher-order function is a function which can take besides natural numbers also functions as arguments. A typical higher-order function is evaluation:

$$\mathrm{ev}(f, x) = f(x)$$

Another one is iteration:

$$\mathrm{it}(f, u) = f^{|u|}(0)$$

Iteration is definable schematically in $\mathcal{B}$ in the following sense.

If $f(\vec{x};\vec{y}, z)$ is definable then so is $g(\vec{x}, u;\vec{y})$ given by

$$\begin{aligned} g(\vec{x}, 0;\vec{y}) &= 0 \\ g(\vec{x}, u;\vec{y}) &= f(\vec{x};\vec{y}, g(\vec{x};\left\lfloor \tfrac{u}{2} \right\rfloor, \vec{y})) \end{aligned}$$

The typed lambda calculus has been invented to facilitate the formulation of such schemata. It uses type expressions built up from N and → to describe the "arity" of a higher-order function.

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

Terms are built up from variables and basic functions by application and functional abstraction:

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & c & \text{constant} \\ & | & (e_1\, e_2) & \text{application} \\ & | & \lambda x\colon A.e & \text{abstraction} \end{array}$$

Application can also be written as $e_1(e_2)$. Iterated application $(e\ e_1\ e_2\ \dots e_n)$ (association to the left) can also be written as $e(e_2, e_3, \dots, e_n)$ or $e(\vec{e})$. The variable x in an abstraction $\lambda x\colon A.e$ is bound. We identify terms up to renaming of bound variables, i.e., $\lambda x\colon A.xy$ is considered equal to $\lambda z\colon A.zy$, but different from $\lambda x\colon A.xu$.

If $\vec{A} = (A_1, \dots, A_n)$ then $\vec{A}{\to}B$ stands for $A_1{\to}A_2{\to}\dots{\to}A_n{\to}B$. The type $A^n{\to}B$ is defined recursively by $A^0{\to}B = B$ and $A^{n+1}{\to}B = A{\to}A^n{\to}B$ (association to the right).

A type of the form $\vec{\mathsf{N}}{\to}\mathsf{N}$ is called a *first-order* type. All the other types are *higher-order*. A term of first order type all whose variables are of base type is called first-order. Note that by iterated abstraction a first-order term can be transformed into a closed term of first-order type.

We fix an assignment of types to constants. For example, we could have $0 : \mathsf{N}$ and $\mathsf{S}_0, \mathsf{S}_1 : \mathsf{N}{\to}\mathsf{N}$ to mean that 0 is an integer constant and $\mathsf{S}_0, \mathsf{S}_1$ are unary functions on integers.

Well-formed terms are assigned types relative to a typing of the variables: a *context*. Such a context is a set $\Gamma = \{x_1\colon A_n, \dots, x_n\colon A_n\}$ of bindings $x\colon A$ such that the x_i are pairwise distinct. We write $\Gamma, x\colon A$ for $\Gamma \cup \{x\colon A\}$ if x is not mentioned in Γ. We write $\mathrm{dom}(\Gamma)$ for the set of variables bound in Γ and $\Gamma(x) = \tau$ if $x\colon \tau \in \Gamma$.

The typing judgement $\Gamma \vdash e : A$ read term e is well-typed of type A in context Γ is then inductively defined by the following rules.

$$\frac{x\colon A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{(T-VAR)}$$

$$\frac{c\colon A}{\Gamma \vdash c : A} \qquad \text{(T-CONST)}$$

$$\frac{\Gamma \vdash e_1 : A{\to}B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B} \qquad \text{(T-ARR-E)}$$

$$\frac{\Gamma, x\colon A \vdash e : B}{\Gamma \vdash \lambda x\colon A.e : A{\to}B} \qquad \text{(T-ARR-I)}$$

The following two rules are easily seen to be admissible:

$$\frac{\Gamma \vdash e : A \qquad \Gamma, \Delta \text{ well-formed}}{\Gamma, \Delta \vdash e : A} \tag{T-Weak}$$

$$\frac{\Gamma, x{:}\, A \vdash e : B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e[e'/x] : B} \tag{T-Subst}$$

In the latter rule $e[e'/x]$ denotes the capture-free substitution of e' for x in e. It is obtained by first renaming all bound variables in e so as to become different from free variables in e' and then literally replacing every occurrence of x in e by e'.

If e is closed, i.e., contains no free variables then we write $e : A$ instead of $\Gamma \vdash e : A$ or $\emptyset \vdash e : A$. The typing rules also make sense if instead of just N we have several basic types. They can also be generalised to encompass other type formers such as cartesian products, lists, etc. We shall encounter such later on.

2.5.1 Set-theoretic interpretation

The typed lambda calculus has an intended set-theoretic interpretation: To each type A we assign a set $[\![A]\!]$ by

$$\begin{aligned}&[\![\mathsf{N}]\!] = \mathbb{N}\\ &[\![A {\to} B]\!] = [\![A]\!] {\to} [\![B]\!]\end{aligned}$$

where $[\![A]\!]{\to}[\![B]\!]$ is the set of all functions from $[\![A]\!]$ to $[\![B]\!]$. An environment η for a context Γ is an assignment $\eta(x) \in [\![A]\!]$ for each binding $x{:}\, A$ in Γ. If $\Gamma \vdash e : A$ and η is an environment for Γ then the meaning $[\![e]\!]\eta$ is given by

$$\begin{aligned}&[\![x]\!]\eta = \eta(x)\\ &[\![e_1\ e_2]\!]\eta = [\![e_1]\!]\eta([\![e_2]\!]\eta)\\ &[\![\lambda x{:}\, A.e]\!]\eta(v) = [\![e]\!]\eta[x \mapsto v]\end{aligned}$$

For example, the meaning of the term $\lambda f{:}\, (\mathsf{N}{\to}\mathsf{N}){\to}\mathsf{N}.f(\lambda x{:}\, \mathsf{N}.x)$ is the function which takes a function $f : (\mathbb{N}{\to}\mathbb{N}){\to}\mathbb{N}$ and applies it to the identity function.

2.5.2 The system $\boldsymbol{PV^\omega}$

The system PV^ω [9] is a higher-order extension of Cobham's function algebra $\mathcal{F}$. It is defined as the typed lambda calculus over base type N and the following constants:

i. The constant zero: $0 : o$.

ii. The two successor functions: $s_0, s_1 : o{\to}o$.

iii. Integer division by two ("mixfix notation"): $\left\lfloor \frac{x}{2} \right\rfloor$.

iv. The (infix) functions *chop*, *pad*, and *smash* : $\dot{-}, \boxplus, \# : o{\to}o{\to}o$.

v. The ternary conditional $\mathit{Cond} : o{\to}o{\to}o{\to}o$.

vi. The bounded recursor $\mathsf{rec} : o{\to}(o{\to}o{\to}o){\to}(o{\to}o){\to}o{\to}o$

The set-theoretic meaning of these functions is given as follows.

$$[\![0]\!] = 0, \quad [\![s_0]\!](x) = 2x, \quad [\![s_1]\!](x) = 2x+1, \quad [\![\left\lfloor \tfrac{}{2} \right\rfloor]\!](x) = \lfloor x/2 \rfloor,$$

$$[\![\#]\!](x)(y) = 2^{|x|\cdot|y|}, \quad [\![\boxplus]\!](x)(y) = x \cdot 2^{|y|}, \quad [\![\dot{-}]\!](x)(y) = \lfloor x/2^{|y|} \rfloor,$$

$$[\![\mathit{Cond}]\!](x)(y)(z) = \begin{cases} y, & \text{if } x = 0 \\ z, & \text{otherwise} \end{cases}$$

Finally, if $g \in [\![\mathsf{N}]\!], h \in [\![\mathsf{N}{\to}\mathsf{N}{\to}\mathsf{N}]\!]$, and $k \in [\![\mathsf{N}{\to}\mathsf{N}]\!]$ then $f(x) = [\![\mathsf{rec}]\!](g,h,k,x)$ is the function defined by

$$\begin{aligned} &f(0) = \min(k(0), g) \\ &f(x) = \min(k(x), h(x, f(\left\lfloor \tfrac{x}{2} \right\rfloor))) \end{aligned}$$

It is easy to see by induction that whenever a function $f(x_1, \ldots, x_n)$ can be defined in $\mathcal{F}$ then there exists a term $e : \mathsf{N}^n{\to}\mathsf{N}$ such that $f(\vec{x}) = [\![e]\!](\vec{x})$.

The advantage of using PV^ω as opposed to plain $\mathcal{F}$ is that it allows for a direct definition of higher-order functions.

For example, we can define the functional $\exists : (\mathsf{N}{\to}\mathsf{N}){\to}\mathsf{N}$ which maps $f : \mathsf{N}{\to}\mathsf{N}$ and x to 0 iff $f(i) = 0$ for some $i \leq |x|$.

$$\exists = \lambda f{:}\,\mathsf{N}{\to}\mathsf{N}.\lambda x{:}\,\mathsf{N}.\mathsf{rec}(f(0), \lambda x{:}\,\mathsf{N}.\lambda e{:}\,\mathsf{N}.\mathit{Cond}(e, 0, f(|x|)))$$

The main result about PV^ω is that its first-order section is contained in *PTIME*:

Theorem 2.5.1 *If* $e : \mathsf{N}^n{\to}\mathsf{N}$ *in* PV^ω *then* $[\![e]\!] : \mathbb{N}^n{\to}\mathbb{N}$ *is a PTIME-function.*

This result is due to Cook and Urquhart [9] who introduced PV^ω in order to define a functional interpretation for Buss' systems of Bounded Arithmetic—subsystems of Peano arithmetic in which all provably total functions are *PTIME* [4].

Their proof uses a translation of *normal forms* of PV^ω terms back into $\mathcal{F}$. We will later on describe a semantic proof which does not make use of normalisation. In doing so we illustrate one of the central concepts used in the soundness proof which forms the heart of this thesis.

2.6 Background on categories

The key technique in this thesis is to establish properties the set-theoretic interpretation of some typed lambda calculus by relating it to another "non-standard" interpretation. Extensive use will be made of categories both to define the appropriate *notion of model* and to construct the particular models needed to establish the desired results. In order to make this thesis self-contained we review the required category theory from scratch starting with the most basic definitions. The reader with prior knowledge might skim the first few sections taking a closer look only from Section 2.6.4 onwards.

2.6.1 Categories

A *category* $\mathbb{C}$ is given by a collection $|\mathbb{C}|$ of *objects* and for any two objects A, B a set $\mathbb{C}(A, B)$ of *morphisms*. We may write $\mathbb{C}$ for $|\mathbb{C}|$ and $f : A \longrightarrow B$ for $f : \mathbb{C}(A, B)$. For each object A there is an *identity morphism* $\mathrm{id}_A : A \longrightarrow A$. We may also use the notations id or A for id_A. For any three objects A, B, C there is a *composition* $\mathrm{comp}_{A,B,C} : \mathbb{C}(B, C) \times \mathbb{C}(A, B) \longrightarrow \mathbb{C}(A, C)$. If $f : B \longrightarrow C$ and $g : A \longrightarrow B$ then we write $f \circ g$ for $\mathrm{comp}_{A,B,C}(f, g)$. Composition and identities are related by the following equations:

$$\begin{aligned} f \circ \mathrm{id} = f = \mathrm{id} \circ f \\ f \circ (g \circ h) = (f \circ g) \circ h \end{aligned}$$

A morphism $f : A \longrightarrow B$ is an *isomorphism* if there exists $g : B \longrightarrow A$ such that $g \circ f = \mathrm{id}$ and $f \circ g = \mathrm{id}$. Two objects A, B are isomorphic, written $A \cong B$ if there exists an isomorphism $f : A \longrightarrow B$.

A morphism $f : A \longrightarrow B$ is a monomorphism if $f \circ u = f \circ v$ implies $u = v$ for all $u, v : X \longrightarrow A$.

A category $\mathbb{C}$ is called *small* if its collection of objects $|C|$ forms a set.

2.6.1.1 Examples

The category **Sets** has the collection of sets as objects; a morphism from set X to Y is a function from X to Y.

A partially ordered set P can be viewed as a category with the elements of P as objects and $P(x, y) = \{\star\}$ if $x \leq y$ and $P(x, y) = \emptyset$, otherwise.

A typed lambda calculus gives rise to a category $\mathbb{S}$ with its typing contexts as objects in the following way. A *substitution* from context Γ to context Δ is a function σ assigning to each $x \in \mathrm{dom}(\Delta)$ a term $\sigma(x)$ such that $\Gamma \vdash \sigma(x) : \Delta(x)$. The application $e[\sigma]$ of a substitution σ to a term e with free variables among the domain of σ is defined as the homomorphic extension of $x[\sigma] = \sigma(x)$ with the understanding that the bound variables in e are chosen different from the free variables in σ.

We have the derived rule

$$\frac{\Delta \vdash e : A \qquad \sigma : \Gamma \longrightarrow \Delta}{\Gamma \longrightarrow e[\sigma] : A} \tag{T-Subst'}$$

The identity substitution is defined by $\mathrm{id}_\Gamma(x) = x$ for $x \in \mathrm{dom}(\Gamma)$. If $\sigma : \Gamma \longrightarrow \Delta$ and $\tau : \Delta \longrightarrow \Theta$ then $\tau \circ \sigma : \Gamma \longrightarrow \Theta$ is defined by $(\tau \circ \sigma)(x) = \tau(x)[\sigma]$. Now, $x[\tau \circ \sigma] = \tau(x)[\sigma] = x[\tau][\sigma]$ by definition, hence $e[\tau \circ \sigma] = e[\tau][\sigma]$ for all terms e by homomorphic extension. Similarly, $e[\mathrm{id}] = e$ for all e. From this, it follows that contexts and substitutions form indeed a category and in particular, that composition of substitutions is associative.

We emphasise that associativity holds with respect to definitional equality of terms, i.e., up to renaming of bound variables, and that no further equations such as $(\lambda x{:}\, A.e)e' = e[e'/x]$ are needed. We also notice that the set of terms of type A in context Γ is in 1-1 correspondence with substitutions from Γ to the context $\{x{:}\, A\}$ where x is an arbitrary variable.

The category $\mathbb{P}$ of *PTIME*-functions is defined as follows. An object of $\mathbb{P}$ is a nonnegative integer (thought of as an arity); a morphism from m to n is a *PTIME*-function from $\mathbb{N}^m$ to $\mathbb{N}^n$. Composition in $\mathbb{P}$ is ordinary composition of functions.

All these categories except **Sets** are small.

2.6.2 Terminal object and cartesian products

An object $\top$ in a category $\mathbb{C}$ is *terminal* if for every $X \in \mathbb{C}$ there is a unique morphism $\langle\rangle_X : X \longrightarrow \top$ called the *terminal projection*. Any two terminal objects are isomorphic and so one often speaks of the terminal object. The one element set $\{\langle\rangle\}$ is a terminal object in **Sets**. The empty context is the terminal object in the category of substitutions. Finally, 0 is a terminal object in the category $\mathbb{P}$ of *PTIME*-functions.

A cartesian product of two objects $A, B \in \mathbb{C}$ is an object $A \times B$ and morphisms $\pi : A \times B \longrightarrow A$ and $\pi' : A \times B \longrightarrow B$ such that for any pair of morphisms $f : C \longrightarrow A$ and $g : C \longrightarrow B$ there is a unique morphism $\langle f, g\rangle : C \longrightarrow A \times B$ satisfying $\pi \circ \langle f, g\rangle = f$ and $\pi' \circ \langle f, g\rangle = g$. More generally, if I is a set and $(A_i)_{i\in I}$ is an I-indexed family of objects of $\mathbb{C}$ then a cartesian product of the family is given by a object $\prod_{i\in I} A_i$ and a family of morphisms $\pi_i : \prod_i A_i \longrightarrow A_i$ such that for every family $f_i : C \longrightarrow A_i$ there is a unique map $\langle f_i | i\in I\rangle : C \longrightarrow \prod_{i\in I} A_i$ such that $\pi_i \circ \langle f_i | i\in I\rangle = f_i$.

Notice that uniqueness of $\langle f_i | i\in I\rangle$ implies the following equations:

$$\begin{aligned}&\langle f_i | i\in I\rangle \circ h = \langle f_i \circ h | i\in I\rangle\\ &\langle \pi_i | i\in I\rangle = \mathrm{id}\end{aligned}$$

which in turn imply uniqueness.

Like a terminal object, a cartesian product of two objects is unique up to isomorphism so that existence of cartesian products is a property of a category rather than extra structure.

In **Sets** cartesian products always exist and are given as sets of pairs in the case of binary cartesian products and as the set of choice functions in the more general case. In the category of substitutions, binary products are given by $\Gamma \times \Delta = \Gamma \cup \Delta'$ where Δ' is obtained from Δ by renaming the variables in $\mathrm{dom}(\Delta)$ so as to become different from those in $\mathrm{dom}(\Gamma)$. In $\mathbb{P}$ binary cartesian products are given by addition of natural numbers.

The categories $\mathbb{S}$ and $\mathbb{P}$ also have indexed cartesian products for finite index set. In general, a category which has binary cartesian products also has cartesian products of finite families.

2.6.3 Functors and natural transformations

A *functor* $F : \mathbb{C} \longrightarrow \mathbb{D}$ from category $\mathbb{C}$ to category $\mathbb{D}$ is given by a function $F : |\mathbb{C}| \longrightarrow |\mathbb{D}|$ and for any two objects A, B a function

$$F : \mathbb{C}(A, B) \longrightarrow \mathbb{D}(F(A), F(B))$$

such that the following equations called *functor laws* or *functoriality* are satisfied whenever they make sense.

$$\begin{aligned}&F(\mathrm{id}) = \mathrm{id}\\&F(f \circ g) = F(f) \circ F(g)\end{aligned}$$

Functors can be composed in the obvious way and for every category we have an identity functor.

2.6.3.1 Opposite and product category

The *opposite category* $\mathbb{C}^{\mathrm{op}}$ of a category $\mathbb{C}$ is given by $|\mathbb{C}^{op}| = |\mathbb{C}|$ and $\mathbb{C}^{op}(X, Y) = \mathbb{C}(Y, X)$. The product category $\mathbb{C}\times\mathbb{D}$ is given by $|\mathbb{C}\times\mathbb{D}| = |\mathbb{C}|\times|\mathbb{D}|$ and $(\mathbb{C}\times\mathbb{D})((X_0, X_1), (Y_0, Y_1)) = \mathbb{C}(X_0, Y_0) \times \mathbb{D}(X_1, Y_1)$.

A functor $F : \mathbb{C}^{\mathrm{op}} \longrightarrow \mathbb{D}$ is called a *contravariant functor* from $\mathbb{C}$ to $\mathbb{D}$. More explicitly, its morphism part sends a morphism $f \in \mathbb{C}(X, Y)$ to a morphism $F(f) \in \mathbb{D}(F(Y), F(X))$. A functor $F : \mathbb{C}\times\mathbb{C} \longrightarrow \mathbb{D}$ is called a *bifunctor* from $\mathbb{C}$ to $\mathbb{D}$. The purpose of opposite and product category is to subsume ordinary, contravariant, and bifunctors under one single notion.

2.6.3.2 Natural transformation

Let $F, G : \mathbb{C} \longrightarrow \mathbb{D}$ be functors from category $\mathbb{C}$ to category $\mathbb{D}$. A *natural transformation* α from F to G written $\alpha : F \longrightarrow G$ is a $|\mathbb{C}|$-indexed family of morphisms $\alpha_X \in \mathbb{D}(FX, GX)$ such that the following equation called *naturality* is satisfied for all $u \in \mathbb{C}(X, Y)$.

$$\alpha_Y \circ Fu = Gu \circ \alpha_X$$

For every functor $F : \mathbb{C} \longrightarrow \mathbb{D}$ there is an identity natural transformation $\mathrm{id} : F \longrightarrow F$ given by $\mathrm{id}_X = \mathrm{id}_{FX}$ and two natural transformations $\alpha : F \longrightarrow G$ and $\beta : G \longrightarrow H$ can be composed by $(\beta \circ \alpha)_X = \beta_X \circ \alpha_X$. If the category $\mathbb{C}$ is small the natural transformations from $\mathbb{C}$ to $\mathbb{D}$ form a set so that we obtain a category $\mathbb{D}^{\mathbb{C}}$: the category of functors from $\mathbb{C}$ to $\mathbb{D}$ with natural transformations as morphisms.

A *natural isomorphism* is a natural transformation $\alpha : F \longrightarrow G$ such that every morphism α_X is an isomorphism. It is easy to see that in this case the family of inverses $(\alpha_X)^{-1}$ forms itself a natural transformation from G to F so that a natural isomorphism can equivalently be defined as an isomorphism in a functor category.

A functor F is *full*, resp. *faithful* if for each X, Y the function $F : \mathbb{C}(X, Y) \rightarrow \mathbb{D}(FX, FY)$ is surjective, resp. injective.

2.6.3.3 Examples

The functor $\Gamma : \mathbb{P} \longrightarrow \textbf{Sets}$ defined by $\Gamma(n) = \mathbb{N}^n$ and $\Gamma(u) = u$ is faithful, but not full.

The set-theoretic interpretation of a typed lambda calculus defines a functor $\mathcal{I} : \mathbb{S} \longrightarrow \textbf{Sets}$ where $\mathcal{I}(\Gamma)$ is the set of environments for Γ and for substitution $\sigma : \Gamma \longrightarrow \Delta$ we have

$$\mathcal{I}(\sigma)(\eta)(x) = [\![\sigma(x)]\!]\eta$$

The functor $\mathcal{I}$ is neither full nor faithful.

2.6.4 Canonical isomorphisms and object equality

This functor preserves cartesian products and terminal objects in the sense that $\mathcal{I}(\Gamma \times \Delta)$ and $\mathcal{I}(\Gamma) \times \mathcal{I}(\Delta)$ are isomorphic sets. An element of the former is a function defined on the disjoint union of dom(Γ) and dom(Δ); an element of the latter is a pair of a function on dom(Γ) and a function on dom(Δ). Furthermore, the isomorphism between the two is canonical in the sense that the following two diagrams, where i denotes the isomorphism, commute whenever they make sense.

$$\begin{tikzcd}
\mathcal{I}(\Gamma) & \mathcal{I}(\Gamma \times \Delta) \arrow[l, "\mathcal{I}(\pi)"'] \arrow[r, "\mathcal{I}(\pi')"] \arrow[d, "i"'] & \mathcal{I}(\Delta) \\
& \mathcal{I}(\Gamma) \times \mathcal{I}(\Delta) \arrow[lu, "\pi"] \arrow[ru, "\pi'"'] &
\end{tikzcd}$$

It greatly simplifies notation and reasoning if one treats such a canonical isomorphism as an identity, i.e., assumes that $\mathcal{I}(\Gamma \times \Delta)$ and $\mathcal{I}(\Gamma) \times \mathcal{I}(\Delta)$ are equal. A technical development based on this simplification should be understood as an abbreviation for a more detailed development in which the isomorphisms are filled in. This filling in of isomorphisms is possible whenever the assumed equality is used in a "reasonable way", which intuitively means that identity of objects is used only insofar as to make certain morphisms composable.

In the particular example at hand canonicity of the isomorphism means that when making the simplifying identification between $\mathcal{I}(\Gamma \times \Delta)$ and $\mathcal{I}(\Gamma) \times \mathcal{I}(\Delta)$ we can at the same time identify $\mathcal{I}(\pi)$ and π as well as $\mathcal{I}(\pi')$ and π', which now formally have equal source and target.

We reiterate the point that treating an isomorphism as an identity does not mean that the to isomorphic objects get actually identified, but that canonical isomorphisms do not show up in the notation.

There are ways by which this convention and its soundness can be formalised and established, for details see [23, 22].

2.6.5 Subcategories

A category $\mathbb{C}$ is a subcategory of $\mathbb{D}$ if $|\mathbb{C}| \subseteq |\mathbb{D}|$ and $\mathbb{C}(X,Y) \subseteq \mathbb{D}(X,Y)$ and composition and identities are the same in both categories. The category $\mathbb{C}$ is a full subcategory if $\mathbb{C}(X,Y) = \mathbb{D}(X,Y)$. If $F : \mathbb{C} \longrightarrow \mathbb{D}$ is a faithful functor then we can identify $\mathbb{C}$ with the subcategory of $\mathbb{D}$ consisting of the objects of the form FX and of morphisms of the form Ff. If F is full and faithful then the thus obtained subcategory is full. Again, such identification should be understood as a means for simplification of notation and reasoning not necessarily as an actual identification.

2.6.6 Global elements

Let $\mathbb{C}$ be a category with terminal object. A *global element* of object X is a morphism $u : \top \longrightarrow X$. We also use the notation $u : X$ to mean that u is a global element of X. The set of global elements of X is denoted $\mathcal{G}(X)$. This assignment extends to a functor $\mathcal{G} : \mathbb{C} \longrightarrow \mathbf{Sets}$ by $\mathcal{G}(f)(u) = f \circ u$.

We may use the notation $f(u)$ instead of $\mathcal{G}(f)(u)$ or $f \circ u$ if $f : A \longrightarrow B$ and $u : A$ thus emphasising the view of morphisms as functions on global elements.

A category is called *well-pointed* if this functor is faithful, i.e., a morphism is uniquely determined by its functional action on global elements. The category **Sets** is obviously well-pointed and so are categories of algebraic structures. Syntactical categories like $\mathbb{S}$ are in general not well-pointed and neither are most functor categories.

If a category $\mathbb{C}$ has terminal objects and cartesian products then $\mathcal{G}(\top) \cong \{\langle\rangle\}$ and $\mathcal{G}(A \times B) \cong \mathcal{G}(A) \times \mathcal{G}(B)$ and these isomorphisms are again canonical allowing us to view them as identities. So, a global element of a cartesian product can be seen as a pair of global elements.

2.6.7 Function spaces

2.6.7.1 Cartesian product as a functor

Let $\mathbb{C}$ be a category and $A \in \mathbb{C}$ be such that for every $X \in \mathbb{C}$ the product $X \times A$ exists. In this case $X \mapsto X \times A$ defines a functor $- \times A$ from $\mathbb{C}$ to $\mathbb{C}$ with morphism part given by $f \times A = \langle f \circ \pi, \pi' \rangle$. Similarly, we have a functor $A \times - : \mathbb{C} \to \mathbb{C}$. If all binary cartesian products exists then $\times$ defines a bifunctor on $\mathbb{C}$ given on objects by $(A, B) \mapsto A \times B$ and on morphisms by $(f, g) \mapsto \langle f \circ \pi, g \circ \pi' \rangle$.

Now let again A be such that all products $X \times A$ exist. If B is another object then the function space or exponential of B by A is given by an object B^A (also written $A \to B$), a morphism $\mathrm{ev} : B^A \times A \longrightarrow B$—the *evaluation map*—and for every morphism $f : X \times A \longrightarrow B$ a unique morphism $\mathrm{curry}(f) : X \longrightarrow B^A$ called the *currying* or *exponential transpose* of f such that $\mathrm{ev} \circ (\mathrm{curry}(f) \times A) = f$.

Instead of uniqueness we can also postulate the following additional equations which in turn are consequences of uniqueness:

$$\begin{array}{l} \mathrm{curry}(f) \circ h = \mathrm{curry}(f \circ (h \times A)) \\ \mathrm{curry}(\mathrm{ev}) = \mathrm{id} \end{array}$$

Currying and evaluation establish a natural bijection

$$\mathbb{C}(X \times A, B) \cong \mathbb{C}(X, B^A)$$

2.6.7.2 Examples

In **Sets** the function space B^A is the set of a all functions from A to B.

If we factor the terms of a typed lambda calculus by $\beta\eta$-equality, i.e. the congruence generated by the following two equations

$$\begin{array}{l} (\lambda x\colon A.e)e' = e[e'/x] \\ (\lambda x\colon A.ex) = e, \text{ if } x \text{ is not free in } e \end{array}$$

then a context $f : A \rightarrow B$ is the exponential $(\{y\colon B\})^{\{x:A\}}$ in $\mathbb{S}$ with evaluation map given by $\mathrm{ev}(y) = fx$ and currying given by abstraction.

A category is called *cartesian closed* if it has a terminal object, all binary cartesian products and all function spaces. The category **Sets** is cartesian closed and so is $\mathbb{S}$ but only in a slightly contrived way.

2.6.7.3 Global elements and function spaces

Applying $\mathcal{G}$ to the evaluation map $\mathrm{ev} : B^A \times A \longrightarrow B$ gives rise to a function

$$\mathcal{G}(\mathrm{ev}) : \mathcal{G}(B^A) \times \mathcal{G}(A) \longrightarrow \mathcal{G}(B)$$

We will abbreviate $\mathcal{G}(\mathrm{ev})(u, a)$ by $u(a)$ thus viewing a global element of a function space as an actual function.

We also note that in view of $\mathbb{C}(\top, A \rightarrow B) \cong \mathbb{C}(\top \times A, B) \cong \mathbb{C}(A, B)$ global elements of function space $A \rightarrow B$ are in 1-1 correspondence with morphisms from A to B.

2.6.8 Interpretation of the typed lambda calculus

Let $\mathbb{C}$ be a category with terminal object and a distinguished subcollection $\mathbf{T} \subseteq \mathbb{C}$ of objects called *type objects* such that all cartesian products of the form $X \times A$ for $A \in \mathbf{T}$ and all exponentials of the form B^A for $A, B \in \mathbf{T}$ exist. Then all products of finite families of type objects exist as well via the encoding

$$\prod_{i \in \{i_1, \ldots, i_n\}} A_i = (\ldots(\top \times A_{i_1}) \times \ldots) \times A_{i_n}$$

Notice here that we always have $\top \times X \cong X$.

Now suppose that we are given a typed lambda calculus with some base types and constants with associated types. For every base type B we assume a type object $[\![B]\!]$. This extends to an assignment of type objects to all types by $[\![A_1 \to A_2]\!] = [\![A_1]\!] \to [\![A_2]\!]$. Furthermore, for every constant c of type A we assume a global element $[\![c]\!] : [\![A]\!]$.

A context Γ now gets interpreted as the $\mathrm{dom}(\Gamma)$-indexed cartesian product of the family $([\![\Gamma(x)]\!])_{x \in \mathrm{dom}(\Gamma)}$. A term $\Gamma \vdash e : A$ gets interpreted as a morphism $[\![\Gamma \vdash e : A]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!]$. The definition of this interpretation is by induction on typing derivations; the defining clauses are as follows.

$$\begin{array}{lcl} [\![\Gamma \vdash x : \Gamma(x)]\!] & = & \pi_x \\ [\![\Gamma \vdash c : A]\!] & = & [\![c]\!] \circ \langle\rangle_{[\![\Gamma]\!]} \\ [\![\Gamma \vdash e_1 e_2 : B]\!] & = & \mathrm{ev} \circ \langle [\![\Gamma \vdash e_1 : A \to B]\!], [\![\Gamma \vdash e_2 : A]\!] \rangle \\ [\![\Gamma \vdash \lambda x{:}\, A.e : A \to B]\!] & = & \mathrm{curry}([\![\Gamma, x{:}\, A \vdash e : B]\!]) \end{array}$$

In the last clause we have identified the $\mathrm{dom}(\Gamma) \cup \{x\}$-indexed product $[\![\Gamma, x{:}\, A]\!]$ with the binary product $[\![\Gamma]\!] \times [\![A]\!]$. This is legitimate as these are canonically isomorphic in the sense of Section 2.6.4. If we wanted to avoid a treatment of canonical isomorphism as identity we would have to rewrite the clause into something like

$$[\![\Gamma \vdash \lambda x{:}\, A.e : A \to B]\!] = \mathrm{curry}(i \circ [\![\Gamma, x{:}\, A \vdash e : B]\!])$$

where $i : [\![\Gamma, x{:}\, A]\!] \longrightarrow [\![\Gamma]\!] \times [\![A]\!]$ is the isomorphism defined by

$$i = \langle \langle \pi_y | y \in \mathrm{dom}(\Gamma) \rangle, \pi_x \rangle$$

One can now show that this interpretation is sound for $\beta\eta$-equality in the sense that if $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ are $\beta\eta$-equal then $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$.

Example. The set-theoretic interpretation of the typed lambda calculus forms an instance of this generic interpretation. If we use the category-theoretic framework then the interpretation of a term will be a function from $[\![\Gamma]\!]$ to $[\![A]\!]$. However, $[\![\Gamma]\!]$ is isomorphic to the set of environments on Γ so the "typing" agrees with the previously defined set-theoretic interpretation.

2.6.8.1 Global elements and category-theoretic semantics

The identification between $\mathcal{G}(A \times B)$ and $\mathcal{G}(A) \times \mathcal{G}(B)$ generalises to indexed cartesian products thus allowing us to view a global element of $[\![\Gamma]\!]$ as a function η defined on $\mathrm{dom}(\Gamma)$ and such that $\eta(x) : [\![\Gamma(x)]\!]$.

If $\Gamma \vdash e : A$ then

$$\mathcal{G}([\![\Gamma \vdash e : A]\!]) : \mathcal{G}([\![\Gamma]\!]) \longrightarrow \mathcal{G}([\![A]\!])$$

is a function sending such environments to global elements of A.

If the category $\mathbb{C}$ is well-pointed then $[\![\Gamma \vdash e : A]\!]$ is uniquely determined by the function $\mathcal{G}([\![\Gamma \vdash e : A]\!])$ which maps environments (elements of $\mathcal{G}([\![\Gamma]\!])$ to values in $\mathcal{G}([\![A]\!])$.

The difference to set-theoretic interpretation is that not all functions are $\mathbb{C}$-morphisms, thus, by appropriately choosing the category $\mathbb{C}$ one can maintain additional invariants.

2.6.9 Logical relations

Logical relations form a tool with which to relate two different interpretations of a typed lambda calculus typically with the aim of using one interpretation in order to prove something about the other one.

The following definition is not the most general one possible, but sufficient for our purposes. Let $\mathbb{C}$, $\mathbb{D}$ be two categories with enough structure to interpret a typed lambda calculus. We denote the respective interpretations by $[\![-]\!]^{\mathbb{C}}$ and $[\![-]\!]^{\mathbb{D}}$.

In the following definition we make use of the simplifying notation for application of global elements of function spaces introduced in Section 2.6.7.3

Definition 2.6.1 (Statman, Plotkin) *A type-indexed family of relations*

$$R_A \subseteq \mathcal{G}([\![A]\!]^{\mathbb{C}}) \times \mathcal{G}([\![A]\!]^{\mathbb{D}})$$

is called a logical relation, *if for all types A and B and $u : [\![A{\to}B]\!]^{\mathbb{C}}$ and $v : [\![A{\to}B]\!]^{\mathbb{D}}$ we have*

$$uR_{A\to B}v \iff \forall a\colon [\![A]\!]^{\mathbb{C}}.\forall b\colon [\![B]\!]^{\mathbb{D}}.aR_Ab \Rightarrow u(a)R_Bv(b)$$

So, a logical relation is determined by its restriction to base types. Let R be a logical relation.

For environments $\eta\colon [\![\Gamma]\!]^{\mathbb{C}}$ and $\rho\colon [\![\Gamma]\!]^{\mathbb{D}}$ we define

$$\eta R_\Gamma \rho \iff \forall x{\in}\,\mathrm{dom}(\Gamma).\eta(x)R_{[\![\Gamma]\!](x)}\rho(x)$$

Theorem 2.6.2 *Suppose that $[\![c]\!]^{\mathbb{C}}R_A[\![c]\!]^{\mathbb{D}}$ for every constant $c : A$. Then for every term $\Gamma \vdash e : A$ and environments $\eta \in \mathcal{G}([\![\Gamma]\!]^{\mathbb{C}})$ and $\rho \in \mathcal{G}([\![\Gamma]\!]^{\mathbb{D}})$, we have*

$$\eta R_\Gamma \rho \Longrightarrow [\![\Gamma \vdash e : A]\!]^{\mathbb{C}}(\eta)\; R_A \;[\![\Gamma \vdash e : A]\!]^{\mathbb{D}}(\rho)$$

Proof. By induction on the derivation of $\Gamma \vdash e : A$. The case T-Const follows from the assumption; if $e = x$ and $\Gamma \vdash e : A$ by T-Var then $[\![\Gamma \vdash x : A]\!](\eta) = \eta(x)$ and again the conclusion is part of the assumption.

Now suppose that $e = e_1e_2$ where $\Gamma \vdash e_1 : A{\to}B$ and $\Gamma \vdash e_2 : A$. If $\eta R_\Gamma \rho$ then, inductively,

$$[\![\Gamma \vdash e_1]\!]^{\mathbb{C}}(\eta)R_{A\to B}[\![\Gamma \vdash e_1]\!]^{\mathbb{D}}(\rho)$$

and

$$[\![\Gamma \vdash e_1]\!]^{\mathbb{C}}(\eta)R_A[\![\Gamma \vdash e_2]\!]^{\mathbb{D}}(\rho)$$

Hence

$$[\![\Gamma \vdash e_1e_2]\!]^{\mathbb{C}}(\eta)R_B[\![\Gamma \vdash e_1e_2]\!]^{\mathbb{D}}(\rho)$$

by definition of $R_{A\to B}$.

Finally, suppose that $e = \lambda x\colon A.e_1$ where $\Gamma, x\colon A \vdash e_1 : B$. In order to show that

$$[\![\Gamma \vdash e : A]\!]^{\mathbb{C}}(\eta) R_{A\to B} [\![\Gamma \vdash e : A]\!]^{\mathbb{D}}(\rho)$$

we assume $u : [\![A]\!]^{\mathbb{C}}$ and $v : [\![A]\!]^{\mathbb{D}}$ with uR_Av. Pairing these with η and ρ, respectively, yields environments $\langle \eta, u\rangle : [\![\Gamma, x\colon A]\!]^{\mathbb{C}}$ and $\langle \rho, v\rangle : [\![\Gamma, x\colon A]\!]^{\mathbb{D}}$. The induction hypothesis then yields

$$[\![\Gamma, x\colon A \vdash e : B]\!]^{\mathbb{C}}(\langle \eta, u\rangle) R_B [\![\Gamma, x\colon A \vdash e : B]\!]^{\mathbb{C}}(\langle \rho, v\rangle)$$

But

$$[\![\Gamma \vdash \lambda x\colon A.e : A\to B]\!]^{\mathbb{C}}(\eta)(u) = [\![\Gamma, x\colon A \vdash e : B]\!]^{\mathbb{C}}(\langle \eta, u\rangle)$$

and similarly for $\mathbb{D}$ by the defining equation for function spaces, hence the result. □

2.6.10 Presheaves

A *presheaf* over a category $\mathbb{C}$ is a functor from $\mathbb{C}^{op}$ to **Sets**. More elementarily, this means that F consists of a $|\mathbb{C}|$-indexed family of sets $(FX)_{X\in\mathbb{C}}$ and for every morphism $f : X \longrightarrow Y$ a "reindexing function" $Ff : FY \longrightarrow FX$ such that

$$\begin{array}{l} F(\mathrm{id}_X)(x) = x \\ F(f \circ g)(x) = F(g)(F(f)(x)) \end{array}$$

We will usually write F_X for FX and $F_f(x)$ for $F(f)(x)$. If the presheaf under consideration is clear from the context we may also write $x[f]$ for $F_f(x)$. We then have $x[\mathrm{id}] = x$ and $x[f][g] = x[f \circ g]$. The attention of the reader is drawn to the formal similarity of the morphism part of a presheaf with substitution.

Notice that a natural transformation from F to G is given by a $|\mathbb{C}|$-indexed family of functions $\alpha_X : F_X \longrightarrow G_X$ such that for every $u \in \mathbb{C}(X, Y)$ and $f \in F_Y$ we have

$$(\alpha_Y(f))[u] = \alpha_X(f[u])$$

Under the abovementioned analogy with explicit substitution this equation corresponds to compatibility of a term former (here α) with substitution.

If $\mathbb{C}$ is a small category then the presheaves form a category which we denote by $\widehat{\mathbb{C}}$ rather than $\mathbf{Sets}^{\mathbb{C}^{op}}$.

For every object $X \in \mathbb{C}$ we have the *representable presheaf* $\mathcal{Y}(X)$ defined by $\mathcal{Y}(X)_Y = \mathbb{C}(Y, X)$ and $\mathcal{Y}(X)_f(u) = u \circ f$. This extends to a functor $\mathcal{Y} : \mathbb{C} \longrightarrow \widehat{\mathbb{C}}$ by $\mathcal{Y}(t)_Z(u) = t \circ u$ whenever $t : X \longrightarrow Y$ and $u \in \mathcal{Y}(X)_Z = \mathbb{C}(Z, X)$.

This functor is called *Yoneda embedding.* The *Yoneda Lemma* states that this functor is full and faithful. Indeed, we can recover t from $\mathcal{Y}(t)$ by $t = \mathcal{Y}(t)_X(\mathrm{id}_X)$ giving faithfulness, and if $\alpha : \mathcal{Y}(X) \longrightarrow \mathcal{Y}(Y)$ is a natural transformation then $\alpha = \mathcal{Y}(t)$ for $t =_{\mathrm{def}} \alpha_X(\mathrm{id}_X)$ hence fullness.

It is convenient to identify a category $\mathbb{C}$ with a subcategory of $\widehat{\mathbb{C}}$ thus writing X instead of $\mathcal{Y}(X)$.

2.6.10.1 Examples

In the category $\widehat{\mathbb{S}}$ we have for every type A the presheaf $\mathrm{Tm}(A)$ given by

$$\begin{aligned}&\mathrm{Tm}(A)_\Gamma = \{e \mid \Gamma \vdash e : A\}\\&\mathrm{Tm}(A)_\sigma(e) = e[\sigma]\end{aligned}$$

This presheaf is isomorphic to the representable presheaf $\mathcal{Y}(\{x\colon A\})$.

In $\widehat{\mathbb{P}}$ we have the presheaf N of *PTIME*-functions given by

$$\begin{aligned}&\mathsf{N}_n = \{f \mid f : \mathbb{N}^n \longrightarrow \mathbb{N} \text{ and } f \in \mathit{PTIME}\}\\&\mathsf{N}_u(f) = f \circ u\end{aligned}$$

This presheaf is isomorphic to the representable presheaf $\mathcal{Y}(1)$.

Thus, by the Yoneda Lemma the $\widehat{\mathbb{P}}$-morphisms from N to N are in 1-1 correspondence with the *PTIME*-functions.

Lemma 2.6.3 *If F is a presheaf then the set of morphisms $\widehat{\mathbb{C}}(\mathcal{Y}(X), F)$ is naturally isomorphic to the set F_X, that is F is isomorphic to the presheaf $\widehat{\mathbb{C}}(\mathcal{Y}(-), F)$.*

Proof. If $m : \mathcal{Y}(X) \longrightarrow F$ then $\hat{m} =_{\mathrm{def}} m_X(\mathrm{id}_X) \in F_X$. Conversely, if $f \in F_X$ then we define $\check{f} : \mathcal{Y}(X) \longrightarrow F$ by $\check{f}_Y(u) = f[u]$. We have $\check{\hat{m}}_Y(u) = m_X(\mathrm{id}_X)[u] = m_Y(\mathcal{Y}(X)_u(\mathrm{id}_X)) = m_Y(\mathrm{id}_X \circ u) = m_Y(u)$. Conversely, $\hat{\check{f}} = f[\mathrm{id}_X] = f$. For naturality, we have to check that for $v : X' \longrightarrow X$ we have $\check{(f[v])}_{X'}(u) = \check{f}_X(v \circ u)$ which is direct from the definitions. □

2.6.10.2 Cartesian closure of $\widehat{\mathbb{C}}$

Every presheaf category $\widehat{\mathbb{C}}$ is cartesian closed. The terminal object is given by the constant presheaf defined by $\top_X = \{\langle\rangle\}$, $\langle\rangle[u] = \langle\rangle$. If $\mathbb{C}$ has a terminal object then $\mathcal{Y}(\top) \cong \top$ and it is convenient to treat this isomorphism as an identity.

Cartesian product $F \times G$ is given pointwise by $(F \times G)_X = F_X \times G_X$ and $((F \times G)_u(f, g) = (F_u(f), G_u(g)$.

If $\mathbb{C}$ has cartesian products then $\mathcal{Y}(X \times Y) \cong \mathcal{Y}(X) \times \mathcal{Y}(Y)$ so that when viewing $\mathbb{C}$ as a subcategory of $\widehat{\mathbb{C}}$ we do not need to make a distinction as to whether we form cartesian products in $\mathbb{C}$ or in $\widehat{\mathbb{C}}$.

More generally, $\widehat{\mathbb{C}}$ has indexed products as follows. If $(F_i)_{i\in I}$ is a family of presheaves then the product $\prod_{i\in I} F_i \in \widehat{\mathbb{C}}$ is defined pointwise by

$$(\prod_{i\in I} F_i)_X = \prod_{i\in I}(F_i)_X$$

and analogously on morphisms.

If $f_i : C \longrightarrow F_i$ then

$$\langle f_i | i\in I\rangle_X = \langle (f_i)_X | i\in I\rangle$$

and

$$(\pi_i)_X = \pi_i$$

where the notation on the right-hand side refers to set-theoretic indexed cartesian products.

For the function space we start from the observation that *if* $\widehat{\mathbb{C}}$ has function spaces then we must necessarily have

$$\begin{array}{ll} & (F{\to}G)_X \\ \cong & \widehat{\mathbb{C}}(\mathcal{Y}(X), F{\to}G) \qquad\qquad \text{by Lemma 2.6.3} \\ \cong & \widehat{\mathbb{C}}(\mathcal{Y}(X) \times F, G) \end{array}$$

which suggests to define $(F{\to}G)_X$ as the set of natural transformations from $\mathcal{Y}(X) \times F$ to G. More elementarily, an element φ of $(F{\to}G)_X$ assigns to each $Y \in \mathbb{C}$, $v : Y \longrightarrow X$ and $f \in F_Y$ and element $\varphi(v, f) \in G_Y$ in such a way that if $u : Z \longrightarrow Y$ then

$$\varphi(v, f)[u] = \varphi(v \circ u, f[u])$$

If $w : X' \longrightarrow X$ we define $\varphi[w] \in (F{\to}G)'_X$ by $\varphi[w](v, f) = \varphi(w \circ v, f)$.

The evaluation map is given by

$$\mathrm{ev}_X(\varphi, f) = \varphi(\mathrm{id}_X, f)$$

If $h : L \times F \longrightarrow G$ then $\mathrm{curry}(h) : L \longrightarrow F{\to}G$ is defined by

$$\mathrm{curry}(h)_X(l)(u, f) = h_Y(l[u], f)$$

when $l \in L_X, u : Y \longrightarrow X, f \in F_Y$. The straightforward verifications are left to the reader.

We draw the attention of the reader to the similarity of function space in functor categories and implication in Kripke semantics. Indeed, Kripke semantics of intuitionistic propositional calculus can be seen as interpretation in the presheaf category $\widehat{\mathbb{W}}$ where $\mathbb{W}$ is the poset of worlds viewed as a category. Every formula φ gets assigned a presheaf $[\![\varphi]\!]$ which has the property that $[\![\varphi]\!]_w$ has at most one element. The defining clauses are $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \times [\![\psi]\!]$ and $[\![\varphi \supset \psi]\!] = [\![\varphi]\!]{\to}[\![\psi]\!]$. One may write $w \Vdash \varphi$ if $[\![\varphi]\!]_w \neq \emptyset$ and then recovers the usual rules for interpretation in Kripke models.

The following characterisation of function spaces involving representable presheaves is crucial for the application of presheaves to lambda calculi with recursion operators.

Proposition 2.6.4 *Let $U \in \mathbb{C}$ and assume that $\mathbb{C}$ has cartesian products of the form $- \times U$. Then for any presheaf $F \in \widehat{\mathbb{C}}$ the function space $U{\rightarrow}F$ $(= \mathcal{Y}(U){\rightarrow}F)$ is isomorphic to the presheaf $F_{-\times U}$ defined by $(F_{-\times U})_X = F_{X \times U}$ and $(F_{-\times U})_u = F_{u \times U}$.*

Proof.

$$\begin{array}{lll} & (U{\rightarrow}F)_X & \\ \cong & \widehat{\mathbb{C}}(X, U{\rightarrow}F) & \text{Lemma 2.6.3} \\ \cong & \widehat{\mathbb{C}}(X \times U, F) & \text{Property of function space} \\ \cong & F_{X \times U} & \text{Lemma 2.6.3} \end{array}$$

□

We can now define evaluation and currying directly on $F_{-\times U}$ thus allowing us to treat the above isomorphism as an identity. Evaluation $\mathrm{ev} : F_{-\times U} \times U \longrightarrow F$ is given by

$$\mathrm{ev}_X(f, u) = f[\langle \mathrm{id}_X, u \rangle]$$

when $f \in F_{X \times U}, u : X \longrightarrow U$. If $h : L \times U \longrightarrow F$ then $\mathrm{curry}(h) : L \longrightarrow F_{-\times U}$ is given by

$$\mathrm{curry}(h)_X(l) = h_{X \times U}(l[\pi], \pi')$$

If F is representable itself, i.e., $F = V$ for $V \in \mathbb{C}$ then we have $(U{\rightarrow}V)_X = V_{X \times U} = \mathbb{C}(X \times U, V)$. If the exponential $U{\rightarrow}V$ exists in $\mathbb{C}$ then $(U{\rightarrow}V)_X \cong \mathbb{C}(X, U{\rightarrow}V) = \mathcal{Y}(U{\rightarrow}V)_X$ so the Yoneda embedding preserves existing exponentials and therefore, when writing $U{\rightarrow}V$ it does not make a difference whether we understand the function space in $\mathbb{C}$ or in $\widehat{\mathbb{C}}$.

The functor category $\widehat{\mathbb{C}}$ therefore can be seen as a generic extension of a category $\mathbb{C}$ with all previously lacking function spaces. It is, however, not a free extension, as $\widehat{\mathbb{C}}$ extends $\mathbb{C}$ with much more structure than merely function spaces.

2.6.10.3 Examples

In $\widehat{\mathbb{S}}$ the exponential $\mathrm{Tm}(A){\rightarrow}\mathrm{Tm}(B)$ can be given by

$$(\mathrm{Tm}(A){\rightarrow}\mathrm{Tm}(B))_\Gamma = \mathrm{Tm}(B)_{\Gamma, x:A}$$

Therefore, we have a global element in $\widehat{\mathbb{S}}$

$$\mathrm{lambda} : \prod_{A,B \text{ types}} (\mathrm{Tm}(A){\rightarrow}\mathrm{Tm}(B)){\rightarrow}\mathrm{Tm}(A{\rightarrow}B)$$

which performs functional abstraction. Similarly, we have

$$\mathrm{app} : \prod_A \prod_B (\mathrm{Tm}(A{\rightarrow}B)){\rightarrow}\mathrm{Tm}(A){\rightarrow}\mathrm{Tm}(B)$$

This can be used to give a semantic proof of adequacy of representations of formal systems using *higher-order abstract syntax* used in the context of *Logical Frameworks*.

We further remark that if $\mathbb{S}$ is understood modulo $\beta\eta$-equality then $\{x\colon A{\to}B\}$ is the exponential of $\{x\colon A\}$ and $\{x\colon B\}$ in $\mathbb{S}$, so in this case $\mathrm{Tm}(A){\to}\mathrm{Tm}(B)$ is isomorphic to $\mathrm{Tm}(A{\to}B)$.

Recall that $\mathsf{N} \in \widehat{\mathbb{P}}$ was defined as the representable presheaf $\mathcal{Y}(1)$. The function space $\mathsf{N}{\to}\mathsf{N}$ (which is not representable) is given by

$$\begin{aligned}(\mathbb{N}{\to}\mathbb{N})_n &= \mathbb{N}_{n+1}\\ &= \{n+1\text{-ary } \mathit{PTIME}\text{-functions}\}\end{aligned}$$

2.6.10.4 Presheaves and global elements

Suppose that $\mathbb{C}$ has a terminal object $\top$. Then by Lemma 2.6.3 the set $\mathcal{G}(F)$ of global elements of a presheaf F is isomorphic to the set $F_\top$. More generally, the global sections functor $\mathcal{G} : \widehat{\mathbb{C}} \longrightarrow \mathbf{Sets}$ is isomorphic to the functor which applies a presheaf F to $\top$. This means that the following diagram commutes for every $m \in \widehat{\mathbb{C}}(F, G)$:

$$\begin{tikzcd}
\mathcal{G}(F) \arrow[r, "\cong"] \arrow[d, "\mathcal{G}(m)"'] & F_\top \arrow[d, "m_\top"] \\
\mathcal{G}(G) \arrow[r, "\cong"'] & G_\top
\end{tikzcd}$$

In view of this we will henceforth notationally identify $\mathcal{G}(F)$ with $F_\top$ whenever $\mathbb{C}$ has a terminal object.

This has the following useful consequence. If $f \in \widehat{\mathbb{C}}(\mathcal{Y}(X), \mathcal{Y}(Y))$ then by the Yoneda-Lemma there exists $g \in \mathbb{C}(X, Y)$ with $f = \mathcal{Y}(g)$, namely $g = f_X(\mathrm{id}_X)$. Now, $\mathcal{G}(\mathcal{Y}(X)) = \mathbb{C}(\top, X) = \mathcal{G}(X)$ hence $\mathcal{G}(\mathcal{Y}(g)) = \mathcal{G}(g)$ and so $\mathcal{G}(f) = \mathcal{G}(g)$.

2.6.10.5 Extensional presheaves

If the underlying category is well-pointed (cf. Section 2.6.6) then it is possible to define a well-pointed subcategory of $\widehat{\mathbb{C}}$ which contains all the representable presheaves and is closed under product and function space.

Definition 2.6.5 *Let $\mathbb{C}$ be a well-pointed category. A presheaf $F \in \widehat{\mathbb{C}}$ is called extensional if for each $X \in \mathbb{C}$ the function $\eta_X : F_X \longrightarrow F_\top^{\mathcal{G}(X)}$ sending $f \in F_X$ to $\mathcal{G}(X) \ni x \mapsto f[x] \in F_\top$ is injective.*

In other words, a presheaf F is extensional if for each $f_1, f_2 \in F_X$ we have $f_1 = f_2$ if and only if $f_1[x] = f_2[x]$ for all global elements $x : X$.

Lemma 2.6.6 *Let $\mathbb{C}$ be a well-pointed category.*

i. Every representable presheaf $\mathcal{Y}(X)$ *for* $X \in \mathbb{C}$ *is extensional.*

ii. Every constant presheaf is extensional, in particular $\top$ *is.*

iii. If F_1, F_2 *are extensional so are* $F_1 \times F_2$*.*

iv. If G *is extensional so is* $F \to G$*.*

Proof. Let X, Y be objects of $\mathbb{C}$. An element $f \in \mathcal{Y}(Y)_X$ is a $\mathbb{C}$-morphism from X to Y. As $\mathbb{C}$ is well-pointed such morphism is uniquely determined by its action on global elements hence $\mathcal{Y}(Y)$ is extensional. If F_1, F_2 are extensional then obviously $F_1 \times F_2$ are extensional as products are taken pointwise in $\hat{\mathbb{C}}$.

Now suppose that G is extensional and F is arbitrary and let $\mu, \mu' \in (F \to G)_X$. Assume furthermore that for each $x \in \mathcal{G}(X)$ we have $(F \to G)_x(\mu) = (F \to G)_x(\mu')$, i.e., if $f \in \mathcal{G}(F)$ then $\mu_\top(x, f) = \mu'_\top(x, f)$. We must show that $\mu = \mu'$. To see this, pick $Y \in \mathbb{C}$ and $u \in \mathbb{C}(Y, X)$ and $f \in F_Y$. Now both $\mu_Y(u, f)$ and $\mu'_Y(u, f)$ are elements of G_Y thus to show that they are equal it suffices to show that $G_y(\mu_Y(u, f)) = G_y(\mu'_Y(u, f))$ for each $y \in \mathcal{G}(Y)$ as G is extensional. By naturality we have $G_y(\mu_Y(u, f)) = \mu_\top(u \circ y, F_y(f))$. By assumption the latter term equals $\mu'_\top(u \circ y, F_y(f))$ hence the result. □

Let us write $\mathrm{Ext}(\mathbb{C})$ for the full subcategory (of $\widehat{\mathbb{C}}$) of extensional presheaves.

Lemma 2.6.7 $\mathrm{Ext}(\mathbb{C})$ *is a well-pointed category.*

Proof. Let $F, G \in \mathrm{Ext}(\mathbb{C})$ and $u, v : F \longrightarrow G$ be natural transformations such that $u(f) = v(f)$ for each $f \in \mathcal{G}(G) \cong G_\top$. We must show that $u = v$, i.e., $u_X(m) = v_X(m)$ for each $X \in \mathbb{C}$ and $m \in F_X$. Since G is extensional this would follow from $u_X(m)[g] = v_X(m)[g]$ for every $g \in \mathcal{G}(G)$. But by naturality and assumption on u, v we have

$$u_X(m)[g] = u_\top(m \circ g) = v_\top(m \circ g) = v_X(m)[g]$$

□

Remark 2.6.8 *In view of this fact extensional presheaves admit a more concrete description as sets with additional structure. Namely, an extensional presheaf can equivalently be given as a set* $|A|$ *together with a* $|\mathbb{C}|$*-indexed family of functions* $A_X \subseteq |A|^{\mathcal{G}(X)}$ *such that whenever* $u \in \mathbb{C}(Y, X)$ *and* $a \in A_X$ *then* $a \circ \mathcal{G}(u) \in A_Y$*. A morphism between two such objects* A *and* B *is a function* $f : |A| \longrightarrow |B|$ *such that for each* $a \in A_X$ *we have* $f \circ a \in B_X$*. These objects form a category equivalent to* $\mathrm{Ext}(\mathbb{C})$*. This category has also been introduced in [12] and implicitly in [39].*

We have now assembled enough technical machinery to give a semantic proof of Theorem 2.5.1

2.6.10.6 Interpretation of $\boldsymbol{PV^\omega}$ in $\widehat{\mathbb{P}}$

We interpret the base type as the representable presheaf $\mathsf{N} = \mathcal{Y}(1)$. Function types are interpreted as the corresponding function spaces in $\widehat{\mathbb{P}}$.

Next, we need to assign a global element $[\![c]\!] : [\![A]\!]$ to each constant $c\colon A$.

The first-order constants are interpreted by applying the Yoneda embedding to their set-theoretic meanings. For example, we define $[\![\mathsf{S}_0]\!] \in \hat{\mathbb{P}}(\top, O{\to}O) \cong \hat{\mathbb{P}}(O,O)$ as $\mathcal{Y}(\lambda x.2x)$. That is, $[\![s_0]\!]_n(f) = \lambda\vec{x}.2f(\vec{x})$.

For the recursor $\mathsf{rec} : A$ where

$$A = o{\to}(o{\to}o{\to}o){\to}(o{\to}o){\to}o{\to}o$$

we first note that in view of Prop. 2.6.4

$$[\![A]\!] \cong \widehat{\mathbb{C}}(\mathsf{N} \times \mathsf{N}_{\_+2} \times \mathsf{N}_{\_+1}, \mathsf{N}_{\_+1})$$

Treating this isomorphism as an identity we can define $[\![\mathsf{rec}]\!]$ directly as follows:

$$[\![\mathsf{rec}]\!]_n(g,h,k) = f$$

where $g \in \mathsf{N}_n, h \in \mathsf{N}_{n+2}, k \in \mathsf{N}_{n+1}$ are arbitrary and $f \in \mathsf{N}_{n+1}$ is defined from g,h,k by bounded recursion on notation according to Def. 2.2.1.

Naturality amounts to the fact that bounded recursion on notation commutes with substitution, i.e., if f is obtained from g,h,k by bounded recursion on notation as above and $s \in \mathbb{P}(m,n)$ then $f\circ s$ can be obtained by bounded recursion on notation from $g\circ s, h\circ(s+2), k\circ(s+1)$. Here, by $s+k \in \mathbb{P}(m+k,n+k)$ we mean the product functor $-\times k$ applied to s.

Remark 2.6.9 *Since $\mathbb{P}$ is a well-pointed category it makes sense to form the subcategory* $\mathrm{Ext}(\mathbb{P})$ *of extensional presheaves and in view of Lemma 2.6.6 the interpretation restricts to* $\mathrm{Ext}(\mathbb{P})$.

Now, if $t : \mathsf{N}{\to}\mathsf{N}$ then $[\![t]\!] : [\![\mathsf{N}]\!]{\to}[\![\mathsf{N}]\!]$ and by the Yoneda-Lemma the function $\mathcal{G}([\![t]\!]) : \mathcal{G}(\mathsf{N}) \longrightarrow \mathcal{G}(\mathsf{N})$ is a *PTIME*-function hence by the Yoneda Lemma $[\![t]\!]$ is a *PTIME*-function. It remains to show that it agrees with the set-theoretic interpretation of t. In order to show this we consider the logical relation between the set-theoretic interpretation and the interpretation in $\widehat{\mathbb{P}}$ generated by

$$xR_{\mathsf{N}}y \iff x = y$$

Notice that this definition makes sense as $\mathsf{N}_\top = \mathsf{N}_0 \cong \mathbb{N}$.

Suppose that $u \in [\![\mathsf{N}^n{\to}\mathsf{N}]\!]^{\mathbf{Sets}}$, i.e., $u : \mathbb{N}^n \longrightarrow \mathbb{N}$ and $u' : [\![\mathsf{N}^n{\to}\mathsf{N}]\!]^{\widehat{\mathbb{P}}}$, i.e., $u' \in \widehat{\mathbb{P}}(\mathsf{N}^n, \mathsf{N}) \cong \mathbb{P}(n,1)$ then by definition of a logical relation u and u' are related in $R_{\mathsf{N}^n\to\mathsf{N}}$ if they send related elements to related elements, which means in this particular case that u and u' are equal functions. Therefore, if $u \in [\![\mathsf{N}^n{\to}\mathsf{N}]\!]^{\mathbf{Sets}}$ is related to some global element of $[\![\mathsf{N}^n{\to}\mathsf{N}]\!]^{\widehat{\mathbb{P}}}$ then it must be a polynomial time computable function.

The promised result that set-theoretic meanings of first-order terms are polynomial time computable therefore is a consequence of Theorem 2.6.2 if we can show that the respective meanings of every constant are related.

For first-order constants this is immediate from the above unfolding of R at first-order types.

For the recursor assume that

$$\begin{array}{l} g\, R_{\mathsf{N}}\, g' \\ h\, R_{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}\, h' \\ k\, R_{\mathsf{N}\to\mathsf{N}}\, k' \end{array}$$

After unfolding the definitions this gives that g, g' and h, h' and k, k' are extensionally equal functions. Therefore, $[\![\mathsf{rec}]\!]^{\mathbf{Sets}}(g, h, k)$ and $[\![\mathsf{rec}]\!]^{\widehat{\mathbb{P}}}(g', h', k')$ are also equal and hence related.

2.6.10.7 Syntactic version

The content of this section is independent of the rest of this thesis.

Cook and Urquhart actually show a stronger result, namely that every first-order PV^ω term is provably equal (w.r.t. some equational theory) to a term in a first-order system PV which consists of terms for all function definitions in $\mathcal{F}$ and an equational logic based on their defining axioms. It is possible to give a semantic proof of this by replacing $\mathbb{P}$ with a category having tuples of PV-terms modulo provable equality as morphisms. The interpretation of PV^ω in $\widehat{\mathbb{PV}}$ then assigns to every first-order term of PV^ω a PV-term.

The proof then goes through in essentially the same way; however, such a syntactic category is not well-pointed (this would mean that extensionally equal function expressions would be provably equal), so we cannot restrict our interpretation to extensional presheaves.

We also note that, as before, the logical relation only establishes that the resulting PV-term is extensionally equal to the PV^ω-term to start off with. If we want to strengthen this to PV^ω-provable equality one must use a more general kind of logical relation. We omit the details as we do not consider these syntactic versions later.

2.6.11 Adjoint functors

The reader who is unfamiliar with adjoint functors and does not want to invest energy could skip the following section and ignore later references to the concept. We include it because in our opinion it substantially clarifies some of the later definitions; e.g., linear function space and comonads.

Let $F : \mathbb{C} \longrightarrow \mathbb{D}$ and $G : \mathbb{D} \longrightarrow \mathbb{C}$. We say that F is left adjoint to G, written $F \dashv G$ (alternatively G is right adjoint to F, written $G \vdash F$) if for each $X \in \mathbb{C}$ and $Y \in \mathbb{D}$ we have a bijective correspondence

$$\varphi_{X,Y} : \mathbb{C}(FX, Y) \cong \mathbb{D}(X, GY)$$

which is natural in X, Y in the sense that if $u \in \mathbb{C}(X, X')$ and $v \in \mathbb{D}(Y, Y')$ then

$$\varphi_{X',Y'}(v \circ f \circ Fu) = Gv \circ \varphi_{X,Y}(f) \circ u$$

The application of φ or φ^{-1} to a morphism f is called the *transpose* of f (along the adjunction).

We have a natural transformation $\varepsilon : F \circ G \longrightarrow \mathrm{id}$ given by $\varepsilon_X = \varphi^{-1}_{GX,X}(\mathrm{id}_{GX})$ called the *counit* of the adjunction.

Another way of defining adjunction consists of postulating the counit ε and in exchange dropping the requirement that φ be bijective. The map φ and the counit ε are then required to satisfy the equations $\varepsilon_Y \circ F\varphi_{X,Y}(u) = u$ and $\varphi_{X,FX}(\varepsilon_X) = \mathrm{id}_{GX}$. An inverse to φ can then be defined by

$$\varphi^{-1}_{X,Y}(u) = \varepsilon_Y \circ Fu$$

Similarly, we have a natural transformation $\eta : \mathrm{id} \longrightarrow G \circ F$ called the *unit* of the adjunction. It is given by $\eta_X = \varphi_{X,FX}(\mathrm{id}_{FX})$. There are alternative presentations of adjunctions taking φ^{-1} and η or η and ε as primitives.

2.6.11.1 Examples

Let $\mathbb{C}$ be a category with terminal object. The constant presheaf functor $\nabla : \mathbf{Sets} \longrightarrow \widehat{\mathbb{C}}$ given by $\nabla S_X = S$ and $s[u] = s$ is left adjoint to the global sections functor $\mathcal{G} : \widehat{\mathbb{C}} \longrightarrow \mathbf{Sets}$. Indeed, if $m : \nabla S \longrightarrow F$ then by naturality, we have

$$m_X(s) = m_X(\nabla S_{\langle\rangle_X}(s)) = F_{\langle\rangle_X}(m_\top(s))$$

So m is uniquely determined by the function $m_\top : S \longrightarrow F_\top$. The unit of the adjunction $\eta_S : S \longrightarrow (\nabla S)_\top$ is the identity function; the counit $\varepsilon_F : \nabla F_\top \longrightarrow F$ is given by $(\varepsilon_F)_X(f) = f[\langle\rangle_X]$.

The global sections functor $\mathcal{G} : \widehat{\mathbb{C}} \longrightarrow \mathbf{Sets}$ also has a right adjoint $\Delta : \mathbf{Sets} \longrightarrow \widehat{\mathbb{C}}$ given by $\Delta(S)_X = S^{\mathcal{G}(X)}$ where the latter $\mathcal{G}$ is the global sections functor from $\mathbb{C}$ to $\mathbf{Sets}$. Since $\Delta(S)$ is always extensional it restricts to an adjunction between $\mathbf{Sets}$ and $\mathrm{Ext}(\mathbb{C})$. The unit $\eta : F \longrightarrow \Delta(\mathcal{G}(F))$ of the latter adjunction is the natural transformation sending $f \in F_X$ to $\mathcal{G}(X) \ni x \mapsto f[x] \in \mathcal{G}(F)$. By definition of "extensional" it is a monomorphism and so it follows from a Theorem in [29] that $\mathcal{G}$ is faithful thus giving another proof of Lemma 2.6.7 above.

If $\mathbb{C}$ has a product functor $- \times A$ and all exponentials of the form $-^A$ exist in $\mathbb{C}$, then the assignment $X \mapsto X^A$ extends to a functor on $\mathbb{C}$ acting on morphisms by

$$u^A =_{\mathrm{def}} \mathrm{curry}(u \circ \mathrm{ev}) : X^A \longrightarrow Y^A$$

whenever $u : X \longrightarrow Y$.

This functor is right adjoint to the product functor $- \times A$ in view of the bijective correspondence $\mathbb{C}(X \times A, Y) \cong \mathbb{C}(X, Y^A)$.

2.6.12 Comonads

Comonads form the semantic analogue of S4-modal operators which we will need in order to interpret safe recursion. They are the dual notion of the more familiar monads [29, 30, 24] in the sense that a comonad on a category $\mathbb{C}$ is a monad on $\mathbb{C}^{op}$ and so the general theorems on monads carry over to comonads mutatis mutandis. We will nevertheless develop the needed material from scratch.

Definition 2.6.10 *Let $\mathbb{C}$ be a category. A comonad on $\mathbb{C}$ is a functor $\Box : \mathbb{C} \longrightarrow \mathbb{C}$ together with two natural transformations $\varepsilon_X : \Box(X) \longrightarrow X$—the counit—, and $\nu_X : \Box(X) \longrightarrow \Box(\Box(X))$—the comultiplication satisfying the following equations:*

$$\begin{tikzcd}
\Box X \arrow[r, "\nu_X"] \arrow[dr, "\mathrm{id}_{\Box X}"'] & \Box\Box X \arrow[d, "\varepsilon_{\Box X}"] \\
 & \Box X
\end{tikzcd}
\qquad
\begin{tikzcd}
\Box X \arrow[r, "\nu_X"] \arrow[dr, "\mathrm{id}_{\Box X}"'] & \Box\Box X \arrow[d, "\Box\varepsilon_X"] \\
 & \Box X
\end{tikzcd}
\qquad
\begin{tikzcd}
\Box X \arrow[r, "\nu_X"] \arrow[d, "\nu_X"'] & \Box\Box X \arrow[d, "\Box\nu_X"] \\
\Box\Box X \arrow[r, "\nu_{\Box X}"'] & \Box\Box\Box X
\end{tikzcd}$$

The reader familiar with monads might wish to note that the comonads of interest to us will *not* have a *strength*, i.e., the morphism part of the functor $\Box$ is in general not applicable in the presence of parameters.

Proposition 2.6.11 *If $G : \mathbb{C} \longrightarrow \mathbb{D}$ and $F : \mathbb{D} \longrightarrow \mathbb{C}$ with $G \vdash F$ is a pair of adjoint functors then $F \circ G : \mathbb{C} \longrightarrow \mathbb{C}$ forms a comonad with the counit taken from the adjunction and comultiplication defined from the unit η by*

$$\nu_X = F(\eta_{GX})$$

Every comonad arises in this way albeit not necessarily in a unique way.

Proof. By dualising the corresponding proof for monads, see [24]. □

2.6.12.1 Kleisli triples

Let $(\Box, \varepsilon, \nu)$ be a comonad on category $\mathbb{C}$.

If $f : \Box A \longrightarrow B$ then we can define $f^\Box : \Box A \longrightarrow \Box B$ by

$$f^\Box = \Box(f) \circ \nu_A$$

We call $f^\Box$ the *Kleisli-lifting* of f. This lifting which is reminiscent of the S4 necessitation rule

$$\frac{\Box\Gamma \vdash \varphi}{\Box\Gamma \vdash \Box\varphi} \tag{Nec}$$

satisfies the following equations:

$$\begin{tikzcd}
\Box A \arrow[r, "f^\Box"] \arrow[dr, "f"'] & \Box B \arrow[d, "\varepsilon_B"] \\
& B
\end{tikzcd}
\qquad
\begin{tikzcd}
\Box A \arrow[d, shift right=2, "\varepsilon_A^\Box"'] \arrow[d, shift left=2, "\mathrm{id}_{\Box A}"] \arrow[d, phantom, "="] \\
\Box A
\end{tikzcd}
\qquad
\begin{tikzcd}
\Box A \arrow[r, "f^\Box"] \arrow[dr, "(g \circ f^\Box)^\Box"'] & \Box B \arrow[d, "g^\Box"] \\
& \Box C
\end{tikzcd}$$

From ε and $f \mapsto f^\Box$ we can define comultiplication by $\nu_X = (\mathrm{id}_{\Box X})^\Box$ and even the morphism part of $\Box$ by $\Box(f) = (f \circ \varepsilon)^\Box$. The equations relating ε and ν, naturality of ε (i.e. $f \circ \varepsilon = \varepsilon \circ \Box(f)$), and the functor laws can be proved from the above equations for $f \mapsto f^\Box$ so that we obtain a more economical definition of comonads starting from a mapping $\Box$ on objects, a family of maps ε, and an operation $f \mapsto f^\Box$ on morphisms subject to the above three equations.

The corresponding concept for monads was called *Kleisli-triple* by Moggi.

2.6.12.2 Preservation of products

Now assume that $\mathbb{C}$ has terminal object and cartesian products. We always have the maps

$$\langle\rangle_{\Box\top} : \Box\top \longrightarrow \top$$

and

$$\langle \Box\pi, \Box\pi' \rangle : \Box(A \times B) \longrightarrow \Box A \times \Box B$$

The comonad $\Box$ *preserves* terminal objects and cartesian products if these maps are isomorphisms. In this case, we can identify $\Box\top$ and $\top$ as well as $\Box(A \times B)$ and $\Box A \times \Box B$.

2.6.12.3 Examples

If $\mathbb{C}$ is a category with terminal object then the adjunction $\mathcal{G} \vdash \nabla$ defines a comonad on $\widehat{\mathbb{C}}$ given explicitly by $(\Box F)_X = F_\top$. In the particular case of $\mathbb{C} = \mathbb{S}$ we have that $\Box\mathrm{Tm}(A)$ is the constant presheaf of *closed* terms of type A, and $\Box(\mathrm{Tm}(A) \to \mathrm{Tm}(B))$ is the constant presheaf $\mathrm{Tm}(B)_{x:A}$, i.e., the set of terms of type B with a distinguished free variable of type A.

This can be used to interpret the induction principles for higher-order abstract syntax studied in [11].

Let $\mathbb{B}$ be the category having pairs of natural numbers as objects. A morphism from (m, n) to (m', n') is a pair (f, g) where $f \in \mathbb{P}(m, m')$ and $g \in \mathbb{P}(m + m', n')$ and in addition

$$\max |g(\vec{x}; \vec{y})| \leq p(|\vec{x}|) + \max(|\vec{y}|)$$

for some polynomial p. This category arises from the "polymax-bounded" functions used in the soundness proof for safe recursion (Thm. 2.3.4). Indeed, safe recursion can be recovered as the following schema in $\mathbb{B}$: If $g : (m, n) \longrightarrow (0, 1)$ and $h : (m+1, n+1) \longrightarrow (0, 1)$ then $f : \mathbb{N}^{m+1} \times \mathbb{N}^n \longrightarrow \mathbb{N}$ defined from g, h by safe recursion is a $\mathbb{B}$-morphism from $(m+1, n)$ to $(0, 1)$.

Using $\widehat{\mathbb{B}}$ we can turn this pattern into a single higher-order constant as follows.

The category $\mathbb{B}$ has a terminal object $\top = (0, 0)$ and cartesian products given on objects by componentwise addition. In the functor category we have the representable presheaf $\mathsf{N} = \mathcal{Y}(0, 1)$ given concretely by

$$\mathsf{N}_{(m,n)} = \{f : \mathbb{N}^m \times \mathbb{N}^n \longrightarrow \mathbb{N} \,|\, f \in \mathit{PTIME} \wedge |f(\vec{x};\vec{y})| \leq p(|\vec{x}|) + \max(|\vec{y}|)\}$$

In $\widehat{\mathbb{B}}$ we have a comonad $\Box$ given by $\Box F_{(m,n)} = F_{(m,0)}$. The counit is given by $(\varepsilon_F)_{(m,n)} = F_\pi$ where $\pi : (m, n) \longrightarrow (m, 0)$ is the projection. The comultiplication is the identity as $\Box^2 = \Box$.

This comonad arises from an adjunction between $\widehat{\mathbb{B}}$ and $\widehat{\mathbb{P}}$ one direction of which is (on objects) the projection $(m, n) \mapsto m$ and the other one sends m to $(m, 0)$.

Now,

$$\Box\mathsf{N}_{(m,n)} = \mathsf{N}_{(m,0)} = \mathbb{B}((m, n)\,,\,(1, 0))$$

So $\Box\mathsf{N} \cong \mathcal{Y}(1, 0)$ and by the Yoneda Lemma we know that the set of $\mathbb{B}$-morphisms from (m, n) to $(0, 1)$ are in 1-1 correspondence with the $\widehat{\mathbb{B}}$-morphisms from $\Box\mathsf{N}^m \times \mathsf{N}^n$ to N.

Lemma 2.6.3 gives for arbitrary presheaf F the characterisations

$$\begin{array}{l}(\mathsf{N}{\to}F)_{(m,n)} \cong F_{(m,n+1)}\\ (\Box\mathsf{N}{\to}F)_{(m,n)} \cong F_{(m+1,n)}\end{array}$$

Safe recursion thus takes the form of a single higher-order constant

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

In the next section we show how to interpret in $\widehat{\mathbb{B}}$ a modal lambda calculus which allows to formulate a higher-order extension of safe recursion analogous to system PV^ω.

2.6.12.4 Interpretation of modal lambda calculus

A cartesian closed category with a product preserving comonad can be used to interpret the modal lambda calculus by Pfenning and Davies [33].[1]

In addition to the constructs of typed lambda calculus it has a unary type former $\Box$:

$$A ::= \ldots \,|\, \Box A$$

[1] We will later on use other formulations of modal lambda calculus. In using Pfenning's calculus here we illustrate the flexibility of the comonad concept and provide a link between our subsequent formulations of modal lambda calculus and existing work.

The grammar for terms is extended by the following clauses:

$$e ::= \ldots \mid \mathrm{box}(e) \mid \mathsf{let}\, e_1{=}\mathrm{box}(x)\, \mathsf{in}\, e_2$$

Contexts are built out of two kinds of binding: $x\colon A$ (ordinary binding) and $x \overset{\Box}{:} A$ (modal binding)[2]. A context with modal bindings only is called modal.

The typing rules for the lambda calculus fragment are as before with the understanding that rule T-Var applies to both modally and ordinarily bound variables and that rule T-Arr-E requires an ordinarily bound variable. The rules for the newly introduced constructs are

$$\frac{\Gamma \vdash e : A \qquad \Gamma \text{ modal}}{\Gamma \vdash \mathrm{box}(e) : \Box A} \tag{T-Box}$$

$$\frac{\Gamma \vdash e_1 : \Box A \qquad \Gamma, x \overset{\Box}{:} A \vdash e_2 : B}{\Gamma \vdash \mathsf{let}\, e_1{=}\mathrm{box}(x)\, \mathsf{in}\, e_2 : B} \tag{T-Let-Box}$$

This system can be modelled in a category with product preserving comonad as follows: types are interpreted as usual with the extra clause

$$[\![\Box A]\!] = \Box[\![A]\!]$$

A context Γ gets interpreted as $\prod_{x\in\mathrm{dom}(\Gamma)} F_x([\![A]\!])$ where F_x is the identity if x is ordinarily bound and $F_x = \Box$ if x is modally bound in Γ.

As before, a typing judgement $\Gamma \vdash e : A$ gets interpreted as a morphism

$$[\![\Gamma \vdash e : A]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!]$$

The defining clauses for the functional fragment are as before.

Since $\Box$ preserves products, the meaning of a modal context is of the form $\Box(X)$ thus allowing us to interpret rule T-Box by Kleisli lifting, i.e.,

$$[\![\Gamma \vdash \mathrm{box}(e)]\!] = [\![\Gamma \vdash e]\!]^{\Box} \; (= \Box([\![\Gamma \vdash e]\!]) \circ \nu)$$

Finally, the rule T-Let-Box is interpreted using composition as follows.

$$[\![\Gamma \vdash \mathsf{let}\, e_1{=}\Box(x)\, \mathsf{in}\, e_2 : B]\!] = [\![\Gamma, x \overset{\Box}{:} A \vdash e_2 : B]\!] \circ \langle \mathrm{id}_{[\![\Gamma]\!]}, [\![\Gamma \vdash e_1 : \Box A]\!]\rangle$$

In order to get a higher-order version of safe recursion we add constants of appropriate type for the basic functions, e.g., $\mathsf{S}_0, \mathsf{S}_1 : \mathsf{N}{\to}\mathsf{N}$ and a constant

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

[2] Pfenning and Davies use two zoned contexts of the form $\Gamma; \Delta$ where Γ records the modal bindings and Δ records the ordinary bindings.

We can embed Bellantoni-Cook's system into this calculus by translating a function $f(\vec{x};\vec{y})$ into a closed term of type $\Box\mathsf{N}^m{\to}\mathsf{N}^n{\to}\mathsf{N}$. Moreover, functionals like the iteration functional $\mathrm{it}(f,u) = f^{|u|}(0)$ from Section 2.5 can now be defined as single constants rather than schemata in $\mathcal{B}$:

$$\mathrm{it} =_{\mathrm{def}} \lambda f{:}\,\mathsf{N}{\to}\mathsf{N}.\lambda u{:}\,\Box\mathsf{N}.\mathsf{rec}\ 0\ (\lambda x{:}\,\Box\mathsf{N}.\lambda y{:}\,\mathsf{N}.f\ y)\ u$$

The resulting calculus can now be interpreted in $\widehat{\mathbb{B}}$ as sketched above using the Yoneda Lemma to interpret first-order constants and the above reasoning in order to interpret the recursor rec. The thus obtained interpretation has the property that, again by the Yoneda Lemma, the meaning of a first-order function is a morphism in $\mathbb{B}$—hence *PTIME*. In order to show that the thus associated *PTIME*-function is the "correct one", we may relate this interpretation to the set-theoretic one by extending the logical relation used for PV^ω by the clause

$$x\,R_{\Box A}\,y \iff x\,R_A\,y$$

Notice here that $\mathcal{G}(\Box F) = \mathcal{G}(F)$. By following the pattern of the corresponding proof for PV^ω it then follows that the set-theoretic meanings of first-order functions in the modal lambda calculus with constants for safe recursion are *PTIME*.

Like in the case of PV^ω the category $\mathbb{B}$ can be replaced by a category whose morphisms are terms denoting definitions in the function algebra $\mathcal{B}$. More precisely, a morphism from (m,n) to (m',n') would consist of an m'-tuple of terms $f_i(\vec{x};)$ in m normal variables $\vec{x}$ together with an n'-tuple of terms $g_j(\vec{x};\vec{y})$ in m normal variables $\vec{x}$ and n safe variables $\vec{y}$.

The semantics then shows that for every term $t : \Box\mathsf{N}{\to}\mathsf{N}$ in modal lambda calculus we can find a term $\hat{t}(x;)$ with the same meaning as t.

One should notice, however, that such a category is no longer well-pointed which means that the definition of the logical relation must be made in a slightly more complicated fashion. Details of this approach will not be discussed in this thesis.

2.7 Affine linear categories

Modelling contexts as cartesian products means that (semantically) a variable can be used as many times as desired: in every category with cartesian products we can define diagonal morphisms $\delta : [\![x{:}\,A]\!] \longrightarrow [\![x_1{:}\,A, x_2{:}\,A]\!]$, namely $\delta = \langle \mathrm{id}, \mathrm{id}\rangle$.

If we want to semantically exploit linearity restrictions we must generalise cartesian products in such a way that diagonal maps do not necessarily exist anymore. This can be done and leads to the notion of symmetric monoidal category [29]. This concept, although well established, is somewhat complicated due to the fact that the associated structure is not determined by universal properties as in the case of cartesian product and thus a number of coherence conditions are required.

Fortunately, all the examples of interest in the present context enjoy an extra property which allows us to avoid coherence. Rather than defining symmetric monoidal categories and afterwards isolating a special case we will define the latter directly.

Definition 2.7.1 *An affine linear category (ALC) is given by the following data:*

- *a category $\mathbb{C}$,*
- *for any two objects $A, B \in \mathbb{C}$ an object $A \otimes B$, called* tensor product*, and morphisms $\pi : A \otimes B \longrightarrow A$ and $\pi' : A \otimes B \longrightarrow B$, called projections, which are jointly monomorphic in the following sense. If $f, g : C \longrightarrow A \otimes B$ and $\pi \circ f = \pi \circ g$ and $\pi' \circ f = \pi' \circ g$ then $f = g$,*
- *for any two maps $f : A \longrightarrow A'$ and $g : B \longrightarrow B'$ a map $f \otimes g : A \otimes B \longrightarrow A' \otimes B'$ such that $\pi \circ (f \otimes g) = f \circ \pi$ and $\pi' \circ (f \otimes g) = g \circ \pi'$,*
- *an isomorphism $\alpha : A \otimes (B \otimes C) \longrightarrow (A \otimes B) \otimes C$ such that $\pi \circ \pi \circ \alpha = \pi$, $\pi' \circ \pi \circ \alpha = \pi \circ \pi'$, $\pi' \circ \alpha = \pi' \circ \pi'$,*
- *an isomorphism $\gamma : A \otimes B \longrightarrow B \otimes A$ such that $\pi \circ \gamma = \pi'$ and $\pi' \circ \gamma = \pi$,*
- *a terminal object $\top$ such that $\pi : A \otimes \top \longrightarrow A$ and $\pi' : \top \otimes A \longrightarrow A$ are isomorphisms*

This definition warrants some explanation. First, and most importantly, we note that in view of the requirement on the projections the other constructions $f \otimes g$, α, and γ, are uniquely determined by their defining equations.

Next we note that if $\mathbb{C}$ happens to have cartesian products then for every pair of objects A, B we have a monomorphism

$$\langle \pi, \pi' \rangle : A \otimes B \longrightarrow A \times B$$

These leads to the intuition that the tensor product can be seen as an extra property on pairs: given $f : C \longrightarrow A$ and $g : C \longrightarrow B$ then it may or may not be the case that $\langle f, g \rangle$ factors through $A \otimes B$. The definition of ALC states in this sense that certain definable maps involving cartesian products factor through tensor products, for example the associativity map $\langle \langle \pi \circ \pi, \pi' \circ \pi \rangle, \pi' \rangle : (A \times B) \times C \longrightarrow A \times (B \times C)$ does.

We notice that global elements always factor through the tensor product: if $a \in \mathcal{G}(A)$ and $b \in \mathcal{G}(B)$ then $a \otimes b : \top \otimes \top \longrightarrow A \otimes B$ and thus $(a \otimes b) \circ \langle\rangle^{-1} \in \mathcal{G}(A \otimes B)$ where we have used the fact that the unique map $\pi = \pi' = \langle\rangle : \top \otimes \top \longrightarrow \top$ is required to be an isomorphism. Now $\pi \circ (a \otimes b) \circ \langle\rangle^{-1} = a \circ \pi \circ \langle\rangle^{-1} = a$ and similarly for b.

What we have essentially used here is the existence of a diagonal map $\delta : \top \longrightarrow \top \otimes \top$ satisfying $\pi \circ \delta = \pi' \circ \delta = \mathrm{id}$. This motivates the following definition.

Definition 2.7.2 *An object D in an ALC is called* duplicable *if there exists a (uniquely determined) morphism $\delta : D \longrightarrow D \otimes D$ such that $\pi \circ \delta = \pi' \circ \delta = \mathrm{id}$.*

We write $\mathbb{C}_{\mathrm{dup}}$ for the full subcategory consisting of the duplicable objects.

Lemma 2.7.3 *If $\mathbb{C}$ is an ALC then $\top \in \mathbb{C}_{\mathrm{dup}}$ and whenever $A, B \in \mathbb{C}_{\mathrm{dup}}$ so is $A \otimes B$. Moreover, $A \otimes B$ is a cartesian product in $\mathbb{C}_{\mathrm{dup}}$.*

Proof. The diagonal for $\top$ is obtained as a special case of the isomorphism $X \cong X \otimes \top$. The diagonal $\delta^{A\otimes B}$ for $A \otimes B$ is obtained from $\delta^A : A \longrightarrow A \otimes A$ and $\delta^B : B \longrightarrow B \otimes B$ as

$$A \otimes B \xrightarrow{\delta^A \otimes \delta^B} (A \otimes A) \otimes (B \otimes B) \xrightarrow{w} (A \otimes B) \otimes (A \otimes B)$$

where w is a wiring map. □

The above discussion of global elements generalises to the following lemma.

Lemma 2.7.4 *If D is duplicable and $f : D \longrightarrow A$, $g : D \longrightarrow B$ then there exists a unique map $h = (f \otimes g) \circ \delta : D \longrightarrow A \otimes B$ such that $\pi \circ h = f$ and $\pi' \circ h = g$.*

For the reader familiar with symmetric monoidal categories (SMC), see [29] we remark that every ALC forms an SMC, but not vice versa.

2.7.1 Examples

Of course, every category with cartesian products and terminal object forms an ALC. A more interesting example which forms the intuitive basis of the central result in this thesis is constructed as follows.

Let us define the category $\mathbb{M}$ of *length spaces* (for lack of better name) as follows. A length space is a pair $A = (|A|, \ell_A)$ where $|A|$ is a set and $\ell_A : |A| \longrightarrow \mathbb{N}$ is a mapping assigning a natural number—the length—to each element of $|A|$. A morphism from length space A to B is a function $f : |A| \longrightarrow |B|$ such that

$$\ell_B(f(a)) \leq c + \ell_A(a)$$

for some "witnessing" constant c independent of a. Composition is ordinary composition of functions. The tensor product is given by $|A \otimes B| = |A| \times |B|$ and $\ell_{A\otimes B}(a, b) = \ell_A(a) + \ell_B(b)$. The tensor product of two morphisms $f : A \longrightarrow A'$ and $g : B \longrightarrow B'$ is given as in **Sets** by $(f \otimes g)(a, b) = (f(a), g(a))$. If c_f and c_g are constants witnessing that f, g are morphisms then we have

$$\ell_{A'\otimes B'}((f \otimes g)(a, b)) \leq c_f + c_g + \ell_A(a) + \ell_B(b) = \ell_{A\otimes B}(a, b) + c_1 + c_2$$

So, $f \otimes g$ is a morphism witnessed by $c_1 + c_2$. Similarly, the set-theoretic isomorphisms witnessing associativity, etc., of cartesian product are morphisms in $\mathbb{M}$.

The terminal object $\top$ is given by $|\top| = \{\langle\rangle\}$ and $\ell_\top(\langle\rangle) = 0$. Clearly, $A \otimes \top \cong A$ is isomorphic to A by the set-theoretic isomorphism between $|A| \times \top$ and $|A|$ so that we can conclude that $\mathbb{M}$ is an ALC.

A length space X is duplicable iff there exists a constant c such that $\ell_X(x) \leq c$ for all $x \in |X|$. Namely, in this case $\ell_{X\otimes X}(x, x) = 2\ell_X(x) \leq c + \ell_X(x)$ so the diagonal map is a morphism from X to $X \otimes X$.

On the other hand, the object N with $|\mathsf{N}| = \mathbb{N}$ and $\ell_{\mathsf{N}}(x) = |x|$ is not duplicable.

The category $\mathbb{M}$ has the subcategory $\mathbb{M}_s$ of strict morphisms which satisfy the stronger requirement

$$\ell_B(f(a)) \leq \ell_A(a)$$

The tensor product on $\mathbb{M}$ restricts to $\mathbb{M}_s$ in the sense that the required isomorphisms α, γ etc. are strict and that tensor product of strict morphisms is strict. In $\mathbb{N}^s$ the only duplicable objects are those for which $\ell_X(x) = 0$.

Both $\mathbb{M}$ and $\mathbb{M}_s$ also have a cartesian product given by $|A \times B| = |A| \times |B|$ and $\ell_{A\times B}(a, b) = \max(\ell_A(a), \ell_B(b))$.

The pairing of morphisms $f : C \longrightarrow A$ and $g : C \longrightarrow B$ is given as in **Sets** by $\langle f, g\rangle(c) = (f(c), g(c))$ and we have

$$\ell_{A\times B}(f(c), g(c)) = \max(\ell_A(f(c)), \ell_B(g(c))) \leq \max(c_f, c_g) + \ell_C(c)$$

Notice that $\langle f, g\rangle$ is *not* an $\mathbb{M}$-morphism from C to $A \otimes B$ unless C is duplicable.

2.7.2 Wiring maps

The morphisms α, γ and the constructor $\otimes$ yield intuitively linear maps. One might ask whether these primitives are sufficient; they are indeed in the following sense.

A morphism in an ALC constructed by composition, tensoring ($\otimes$) from the isomorphisms α, γ, etc., projections, identities will be called a *wiring map* (MacLane calls the corresponding notion for SMC *canonical maps*).

Suppose that object A can be decomposed into a term built up via $\top$ and $\otimes$ from objects $A_1, \dots, A_n$ in this order, e.g., $n = 4$ and $A = ((A_1 \otimes A_2) \otimes (\top \otimes A_3)) \otimes A_4$. Suppose further that ι is an injection from m elements into n elements and that B is built up from $A_{\iota(1)}, \dots, A_{\iota(m)}$ in this order, e.g., $B = A_4 \otimes (A_3 \otimes A_2)$. Then there exists a wiring map $w_\iota : A \longrightarrow B$ which "sends A_i in A to A_i in B" in the sense that it satisfies the obvious specification in terms of projections. In the example the wiring map is given by

$$\begin{array}{ll} & ((A_1 \otimes A_2) \otimes (\top \otimes A_3)) \otimes A_4 \\ \xrightarrow{\gamma} & A_4 \otimes ((A_1 \otimes A_2) \otimes (\top \otimes A_3)) \\ \xrightarrow{\mathrm{id}_{A_4} \otimes (\pi' \otimes \pi')} & A_4 \otimes (A_2 \otimes A_3) \\ \xrightarrow{\mathrm{id}_{A_4} \otimes \gamma} & A_4 \otimes (A_3 \otimes A_2) \end{array}$$

and it is uniquely determined by the specification $\pi \circ w = \pi'$, $\pi \circ \pi' \circ w = \pi' \circ \pi' \circ \pi$, $\pi' \circ \pi' \circ w = \pi' \circ \pi \circ \pi$.

It should be clear from this example how these wiring maps are defined and specified in general using an appropriate inductive definition.

2.7.3 Indexed tensor product

Recall that in a category with cartesian products we can define all finite indexed cartesian products using some arbitrary enumeration of the index set. A similar thing can be done in ALC. Suppose that $(A_i)_{i \in I}$ is a finite family of objects of an ALC $\mathbb{C}$. We define the tensor product $\bigotimes_{i \in I} A_i$ by

$$\bigotimes_{i \in I} A_i = (\dots(A_{i_1} \otimes A_{i_2})\dots) \otimes A_{i_n}$$

where $I = \{i_1, \dots, i_n\}$ is some arbitrary enumeration of the elements of the index set I. Associativity and symmetry show that this tensor product is independent up to isomorphism of the chosen enumeration. We can define projections $\pi_i : \bigotimes_{i \in I} A_i \longrightarrow A_i$ and wiring maps in the obvious sense.

2.7.4 Linear function spaces

In this section we define an analogue of function spaces for ALC. Let A, B be objects in an ALC $\mathbb{C}$. The linear function space or linear exponential of B by A is given by an object $A \multimap B$, a morphism $\mathrm{ev} : (A \multimap B) \otimes A \longrightarrow B$—the evaluation map—and for every morphism $f : X \otimes A \longrightarrow B$ a unique morphism $\mathrm{curry}(f) : X \longrightarrow A \multimap B$ called the currying or exponential transpose of f such that $\mathrm{ev} \circ (\mathrm{curry}(f) \otimes \mathrm{id}_A) = f$.

Again, instead of uniqueness we can postulate the following additional equations which in turn are consequences of uniqueness:

$$\begin{aligned}&\mathrm{curry}(f) \circ h = \mathrm{curry}(f \circ (h \otimes \mathrm{id}))\\&\mathrm{curry}(\mathrm{ev}) = \mathrm{id}\end{aligned}$$

Currying and evaluation establish a natural bijection

$$\mathbb{C}(X \otimes A, B) \cong \mathbb{C}(X, A \multimap B)$$

An ALC in which all linear function spaces exist will be called *affine linear closed category* (ALCC). We remark that the analogue for SMC is called *symmetric monoidal closed category.* In an ALCC the mapping $X \mapsto A \multimap X$ extends to a functor on $\mathbb{C}$ which is right adjoint to the tensor product functor $\_ \otimes A$.

2.7.4.1 Examples

If tensor product is given by cartesian product then ordinary function spaces in the sense of Section 2.6.7 are the linear function spaces. In $\mathbb{M}$ the function space $A \multimap B$ is given by

$$\begin{aligned}&|A \multimap B| = \mathbb{M}(A, B)\\&\ell_{A \multimap B}(f) = \min\{c \,|\, \forall a \in |A|. \ell_B(f(a)) \leq c + \ell_A(a)\} = \max_{a \in |A|} \ell_B(f(a)) \dot{-} \ell_A(a)\end{aligned}$$

Evaluation sends $(f, a) \in |(A \multimap B) \otimes A|$ to $f(a) \in B$ and we have

$$\ell_B(f(a)) \leq \ell_{A \multimap B}(f) + \ell_A(a) = \ell_{(A \multimap B) \otimes A}(f, a)$$

so evaluation is actually a map in $\mathbb{M}_s$. Conversely, if $f : X \otimes A \longrightarrow B$ then $\mathrm{curry}(f)(x) = f_x$ where $f_x(a) = f(x,a)$. We have $\ell_B(f_x(a)) \leq \ell_X(x)+\ell_A(a)+c$ for some c, so $f_x \in |A{\multimap}B|$ and moreover $\ell_{A\multimap B}(f_x) \leq c + \ell_X(x)$, so $\mathrm{curry}(f) \in \mathbb{M}(X, A{\multimap}B)$.

We notice that if f is an $\mathbb{M}_s$-morphism then so is $\mathrm{curry}(f)$. This—together with the fact that evaluation is a $\mathbb{M}_s$-morphism—implies that $\mathbb{M}_s$ also has linear function spaces and that they agree with those in $\mathbb{M}$. Somewhat astonishingly we do *not* have $|A{\multimap}B| = \mathbb{M}_s(A,B)$ in $\mathbb{M}_s$.

2.7.4.2 Affine linear lambda calculus

In the same way as the typed lambda calculus provides an "internal language" for categories with function spaces we can devise a linear typed lambda calculus which admits a natural interpretation in an ALC with enough linear function spaces.

The types of linear lambda calculus[3] are the same as those of ordinary typed lambda calculus except that we write $A{\multimap}B$ instead of $A{\rightarrow}B$ to emphasize the linear character. The typing rules T-Var, T-Const, and T-Arr-I remain unchanged. Rule T-Arr-E which corresponds to application is replaced by the following weaker rule:

$$\frac{\Gamma \vdash e_1 : A{\multimap}B \qquad \Delta \vdash e_2 : A \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset}{\Gamma, \Delta \vdash e_1 e_2 : B} \tag{T-Arr-E-Lin}$$

Induction on derivations shows that if $\Gamma \vdash e : A$ then every variable $x \in \mathrm{dom}(\Gamma)$ appears at most once in e. So, for example, $\lambda f\colon A{\multimap}A{\multimap}A.\lambda x\colon A.\lambda y\colon A.fyx$ is well typed (of type $(A{\multimap}A{\multimap}A){\multimap}\ A{\multimap}A{\multimap}A$), but

$$\lambda f\colon A{\multimap}A{\multimap}A.\lambda x\colon A.fxx$$

is ill-typed as is

$$\lambda f\colon A{\multimap}A.\lambda x\colon A.f(fx)$$

The linear lambda calculus admits an interpretation in an ALC with a distinguished class of objects closed under linear function space and containing interpretations for base types. Types are interpreted by the obvious clause $[\![A{\multimap}B]\!] = [\![A]\!]{\multimap}[\![B]\!]$. A context Γ is interpreted as a $\mathrm{dom}(\Gamma)$-index tensor product:

$$[\![\Gamma]\!] = \bigotimes_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]$$

A variable is interpreted by the corresponding projection; functional abstraction is interpreted by currying. For linear application assume that $\Gamma \vdash e_1 : A{\multimap}B$ and $\Delta \vdash e_2 : A$. Then

$$[\![\Gamma, \Delta, \vdash e_1 e_2 : B]\!]$$

[3] We henceforth omit the attribute "affine" in this context.

is given by

$$[\![\Gamma, \Delta]\!] \xrightarrow{w} [\![\Gamma]\!] \otimes [\![\Delta]\!] \xrightarrow{f_1 \otimes f_2} [\![A \multimap B]\!] \otimes [\![A]\!] \xrightarrow{\mathrm{ev}} [\![B]\!]$$

where w is a wiring map and f_1, f_2 are the interpretations of e_1, e_2, respectively.

2.7.5 Comonads in ALC

A comonad $\Box : \mathbb{C} \longrightarrow \mathbb{C}$ is said to preserve tensor product and terminal object if the morphisms $\langle\rangle : \Box\top \longrightarrow \top$ is an isomorphism and there exists a (uniquely determined) isomorphism $\varphi : \Box(X \otimes Y) \longrightarrow \Box X \otimes \Box Y$ such that $\pi \circ \varphi = \Box\pi$ and $\pi' \circ \varphi = \Box\pi'$. If this is the case, we can notationally identify $\Box(X \otimes Y)$ with $\Box X \otimes \Box Y$ and $\Box\top$ with $\top$.

2.7.6 ALC and the Yoneda embedding

In this section we show how to add linear function spaces and also a comonad ! giving duplicability to an arbitrary well-pointed ALC. This is one of the main tools for our soundness proof for the calculus SLR. We emphasise at this point that the technique described in this section is independent of this intended application and could be applied in various other situations.

Recall that the Yoneda embedding $\mathcal{Y} : \mathbb{C} \longrightarrow \widehat{\mathbb{C}}$ preserves cartesian products and also existing function spaces in a category $\mathbb{C}$ with cartesian products. In this way, functor categories provide a way to generically add lacking function spaces to a category with cartesian products.

A similar effect can be achieved for the linear world using an appropriate tensor product construction for presheaves. For SMC this has been done by Day [10]. However, even if $\mathbb{C}$ is an ALC then the presheaf category $\widehat{\mathbb{C}}$ equipped with Day's tensor product will in general fail to be an ALC because the projections (which in this case can be defined) fail to be jointly monomorphic. However, if the underlying ALC $\mathbb{C}$ is well-pointed then we can define a tensor product on the full subcategory $\mathrm{Ext}(\mathbb{C})$ of extensional presheaves making it an ALCC in such a way that the Yoneda embedding preserves tensor products and existing linear function spaces. The tensor product we define agrees in fact with Day's tensor product which can also be defined for $\mathrm{Ext}(\mathbb{C})$. In this particular case there is, however, a more concrete description which in particular avoids the notion of "end" [29]. We will work exclusively with this more concrete description.

Assume for the rest of this section that $\mathbb{C}$ is a well-pointed ALC.

2.7.6.1 Tensor product of extensional presheaves

For extensional presheaves $A, B \in \mathrm{Ext}(\mathbb{C})$ the tensor product $A \otimes B \in \mathrm{Ext}(\mathbb{C})$ is defined as follows. For $X \in \mathbb{C}$ the set $(A \otimes B)_X$ consists of those pairs $(a, b) \in A_X \times B_X$ for which there exist objects $U, V \in \mathbb{C}$, a map $t : X \longrightarrow U \otimes V$ and elements $\bar{a} \in A_U$, $\bar{b} \in B_V$ such that $a = \bar{a}[\pi \circ t]$ and $b = \bar{b}[\pi' \circ t]$.

$$(a,b) \in A_X \times B_X \qquad\qquad \bar{a} \in A_U \qquad \bar{b} \in A_V$$

$$\begin{tikzcd}
X \arrow[dr, "t"'] & & U & V \\
& U \otimes V \arrow[ur, "\pi"] \arrow[urr, "\pi'"'] & &
\end{tikzcd}$$

In this case we say that $t, \bar{a}, \bar{b}$ witness that $(a,b) \in (A \otimes B)_X$. We may also use the notation $(a, b|t, \bar{a}, \bar{b})$ for such a pair with the understanding that $t, \bar{a}, \bar{b}$ are not part of the element but merely required to exist.

If $u : Y \longrightarrow X$ and $(a,b) \in (A \otimes B)_X$ witnessed by $t, \bar{a}, \bar{b}$ then we define $(A \otimes B)_u(a,b)$ as $(a[u], b[u])$ which can be witnessed by $(t \circ u, \bar{a}, \bar{b})$. Using the above notation we can write more compactly

$$(A \otimes B)_u(a, b|t, \bar{a}, \bar{t}) =_{\mathrm{def}} (a[u], b[u]|t \circ u, \bar{a}, \bar{b})$$

which is a valid definition as the right hand side does not depend on the choice of the witnesses $t, \bar{a}, \bar{b}$.

Since the thus defined presheaf $A \otimes B$ is a sub-presheaf of the cartesian product $A \times B$ it is clearly extensional, i.e., lies in $\mathrm{Ext}(\mathbb{C})$.

The projections are defined in the obvious way by $\pi_X(a,b) = a$ and $\pi'_X(a,b) = b$ which are clearly jointly monomorphic.

If $f : A \longrightarrow A'$ and $g : B \longrightarrow B'$ then $f \otimes g : A \otimes B \longrightarrow B \otimes B'$ is given by

$$(f \otimes g)_X(a, b|t, \bar{a}, \bar{b}) =_{\mathrm{def}} (f(a), g(b)|t, f(\bar{a}), g(\bar{b}))$$

Associativity is defined by $\alpha(a, (b, c)) = ((a, b), c)$. If $(a, (b, c))$ is witnessed by $t : X \longrightarrow U \otimes V$ and $\bar{a} \in A_U$ and $t' : V \longrightarrow W \otimes Z$ and $\bar{b} \in B_W$ and $\bar{c} \in C_Z$ then we have

$$s =_{\mathrm{def}} \alpha \circ (\mathrm{id} \otimes t') \circ t : X \longrightarrow (U \otimes W) \otimes Z$$

and we obtain a witness for $((a, b), c)$ as $s, \mathrm{id}_{U \otimes W}, \bar{a}, \bar{b}, \bar{c}$. Similarly, the other isomorphisms are defined and it follows that $\mathrm{Ext}(\mathbb{C})$ with the above tensor product is an ALC.

Let us see how the Yoneda embedding preserves the tensor product. Suppose that $(a, b|t, \bar{a}, \bar{b}) \in (\mathcal{Y}(A) \otimes \mathcal{Y}(B))_X$, i.e., $a : X \longrightarrow A, b : X \longrightarrow B, t : X \longrightarrow U \otimes V, \bar{a} : U \longrightarrow A, \bar{b} : V \longrightarrow B, a = \bar{a} \circ \pi \circ t, b = \bar{b} \circ \pi' \circ t$.

Then

$$h =_{\mathrm{def}} (\bar{a} \otimes \bar{b}) \circ t : X \longrightarrow A \otimes B$$

is an element of $\mathcal{Y}(A \otimes B)$ and it is independent of the witness as $\pi \circ h = \bar{a} \circ \pi \circ t = a$ and similarly $\pi' \circ h = b$. Conversely, if $h \in \mathcal{Y}(A \otimes B)_X$ then $(\pi \circ h, \pi' \circ h|h, \mathrm{id}, \mathrm{id}) \in \mathcal{Y}(A) \otimes \mathcal{Y}(B)$

and it is readily seen that these constructions yield a natural isomorphism $\iota : \mathcal{Y}(A) \otimes \mathcal{Y}(B) \cong \mathcal{Y}(A \otimes B)$ which is canonical in the sense that $\mathcal{Y}(\pi) \circ \iota = \pi$ and $\mathcal{Y}(\pi') \circ \iota = \pi'$ which allows us to treat it as an identity.

Lemma 2.7.5 *Let $A, B \in \mathrm{Ext}(\mathbb{C})$. If $D \in \mathbb{C}_{\mathrm{dup}}$ then $(A \otimes B)_D = (A \times B)_D$.*

Proof. Any element $(a, b) \in (A \times B)_D$ can be witnessed by δ_D, a, b. □

2.7.6.2 Linear function space of extensional presheaves

Proposition 2.7.6 *The ALC $\mathrm{Ext}(\mathbb{C})$ has all linear function spaces and the Yoneda embedding $\mathcal{Y} : \mathbb{C} \longrightarrow \mathrm{Ext}(\mathbb{C})$ preserves existing linear function spaces up to canonical isomorphism.*

Proof. We only give the constructions leaving most of the routine verifications to the reader.

The linear function space $A{\multimap}B$ of extensional presheaves is defined similarly to the ordinary function space:

$$(A{\multimap}B)_X =_{\mathrm{def}} \mathrm{Ext}(\mathbb{C})(\mathcal{Y}(X) \otimes A, B)$$

If $f : C \otimes A \longrightarrow B$ then $\mathrm{curry}(f) : C \longrightarrow A{\multimap}B$ is defined as follows. Suppose that $X \in \mathbb{C}, c \in C_X, Y \in \mathbb{C}$, and $(u, a) \in (\mathcal{Y}(X) \otimes A)_Y$. The latter means that $u : Y \longrightarrow X, a \in A_Y$ and that there exist $t : Y \longrightarrow U \otimes V$, $\bar{u} : U \longrightarrow X$, $\bar{a} : A_V$ such that $a = \bar{a}[\pi' \circ t]$ and $u = \bar{u} \circ \pi \circ t$. We define

$$\mathrm{curry}(f)_X(c)_Y(u, a) =_{\mathrm{def}} f_Y(c[u], a)$$

The argument to f_Y can be witnessed by $t, c[\bar{u}], \bar{a}$.

Next, we define the evaluation map $\mathrm{ev} : (A{\multimap}B) \otimes A \longrightarrow B$. Suppose we are given $X \in \mathbb{C}, f \in (A{\multimap}B)_X, a \in A_X$, and witnesses $t : X \longrightarrow U \otimes V, \bar{f} \in (A{\multimap}B)_U, \bar{a} \in A_V$. This means that whenever $(x, a') \in (\mathcal{Y}(X) \otimes A)_Y$, i.e., $x : Y \longrightarrow X$ and $a' \in A_Y$ plus appropriate witnesses, then $f_Y(x, a') = \bar{f}_Y(\pi \circ t \circ x, a')$.

We might be tempted to define

$$\mathrm{ev}_X(f, a) =_{\mathrm{def}} f_X(\mathrm{id}_X, a)$$

However, this does not work since (id_X, a) need not be an element of $(\mathcal{Y}(X) \otimes A)_X$. The only possible witness would use t and $\bar{a}$, but then we would have to give $\bar{x} : U \longrightarrow X$ such that $\bar{x} \circ \pi \circ t = \mathrm{id}_X$ and there is in general no reason as to why such $\bar{x}$ should exist.

Instead, we define

$$\mathrm{ev}_X(f, a) =_{\mathrm{def}} \bar{f}_X(\pi \circ t, a)$$

and now the argument can be witnessed by $t, \mathrm{id}_U, \bar{a}$. But since we have used part of the witness in the definition we must show that this is at all well-defined, i.e., does not depend on the witness chosen. This is where extensionality comes in. If $x \in \mathcal{G}(X)$ then

$$\begin{array}{lll}
 & (\mathrm{ev}_X(f,a))[x] & \\
= & \bar{f}_X(\pi \circ t, a)[x] & \\
= & \bar{f}_\top(\pi \circ t \circ x, a[x]) & \text{naturality of } f \\
= & f_\top(x, a[x]) &
\end{array}$$

where the last step follows from the assumption $\bar{f}[\pi \circ t] = f$ using the fact that $(x, a[x]) \in (\mathcal{Y}(X) \otimes A)_\top$ by Lemma 2.7.5 e.g. by $\delta_\top, x, a$. So, the restriction of $\mathrm{ev}_X(f,a)$ along a global element does not depend on the chosen witness and so $\mathrm{ev}_X(f,a)$ is well-defined by extensionality of B. □

Proposition 2.7.7 *If $A, B \in \mathbb{C}$ and $A \multimap B$ exists in $\mathbb{C}$ then $\mathcal{Y}(A \multimap B)$ is canonically isomorphic to $\mathcal{Y}(A) \multimap \mathcal{Y}(B)$.*

Proof. By Lemma 2.6.3

$$\begin{array}{lll}
 & \mathcal{Y}(A \multimap B)_X & \\
\cong & \mathbb{C}(X\,,\, A \multimap B) & \\
\cong & \mathbb{C}(X \otimes A\,,\, B) & \\
\cong & \mathrm{Ext}(\mathbb{C})(\mathcal{Y}(X \otimes A)\,,\, \mathcal{Y}(B)) & \text{since } \mathcal{Y} \text{ is full and faithful} \\
\cong & \mathrm{Ext}(\mathbb{C})(\mathcal{Y}(X) \otimes \mathcal{Y}(A), \mathcal{Y}(B)) & \\
\cong & (\mathcal{Y}(A) \multimap \mathcal{Y}(B))_X &
\end{array}$$

□

Finally, we have an affine linear analogue to Prop. 2.6.4.

Proposition 2.7.8 *Let $U \in \mathbb{C}$. For any presheaf $F \in \mathrm{Ext}(\mathbb{C})$ the linear function space $U \multimap F\,(= \mathcal{Y}(U) \multimap F)$ is isomorphic to the presheaf $F_{-\otimes U}$ obtained by precomposing F with the functor $X \mapsto X \otimes U$.*

Proof. Like the proof of Prop. 2.6.4 with × replaced by ⊗ and → replaced by ⊸. □

2.7.6.3 Duplication in $\mathrm{Ext}(\mathbb{C})$

Obviously, if $D \in \mathbb{C}_{\mathrm{dup}}$ then $\mathcal{Y}(D)$ is duplicable in $\mathrm{Ext}(\mathbb{C})$ since $\mathcal{Y}(\delta_D)$ gives the required diagonal.

In $\mathrm{Ext}(\mathbb{C})$ we can associate to any presheaf a "best approximating" duplicable presheaf.

Proposition 2.7.9 *There exists a comonad* $! : \mathrm{Ext}(\mathbb{C}) \longrightarrow \mathrm{Ext}(\mathbb{C})_{\mathrm{dup}} \subseteq \mathrm{Ext}(\mathbb{C})$ *such that*

- *the counit written* $\mathsf{derelict} : !\,A \longrightarrow A$ *is an inclusion,*
- *if* $D \in \mathbb{C}_{\mathrm{dup}}$ *then* $!\,\mathcal{Y}(D) = \mathcal{Y}(D)$*,*
- $!\,\top = \top$*,*
- $!$ *preserves* $\otimes$ *up to equality, in fact both* $!(A \otimes B)$ *and* $!\,A \otimes !\,B$ *are equal to* $!(A \times B)$*,*
- $\mathcal{G}(!\,A) = \mathcal{G}(A)$*.*

Proof. The set $(!\,A)_X$ consists of those elements $a \in A_X$ for which there exists a witness consisting of a duplicable object $D \in \mathbb{C}_{\mathrm{dup}}$, a map $t : X \longrightarrow D$ and $\bar{a} \in A_D$ such that $a = \bar{a}[t]$. In particular, if $A = \mathcal{Y}(A)$ is representable then $!\,A_X$ consists of those maps into A which factor through a duplicable object. We remark that such factorisation is implicit in the notion of "affinations" in [2].

The morphism part of $!\,A$ is inherited from A.

The counit $\mathsf{derelict} : !\,A \longrightarrow A$ is simply the inclusion map $!\,A_X \subseteq A_X$.

The diagonal $\delta : !\,A \longrightarrow !\,A \otimes !\,A$ is obtained as follows. If $X \in \mathbb{C}$ and $a \in (!\,A)_X$ witnessed by $t : X \longrightarrow D$ and $\bar{a} \in A_D$ where $D \in \mathbb{C}_{\mathrm{dup}}$ and $a = \bar{a}[t]$ then $(a, a) \in !\,A \otimes !\,A$ as witnessed by $\delta_D \circ t : X \longrightarrow D \otimes D$ and $\bar{a}$, $\bar{a}$.

$$\begin{tikzcd}
a \in A_X & A_D \ni \bar{a} \arrow[l, "A_t"']
\end{tikzcd}$$

$$\begin{tikzcd}
X \arrow[r, "t"'] & D \arrow[r, "\delta"'] & D \otimes D
\end{tikzcd}$$

To see that $!\,\top = \top$ we notice that $\langle\rangle \in !\,\top_X$ is witnessed by $\langle\rangle : X \longrightarrow \top$ and $\langle\rangle \in \top_\top$.

For preservation of $\otimes$ suppose that $A, B \in \mathrm{Ext}(\mathbb{C})$. If $(a, b) \in !(A \times B)_X$ witnessed by $t : X \longrightarrow D$ and $\bar{a} \in A_D, \bar{b} \in B_D$ with $a = \bar{a}[t]$ and $b = \bar{b}[t]$ then $(a, b) \in (!\,A \otimes !\,B)_X$ can be witnessed by $\delta \circ t : X \longrightarrow D \otimes D, \mathrm{id}_D, \mathrm{id}_D, \bar{a}, \bar{b}$.

Similarly, $(a, b) \in !(A \otimes B)_X$ can be witnessed by $t : X \longrightarrow D$ and $\delta : D \longrightarrow D \otimes D$ and $\bar{a}, \bar{b}$.

So $!(A \times B) \subseteq !\,A \otimes !\,B$ and $!(A \times B) \subseteq !(A \otimes B)$.

For the inclusion $!\,A \otimes !\,B \subseteq !(A \times B)$ assume that $(a, b) \in (!\,A \otimes !\,B)_X$ witnessed by $t : X \longrightarrow U \otimes V$ and $s_1 : U \longrightarrow D_1$ and $s_2 : V \longrightarrow D_2$ and $\bar{a} \in A_{D_1}, \bar{b} \in B_{D_2}$. Then, since $D = D_1 \otimes D_2$ is duplicable, $(a, b) \in\, !(A \times B)_X$ can be witnessed by $(s_1 \otimes s_2) \circ t$ and $\bar{a}[\pi] \in A_D$, $\bar{b}[\pi'] \in B_D$.

$$\begin{tikzcd}
(a, b) \in A_X \times B_X & \bar{a}[s_1] \in A_U, \bar{b}[s_2] \in B_V & \bar{a} \in A_{D_1}, \bar{b} \in B_{D_2} \\
X \arrow[r, "t"'] & U \otimes V \arrow[r, "s_1 \otimes s_2"'] & D_1 \otimes D_2
\end{tikzcd}$$

Finally, $!(A \otimes B) \subseteq\, !(A \times B)$ is obvious by applying ! to the inclusion $A \otimes B \subseteq A \times B$.

The last part $\mathcal{G}(A) = \mathcal{G}(!\, A)$ follows from the fact that a global element $a : \top \longrightarrow A$ □

Chapter 3

The type system SLR

In this chapter we define and study the syntax and set-theoretic semantics of a lambda calculus with modal and linear function spaces. Apart from the definition itself the main result proved in this chapter is decidability of type checking by a syntax-directed procedure.

This result does not follow trivially by inverting the typing rules as we have only one abstraction construct for the different function spaces. That is to say, a functional abstraction $\lambda x{:}\,A.e$ can receive any of the three types $A{\multimap}B$, $A{\to}B$, $\Box A{\to}B$ according to how the variable x is used in the body of e.

We also describe an alternative system which boasts modal types $\Box(A)$ and $!(A)$ and in which the function spaces can be defined as $\Box(A){\multimap}B$ and $!(A){\multimap}B$. This system is slightly more expressive but does not provide any modality inference.

3.1 Types and subtyping

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

A, B	$::=$	$\mid\ \mathsf{N}$	natural numbers
		$\mid\ X$	type variable
		$\mid\ \mathsf{L}(A)$	lists over A
		$\mid\ \mathsf{T}(A)$	binary trees labelled over A
		$\mid\ \forall X.A$	polymorphic type
		$\mid\ A{\multimap}B$	linear function space
		$\mid\ \Box A{\to}B$	nonlinear, modal function space
		$\mid\ A{\to}B$	nonlinear, nonmodal function space
		$\mid\ A \times B$	cartesian product
		$\mid\ A \otimes B$	tensor product

The type formers $\times$ and $\otimes$ bind stronger than $\multimap$, $\to$, $\Box - \to -$. All these type formers associate to the right so $A{\multimap}B{\multimap}C$ reads $A{\multimap}(B{\multimap}C)$.

The type former $\forall X.A$ binds a free type variable X in A. A variable which is not bound by a $\forall$-quantifier is free. For example, the variables X, Y are free in $\forall Z \forall U.(X \times Z){\multimap}U{\multimap}Y$, whereas Z, U are bound. Types are understood as equivalence classes modulo renaming of bound variables, e.g., $\forall X.X{\multimap}\mathsf{N} \otimes X = \forall Y.Y{\multimap}\mathsf{N} \otimes Y$. If X is a free variable in the type expression B and A is another type then the *capture free substitution* of A for X in B written $B[A/X]$ is defined by first choosing a representative for B all of whose bound variables are different from the free variables in A and then replacing all occurrences of X in B by A. For example

$$(\forall X.Y{\multimap}X)[X \otimes \mathsf{N} \,/\, Y] = \forall X'.(X \otimes \mathsf{N}){\multimap}X'$$

We will now introduce a generic notation for the three function spaces which on the one hand facilitates the formulation of the typing rules and on the other hand allows us to introduce further modalities without having to change the typing rules and the type checking algorithm.

Definition 3.1.1 *An* aspect *is a pair* (l, m) *where* $l \in \{\text{linear}, \text{nonlinear}\}$ *and* $m \in \{$nonmodal, modal$\}$ *with the exception* (linear, modal) *which is not an aspect.*

The aspects are ordered componentwise by nonlinear <: linear *and* modal <: nonmodal.

Now we use the following generic notations for the three function spaces.

$$\begin{array}{llll} A \xrightarrow{a} B & \text{is} & A{\multimap}B & \text{when} \quad a = (\text{linear}, \text{nonmodal}) \\ A \xrightarrow{a} B & \text{is} & A{\to}B & \text{when} \quad a = (\text{nonlinear}, \text{nonmodal}) \\ A \xrightarrow{a} B & \text{is} & \Box A{\to}B & \text{when} \quad a = (\text{nonlinear}, \text{modal}) \end{array}$$

Our applications do not use the fourth theoretically possible aspect (linear, modal) because (semantically) modal use of a variable always implies nonlinear use. The typing rules, however, make perfect sense with modal, linear function space and it may be that subsequent use will be found for this aspect.

The ordering $a <: a'$ should be read as "a offers more capabilities than a'". So a variable which admits nonlinear use is worth more than a variable restricted to linear use, hence nonlinear <: linear; a variable allowing for modal use, i.e., being recursed over, is worth more than a variable which may not be used in this way. Since the aspects refer to the argument positions, the corresponding function spaces are ordered in the opposite direction. A function of type $\Box A{\to}B$ places a stronger requirement on its input than a function of type $A{\to}B$, so we have a subtyping $A{\to}B <: \Box A{\to}B$.

Definition 3.1.2 *The subtyping relation between types is defined inductively by the following rules.*

$$\frac{A' <: A \qquad B <: B' \qquad a' <: a}{A \xrightarrow{a} B <: A' \xrightarrow{a'} B'} \qquad \textit{(}\text{S-A}{\small\text{RR}}\textit{)}$$

$$\frac{A <: A' \qquad B <: B'}{A \times B <: A' \times B'} \qquad \textit{(}\text{S-P}{\small\text{ROD}}\textit{)}$$

$$\frac{A <: A' \qquad B <: B'}{A \otimes B <: A' \otimes B'} \qquad (\text{S-Tens})$$

$$\frac{A <: B}{\forall X.A <: \forall X.B} \qquad (\text{S-All})$$

$$\frac{A <: B}{\mathsf{N}{\rightarrow}A <: \mathsf{N}{\multimap}B} \qquad (\text{S-Ax})$$

$$A <: A \qquad (\text{S-Refl})$$

Notice the contravariance of the first rule w.r.t. the ordering of aspects so that, e.g., $A{\multimap}B <: A{\rightarrow}B$ and $A{\rightarrow}B <: \Box A{\rightarrow}B$.

The subtyping rule for function types contains (using S-Refl) the special cases $A{\multimap}B <: A{\rightarrow}B <: \Box A{\rightarrow}B$ and $A{\multimap}B <: \Box A{\multimap}B <: \Box A{\rightarrow}B$.

The axiom S-Ax is optional. It expresses that variables of type N can be duplicated without sacrificing linearity. We give one interpretation which validates it and another which does not, but in exchange allows for stronger basic functions such as multiplication as a basic function of type $\mathsf{N}{\multimap}\mathsf{N}{\multimap}\mathsf{N}$.

Remark 3.1.3 *We remark at this point that without axiom* S-Ax *the distinction between* $A{\rightarrow}B$ *and* $\Box A{\rightarrow}B$ *is immaterial because we could then identify the two function spaces while preserving typability. Obviously, such identification would be unsound with* S-Ax *as we would then identify* $\Box\mathsf{N}{\rightarrow}A$ *with* $\mathsf{N}{\multimap}A$.

The following properties of subtyping are direct.

Proposition 3.1.4 – *The following rule of transitivity is admissible*

$$\frac{A <: B \qquad B <: C}{A <: C} \qquad (\text{S-Trans})$$

– *Subtyping is decidable by a syntax directed procedure.*

3.2 Terms and typing rules

The expressions of SLR are given by the grammar

$$\begin{array}{rlll}
e ::= & x & & \text{(variable)} \\
| & (e_1\ e_2) & & \text{(application)} \\
| & \lambda x{:}\,A.e & & \text{(abstraction)} \\
| & \Lambda X.e & & \text{(type abstraction)} \\
| & e[A] & & \text{(type application)} \\
| & \langle e_1, e_2 \rangle & & \text{(pairing w.r.t. } \times\text{)} \\
| & e.1 & & \text{first projection} \\
| & e.2 & & \text{second projection} \\
| & e_1 \otimes e_2 & & \text{(pairing w.r.t. } \otimes\text{)} \\
| & \mathsf{let}\, e_1{=}x \otimes y \,\mathsf{in}\, e_2 & & \text{(}\otimes\text{-elimination)} \\
| & \mathsf{let}\, e_1{=}x \,\mathsf{in}\, y & & \text{untyped abbreviation} \\
| & c & & \text{(constants)}
\end{array}$$

Here x ranges over a countable set of variables and c ranges over the following set of constants with types as indicated:

$$\begin{array}{l}
0 : \mathsf{N} \\
\mathsf{S}_0, \mathsf{S}_1 : \mathsf{N} \multimap \mathsf{N} \\
\mathsf{nil} : \forall A.\mathsf{L}(A) \\
\mathsf{cons} : \forall A.A \multimap \mathsf{L}(A) \multimap \mathsf{L}(A) \\
\mathsf{leaf} : \forall A.A \multimap \mathsf{T}(A) \\
\mathsf{node} : \forall A.A \multimap \mathsf{T}(A) \multimap \mathsf{T}(A) \\
\mathsf{rec}^{\mathsf{N}} : \forall X.X \multimap (\Box\mathsf{N} \to X \multimap X) \to \Box\mathsf{N} \to X \\
\mathsf{case}^{\mathsf{N}} : \forall X.(X \times (\mathsf{N} \multimap X) \times (\mathsf{N} \multimap X)) \multimap \mathsf{N} \\
\mathsf{rec}^{\mathsf{L}} : \forall A.\forall X.X \multimap (\Box A \to \Box\mathsf{L}(A) \to \Box\mathsf{L}(A) \to X \multimap X) \to \Box\mathsf{L}(A) \to X \\
\mathsf{case}^{\mathsf{L}} : \forall A.\forall X.X \times (A \multimap \mathsf{L}(A) \multimap X) \multimap \mathsf{L}(A) \multimap X \\
\mathsf{rec}^{\mathsf{T}} : \forall A.\forall X.(\Box A \to X) \to (\Box A \to \Box\mathsf{T}(A) \to \Box\mathsf{T}(A) \to X \multimap X \multimap X) \to \Box\mathsf{T}(A) \to X \\
\mathsf{case}^{\mathsf{T}} : \forall A \forall X.((A \multimap X) \times (A \multimap \mathsf{T}(A) \multimap \mathsf{T}(A) \multimap X)) \multimap \mathsf{T}(A) \multimap X
\end{array}$$

Remark 3.2.1 *We remark that without nonlinear function space we would have to give the following slightly weaker types to the recursors.*

$$\begin{array}{l}
\mathsf{rec}^{\mathsf{N}} : \forall X.X \multimap \Box(\Box\mathsf{N} \to X \multimap X) \to \Box\mathsf{N} \multimap X \\
\mathsf{rec}^{\mathsf{L}} : \forall A.\forall X.X \multimap \Box(\Box A \to \Box\mathsf{L}(A) \to \Box\mathsf{L}(A) \to X \multimap X) \to \Box\mathsf{L}(A) \to X \\
\mathsf{rec}^{\mathsf{T}} : \forall A.\forall X.(\Box A \to X) \to \Box(\Box A \to \Box\mathsf{T}(A) \to \Box\mathsf{T}(A) \to X \multimap X \multimap X) \to \Box\mathsf{T}(A) \to X
\end{array}$$

This typing rules out step functions containing safe parameters of type N *whereas such are allowed with the finer typing involving nonlinear function space using rule* S-Ax*. As said before, without rule* S-Ax *the distinction between the two function spaces can be given up.*

3.2.1 Contexts

A typing judgement $e : A$ read "e has type A" is made relative to a *typing context* which records typing and aspect assumptions for variables. For example, if e has type B in a context in which variable x has type A and linear, nonmodal aspect then $\lambda x\colon A.e$ has type $A \multimap B$ in the context with the assumption on x removed. This typically happens if x appears at most once in e and not as a subterm of an argument to a nonlinear function. If e can be given type B only under the stronger assumption that x has modal, nonlinear aspect then the weaker type $\Box A \to B$ will be assigned to the abstraction $\lambda x\colon A.e$. Typically, this happens if x appears as a subterm of a term of type $\mathsf{N}, \mathsf{L}(A), \mathsf{T}(A)$ which is recursed on, i.e., appears as last argument to a recursor.

Definition 3.2.2 *A* context *is a partial function from term variables to pairs of aspects and types. It is typically written as a list of bindings of the form $x \overset{a}{:} A$. 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.*

A context Γ is nonlinear if all its bindings are of nonlinear aspect.

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.*

Notation. We write $x \hat{:} A$ for the binding $x \overset{a}{:} A$ where a is $(\text{linear}, \text{nonmodal})$, i.e., the maximal aspect. For type A and aspect a we define an aspect $\mathrm{adj}(a, A)$ as follows:

$$\begin{array}{l}
\mathrm{adj}((\text{nonlinear}, \text{nonmodal}), \mathsf{N}) = (\text{linear}, \text{nonmodal}) \\
\mathrm{adj}(a, A) = a \text{ in all other cases}
\end{array}$$

This notation allows us to subsume rules S-Arr and S-Ax under the following more general rule.

$$\frac{A' <: A \qquad B <: B' \qquad a' <: \mathrm{adj}(a, A)}{A \xrightarrow{a} B <: A' \xrightarrow{a'} B'} \qquad (\text{S-Arr-Ax})$$

For context Γ and aspect a we write $\Gamma <: a$ to mean that $\Gamma((x)) <: a$ for every variable $x \in \mathrm{dom}(\Gamma)$.

3.2.2 Typing rules

Definition 3.2.3 *The subset of* safe types *is defined by the following grammar.*

$$A, B ::= \mathsf{N} \mid \mathsf{L}(A) \mid \mathsf{T}(A) \mid X \mid A \multimap B \mid A \times B \mid A \otimes B$$

The safe types are those which may appear as result types of primitive recursive definitions. By definition, type variables range over safe types; accordingly polymorphic application will be restricted to safe types. This means that since our constructor functions for lists and trees are polymorphic we can only form lists and trees over safe type of entries. Semantically, we could justify lists and trees over non safe type of entries, but these would not be allowed as result types of recursive definitions.

The typing relation $\Gamma \vdash e : A$ between contexts, expressions, and types is defined inductively by the following rules. 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. The typing rules described here are the affine ones from [18].

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \tag{T-Var}$$

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B} \tag{T-Sub}$$

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

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A \xrightarrow{a} B \qquad \Gamma, \Delta_2 \vdash e_2 : A \qquad \Gamma \text{ nonlinear} \qquad \Gamma, \Delta_2 <: a}{\Gamma, \Delta_1, \Delta_2 \vdash (e_1\ e_2) : B} \tag{T-Arr-E}$$

$$\frac{\Gamma \vdash e : A \quad X \text{ not free in } \Gamma}{\Gamma \vdash \Lambda X.e : \forall X.A} \tag{T-All-I}$$

$$\frac{\Gamma \vdash e : \forall X.A \quad S \text{ safe}}{\Gamma \vdash e[S] : A[S/X]} \tag{T-All-E}$$

$$\frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2} \tag{T-Prod-I}$$

$$\frac{\Gamma \vdash e : A_1 \times A_2 \quad i \in \{1, 2\}}{\Gamma \vdash e.i : A_i} \tag{T-Prod-E}$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A_1 \quad \Gamma, \Delta_2 \vdash e_2 : A_2 \quad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2} \tag{T-Tens-I}$$

$$\frac{\begin{array}{c}\Gamma, \Delta_1 \vdash e_1 : A_1 \otimes A_2 \qquad \Gamma, \Delta_2, x \overset{a_1}{:} A_1, y \overset{a_2}{:} A_2 \vdash e_2 : B \\ \Gamma \text{ nonlinear} \qquad \Delta_1 \leq a_1 \wedge a_2\end{array}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1 {=} x \otimes y \,\mathsf{in}\, e_2 : B} \qquad (\text{T-Tens-E})$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A \qquad \Gamma, \Delta_2, x \overset{a}{:} A \vdash e_2 : B \qquad \Gamma \text{ nonlinear} \qquad \Delta_1 \leq a}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1 {=} x \,\mathsf{in}\, e_2 : B} \qquad (\text{T-Let})$$

$$\frac{c : A}{\Gamma \vdash c : A} \qquad (\text{T-Const})$$

3.2.3 Examples

The function sq. 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 $a = (\text{nonmodal}, \text{linear})$, $\Gamma = \Delta_1 = \emptyset$, $\Delta_2 = y{:}\,\mathsf{N}, q{:}\,\mathsf{N}$. Similarly, we get $y{:}\,\mathsf{N}, q{:}\,\mathsf{N} \vdash \mathsf{S}_0(\mathsf{S}_0\, q) : \mathsf{N}$. Therefore, $t =_{\text{def}} \lambda y{:}\,\mathsf{N}.\lambda q{:}\,\mathsf{N}.\mathsf{S}_0(\mathsf{S}_0\, q) : \mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$, and $t : \Box\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ by T-Sub. Now $x{:}\,\Box\mathsf{N} \vdash \mathsf{rec}^{\mathsf{N}}\, x\, 1\, t : \mathsf{N}$ by T-Const and three instances of T-Arr-E; the first with $a = (\text{modal}, \text{nonlinear})$. Finally, T-Arr-I gives $\text{sq} =_{\text{def}} \lambda x{:}\,\mathsf{N}.\mathsf{rec}^{\mathsf{N}}\, x\, 1\, t : \Box\mathsf{N} \to \mathsf{N}$.

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

However, as expected, we cannot define an exponentially growing function by

$$\exp =_{\text{def}} \lambda x{:}\,\mathsf{N}.\mathsf{rec}^{\mathsf{N}}\ 1\ (\lambda z{:}\,\mathsf{N}.\text{sq})\ x$$

as typechecking exp in the empty context fails. The reason is that the principal type of the step function $\lambda z{:}\,\mathsf{N}.\text{sq}$ is $\mathsf{N} \multimap \Box\mathsf{N} \to \mathsf{N}$ which is a subtype of $\Box\mathsf{N} \to \Box\mathsf{N} \to \mathsf{N}$, but not of $\Box\mathsf{N} \to \mathsf{N} \multimap \mathsf{N}$ as would be required in order to typecheck the application.

Iterator from recursor We can define

$$\text{it}^{\mathsf{N}} =_{\text{def}} \Lambda X.\lambda g{:}\, X.\lambda h{:}\, X \multimap X.\lambda n{:}\,\mathsf{N}.\mathsf{rec}^{\mathsf{N}}\ g\ (\lambda x{:}\,\mathsf{N}.\lambda y{:}\, X.h\ x)$$

and we obtain the typing

$$\text{it}^{\mathsf{N}} : \forall X.X \multimap (X \multimap X) \to \Box\mathsf{N} \to X$$

If-then-else We can define test for zero by

$$\text{ifz}_1 =_{\text{def}} \Lambda X.\lambda n{:}\,\mathsf{N}.\lambda x{:}\, X.\lambda y{:}\, X.\mathsf{case}^{\mathsf{N}}[X]\langle x, \langle \lambda n{:}\,\mathsf{N}.y, \lambda n{:}\,\mathsf{N}.y\rangle\rangle$$

and we obtain the typing

$$\text{ifz}_1 : \forall X.\mathsf{N} \multimap X \multimap X \multimap X$$

Notice that although y appears twice in the body of the case-expression it is counted linearly according to rule T-PROD-E. This does not happen with our newly defined if-expression: The function

$$F =_{\mathrm{def}} \lambda f\colon \mathsf{N}{\multimap}\mathsf{N}.\lambda x\colon \mathsf{N}.\mathrm{ifz}_1[\mathsf{N}]\ x\ (f\ 0)\ (f\ 1)$$

gets the type

$$F : (\mathsf{N}{\multimap}\mathsf{N}){\to}\mathsf{N}{\multimap}\mathsf{N}$$

according to rule T-ARR-E. Without expanding the definition of ifz_1 and merely looking at its type the system does not recognise that in fact f will be applied at most once. If we desire such behaviour we should use the cartesian product in the typing of if-then-else:

$$\begin{aligned}&\mathrm{ifz}_2 = \Lambda X.\lambda n\colon \mathsf{N}.\lambda p\colon X \times X.\mathsf{case}^{\mathsf{N}}[X]\langle p.1, \langle \lambda n\colon \mathsf{N}.p.2, \lambda n\colon \mathsf{N}.p.2\rangle\rangle :\\ &\qquad \forall X.\mathsf{N}{\multimap}(X \times X){\multimap}X\end{aligned}$$

Then we obtain the more refined typing for F:

$$F =_{\mathrm{def}} \lambda f\colon \mathsf{N}{\multimap}\mathsf{N}.\lambda x\colon \mathsf{N}.\mathrm{ifz}_2[\mathsf{N}]\ x\ \langle f\ 0, f\ 1\rangle : (\mathsf{N}{\multimap}\mathsf{N}){\multimap}\mathsf{N}{\multimap}\mathsf{N}$$

We emphasise how the presence of polymorphism and cartesian product types enables such general definitions. In the earlier system described in [18] only ifz_1 could be formulated and only for fixed result type X.

We also remark that by rule T-ALL-E the result types of case distinctions are bound to be safe although this is not strictly necessary from the point of view of soundness. Indeed, case distinction for arbitrary result type can be defined schematically; for example, if $A = \Box\mathsf{N}{\to}\mathsf{N}$ then we can define

$$\mathrm{ifz}^A = \lambda n\colon \mathsf{N}.\lambda p\colon A \times A.\lambda m\colon \mathsf{N}.\mathrm{ifz}_2[\mathsf{N}]\ n\ \langle p.1\ n, p.2\ n\rangle : \mathsf{N}{\multimap}(A \times A){\multimap}A$$

There would be no principal obstacle against introducing a second kind of type variables ranging over arbitrary types, but we have refrained from doing so in order to keep the syntax manageable.

Ordinary safe recursion. Let us see how ordinary safe recursion can be recovered in SLR. Suppose we are given $g : \Box\mathsf{N}^m{\to}\mathsf{N}^n{\multimap}\mathsf{N}$ and $h : \Box\mathsf{N}^{m+1}{\to}\mathsf{N}^{n+1}{\multimap}\mathsf{N}$ and we wish to define $f : \Box\mathsf{N}^{m+1}{\to}\mathsf{N}^n{\multimap}\mathsf{N}$ such that, semantically,

$$\begin{aligned}&f(0, \vec{x}; \vec{y}) = g(\vec{x}; \vec{y})\\ &f(x, \vec{x}; \vec{y}) = h(x, \vec{x}; f(\left\lfloor \tfrac{x}{2} \right\rfloor, \vec{x}; \vec{y}), \vec{y})\end{aligned}$$

Let Γ be the context which binds the x-variables to N with aspect (modal, nonlinear) and the y-variables to N with aspect (nonmodal, nonlinear) and write

$$\begin{aligned}&\mathrm{base} =_{\mathrm{def}} g(\vec{x}; \vec{y})\\ &\mathrm{step} =_{\mathrm{def}} \lambda x\colon \mathsf{N}.\lambda y\colon \mathsf{N}.h(x, \vec{x}; y, \vec{y})\end{aligned}$$

for base case and step function of the above definition. We have

$$\begin{array}{l}\Gamma \vdash \text{base} : \mathsf{N} \\ \Gamma \vdash \text{step} : \Box\mathsf{N}\to\mathsf{N}\multimap\mathsf{N}\end{array}$$

Since Γ is a nonlinear context, the side conditions to rule T-Arr-E are satisfied in the application $\mathsf{rec}^{\mathsf{N}}[\mathsf{N}]$ base step and we have

$$\Gamma \vdash \mathsf{rec}^{\mathsf{N}}[\mathsf{N}] \text{ base step} : \Box\mathsf{N}\to\mathsf{N}$$

Finally, rule T-Arr-I gives

$$\emptyset \vdash \lambda x{:}\,\mathsf{N}.\lambda\vec{x}{:}\,\mathsf{N}^n.\lambda\vec{y}{:}\,\mathsf{N}.\mathsf{rec}^{\mathsf{N}}[\mathsf{N}] \text{ base step} : \Box\mathsf{N}^{m+1}\to\mathsf{N}^n\to\mathsf{N}$$

and we finally get the desired typing $\Box\mathsf{N}^{m+1}\to\mathsf{N}^n\multimap\mathsf{N}$ using S-Ax and T-Sub. We remark that the type inference algorithm implicit in the proof of Theorem 3.3.7 below performs all these steps automatically, and in the implementation we merely type in the term and its type will be computed.

Bounded recursion on notation With higher result type it is relatively straightforward to translate bounded recursion on notation into safe recursion and thus to show that all *PTIME*-functions are definable in SLR.

We have the following correspondence [3].

Proposition 3.2.4 *If $f(\vec{x})$ is definable in $\mathcal{F}$ then we can find terms $u_f : \Box\mathsf{N}^n\to\mathsf{N}$ and $v_f : \Box\mathsf{N}\to\mathsf{N}^n\multimap\mathsf{N}$ such that (in the set-theoretic semantics)*

$$v_f(t;\vec{x}) = f(\vec{x})$$

whenever $t \geq u_f(\vec{x};)$.

Proof. By induction on a definition of f in $\mathcal{F}$. The crucial step is bounded recursion on notation. Suppose that

$$\begin{array}{l}f(0,\vec{x}) = g(\vec{x}) \\ f(x,\vec{x}) = h(x,\vec{x},f(\left\lfloor\frac{x}{2}\right\rfloor,\vec{x}))\end{array}$$

and that $f(x,\vec{x}) \leq k(x,\vec{x})$.

By the induction hypothesis we have

$$\begin{array}{l}f(0,\vec{x}) = v_g(t;\vec{x}) \\ f(x,\vec{x}) = v_h(t;x,f(\left\lfloor\frac{x}{2}\right\rfloor,\vec{x}),\vec{x})\end{array}$$

provided that

$$t \geq \max(u_g(\vec{x}),u_h(x,v_k(u_k(x,\vec{x};);x,\vec{x}),\vec{x};))$$

Now define v_f by safe recursion with result type $\mathsf{N}\multimap\mathsf{N}$ as

$$\begin{array}{l}
v_f(0;x,\vec{x}) = 0\\
v_f(t;x,\vec{x}) = \textbf{if } x{=}0\\
\qquad\textbf{then } v_g(t;\vec{x})\\
\qquad\textbf{else } v_h(t;x,v_f(\left\lfloor \frac{t}{2}\right\rfloor;\left\lfloor \frac{x}{2}\right\rfloor,\vec{x}),\vec{x})
\end{array}$$

Induction then shows that $f(x,\vec{x}) = v_f(t;x,\vec{x})$ whenever

$$t \geq \max(u_g(\vec{x}), u_h(x, v_k(u_k(x,\vec{x};);x,\vec{x}),\vec{x};)) + |x|$$

Defining v_f sufficiently large so as to majorise the right hand side then yields the result.

□

We remark that the above definition would not have been possible without rule S-Ax. If we wanted to show that even the system without duplication at ground type defines all *PTIME*-functions it seems that we would have to go down one level of abstraction and encode polynomially bounded Turing machines directly. The details remain unexplored.

Tree recursion Here is an iterator for trees which does not provide access to the recursion variable itself but only to results of recursive calls.

$$\begin{array}{l}
\mathrm{it}^{\mathsf{T}} =_{\mathrm{def}} \Lambda A.\Lambda X.\lambda g\colon \Box A{\rightarrow}X.\lambda h\colon \Box A{\rightarrow}X{\multimap}X{\multimap}X.\\
\qquad \mathsf{rec}^{\mathsf{T}}[A][X]\ g\ (\lambda a\colon A.\lambda l\colon \mathsf{T}(A).\lambda r\colon \mathsf{T}(A).h\ a) :\\
\qquad\quad \forall A.\forall X.(\Box A{\rightarrow}X){\rightarrow}(\Box A{\rightarrow}X{\multimap}X{\multimap}X){\rightarrow}\Box\mathsf{T}(A){\rightarrow}X
\end{array}$$

Here is a definition of a function of type $\forall A.\Box\mathsf{T}(A){\rightarrow}\mathsf{T}(A)$ which hereditarily swaps left and right subtrees:

$$\mathrm{treerev} =_{\mathrm{def}} \Lambda A.\mathrm{It}^{\mathsf{T}}[A][\mathsf{T}(A)]\ (\mathsf{leaf}[A])\ (\lambda a\colon A.\lambda l\colon \mathsf{T}(A).\lambda r\colon \mathsf{T}(A).\mathsf{node}[A]\ a\ r\ l)$$

We have

$$\begin{array}{l}
\mathrm{treerev}(\mathsf{leaf}[A]\ a) = \mathsf{leaf}[A]\ a\\
\mathrm{treerev}(\mathsf{node}[A]\ a\ l\ r) = \mathsf{node}[A]\ a\ (\mathrm{treerev}\ r)\ (\mathrm{treerev}\ l)
\end{array}$$

Insertion sort Finally, we give a sugared version of the familiar insertion sort algorithm. Suppose that we have a function $\leq\colon \Box\mathsf{N}{\rightarrow}\mathsf{N}{\multimap}\mathsf{N}$. We define an insertion function $\mathsf{insert} : \Box\mathsf{L}(\mathsf{N}){\rightarrow}\Box\mathsf{N}{\rightarrow}\ \mathsf{L}(\mathsf{N}){\multimap}\mathsf{L}(\mathsf{N})$ such that $\mathsf{insert}(l,a,l')$ inserts a into l' in the correct place assuming that l is longer than l' and that l' is already sorted. The extra parameter l is used to "drive" the recursion enabling us to use l' in a linear way:

$$\begin{array}{l}
\mathsf{insert}([\ ],a,l') = [\ ]\\
\mathsf{insert}(x :: y, a, a' :: l') = \mathsf{if}\ a \leq a'\\
\quad \mathsf{then}\ a :: a' :: l\\
\quad \mathsf{else}\ a' :: \mathsf{insert}(y,a,l')
\end{array}$$

This definition can be formalised using the higher-typed recursion operator

$$\mathsf{rec}^{\mathsf{L}}[\mathsf{N}][\mathsf{L}(\mathsf{N})\multimap\mathsf{L}(\mathsf{N})]$$

The sorting function of type $\Box\mathsf{L}(\mathsf{N})\to\mathsf{L}(\mathsf{N})$ is then defined by

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

Here $\mathsf{rec}^{\mathsf{L}}[\mathsf{N}][\mathsf{L}(\mathsf{N})]$ has been used.

The correctness of this code hinges on the fact that $\mathsf{sort}(l)$ is not longer than l, hence the particular instance of insert behaves correctly.

The usual recursive definition of insert without extra parameter would yield a function of type $\Box\mathsf{L}(\mathsf{N})\to\Box\mathsf{N}\to\mathsf{L}(\mathsf{N})$ which cannot be iterated due to its modal type.

This use of extra parameters to drive recursions is intrinsic to the pattern of safe recursion and already appears in Bellantoni-Cook's proof that all polynomial time functions are definable in their first-order system. The inconvenience caused by this necessity is somewhat palliated by the presence of higher result types as can be seen from the above definition of insert which would be difficult to define with $\mathsf{rec}^{\mathsf{L}}[\mathsf{N}][\mathsf{L}(\mathsf{N})]$ alone.

Splitting and quicksort. In order to illustrate the use of tensor products we show how to split a list in two components. Let test : $N\multimap N$ be a variable. We define a function

$$\text{split} : \Box\mathsf{N}\to\mathsf{L}(\mathsf{N})\multimap\mathsf{L}(\mathsf{N})\otimes\mathsf{L}(\mathsf{N})$$

such that when

$$\text{split}(n, l) = (\text{yes}, \text{no})$$

then *yes* is a list containing those elements a of l for which $\text{test}(a) = 0$ and *no* contains the other ones provided that the length of l is smaller or equal to $|n|$. Otherwise the behaviour of split is undefined. The formal definition of split is as follows.

$$\begin{array}{l}
\text{split} = \lambda\text{test}\colon\mathsf{N}\multimap\mathsf{N}.\text{it}^{\mathsf{N}}[\mathsf{L}(\mathsf{N})\multimap\mathsf{L}(\mathsf{N})\otimes\mathsf{L}(\mathsf{N})]\\
\qquad\quad(\lambda l\colon\mathsf{L}(\mathsf{N}).\mathsf{nil}[\mathsf{N}]\otimes\mathsf{nil}[\mathsf{N}])\\
\qquad\quad(\lambda\text{spl}\colon\mathsf{L}(\mathsf{N})\multimap\mathsf{L}(\mathsf{N})\otimes\mathsf{L}(\mathsf{N}).\lambda l\colon\mathsf{L}(\mathsf{N}).\\
\qquad\qquad\mathsf{case}^{\mathsf{L}}[\mathsf{L}(\mathsf{N})\otimes\mathsf{L}(\mathsf{N})]\\
\qquad\qquad\quad(\mathsf{nil}[\mathsf{N}]\otimes\mathsf{nil}[\mathsf{N}],\\
\qquad\qquad\quad\lambda\text{hd}\colon\mathsf{N}.\lambda\text{tl}\colon\mathsf{L}(\mathsf{N}).\\
\qquad\qquad\qquad\mathsf{let}\,\text{spl}\ l'{=}\text{yes}\otimes\text{no}\,\mathsf{in}\\
\qquad\qquad\qquad\quad\text{ifz}[\mathsf{L}(\mathsf{N})\otimes\mathsf{L}(\mathsf{N})]\ (\text{test}\ x)\\
\qquad\qquad\qquad\qquad(\mathsf{cons}[\mathsf{N}]\ x\ \text{yes}\otimes\text{no},\\
\qquad\qquad\qquad\qquad\text{yes}\otimes\mathsf{cons}[\mathsf{N}]\ x\ \text{no})))\ l
\end{array}$$

From this splitting function one can define a sorting function based on the quicksort algorithm. It seems difficult to do this without using the tensor product.

3.3 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 3.3.1 *The following rules are admissible.*

$$\frac{\Gamma \vdash e : A}{\Gamma, \Delta \vdash e : A} \qquad \textit{(}\text{T-Weak}\textit{)}$$

$$\frac{\Gamma, x \overset{a}{:} A \vdash e : C \qquad B <: A \qquad b \leq a}{\Gamma, x \overset{b}{:} B \vdash e : C} \qquad \textit{(}\text{T-Weak'}\textit{)}$$

$$\frac{\Gamma, x \overset{a}{:} A \vdash e : C \qquad x \textit{ not free in } e}{\Gamma \vdash e : C} \qquad \textit{(}\text{T-Strength}\textit{)}$$

$$\frac{\Gamma, \Delta_1, x \overset{a}{:} A \vdash e_1 : B \quad \Gamma, \Delta_2 \vdash e_2 : B \qquad \Gamma \textit{ nonlinear} \qquad \Gamma, \Delta_2 <: a}{\Gamma, \Delta_1, \Delta_2 \vdash e_1[e_2/x] : B}$$

$\textit{(}$T-Subst$\textit{)}$

Proposition 3.3.2 (Generation of typing) *i. If $\Gamma \vdash x : A$ then $x \in \mathrm{dom}(\Gamma)$ and $\Gamma(x) <: A$.*

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

iii. 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. $x \overset{a'}{:} X$ in Γ', Δ_1 then $a' <: \mathrm{adj}(a, A)$.

iv. If $\Gamma \vdash \Lambda X.e : B$ then $B = \forall X.C$ for some X not free in Γ and $\Gamma \vdash e : C$.

v. If $\Gamma \vdash e[S] : C$ then there exists S, B with S safe such that $\Gamma \vdash e : \forall X.B$ and $B[S/X] <: C$ and X not free in Γ.

vi. If $\Gamma \vdash \langle e_1, e_2 \rangle : C$ then $C = C_1 \times C_2$ and $\Gamma \vdash e_i : C_i$.

vii. If $\Gamma \vdash e.i : C_i$ for $i = 1$ or $i = 2$ then we can find C_{3-i} such that $\Gamma \vdash e : C_1 \times C_2$.

viii. If $\Gamma \vdash e_1 \otimes e_2 : C_1 \otimes C_2$ then we can find a decomposition $\Gamma = \Gamma', \Delta_1, \Delta_2$ with Γ' nonlinear and $\Gamma', \Delta_i \vdash e_i : C_i$.

ix. *If* $\Gamma \vdash \mathsf{let}\, e_1{=}x\otimes y\, \mathsf{in}\, e_2 : C$ *then we can find a decomposition* $\Gamma = \Gamma', \Delta_1, \Delta_2$ *with* Γ' *nonlinear and types* A_1, A_2 *such that* $\Gamma', \Delta_1 \vdash e_1 : A_1 \otimes A_2$ *and* $\Gamma', \Delta_2, x\overset{a_1}{:}A_1, x\overset{a_2}{:}A_2 \vdash e_2 : C$ *and* $\Delta_1(x) \leq a_1$*,* $\Delta_1(x) \leq (a_2)$ *for each* $x \in \mathrm{dom}\,\Delta_1$*.*

x. *If* $\Gamma \vdash \mathsf{let}\, e_1{=}x\, \mathsf{in}\, e_2 : C$ *then we can find a decomposition* $\Gamma = \Gamma', \Delta_1, \Delta_2$ *with* Γ' *nonlinear and a types* A *such that* $\Gamma', \Delta_1 \vdash e_1 : A$ *and* $\Gamma', \Delta_2, x\overset{a}{:}A \vdash e_2 : C$ *and* $\Delta_1(x) \leq a$ *for each* $x \in \mathrm{dom}\,\Delta_1$*.*

xi. *If* $\Gamma \vdash c : A$ *then* $\tau(c) <: A$*.*

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 3.3.3 *Let* Δ, Δ' *be contexts. We write* $\Delta \ll \Delta'$ *to mean that* $\Delta((x)) \leq \Delta'((x))$ *for each* $x : \mathrm{dom}(\Delta')$*.*

The following is immediate.

Proposition 3.3.4 *If* $\Delta \ll \Delta'$ *and* $\Delta' \vdash e : A$ *then* $\Delta \vdash e : A$*.*

Definition 3.3.5 *Let* Γ *be a context and* e *be an expression. Say that* typechecking e under Γ fails *if* $\Delta \not\vdash e : A$ *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.

Lemma 3.3.6 *Suppose that* $(\Gamma, x{:}\,A\,,\, e)$ *has a principal solution* (Δ, C) *and assume that* $B <: A$*.*

If $x \notin \mathrm{dom}(\Delta)$ *then* $(\Gamma, x{:}\,B\,,\, e)$ *has principal solution* (Δ, C)*. If* $\Delta = \Delta', x\overset{a}{:}A$ *then* $(\Gamma, x{:}\,B\,,\, e)$ *has principal solution* $(\Theta, x\overset{b}{:}B\,,\, D)$ *for some* $D <: C$*,* $\Delta' \ll \Theta$ *and* $a \leq b$*.*

Proof. The first case is direct from T-Strength. For the second case we first notice that by T-Weak' and T-Strength the problem $(\Gamma, x\colon B\,,\, e)$ must have a solution of the form $(\Theta, x\overset{b}{:}B\,,\, D)$. The assumption together with rule Weak' gives $\Delta', x\overset{a}{:}B \vdash e : C$. Principality then yields $\Delta' \ll \Theta$ and $a \leq b$ and $C \leq D$. □

Theorem 3.3.7 *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 $e = c$**.** 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 $\Gamma \ll \emptyset$.

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 $\Delta = \Delta', x\overset{a}{:}X$. Now, since Δ is a subcontext of Γ we must have $X = \Delta(x) = \Gamma(x) \leq A$, hence $\Delta \ll \{x\hat{:}\Gamma(x)\}$.

Case $e = \lambda x\colon A.e'$**.** 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 $(\Gamma, x\colon A\,,\, e')$ fails. Assume that this is not the case and let (Δ, B) be a principal solution of $(\Gamma, x\colon A, e')$. We have two cases to distinguish.

i. $x \notin \mathrm{dom}(\Delta)$. Then $(\Delta, A \multimap B)$ is a principal solution.

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

In case i we have $\Delta, x\colon A \vdash e' : B$ by T-Weak and thus $\Delta \vdash e : A \multimap B$. In case ii $\Delta_1 \vdash e : A \xrightarrow{a} B$ 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 $\Delta' \ll \Delta$. Thus, $A \multimap B <: A \xrightarrow{a'} B' <: C$. 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. If A_2 is not a subtype of B then typechecking $e_1\ e_2$ obviously fails.

We claim that the principal solution is (Δ, C) where Δ is the largest (w.r.t. $\ll$) context satisfying the following requirements.

i. $\mathrm{dom}(\Delta) = \mathrm{dom}(\Delta_1) \cup \mathrm{dom}(\Delta_2)$

ii. $x \in \mathrm{dom}(\Delta_1)$ implies $\Delta((x)) \leq \Delta_1((x))$

iii. $x \in \mathrm{dom}(\Delta_2)$ implies $\Delta((x)) \leq \Delta_2((x))$

iv. $x \in \mathrm{dom}(\Delta_2)$ implies $\Delta((x)) \leq \mathrm{adj}(B, a)$

v. $x \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$ 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 (v) we know that Θ is nonlinear. Furthermore, if $x \overset{a'}{:} X$ in Θ, Λ_2 then $a' \leq \mathrm{adj}(a, B)$ by (iv) 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 generation of subtyping we get $a' \leq \mathrm{adj}(a, B)$ and $C <: C'$.

So to show $\Delta' \ll \Delta$ it is enough to show that Δ' restricted to $\mathrm{dom}(\Delta)$ meets requirements (ii–v) 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 $e = \Lambda X.e'$. By bound renaming we may assume w.l.o.g. that type variable X does not occur in Γ. Let (Δ, A) be a principal solution of problem (Γ, e'). By assumption and the fact that Δ is a subcontext of Γ we know that X does not occur in Δ and so $\Delta \vdash e : \forall X.A$. Principality follows directly from the induction hypothesis and Prop. 3.3.2.

Case $e = e'[S]$**.** If S is not safe then there is no solution. For (Γ, e) to be solvable it is necessary that (Γ, e') has a principal solution of the form $(\Delta, \forall X.A)$. In this case, $(\Delta, A[S/X])$ is a principal solution for (Γ, e). Indeed, if $\Delta' \vdash e : C$ then by generation of typing we must have $\Delta' \vdash e' : \forall X.B$ and $B[S/X] <: C$. The induction hypothesis gives $\Delta \ll \Delta'$ and $\forall X.A <: \forall X.B$. Therefore, $A[S/X] <: B[S/X] <: C$.

Case $e = (e_1, e_2)$. Let (Δ_1, A_1) and (Δ_2, A_2) be the principal solutions of (Γ, e_1) and (Γ, e_2) respectively. If none exist then typechecking (Γ, e) fails. Otherwise, we claim that the principal solution is $(\Delta, A_1 \times A_2)$ where Δ is the greatest lower bound w.r.t. $\ll$ of Δ_1 and Δ_2.

It is clear from the definition that $\Delta \vdash (e_1, e_2) : A_1 \times A_2$. On the other hand, if $\Delta' \vdash e : A'$ then $A' = A'_1 \times A'_2$ and $\Delta' \vdash e_i : A'_i$. The induction hypothesis yields $\Delta' \ll \Delta_1$ and $\Delta' \ll \Delta_2$. Hence, $\Delta' \ll \Delta$ by construction of Δ. The i.h. also yields $A_i <: A'_i$ hence $A_1 \times A_2 <: A'$ by rule S-Prod.

Case $e = e'.i$. For (Γ, e) to typecheck it is necessary that $(\Gamma, e'.i)$ has a principal solution of the form $(\Delta, A_1 \times A_2)$. In this case (Δ, A_i) is a principal solution for (Γ, e).

Case $e = e_1 \otimes e_2$. Let (Δ_i, A_i) be principal solutions for (Γ, e_i). If none exist typechecking fails. We claim that the principal solution for (Γ, e) is given by $(\Delta, A_1 \otimes A_2)$ where Δ is the largest w.r.t. $\ll$ subcontext of Γ satisfying

i. $\mathrm{dom}(\Delta) = \mathrm{dom}(D_1) \cup \mathrm{dom}(\Delta_2)$,

ii. $\Delta((x)) \leq \Delta_i((x))$ for $i = 1, 2$ and $x \in \mathrm{dom}(\Delta_i)$,

iii. $\Delta((x))$ nonlinear when $x \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$.

It is clear from Prop. 3.3.4 and T-Tens-I that $\Delta \vdash e_1 \otimes e_2 : A_1 \otimes A_2$. Conversely, if $\Delta' \vdash e_1 \otimes e_2 : A'_1 \otimes A'_2$ then $\Delta' = \Theta, \Lambda_1, \Lambda_2$ where Θ is nonlinear and $\Theta, \Lambda_i \vdash e_i : A'_i$. The induction hypothesis gives $A_i \ll A'_i$ and $\Theta, \Lambda_i \ll \Delta_i$. So, Δ' meets requirements i–iii and hence $\Delta' \ll \Delta$.

Case $e = \mathsf{let}\, e_1{=}x \,\mathsf{in}\, e_2$. We may assume that $x \notin \mathrm{dom}(\Gamma)$. If (Γ, e) has a solution at all then generation of typing gives us $\Theta^0, \Lambda^0_1 \vdash e_1 : A^0$ and $\Theta^0, \Lambda^0_2, x \overset{a^0}{:} A^0 \vdash e_2 : B^0$ for subcontexts $\Theta^0, \Lambda^0_1, \Lambda^0_2$ of Γ and arbitrary types and aspect A^0, B^0, a^0. Therefore, the typechecking problem (Γ, e_1) must have a solution, say (Δ_1, A) where $A \leq A^0$ and $\Theta^0, \Lambda^0_1 \ll \Delta_1$. Rule Weak' then gives $\Theta^0, \Lambda^0_2, x \overset{a^0}{:} A \vdash e_2 : B^0$, so $(\Gamma, x{:}\, A\, , e_2)$ must also have a principal solution, say (Δ_2, C). We claim that the principal solution of (Γ, e) is (Δ, C) where Δ is the largest w.r.t. $\ll$ subcontext of Γ satisfying the following requirements.

i. $\mathrm{dom}(\Delta) = \mathrm{dom}(\Delta_1) \cup \mathrm{dom}(\Delta_2) \setminus \{x\}$,

ii. $\Delta((y))$ nonlinear for $y \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$

iii. $\Delta((y)) \leq \Delta_1((y))$ for all $y \in \mathrm{dom}(\Delta_1)$,

iv. $\Delta((y)) \leq \Delta_2((y))$ for all $y \in \mathrm{dom}(\Delta_2) \setminus \{x\}$,

v. if $x \in \mathrm{dom}(\Delta_2)$ then $\Delta((y)) \leq \Delta_2((x))$ for all $y \in \mathrm{dom}(\Delta_2) \setminus \{x\}$.

It follows from Weak and Subst that $\Delta \vdash e : C$. Principality is immediate from the analysis above and the induction hypothesis.

Case $e = \mathsf{let}\, e_1{=}x_1 \otimes x_2 \,\mathsf{in}\, e_2$**.** Analogous to the previous case. □

Corollary 3.3.8 *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 3.3.9 *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. □

3.4 Comparison with other systems

The modal part of SLR builds upon the modal lambda calculi developed by Pfenning et al. [11, 34]. The new feature of SLR as compared to these latter systems is the absence of special term formers related to modality. Aspects of functions are inferred automatically; explicit coercions are avoided by way of a subtyping.

The idea of restricting modalities to the argument position of function types arises Pfenning and Cervesato's *Linear Logical Framework* [6] and also in Plotkin's formulation of linear system F [37]. These systems do not use subtyping and have two kinds of abstraction and application for linear and nonlinear function spaces, respectively.

A problem we have not studied for SLR is automatic inference of type annotations to functional abstractions and polymorphic applications. Concerning annotations in abstractions, an obstacle against a straightforward generalisation of Hindley-Milner type inference to a system like SLR is that untyped terms need not have principal types. For example, the term $e = \lambda x.\lambda f.fx$ can be given any of the following three incomparable type schemes:

$$\begin{aligned}&e : A{\multimap}(A{\multimap}B){\multimap}B\\&e : A{\to}(A{\multimap}B){\multimap}B\\&e : \Box A{\to}(\Box A{\to}B){\multimap}B\end{aligned}$$

These three are subsumed under the following general pattern:

$$e : A \xrightarrow{a} (A \xrightarrow{a}) \multimap B$$

where a is an arbitrary aspect. For purely linear lambda calculus Wadler *et. al.* [42, 41] have described an inference algorithm which assigns type schemes containing variable aspects (called *use variables* in loc. cit.). It could be possible to extend their algorithm to SLR; the details have not been explored.

Automatic inference of type applications would for example allow one to omit type arguments to constructor functions like $\mathsf{cons}, \mathsf{leaf}$ and to the recursors as well as derived patterns like iterators.

Inference of polymorphic type applications has been shown to be undecidable by Pfenning [32] for system F. However, since the polymorphism in SLR is predicative (type arguments to polymorphic function cannot contain polymorphic types as those are not safe.) Pfenning's result does not apply directly and there is still some hope.

Another very promising approach has recently been put forward by Pierce and Turner [35]. In their system the user is required to give the type of a term to be checked and also to give the type of all local definitions, i.e., to write $\mathsf{let}\, x{=}e_1 : A \,\mathsf{in}\, e_2$. In exchange, type annotations to functional abstractions and type applications can be omitted.

3.5 Set-theoretic semantics

The calculus SLR has an intended set-theoretic interpretation which in particular associates a function $\mathbb{N} \longrightarrow \mathbb{N}$ to a closed term of type $\Box\mathsf{N}{\to}\mathsf{N}$. The central result of this thesis is that all these functions are computable in polynomial time.

The purpose of this set-theoretic interpretation is merely to specify the meaning of the terms. In order to effectively compute these meanings one has two choices.

Either one views the terms as ordinary functional programs by disregarding modality and linearity information and evaluates them using standard evaluation techniques for functional programs. This is what we have done in our prototype implementation. Notice that we will *not* prove that such evaluation can be performed in polynomial time and, at least in general this will not be true since already expansion of higher-order definitions (i.e., $\beta\eta$-reduction in simply-typed lambda calculus) takes superexponential time.

A more ambitious and arguably more efficient way of computing meanings would be to extract a compiler from the soundness proof to be given and use the latter to generate *PTIME* code from first-order terms in SLR. A practical implementation of this procedure is intended, but no concrete results are available at the time of writing.

Notation. If A, B are sets we write $A \times B$ and $A{\to}B$ for their cartesian product and function space. If A is a set let $\mathsf{L}(A)$ stand for the set of finite lists over A constructed by nil and cons. If A is a set let $\mathsf{T}(A)$ be the set of binary A-labelled trees over A inductively defined by $\mathsf{leaf}(a) \in \mathsf{T}(A)$ when $a \in A$ and $\mathsf{node}(a, l, r) \in \mathsf{T}(A)$ when $a \in A$ and $l, r \in \mathsf{T}(A)$. Let $\mathcal{U}$ be a set which contains $\mathbb{N}$ and is closed under $\times, \to, \mathsf{L}, \mathsf{T}$.

If η is a partial function from type variables to $\mathcal{U}$ and A is a type then we define a set $[\![A]\!]\eta$ by

$$
\begin{aligned}
&[\![X]\!]\eta = \eta(X)\\
&[\![\mathsf{N}]\!]\eta = \mathbb{N}\\
&[\![\mathsf{L}(A)]\!]\eta = \mathsf{L}([\![A]\!]\eta)\\
&[\![\mathsf{T}(A)]\!]\eta = \mathsf{T}([\![A]\!]\eta)\\
&[\![A \xrightarrow{a} B]\!]\eta = [\![A]\!]\eta \to [\![B]\!]\eta\\
&[\![\forall X.A]\!]\eta = \textstyle\prod_{B\in\mathcal{U}}[\![A]\!]\eta[X \mapsto B]\\
&[\![A \times B]\!]\eta = [\![A \otimes B]\!]\eta = [\![A]\!]\eta \times [\![B]\!]\eta
\end{aligned}
$$

If A is a closed type then $[\![A]\!]\eta$ is independent of η and we thus write $[\![A]\!]$ in this case. Notice that the interpretation of a safe type always lies in $\mathcal{U}$. Also notice that if $A <: B$ then $[\![A]\!]\eta = [\![B]\!]\eta$.

To each constant $c : A$ we associate an element $[\![c]\!] \in [\![A]\!]$ by the clauses given in the introduction.

The interpretation of terms is w.r.t. an partial function η which maps type variables to elements of $\mathcal{U}$ and term variables to arbitrary values:

$$
\begin{array}{lcl}
[\![x]\!]\eta & = & \eta(x)\\
[\![\lambda x\colon A.e]\!] & = & \lambda v\in [\![A]\!].[\![e]\!]\eta[x\mapsto v]\\
[\![e_1\, e_2]\!]\eta & = & [\![e_1]\!]\eta([\![e_2]\!]\eta)\\
[\![\Lambda X.e]\!]\eta & = & \lambda A\in\mathcal{U}.[\![e]\!]\eta[X \mapsto A]\\
[\![e[A]]\!]\eta & = & [\![e]\!]\eta([\![A]\!]\eta)\\
[\![\langle e_1, e_2\rangle]\!]\eta & = & ([\![e_1]\!]\eta, [\![e_2]\!]\eta)\\
[\![e.i]\!]\eta & = & v_i \text{ where } [\![e]\!]\eta = (v_1, v_2)\\
[\![e_1 \otimes e_2]\!]\eta & = & ([\![e_1]\!]\eta, [\![e_2]\!]\eta)\\
[\![\mathsf{let}\, e_1 = x \otimes y\, \mathsf{in}\, e_2]\!]\eta & = & [\![e_2]\!]\eta[x \mapsto v_1, y \mapsto v_2] \text{ where } [\![e_1]\!]\eta = (v_1, v_2)\\
[\![c]\!]\eta & = & [\![c]\!]
\end{array}
$$

The purpose of this set-theoretic semantics is to specify the meaning of SLR terms. It allows us to do without any notion of term rewriting or evaluation. Of course, by directing the defining equations of the recursors one obtains a normalising rewrite system which computes the set-theoretic meaning of first-order functions. However, there is no reason why such a rewrite system should terminate in polynomial time. In order to obtain polynomial time algorithms from SLR-terms one must rather study the soundness proof we give and from it extract a compiler which transforms SLR-programs of first-order type into polynomial time algorithms. That this is possible in principle follows from the fact that our soundness proof is constructive; a practical implementation, however, must await further work.

3.6 Reduction

We do not define a reduction relation for SLR-terms but will rather specify their meaning by a set-theoretic model. Nevertheless, it seems worth discussing possible reduction rules for

SLR. If the rule S-Ax is omitted from SLR then the obvious reduction relation based on β-reduction for function spaces ($(\lambda x\colon A.e)e' \rightsquigarrow e[e'/x]$) and rules like $\mathsf{let}\, e_1{\otimes}e_2{=}x_1{\otimes}x_2 \,\mathsf{in}\, e_3 \rightsquigarrow e_3[e_1/x_1, e_2/x_2]$ satisfies the subject reduction property, i.e., if $e \rightsquigarrow e'$ and $\Gamma \vdash e : A$ then $\Gamma \vdash e' : A$. This follows directly from T-Subst. However, with rule S-Ax subject reduction fails for the following reason: If $g : \mathsf{N}{\multimap}\mathsf{N}{\multimap}\mathsf{N}$ is a variable then

$$e = \lambda x\colon \mathsf{N}.gxx$$

gets the type $\mathsf{N}{\to}\mathsf{N}$ by rule T-Arr-I and hence also the type $\mathsf{N}{\multimap}\mathsf{N}$ by rule S-Ax and T-Sub. Therefore, the term

$$e' = \lambda f\colon \mathsf{N}{\multimap}\mathsf{N}.e(f0)$$

gets the type $(\mathsf{N}{\multimap}\mathsf{N}){\multimap}\mathsf{N}$ by T-Arr-E and T-Arr-I. However, the reduct

$$e'' = \lambda f\colon \mathsf{N}{\multimap}\mathsf{N}.g(f0)(f0)$$

gets the type $(\mathsf{N}{\multimap}\mathsf{N}){\to}\mathsf{N}$ because f appears twice in the body of the abstraction. Yet, $(\mathsf{N}{\multimap}\mathsf{N}){\to}\mathsf{N}$ is not a subtype of $(\mathsf{N}{\multimap}\mathsf{N}){\multimap}\mathsf{N}$.

A type system closed under reduction would necessarily have to be as complex as normalisation of SLR terms, thus basically require to run programs in order to type check them. The reason is as follows. Let e_1, e_2 be two term of type N containing a free variable $f : \mathsf{N}{\multimap}\mathsf{N}$, i.e., $f\colon \mathsf{N}{\multimap}\mathsf{N} \vdash e_1, e_2 : \mathsf{N}$. As before, let $g\colon \mathsf{N}{\multimap}\mathsf{N}{\multimap}\mathsf{N}$ be a variable and consider the term

$$e = \lambda f\colon \mathsf{N}{\multimap}\mathsf{N}.ge_1e_2$$

If there exists a term e_3 such that e_3 reduces to both e_1 and e_2 then

$$\lambda f\colon \mathsf{N}{\multimap}\mathsf{N}.(\lambda x\colon \mathsf{N}.gxx)e_3 \rightsquigarrow^* e$$

so e would have to be given type $(\mathsf{N}{\multimap}\mathsf{N}){\multimap}\mathsf{N}$. Otherwise, it should get type $(\mathsf{N}{\multimap}\mathsf{N}){\to}\mathsf{N}$. Whether such common ancestor e_3 exists can be decided by normalising e assuming that reduction is confluent.

This is exactly what Bellantoni, Niggl, Schwichtenberg do in order to enforce that their type system be closed under reduction. They use the term *affination* for the collection of several occurrences of a variable stemming from the same subterm of type N. The reader may wish to compare this with the definition of the comonad ! in 2.7.6.3.

Since we do not rely on reduction in the proof of soundness we do not need this.

3.7 Alternative syntax with modal types

It is sometimes convenient to have a unary type former $\square(-)$ and to define the modal function space from the linear one as

$$\square A{\to}B =_{\mathrm{def}} \square(A){\multimap}B$$

Similarly, following Girard, we may introduce a modality $!(-)$ for duplication and define the nonlinear function space as

$$A \to B =_{\mathrm{def}} !(A) \multimap B$$

The reason why we have not done this in the official version is that we were aiming for a system which does not use any new term formers or constants in conjunction with the modalities and can automatically infer the best possible typing of a given term. This is not possible with unary modalities as shown by the following example: The term

$$e =_{\mathrm{def}} \lambda f \colon A \multimap B . \lambda x \colon A . f x$$

can be given any of the following incomparable types

$$\begin{array}{l} \Box(A \multimap B) \multimap \Box A \multimap \Box B \\ !(A \multimap B) \multimap \, ! \, A \multimap \, ! \, B \\ (A \multimap \, ! \, B) \multimap A \multimap \, ! \, B \\ (A \multimap \Box B) \multimap A \multimap \Box B \end{array}$$

and a few more. So, a flexible yet relatively simple type system like the one for SLR does not seem possible for such a system. It could be that a system with “aspect variables” in the style of Hindley-Milner type variables could lead to a viable solution; again details remain to be studied.

If we are more modest and refrain from all inference then we can have a very simple system with modalities and single linear function space. Such system will fail to have the subject reduction property, but since our semantic soundness proof does not need any notion of reduction on terms this need not concern us.

The types of this system, to be called $\lambda^{\Box !}$ for the moment are given by the grammar

$$A ::= X \mid A_1 \multimap A_2 \mid \; !(A) \mid \Box(A) \mid \forall X.A \mid A_1 \otimes A_2 \mid A_1 \times A_2 \mid \mathsf{N} \mid \mathsf{L}(A) \mid \mathsf{T}(A)$$

The terms are those of SLR and in addition we have new term formers referring to the modalities:

$$e ::= \dots \mid \Box(e) \mid \; !(e) \mid \mathrm{unbox}(e) \mid \mathrm{derelict}(e)$$

as well as a constant

$$\delta : \mathsf{N} \multimap !(\mathsf{N})$$

allowing us to duplicate terms of ground type.

Contexts are sets of bindings $x \colon A$ with disjoint variables like in the simply-typed lambda calculus. There are no aspects and no subtyping.

A context Γ is called *modal* if $\Gamma(x)$ is of the form $\Box(A)$ for every variable $x \in \mathrm{dom}(\Gamma)$. It is called *nonlinear* if $\Gamma(x)$ is of the form $\Box(A)$ or $!(A)$ in this case. So, a modal context is in particular nonlinear.

The constants of this system are the same as the ones for SLR with their types amended according to the above definition.

The typing rules are as follows.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \tag{T-Var}$$

$$\frac{\Gamma \vdash e : A \quad \Gamma \text{ modal}}{\Gamma, \Delta \vdash \Box(e) : \Box(A)} \tag{T-Box-I}$$

$$\frac{\Gamma \vdash e : \Box(A)}{\Gamma \vdash \mathsf{unbox}(e) : !(A)} \tag{T-Box-E}$$

$$\frac{\Gamma \vdash e : !(A)}{\Gamma \vdash \mathsf{derelict}(e) : A} \tag{T-Bang-E}$$

$$\frac{\Gamma \vdash e : A \quad \Gamma \text{ nonlinear}}{\Gamma \vdash !(e) : !(A)} \tag{T-Bang-I}$$

$$\frac{\Gamma, x\colon A \vdash e : B}{\Gamma \vdash \lambda x\colon A.e : A \multimap B} \tag{T-Arr-I}$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A \xrightarrow{a} B \qquad \Gamma, \Delta_2 \vdash e_2 : A \qquad\qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash (e_1\ e_2) : B} \tag{T-Arr-E}$$

$$\frac{\Gamma \vdash e : A \quad X \text{ not free in } \Gamma}{\Gamma \vdash \Lambda X.e : \forall X.A} \tag{T-All-I}$$

$$\frac{\Gamma \vdash e : \forall X.A \quad S \text{ safe}}{\Gamma \vdash e[S] : A[S/X]} \tag{T-All-E}$$

$$\frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2} \tag{T-Prod-I}$$

$$\frac{\Gamma \vdash e : A_1 \times A_2 \quad i \in \{1, 2\}}{\Gamma \vdash e.i : A_i} \tag{T-Prod-E}$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A_1 \quad \Gamma, \Delta_2 \vdash e_2 : A_2 \quad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2} \qquad (\text{T-}\textsc{Tens}\text{-I})$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A_1 \otimes A_2 \qquad \Gamma, \Delta_2, x{:}\, A_1, x{:}\, A_2 \vdash e_2 : B \qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1{=}x \otimes y \,\mathsf{in}\, e_2 : B} \qquad (\text{T-}\textsc{Tens}\text{-E})$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A \qquad \Gamma, \Delta_2, x{:}\, A \vdash e_2 : B \qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1{=}x \,\mathsf{in}\, e_2 : B} \qquad (\text{T-}\textsc{Let})$$

$$\frac{c : A}{\Gamma \vdash c : A} \qquad (\text{T-}\textsc{Const})$$

We will show later in Section 4.4.2 how our semantic soundness proof also covers this system with only superficial amendments.

As said before, the disadvantage of this system is that due to the presence of the extra term formers our programs will be more verbose than those in SLR. For example, the squaring function will be defined as

$$\mathrm{sq} =_{\mathrm{def}} \lambda x{:}\, \Box(\mathsf{N}).\mathsf{rec}^{\mathsf{N}}[\mathsf{N}]\, 0\; !(\lambda x{:}\, \Box(\mathsf{N}).\lambda y{:}\, \mathsf{N}.\mathsf{S}_0(\mathsf{S}_0(y))) : \Box(\mathsf{N}) \multimap \mathsf{N}$$

If we want to apply squaring to a constant, say $2 : \mathsf{N}$ then we have to write

$$\mathrm{sq}(\Box(2)) : \mathsf{N}$$

If we want to apply squaring again, then we have to do this each time:

$$\mathrm{sq}(\Box(\mathrm{sq}(\Box(2))))$$

We notice that typing in this system is not closed under well-typed substitution, i.e., an analogue of rule T-Subst is not admissible. The reason is that rule T-Box-I does not obviously commute with substitutions; an explicit counterexample and a detailed discussion can be found in [34].

3.7.0.1 Failure of subject reduction for $\lambda^{\Box\,!}$

We remark that typing in system $\lambda^{\Box\,!}$ is not stable under substitution thus is not preserved by untyped reduction. To see this, consider the context $\Delta =_{\mathrm{def}} f : X \multimap \Box(Y), x : X$ and define $\mathrm{d} =_{\mathrm{def}} \lambda y{:}\, \Box(Y).\Box(y)$. We have

$$\Delta \vdash \mathrm{d} : \Box(Y) \multimap \Box(\Box(Y))$$

by rule T-Box-I with $\Gamma = \emptyset$. We also have $\Delta \vdash (f\ x) : \square(Y)$ by rule T-Arr-E. Therefore,

$$\Delta \vdash \mathrm{d}(f\ x) : \square(\square(Y))$$

However,

$$\Delta \not\vdash \square(f\ x) : \square(\square(Y))$$

because the side condition to rule T-Box-I is not satisfied in this case.

It should be possible to construct a system along the lines of Pfenning and Wong's systems [34], see also Section 2.6.12.4. Since our soundness proof does not rely on reduction we have not explored this avenue.

3.8 Other variations of SLR

For expository reasons we have defined SLR with specific base types and constants built in. It should be clear, however, that the meta-theoretic development is independent of the choice of constants and also relatively flexible w.r.t. to the addition of new types. Other base types can be straightforwardly added via type variables. If one wants to add a new type former with associated operations on terms one has to give subtyping and typing rules and slightly extend some of the proofs.

Another line of extension is the addition of aspects. Indeed, the typing rules and proofs have been formulated in such a way that the aspects can be replaced by an arbitrary $\wedge$-semilattice A together with a monotone function into the two-element poset nonlinear $\leq$ linear. We will briefly discuss an application of this in Chapter 5.

Chapter 4

Semantics of SLR

This chapter constitutes the heart of this thesis. It is devoted to the construction of models for SLR which enable us to deduce that all definable functions are polynomial time computable.

In Section 4.1 we review the concept of a *BCK*-algebra which corresponds to affine linear lambda calculus in the same way as combinatory or *SK*-algebra corresponds to ordinary lambda calculus.

The main result of this section is the construction of a particular *BCK*-algebra consisting of *PTIME* algorithm satisfying a certain linear growth restriction.

In Section 4.2 we study the category $\mathcal{H}$ of modest realisability sets over this algebra. The absence of diagonalisation or an S-combinator in H is reflected by the fact that this category is not cartesian closed but rather affine linear closed in the sense of Section 2.7.

This category serves as interpretation of safe types and functions between them. In particular, we show how to interpret in $\mathcal{H}$ natural numbers and trees along with their constructor functions and operators for case distinction.

Next, we define a notion of *PTIME*-function between such realisability sets which generalises the morphisms in $\mathcal{H}$. Every morphism is a *PTIME*-function but not vice versa. We then identify recursion patterns for natural numbers as operators on *PTIME*-functions. E.g., if h is a *PTIME*-function from $P \otimes \mathsf{N}$ to $A \multimap A$ then so will be $\mathsf{rec}(h)$ defined by $\mathsf{rec}(h)(p, 0, a) = a$ and $\mathsf{rec}(h)(p, x, a) = h(p, x, \mathsf{rec}(h)(p, \left\lfloor \frac{x}{2} \right\rfloor, a))$.

In order to obtain a fully-fledged model of SLR we group the *PTIME*-functions together with the $\mathcal{H}$-morphisms to form a new category $\mathcal{H}^{\square}$ in which objects are pairs $X = (X_0, X_1)$ of H-sets and where a morphism from (X_0, X_1) to (Y_0, Y_1) is a pair $f = (f_0, f_1)$ of *PTIME*-functions (in a suitably defined sense) $f_0 : X_0 \longrightarrow Y_0$ and $f_1 : X_0 \longrightarrow (X_1 \multimap Y_1)$ where $X_1 \multimap Y_1$ is linear function space in $\mathcal{H}$.

The two-zoned structure of this category is reminiscent of the two zones in Bellantoni-Cook's original first-order function algebra and also of the structure of the category used in [15] to model a higher-type extension thereof.

Like $\mathcal{H}$ this category is a well-pointed ALC. However, linear function spaces only exist between safe objects, i.e., those of the from $(\top, X)$.

The desired model of SLR is then obtained as the functor category $\mathrm{Ext}(\mathcal{H}^{\square})$ of extensional presheaves over $\mathcal{H}^{\Box}$. This category supports all linear function spaces as well as modalities $\Box$ and ! referring to modality and nonlinearity. On representable presheaves we have $\Box(A_0, A_1) = (A_0 \otimes A_1, \top)$.

The category $\mathcal{H}$ has the property that only finite objects are duplicable; in particular the object of integers is not. Thus, it provides a model only for SLR without rule S-Ax; in exchange it supports stronger basic functions such as multiplication as a constant of type $\mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$.

To accommodate the equation $\mathsf{N} \multimap A = \mathsf{N} \to A$ we construct another *BCK*-algebra M which has the property that the ensuing category of realisability sets does have a duplicable object of integers. The previous setup has been sufficiently modular so that no further work beyond the construction of this algebra needs to be done.

Finally, by using a logical relation we conclude the desired result that all functions definable in SLR are polynomial time computable.

4.1 A *BCK*-algebra of *PTIME*-functions

It is well-known that the concept of functional abstraction found in typed and untyped lambda calculi can be captured algebraically using the notion of combinatory algebra. Such a combinatory algebra is given by a set A, a binary operation written as juxtaposition for application associating to the left and constants $S, K \in A$ such that the following equations are valid.

$$\begin{aligned} &Sxyz = (xz)(yz) \\ &Kxy = x \end{aligned}$$

The main result about combinatory algebra is that if t is a term built up using application from variables and the constants S, K then whenever x is a variable we can effectively produce a term $\lambda x.t$ which does not contain x and which has the property that $(\lambda x.t)x = t$ is a logical consequence of the above two equations. The definition of $\lambda x.t$ is by induction on the structure of t the crucial step being $\lambda x.t_1 t_2 = S(\lambda x.t_1)(\lambda x.t_2)$.

A similar algebraisation also exists for the (affine) linear lambda calculus in the form of *BCK*-algebras.

Definition 4.1.1 *A BCK-algebra is given by a set A, a binary operation (written as juxtaposition) associating to the left and three constants $B, C, K \in A$ such that the following equations are valid.*

$$\begin{aligned} &Kxy = x \\ &Bxyz = x(yz) \\ &Cxyz = xzy \end{aligned}$$

To rule out the trivial example we additionally require that $|A| > 1$.

An identity combinator I with $Ix = x$ can be defined as $I = CKK$[1]. Notice that since $x \mapsto Kx$ is injective but not onto every *BCK*-algebra must be infinite.

Lemma 4.1.2 *Let A be a BCK-algebra and t be a term in the language of BCK-algebras and containing constants from A. If free variable x appears at most once in t then we can find a term $\lambda x.t$ not containing x such that for every other term s the equation $(\lambda x.t)s = t[s/x]$ is valid in A, i.e., all ground instances of the equation hold in A.*

Proof. By induction on the structure of t. If t does not contain x then we put $\lambda x.t = Kt$. If $t = x$ then $\lambda x.t = I$. If $t = t_1 t_2$ and x does not appear in t_1 then $\lambda x.t = Bt_1(\lambda x.t_2)$. If $t = t_1 t_2$ and x does not appear in t_2 then $\lambda x.t = C(\lambda x.t_1)t_2$. □

The reason why *BCK*-algebras are interesting in the context of polynomial time computation is that all functions which are computable in time $O(|x|^p)$ for a fixed p and which are bounded by a linear function with unit slope can be organised as a *BCK*-algebra as we will now show.

4.1.1 A new length measure

Before giving the construction we must overcome the messy technical problem that there does not exist an injective function $\langle \cdot, \cdot \rangle : \mathbb{N}^2 \longrightarrow \mathbb{N}$ such that

$$|\langle x, y \rangle| \leq |x| + |y| + c$$

for a fixed constant c.[2]

In order to circumvent the technical difficulties arising from this we will use another length measure defined using the pairing function from Lemma 2.1.1. Recall that in this lemma we defined a pairing function $\langle -, - \rangle : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ such that $|\langle x, y \rangle| = |x| + |y| + 2||y|| + 3$ and an injection $\mathsf{num} : \mathbb{N} \longrightarrow \mathbb{N}$ such that $|\mathsf{num}(x)| = |x| + 1$, as well as projections and characteristic functions for the images of the two functions.

Definition 4.1.3 *The length function $\ell(x)$ is defined recursively by*

$$\begin{aligned} &\ell(\mathsf{num}(x)) = |x| + 1 \\ &\ell(\langle x, y \rangle) = \ell(x) + \ell(y) + 3 \\ &\ell(x) = |x|, \textit{ otherwise} \end{aligned}$$

We have $\ell(0) = 0$.

Lemma 4.1.4 *The following inequalities hold for every $x \in \mathbb{N}$.*

$$|x| \geq \ell(x) \geq |x|/(1 + ||x||)$$

[1] Thanks to Andrzej Filinski for pointing this out to me.

[2] Thanks to John Longley for a short proof of this.

Proof. By course-of-values induction on x. If x is not of the form $\langle u, v\rangle$ then the result is direct. So assume the latter and that the inequalities have been established for u and v.

Now, $|\langle u, v\rangle| \geq |u| + |v| + 3 \overset{\mathsf{IH}}{\geq} \ell(u) + \ell(v) + 3 = \ell(\langle u, v\rangle)$ so the first inequality holds. For the second one we calculate as follows.

$$\begin{array}{ll} & |\langle u, v\rangle| \\ = & |u| + |v| + 2||v|| + 3 \\ \leq & \ell(u)(1 + ||u||) + \ell(v)(1 + ||v||) + 2||v|| + 3 \qquad \text{IH} \\ \leq & (\ell(u) + \ell(v))(1 + ||\langle u, v\rangle||) + 2||\langle u, v\rangle|| + 3 \\ \leq & (\ell(u) + \ell(v) + 3)(1 + ||\langle u, v\rangle||) \end{array}$$

□

It follows from the first inequality that if a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is computable in time $O(\ell(x)^n)$ then it is in particular computable in time $O(|x|^n)$. Conversely, if $f : \mathbb{N} \longrightarrow \mathbb{N}$ is computable in time $O(|x|^n)$ then the function $\lambda x.f(\mathsf{num}(x))$ is computable in time $O(\ell(x)^n)$.

The second inequality shows that, in this case f itself is computable in time $O(\ell(x)^{n+1})$ as $|x|/(1 + ||x||) \geq |x|^{1-1/n}$ for large x. so $|x| = O(\ell(x)^{1+\varepsilon})$ for each $\varepsilon > 0$.

4.1.2 Construction of H_p

Recall the notation concerning Turing machine computations from Section 2.1 Let $p > 2$ be a fixed integer.

Definition 4.1.5 *A computation $\{e\}(x)$ is called* short *(w.r.t. p) if it terminates in not more than $d(\ell(e) + \ell(x))^p$ steps where $d = \ell(e) + \ell(x) - \ell(\{e\}(x))$.*

An algorithm e is called short if $\{e\}(x)$ is short for all x.

The difference d between $\ell(e)+\ell(x)$ and $\ell(\{e\}(x))$ is called the *defect* of computation $\{e\}(x)$. Notice that if $\{e\}(x)$ is short then it must have nonzero defect so $\ell(\{e\}(x)) < \ell(e) + \ell(x)$ for every x. Also notice that if f is computable in time $O(\ell(x)^p)$ and $\ell(f(x)) = \ell(x) + O(1)$ then by padding (inserting comments) we can obtain a short algorithm e for f.

Proposition 4.1.6 *There exists a function $\mathsf{app} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ and a constant γ such that*

- *$\mathsf{app}(e, x)$ is computable in time $d(\ell(e) + \ell(x) + \gamma)^p$ where $d = \ell(e) + \ell(x) - \ell(\mathsf{app}(e, x))$*
- *If $\{e\}(x)$ is short then $\{e\}(x) = \mathrm{app}(e, x)$.*

Proof. The following algorithm satisfies the required specification:

```
read $e, x$
$\mathit{conf} := \mathsf{init}(e, x)$
while $t < (\ell(e) + \ell(x))^{p+1}$ and $\mathsf{term}(\mathit{conf}) \neq 0$ do begin
   $\mathit{conf} := \mathbf{step}(\mathit{conf}) \quad t := t + 1$
end
$r := \mathsf{out}(\mathit{conf})$
$d := \lceil t/(\ell(e) + \ell(x))^p \rceil$
if $\mathsf{term}(\mathit{conf}) = 0$ and $\ell(r) + d \leq \ell(e) + \ell(x)$
   then write $r$
   else write $0$
```

The algorithm performs at most $d(\ell(e) + \ell(x))^p$ simulation steps where $d = \ell(e) + \ell(x) - \ell(\mathsf{app}(e, x))$ regardless of which branch of the conditional is taken. If the computation $\{e\}(x)$ is short then the dedicated time suffices to finish it so that $\{e\}(x) = \mathsf{app}(e, x)$ in this case. The total running time of $\mathsf{app}(e, x)$ consists of the simulation steps plus a certain number of steps needed for initialisation, some arithmetic, and moving around intermediate results. The number of these steps is linear in the binary length of the input, thus quadratic in $\ell(e) + \ell(x)$ by Lemma 4.1.4 and thus can be accounted for by an appropriate choice of the constant γ in view of $p > 2$. □

We will henceforth write $\mathsf{app}(e, x)$ as ex where appropriate.

Before embarking on the proof that the above defined application function induces a *BCK*-algebra structure on the natural numbers we will try to motivate the notion of short computation and in particular the role of the defect.

The starting point is that we want to construct an untyped universe of such computations which can later on serve as step functions in safe recursions. Certainly, these algorithms should themselves be polynomial time computable. Moreover, in order that their use as step functions does not lead beyond polynomial time we must require a growth restriction of the form $\ell(f(x)) = \ell(x) + O(1)$. Remember that if, e.g., $\ell(f(x)) = 2\ell(x)$ then $\ell(f^{|y|}(x)) = 2^{|y|}\ell(x)$, thus we quit polynomial time.

Next, in order that application itself be polynomial time computable we must restrict to algorithms running in time $O(\ell(x)^p)$ for some fixed p.

Next, we have to look at the coefficient of the leading term of the polynomial governing the runtime. If our algorithms have running time $d\ell(x)^p + O(\ell(x)^{p-1})$ for arbitrary d then the application runs in time $O(\ell(x)^{p+1})$ thus, again, application is not among the algorithms considered and accordingly, no higher-order functions are possible. If we also bound the coefficient of the leading term and only consider algorithms running in $d\ell(x)^p + O(\ell(x)^{p-1})$ for fixed d and p then once again we lose closure under composition, as in order to evaluate $f(g(x))$ we must evaluate both f and g requiring time $2d\ell(x)^p + O(\ell(x)^{p-1})$. The solution is to couple runtime and output size via the defect so that if $u := g(x)$ is large (which would mean that the second computation $f(u)$ runs longer) then this is made up for by a shorter runtime of $g(x)$. We shall now see formally that this works and in particular that a B-combinator is definable.

Lemma 4.1.7 (Parametrisation) *For every e there exists an algorithm e' such that $e\langle x, y\rangle = e'xy$.*

Proof. Let e_0 be the following algorithm:

```
read $x$
write "read $y$; write app$(e, \langle `x, y\rangle)$"
```

Here $`x$ refers to the actual value of x rather than the name x itself. Now, assuming that e_0 has been reasonably encoded, we have $\ell(\{e_0\}(x)) \leq \ell(e) + \ell(x) + c_0$ where c_0 is some fixed constant. Note that we use here the fact that $\ell(\langle u, v\rangle) = \ell(u) + \ell(v) + 3$ so it is possible to "hardwire" both e and x without sacrificing essentially more than $\ell(e) + \ell(x)$ in length. By padding e_0 we obtain an algorithm e_1 with the same behaviour as e_0 and such that $\ell(\{e_1\}(x)) < \ell(e_1) + \ell(x)$. Since $\{e\}(x)$ terminates in time linear in $|x|$, thus quadratic in $\ell(x)$, we can—by further padding e_1—obtain an algorithm e_2 such that $e_2x = \{e_0\}(x)$.

Now, by construction, we have $\{e_2x\}(y) = e\langle x, y\rangle$ and the computation $\{e_2x\}(y)$ takes less than $d(\ell(e) + \ell(x) + \ell(y) + 3 + \gamma)^p$ steps where γ is the constant from Prop. 4.1.6 and $d = \ell(e) + \ell(x) + \ell(y) + 3 - \ell(e\langle x, y\rangle)$. Therefore, by further padding e_2 to make up for $\gamma + 3$ we obtain the desired algorithm e'. □

Theorem 4.1.8 *The set of natural numbers together with the above application function* app *is a BCK-algebra.*

Proof. The combinator K is obtained by parametrising the (linear time computable) left inverse to the pairing function.

For the composition combinator B we start with the following three-input algorithm:

```
$B_0 \equiv$ read $w$
        write app$(w.1.1, \mathsf{app}(w.1.2, w.2))$
```

We have $\{B_0\}(\langle\langle x, y\rangle, z\rangle) = x(yz)$ and the time t_{tot} needed to evaluate $\{B_0\}(\langle\langle x, y\rangle, z\rangle)$ is less than $t_1 + t_2 + t_b$ where

$$
\begin{aligned}
&u = yz\\
&w = x(yz)\\
&d_1 = \ell(y) + \ell(z) - \ell(u)\\
&d_2 = \ell(x) + \ell(u) - \ell(w)\\
&t_1 = d_1(\ell(y) + \ell(z) + \gamma)^p\\
&t_2 = d_2(\ell(x) + \ell(u) + \gamma)^p
\end{aligned}
$$

and where t_b—the time needed for shuffling around intermediate results—is linear in $|x| + |y| + |z| + |u| + |w|$ thus $O((\ell(x) + \ell(y) + \ell(z))^2)$ where we have used the inequalities $\ell(u) \leq \ell(y) + \ell(z)$ and $\ell(w) \leq \ell(x) + \ell(y) + \ell(z)$ to get rid of the u and w.

This means that we can find a constant c_2 such that

$$t_{\mathsf{tot}} \leq (d_1 + d_2)(\ell(x) + \ell(y) + \ell(z) + c_2)^p$$

Here we have used the fact that $\ell(u) \leq \ell(y) + \ell(z)$.

Now the defect of the computation $\{B_0\}(\langle\langle x, y\rangle, z\rangle)$ equals $\ell(B_0) + \ell(x) + \ell(y) + \ell(z) + 6 - \ell(w) = \ell(B_0) + 6 + d_1 + d_2$. Therefore, by choosing $\ell(B_0)$ large enough we obtain

$$\mathsf{app}(B_0, \langle\langle x, y\rangle, z\rangle) = \{B_0\}(\langle\langle x, y\rangle, z\rangle) = x(yz)$$

The desired algorithm B is then obtained by applying Lemma 4.1.7 twice.

Notice, that the existence of the B combinator hinges on the fact that the time of a computation decreases as the size of the output goes up. Had we not imposed the dependency of running time on output size via the defect it would not have been possible to define the B combinator.

Let us finally define the C combinator. We start with the following algorithm

```
$C_0 \equiv$ read $w$
        write $\mathsf{app}(\mathsf{app}(w.1.1, w.2), w.1.2)$
```

Clearly,

$$\{C_0\}(\langle\langle x, y\rangle, z\rangle) = xzy$$

The total time t_{tot} needed for this computation is bounded by $t_1 + t_2 + t_b$ where

$$\begin{aligned}
&u = xz\\
&w = uy\\
&d_1 = \ell(x) + \ell(z) - \ell(u)\\
&d_2 = \ell(u) + \ell(y) - \ell(w)\\
&t_1 = d_1(\ell(x) + \ell(z) + \gamma)^p\\
&t_2 = d_2(\ell(u) + \ell(y) + \gamma)^p
\end{aligned}$$

and, again, t_b is $O((\ell(x) + \ell(y) + \ell(z))^2)$. Therefore, we can find a constant c_0 such that

$$t_{\mathsf{tot}} \leq (d_1 + d_2)(\ell(x) + \ell(y) + \ell(z) + c_0)^p$$

The defect of the computation $\{C_0\}(\langle\langle x, y\rangle, z\rangle)$ is

$$d = \ell(C_0) + 6 + \ell(x) + \ell(y) + \ell(z) - \ell(w) = \ell(C_0) + 6 + d_1 + d_2$$

Therefore, assuming w.l.o.g. that $\ell(C_0) \geq c_0$ we obtain

$$\mathsf{app}(C_0, \langle\langle x, y\rangle, z\rangle) = \{C_0\}(\langle\langle x, y\rangle, z\rangle) = xzy$$

The desired combinator C is again obtained by applying Lemma 4.1.7 twice. □

Definition 4.1.9 *The BCK-algebra thus constructed will be called H_p.*

Abbreviations. Let H be a BCK-algebra. In view of Lemma 4.1.2 we will freely use linear lambda terms involving constants from H in order to denote particular elements of H. Moreover, we write $\lambda x_1 x_2 \ldots x_n.t$ for $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.t$. We write T for the pairing combinator $\lambda xyf.fxy$ and P_1, P_2 for the projections $\lambda p.p(\lambda xy.x)$ and $\lambda p.p(\lambda xy.y)$. Note that $P_i(Tt_1t_2) = t_i$.

It is in general not a good idea, to use projections in order to decompose a variable meant to encode a pair. The reason is that in order to maintain linearity we can use either P_1 or P_2, but not both. The correct way to decompose a pair is to apply it to a function of two arguments which are then bound to the components of a pair. Suppose, for example, that $u, v \in H$ and that we want to define an element $u \otimes v \in H$ such that $(u \otimes v)(Txy) = T(ux)(vy)$. Writing

$$(u \otimes v) =_{\text{def}} \lambda p.T(u(P_1\ p))(v(P_2\ p))$$

does not work since the λ-abstraction is not defined because p occurs twice in its body. We can, however, achieve the desired effect by putting

$$(u \otimes v) =_{\text{def}} \lambda p(\lambda xy.T(ux)(vy))$$

4.1.3 Truth values and numerals

In every BCK-algebra truth values and numerals can be encoded. In concrete examples it is, however, often convenient to use other representations for these basic datatypes than the canonical ones which is why we give them the status of extra structure.

Definition 4.1.10 *A BCK-algebra H supports truth values and natural numbers if there are distinguished elements $\mathsf{tt}, \mathsf{ff}, D, \mathsf{S}_0, \mathsf{S}_1, G$ and an injection $\mathsf{num} : \mathbb{N} \to H$ such that the following equations are satisfied.*

$$\begin{array}{l}
D\ \mathsf{tt}\ x\ y = x \\
D\ \mathsf{ff}\ x\ y = y \\
\mathsf{S}_0\ \mathsf{num}(x) = \mathsf{num}(2x) \\
\mathsf{S}_1\ \mathsf{num}(x) = \mathsf{num}(2x+1) \\
G\ \mathsf{num}(0) = T\ \mathsf{tt}\ (T\ \mathsf{tt}\ \mathsf{tt}) \\
G\ \mathsf{num}(2(x+1)) = T\ \mathsf{ff}\ (T\ \mathsf{tt}\ \mathsf{num}(x+1)) \\
G\ \mathsf{num}(2x+1) = T\ \mathsf{ff}\ (T\ \mathsf{ff}\ \mathsf{num}(x))
\end{array}$$

If φ is an informal statement let $[\varphi]$ be tt if φ is true and ff otherwise. We have

$$\begin{array}{l}
P_1(G\ \mathsf{num}(n)) = [n{=}0] \\
P_1(P_2(G\ \mathsf{num}(n))) = [n \text{ is even}] \\
P_2(P_2(G\ \mathsf{num}(n+1))) = \mathsf{num}(\left\lfloor \frac{n+1}{2} \right\rfloor)
\end{array}$$

Proposition 4.1.11 *Every BCK-algebra supports truth values and natural numbers.*

Proof. Define $\mathsf{tt} =_{\text{def}} \lambda xy.x$ and $\mathsf{ff} =_{\text{def}} \lambda xy.y$ and $D =_{\text{def}} \lambda txy.txy$. This accounts for the truth values. The injection num is defined recursively by $\mathsf{num}(0) = T\ \mathsf{tt}(T\ \mathsf{tt}\ \mathsf{tt})$, $\mathsf{num}(2(x+1)) = T\ \mathsf{ff}(T\ \mathsf{tt}\ \mathsf{num}(x))$, $\mathsf{num}(2x+1) = T\ \mathsf{ff}(T\mathsf{ff}\ \mathsf{num}(x))$. Now put

$$\begin{array}{ll} \mathsf{S}_0 = \lambda x.x(\lambda ty.D\ t & \\ \quad \mathsf{num}(0) & \text{case } x = 0 \\ \quad (T\ \mathsf{ff}(T\ \mathsf{tt}\ (T\ \mathsf{ff}\ y)) & \text{case } x \neq 0 \\ \mathsf{S}_1 = \lambda x.T\ \mathsf{ff}(T\ \mathsf{ff}\ x) & \\ G = I & \end{array}$$

□

Proposition 4.1.12 *The algebras H_p support natural numbers and truth values with the settings $\mathsf{tt} = 1$, $\mathsf{ff} = 0$, and num defined as in Lemma 2.1.1, i.e., $\mathsf{num}(x) = 2x + 1$.*

Proof. The missing constants are obtained by parametrisation from the obvious algorithms computing them. □

4.2 Realisability sets

In this section we define and explore an analogue of the category of modest sets introduced by Moggi and others based on a *BCK*-algebra supporting truth values and natural numbers. We refer to, e.g., [17] for an introduction to modest sets and realisability. We shall see that due to the absence of an S-combinator hence of diagonalisation, the thus obtained category of modest sets is not cartesian closed. It is, however, an affine linear category w.r.t. to a natural tensor product based on the pairing function and it also has cartesian products, which, however, lack right adjoints, i.e., function spaces.

Unless stated otherwise let H be an arbitrary *BCK*-algebra supporting truth values and natural numbers. For a concrete example the reader may of course think of H_p for H.

Definition 4.2.1 *An H-set is a pair $X = (|X|, \Vdash_X)$ where $|X|$ is a set and $\Vdash_X \subseteq H \times X$ is a relation such that*

i. $\forall x{\in}X.\exists n{\in}H.n \Vdash_X x$.

ii. $\forall x, y{\in}X.\forall n{\in}H.n \Vdash_X x \wedge n \Vdash_X y \Rightarrow x = y$

A morphism from H-set X to H-set Y is a function $f : |X| \longrightarrow |Y|$ such that there exists an element $e \in H$ with

$$\forall x{\in}X.\forall t{\in}H.t \Vdash_X x \Rightarrow e\ t \Vdash_Y f(x)$$

We write $e \Vdash_{X \multimap Y} f$ in this case.

Remark 4.2.2 *We remark in the classical case of realisability sets over a partial combinatory algebra H the term H-set is used for objects which only satisfy requirement i) above. Those which also satisfy ii) are called* modest *H-sets. Since we do not need this generalisation we do not use the attribute "modest". The requirement i) that every element have a realiser is needed in order that the category of H-set is well-pointed which will allow us to use extensional presheaves.*

If $f : |X| \longrightarrow |Y|$ is a set-theoretic function then we say that f is realised by e, if $e \Vdash_{X \multimap Y} f$. So an H-set morphism from X to Y is a function that can be realised.

We will sometimes write X instead of $|X|$ and $\Vdash$ instead of $\Vdash_X$.

Definition and Theorem 4.2.1 *The H-sets together with their morphisms form a well-pointed affine linear category $\mathcal{H}$ with the following settings.*

- *Identities and composition are given by set-theoretic identity and composition which are realised by virtue of the I and B combinators.*
- *The tensor product of H-sets X, Y is given by*

$$\begin{aligned} |X \otimes Y| &= |X| \times |Y| \text{ (set-theoretic cartesian product)} \\ t \Vdash_{X \otimes Y} (x, y) &\iff \exists u, v. t = Tuv \wedge u \Vdash x \wedge v \Vdash y \end{aligned}$$

- *The projections are given by $\pi(a, b) = a$ and $\pi'(a, b) = b$.*
- *The terminal object is $\top = \{\langle\rangle\}$ with $\mathtt{tt} \Vdash_\top \langle\rangle$.*

Proof. The projections are obviously jointly monic and the defining equations for the associated morphisms and operators imply that those are defined as in the category of sets, e.g., associativity is given by $\alpha(x, (y, z)) = ((x, y), z)$. Therefore, all that remains to be shown is that the projections as well as these associated morphisms are realisable.

The projections are realised by P_1 and P_2.

If $f : X_1 \longrightarrow Y_1$ and $g : X_2 \longrightarrow Y_2$ then $(f \otimes g)(x_1, x_2)$ equals $(f(x_1), g(x_2))$ and this can be realised by $\lambda p.p(\lambda t_1 t_2.T(dt_1)(et_2))$ when $d \Vdash f$ and $e \Vdash g$.

Symmetry ($X \otimes Y \cong Y \otimes X$) is realised by $\lambda p.p(\lambda xy.Tyx)$.

Associativity ($X \otimes (Y \otimes Z) \cong (X \otimes Y) \otimes Z$) is realised by $\lambda p.p(\lambda xv.v(\lambda yz.T(Txy)z))$.

The unique map $\langle\rangle : X \longrightarrow \top$ is realised by $\lambda x.\mathtt{tt}$.

The isomorphism $X \cong X \otimes \top$ is realised by $\lambda x.Tx\mathtt{tt}$; its inverse is realised by $\lambda t.t(\lambda xy.x)$. Similarly, $X \cong \top \otimes X$.

Finally, if $x \in |X|$ then the function $f_x : |\top| \longrightarrow |X|$ is a morphism realised, e.g., by Ke where e is a realiser for x. So all elements of $|X|$ appear as global elements and it follows that $\mathcal{H}$ is a well-pointed. Notice that this hinges on the requirement that every element have a realiser. □

Proposition 4.2.3 *In $\mathcal{H}$ all linear function spaces exist and are given as follows. If $X, Y \in \mathcal{H}$ then $X \multimap Y$ has as underlying set the set of morphisms from X to Y. The realisability relation $\Vdash_{X \multimap Y}$ is as defined above in Def. 4.2.1, i.e.,*

$$e \Vdash_{X \multimap Y} f \iff \forall t, x. t \Vdash_X x \Rightarrow et \Vdash_Y f(x)$$

The application map $\mathsf{ev} : (X \multimap Y) \otimes X \longrightarrow Y$ *is defined by* $\mathsf{ev}(f, x) = f(x)$.

Proof. Application is realised by $\lambda z.z(\lambda fx.fx)$. If $f : Z \otimes X \longrightarrow Y$ is realised by e then for each $z \in Z$ the function $x \mapsto f(z, x)$ is realised by $\lambda v.e(\lambda k.kuv)$ when u is a realiser for z. Therefore, we have a function from Z to $(X \multimap Y)$. This function itself is realised by $\lambda uv.e(\lambda k.kuv)$. □

Proposition 4.2.4 *The category $\mathcal{H}$ has cartesian products given by*

$$\begin{array}{l} |X \times Y| = |X| \times |Y| \\ e \Vdash_{X \times Y} (x, y) \iff e\ \mathsf{tt} \Vdash x \wedge e\ \mathsf{ff} \Vdash y \end{array}$$

Proof. Projections $X \times Y \longrightarrow X$ and $X \times Y \longrightarrow Y$ are realised by $\lambda e.e\ \mathsf{tt}$ and $\lambda e.e\ \mathsf{ff}$, respectively. If $f : Z \longrightarrow X$ and $g : Z \longrightarrow Y$ are realised by d and e then the "target-tupled" function $\langle f, g\rangle : Z \longrightarrow X \times Y$ defined by $\langle f, g\rangle(c) = (f(c), g(c))$ is realised by $\lambda xt.(Dtde)x$. □

4.2.1 Natural numbers and other datatypes

Definition 4.2.5 *The H-set of natural numbers N has $\mathbb{N}$ as underlying set and realising relation defined by* $\mathsf{num}(n) \Vdash_{\mathsf{N}} n$.

Lemma 4.2.6 *Suppose that $H = H_p$. The $\mathcal{H}$-morphisms from $\mathsf{N} \otimes \cdots \otimes \mathsf{N}$ (n factors) are the functions $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ which are computable in time $O((|x_1| + \cdots + |x_n|)^p)$ and moreover satisfy $|f(\vec{x})| = |x_1| + \cdots + |x_n| + O(1)$.*

Proof. Direct from Proposition 4.1.6 and the discussion following Lemma 4.1.4. □

Proposition 4.2.7 (Constructors and case distinction) *Let X be an H-set. The functions*

$$\begin{array}{l}
0 : |\top| \longrightarrow |\mathsf{N}| \\
\mathsf{S}_0 : |\mathsf{N}| \longrightarrow |\mathsf{N}| \\
\mathsf{S}_1 : |\mathsf{N}| \longrightarrow |\mathsf{N}| \\
\mathsf{case}^{\mathsf{N}} : |C \times ((\mathsf{N} \multimap C) \times (\mathsf{N} \multimap C))| \longrightarrow |\mathsf{N} \multimap C|
\end{array}$$

with $0(\langle\rangle) = 0$, $\mathsf{S}_0(x) = 2x$, $\mathsf{S}_1(x) = 2x{+}1$, *and* $\mathsf{case}^{\mathsf{N}}(g, h_0, h_1)(0) = g$, $\mathsf{case}^{\mathsf{N}}(g, h_0, h_1)(2x{+}i) = h_i(x)$, *otherwise, are morphisms in* $\mathcal{H}$.

Proof. Obvious for $0, \mathsf{S}_0, \mathsf{S}_1$. A realiser for $\mathsf{case}^{\mathsf{N}}$ is obtained as follows.

$$\begin{array}{ll}
\lambda p.\lambda x.G\ x(\lambda uv.Du & \\
\quad (p\mathsf{tt}) & \text{case } x{=}0 \\
\quad (v(\lambda qy & q{=}[x \text{ is even}], y = \left\lfloor \frac{x}{2} \right\rfloor \\
\quad\ .p\mathsf{ff}\ q\ y)) &
\end{array}$$

$\square$

Definition 4.2.8 *Let A be an H-set. We define $\mathsf{T}(A) \in \mathcal{H}$ by $|\mathsf{T}(A)| = \mathsf{T}(|A|)$, i.e., the set of $|A|$-labelled binary trees. The realisability relation $\Vdash_{\mathsf{T}(A)}$ is defined inductively by*

- *If $x \Vdash a$ then $T\ \mathsf{tt}\ x \Vdash \mathsf{leaf}(x)$.*
- *If $x \Vdash a$ and $y \Vdash l$ and $z \Vdash r$ then $T\mathsf{ff}\ (Ta(Tlr)) \Vdash \mathsf{node}(a, l, r)$.*

Proposition 4.2.9 *The constructors* leaf *and* node *give rise to morphisms of the following type*

$$\begin{array}{l}
\mathsf{leaf}_A : A \longrightarrow \mathsf{T}(A) \\
\mathsf{node}_A : A \otimes \mathsf{T}(A) \otimes \mathsf{T}(A) \longrightarrow \mathsf{T}(A)
\end{array}$$

Furthermore, the function

$$\mathsf{case}_X^{\mathsf{T}(A)} : (A \multimap X) \times (A \otimes \mathsf{T}(A) \otimes \mathsf{T}(A) \multimap X) \multimap \mathsf{T}(A) \multimap X$$

given by

$$\begin{array}{l}
\mathsf{case}_X^{\mathsf{T}(A)}(h_{\mathsf{leaf}}, h_{\mathsf{node}})(\mathsf{leaf}(a)) = h_{\mathsf{leaf}}(a) \\
\mathsf{case}_X^{\mathsf{T}(A)}(h_{\mathsf{node}}, h_{\mathsf{node}})(\mathsf{leaf}(a)) = h_{\mathsf{node}}(a, l, r)
\end{array}$$

is realisable.

Proof. Analogous to Prop. 4.2.7 □

Similarly, we can define H-sets of lists and other inductively defined data types (see also Section 4.3.1).

4.3 Interpreting recursion

In general (e.g. when $H = H_p$) the category $\mathcal{H}$ itself does not contain all *PTIME*-functions and thus does not allow us to represent patterns of safe recursion on notation.

In order to achieve this we introduce a notion of polynomial time computable function between H-sets which strictly contains the H-set morphisms. Safe recursion then takes the form of an operator on such polynomial time computable functions with the proviso that the step functions be H-set morphisms. In order to give safe recursion the shape of a higher-typed constants we move to extensional presheaves over a category obtained by integrating the polynomial time computable functions with the H-set morphisms.

In order to be able to define these polynomial time computable functions we need to restrict our attention to *BCK*-algebras in which application is polynomial time computable. In addition we need to impose further technical conditions all of which are met by the algebras H_p. The reason for introducing another level of abstraction and not working directly with the H_p is that we will later on instantiate these results with two other *BCK*-algebras.

Definition 4.3.1 *A BCK-algebra H supporting truth values and natural numbers is called* polynomial time computable *(polynomial or PTIME for short) if its carrier set can be identified with a subset of strings so that it is amenable to algorithmic manipulation and there exists a function $\ell : H \longrightarrow \mathbb{N}$ such that*

- *there exists an algorithm f and a polynomial p such that for each $x, y \in H$ the computation $f(x, y)$ terminates after not more than $p(\ell(x) + \ell(y))$ steps with result xy,*

- *for each $x, y \in H$ the following inequalities hold:*

$$\begin{array}{l}
\ell(xy) \leq \ell(x) + \ell(y) \\
\ell(x) \leq |x| \\
|Kx| > |x| \\
|Bx| > |x| \\
|Cx| > |x| \\
|Bxy| > |x| + |y| \\
|Cxy| > |x| + |y| \\
|\mathsf{num}(n)| \geq |n|
\end{array}$$

The first two inequalities are abstracted from the particular example H_p. They are needed in order to show that iterations of functions represented in H are polynomial time computable.

The other inequalities are sanity conditions ensuring that the partial applications of B, C, K copy their input into the result and do not reduce the size by some sort of information compression.

We summarise a few basic facts about polynomial *BCK*-algebra in the following lemma:

Lemma 4.3.2 *Let H be a polynomial time computable BCK-algebra.*

i. The BCK-algebras H_p are polynomial time computable;

ii. For each $x, y \in H$ we have $|Txy| > |x| + |y|$;

iii. The requirement $|\mathsf{num}(n)| \geq |n|$ is satisfied for the canonical encoding (Section 4.1.11) of natural numbers in BCK-algebra;

iv. $\ell(\mathsf{num}(n)) = O(|n|)$, hence $\Theta(|n|)$;

v. There exists a PTIME-function getnum *such that* $\mathsf{getnum}(\mathsf{num}(n)) = n$.

Proof. Part i is obvious from the definition of the H_p. Part ii follows from $Txy = C(CIx)y$, hence $|Txy| > |CIx| + |y| > |CI| + |x| + |y| \geq |x| + |y|$. For iii we use ii and induction over n. Part iv follows from the existence of constructor functions S_0 and S_1 together with the inequation $\ell(xy) \leq \ell(x) + \ell(y)$.

For part v we note that the function getnum admits a recursive definition in terms of G. The total number of unfoldings of the recursive definition can be a priori bounded by $|x|$.

□

We assume henceforth that our generic *BCK*-algebra H is polynomial.

Definition 4.3.3 *Let X, Y be H-sets. A PTIME-function from X to Y is a function $f : |X| \longrightarrow |Y|$ such that there exists a one-argument algorithm g and a polynomial p such that whenever $e \Vdash_X x$ then the computation $g(e)$ terminates after not more than $p(|e|)$ steps and $g(e) \Vdash_Y f(x)$.*

We use the notation $f : X \xrightarrow{PTIME} Y$ to indicate that f is a PTIME-function from X to Y

An algorithm g together with polynomial p as in the above definition will often be called a *realiser* for *PTIME*-function f. If g is a realiser with polynomial p then $\ell(g(e)) \leq p(|e|)$ by $\ell(x) \leq |x|$.

A realiser for a *PTIME*-function f from $X \otimes Y$ to Z can equivalently be given as a two-argument algorithm g together with a two-variable polynomial p such that $d \Vdash x$ and

$e \Vdash y$ implies that $\{g\}(x, y)$ terminates in not more than $p(|x|, |y|)$ steps and yields a realiser for $f(x, y)$.

Notice that every morphism of H-sets is a *PTIME*-function between H-sets, but not vice versa. The *PTIME*-functions with ordinary composition clearly form a category but this fact will not be needed.

Also notice that by the estimates of $|\mathsf{num}(n)|$ in Lemma 4.3.2 a map $f : \mathsf{N} \xrightarrow{PTIME} \mathsf{N}$ is the same as a polynomial time computable function on the integers in the usual complexity-theoretic sense.

If $A_1, \ldots, A_m, B_1, \ldots, B_n, C$ are H-sets then a map

$$f : A_1 \otimes \ldots \otimes A_m \xrightarrow{PTIME} (B_1 \otimes \ldots \otimes B_n) \multimap C$$

can be viewed as an $m + n$-ary map with the first m inputs "normal" in the sense that the result depends polynomially on their size and the second n inputs "safe" in the sense that the size of the result is majorised by the sum of their sizes plus a constant independent of their sizes. More, formally, if k, q is a realiser for such map f then we have

$$\ell(f(\vec{x}; \vec{y})) \leq q(\ell(\vec{x})) + \sum_{i=1}^{n} \ell(y_i)$$

In this way the polynomial time functions between H-sets generalise Bellantoni's "polymax-bounded"-functions to linearity and higher types. We see that the maximum operation is now replaced by summation which exploits the greater generality offered by linearity.

Theorem 4.3.4 (Safe recursion on notation) *Let P, X be H-sets. If*

$$h : P \otimes \mathsf{N} \xrightarrow{PTIME} X \multimap X$$

is a PTIME-function as indicated then so is the function $f : |P \otimes \mathsf{N}| \longrightarrow |X \multimap X|$ defined by

$$\begin{aligned} &f(p, 0)(g) = g \\ &f(p, x)(g) = h(p, x)(f(p, \lfloor x/2 \rfloor)(g)) \qquad \textit{when } x > 0 \end{aligned}$$

Proof. Let k be a realiser for h, i.e. an algorithm such that $t \Vdash_P p$ implies that $k(t, \mathsf{num}(n))$ terminates in not more than than $q(|t|, |\mathsf{num}(n)|)$ steps for some fixed polynomial q. In view of $\ell(x) \leq |x|$ this implies $\ell(k(t, \mathsf{num}(n))) \leq q(|t|, |\mathsf{num}(n)|)$.

In order to realise $\mathsf{rec}^{\mathsf{N}}(h)$ we consider the recursive algorithm $g : H \times H \longrightarrow H$ defined by

```
$g(t, x) =$ if $(P_1(G\ x)) = \mathtt{tt}$
  then $\lambda v.v$                    (Case $x = 0$)
  else $\lambda v.k(t, x)\ (g(t, \mathsf{div2}\ x)\ v)$
```

where

$$\mathsf{div2} = \lambda x.P_2(P_2(G\ x))$$

It is clear from the definition that g is a realiser for g provided we can show that it has polynomial runtime for inputs of the form $t \in \mathrm{dom}(P)$ and $x = \mathsf{num}(n)$ for $n \in \mathbb{N}$. Notice that for other inputs $g(t, x)$ may diverge even if k is assumed total. For example x could be such that $\mathsf{div2}\ x = x$.

The length of $g(t, x)$ satisfies the following estimate.

$$\begin{array}{lcl} \ell(g(t, \mathsf{num}(0))) & \leq & c \\ \ell(g(t, \mathsf{num}(n))) & \leq & c + q(|t|, |\mathsf{num}(n)|) + \ell(g(t, \mathsf{num}(\left\lfloor \frac{n}{2} \right\rfloor))), \text{ if } n > 0 \end{array}$$

Therefore,

$$\ell(g(t, \mathsf{num}(n))) \leq |n| \cdot q(|t|, |\mathsf{num}(n)|) + c$$

In order to estimate the time $T(t, x)$ needed to compute $g(t, x)$ we first note that, in fact, g can be written as

$$\begin{array}{ll} g(t, x) = \mathbf{if}(P_1(G\ x)) = \mathtt{tt} & \\ \quad \mathbf{then}\ I & \text{(Case } x = 0) \\ \quad \mathbf{else}\ B\ k(t, x)\ (g(t, \mathsf{div2}\ x)\ v) & \end{array}$$

so that

$$\begin{array}{lcl} T(t, \mathsf{num}(0)) & \leq & c \\ T(t, \mathsf{num}(n)) & \leq & q(|t|, |\mathsf{num}(n)|) + T(t, \mathsf{num}(\left\lfloor \frac{n}{2} \right\rfloor)) \\ & & +p'(q(|t|, |\mathsf{num}(n)|) + \ell(g(t, \mathsf{num}(\left\lfloor \frac{n}{2} \right\rfloor)))), \text{ if } n > 0 \end{array}$$

Here c is a constant and p' is a polynomial obtained from enlarging the polynomial p witnessing that application in H is polynomial time w.r.t. ℓ a little bit so as to account for bookkeeping.

Hence, by induction on n:

$$\begin{array}{l} T(t, \mathsf{num}(n)) \\ \quad \leq c + |n| \cdot (q(|t|, |\mathsf{num}(n)|) + \\ \qquad +p'(q(|t|, |\mathsf{num}(n)|) + |n| \cdot q(|t|, |\mathsf{num}(n)|) + c)) \\ \quad \leq c + |\mathsf{num}(n)|(q(|t|, |\mathsf{num}(n)|) + \\ \qquad +p'(q(|t|, |\mathsf{num}(n)|) + \mathsf{num}(n) \cdot q(|t|, |\mathsf{num}(n)|) + c)) \end{array}$$

which is polynomial in $|t|, |\mathsf{num}(n)|$ as required. □

Corollary 4.3.5 (Duplicable safe parameters) *If P, D, X are H-sets and D is duplicable and $h : P \otimes \mathsf{N} \xrightarrow{PTIME} D \otimes X \multimap X$ is a PTIME-function then the function $f : |P \otimes \mathsf{N}| \longrightarrow |D \otimes X \multimap X|$ defined by*

$$\begin{aligned} &f(p, 0)(d, g) = g \\ &f(p, x)(d, g) = h(p, x)(d, f(p, \left\lfloor \tfrac{x}{2} \right\rfloor)(d, g)) \end{aligned}$$

is a PTIME-function.

Proof. Define $Y = D \multimap X$ and $h' : P \otimes \mathsf{N} \xrightarrow{PTIME} Y \multimap Y$ by

$$h'(p, n)(u)(d) = \mathsf{let}\,(d_1, d_2){=}\delta(d)\,\mathsf{in}\,h(p, x)(d_1, u(d_2))$$

where δ is the diagonal morphism for D. This definition can be formalised as a composition of h with a $\mathcal{H}$-map from $D \otimes X \multimap X$ to $Y \multimap Y$ definable in the language of ALCC using δ and therefore yields a *PTIME*-function without further proof.

Applying Theorem 4.3.4 yields $f' : P \otimes \mathsf{N} \xrightarrow{PTIME} Y \multimap Y$ satisfying the corresponding recurrence. The desired f is obtained from f' by

$$f(p, x)(d, g) = (f'(p, x)(\lambda dg.g))\ d\ g$$

Again, this can be seen as composition of f with a map from $(Y \multimap Y)$ to $D \multimap X \multimap X$. □

Theorem 4.3.6 (Safe tree recursion) *Let $P, X \in \mathcal{H}$ and*

$$\begin{aligned} &h_{\mathsf{leaf}} : P \otimes A \xrightarrow{PTIME} X \\ &h_{\mathsf{node}} : P \otimes A \otimes \mathsf{T}(A) \otimes \mathsf{T}(A) \xrightarrow{PTIME} X \otimes X \multimap X \end{aligned}$$

be PTIME-functions as indicated. then the function $f : |P \otimes \mathsf{T}(A)| \longrightarrow |X|$ defined by

$$\begin{aligned} &f(p, \mathsf{leaf}(a)) = h_{\mathsf{leaf}}(p, a) \\ &f(p, \mathsf{node}(a, l, r)) = h_{\mathsf{node}}(p, a, l, r)(f(p, l), f(p, r)) \end{aligned}$$

is a PTIME-function from $P \otimes \mathsf{T}(A)$ to X.

Proof. Let k_{leaf} and k_{node} be realisers for h_{leaf} and h_{node} viewed as binary, resp. quaternary *PTIME*-functions with witnessing polynomials $q_{\mathsf{leaf}}(t, a)$ and $q_{\mathsf{node}}(t, a, l, r)$. In order to realise f we recursively define a function $g : H \times H \longrightarrow H$ by

$$\begin{aligned} &g(t, x) = D\ x_1 \\ &\quad (k_{\mathsf{leaf}}(t, x_2)) \\ &\quad (k_{\mathsf{node}}(t, x_{21}, x_{22}, x_{23})(T\ f(z_0, x_{22})\ f(z_0, x_{23}))) \\ &\text{where } x_1 = P_1 x, x_2 = P_2 x, x_{21} = P_1(P_2 x), x_{22} = P_1(P_2(P_2 x)), x_{23} = P_2(P_2(P_2 x)) \end{aligned}$$

Again, it is clear that if this algorithm runs sufficiently fast then it will realise the above function on trees. It thus remains to show that if $t \in \mathrm{dom}(P)$ and $x \in \mathrm{dom}(\mathsf{T}(A))$ then $f(t, x)$ is computable in polynomial time.

Now, we have

$$\begin{aligned}
&\ell(f(t, T\mathsf{tt}\ a)) \leq q_{\mathsf{leaf}}(|t|, |a|)\\
&\ell(f(t, T\mathsf{ff}(Ta(Tlr)))) \leq c + q_{\mathsf{node}}(|t|, |a|, |l|, |r|) + \ell(f(t, l)) + \ell(f(t, r))
\end{aligned}$$

where c is a constant accounting for the ℓ-length of the function combining the results of the recursive calls.

Therefore, if $x \in \mathrm{dom}(\mathsf{T}(A))$ then using $|Txy| > |x| + |y|$ we obtain

$$\ell(f(t, x)) \leq |x| \cdot q(|t|, |x|)$$

where $q(|t|, |x|)$ majorise $q_{\mathsf{leaf}}(|t|, |a|)$ and $q_{\mathsf{node}}(|t|, |x|, |x|, |x|) + c$.

Now it follows that the runtime of $f(t, x)$ can be estimated by

$$\begin{aligned}
T(t, T\mathsf{tt}\ a) &\leq p'(q(|t|, |a|)) \leq p'(|t|, |T\mathsf{tt}a|)\\
T(t, T\mathsf{ff}\ x) &\leq q(|t|, |x|) + T(t, l) + T(t, r) +\\
&\quad p'(q(|t|, |x|) + |l|q(|t|, |l|) + |r|q(|t|, |r|))\\
&\quad \leq T(t, l) + T(t, r) + p'(|t|, |x|)
\end{aligned}$$

where $x = T\mathsf{ff}(Ta(Tlr))$ and p' is a suitably large polynomial. Therefore, (always under the assumption that $x \in \mathrm{dom}(\mathsf{T}(A))$)

$$T(t, x) \leq |x| \cdot p'(|t|, |x|)$$

□

Corollary 4.3.7 *Let $P, X, D \in \mathcal{H}$ with D duplicable and suppose that*

$$\begin{aligned}
&h_{\mathsf{leaf}} : P \otimes A \xrightarrow{PTIME} D \multimap X\\
&h_{\mathsf{node}} : P \otimes A \otimes \mathsf{T}(A) \otimes \mathsf{T}(A) \xrightarrow{PTIME} D \otimes X \otimes X \multimap X
\end{aligned}$$

be PTIME-functions as indicated. then the function $f : |P \otimes \mathsf{T}(A)| \longrightarrow |D \multimap X|$ defined by

$$\begin{aligned}
&f(p, \mathsf{leaf}(a))(d) = h_{\mathsf{leaf}}(f, a)(d)\\
&f(p, \mathsf{node}(a, l, r))(d) = h_{\mathsf{node}}(p, a, l, r)(d, f(p, l)(d), f(p, r)(d))
\end{aligned}$$

is a PTIME-function from $P \otimes \mathsf{T}(A)$ to X.

Proof. Analogous to the proof of Cor. 4.3.5 using Thm. 4.3.6 with result type $D \multimap X$.

□

Again, we omit the treatment of lists as it is analogous to the previously treated cases.

4.3.1 Leivant trees

Let us finally consider the "Leivant trees" mentioned in the introduction. For H-set A we define $\mathsf{T}'(A)$ by $|\mathsf{T}'(A)| = |\mathsf{T}(A)|$ and

- If $x \Vdash a$ then $T\ \mathsf{tt}\ x \Vdash \mathsf{leaf}(x)$.
- If $x \Vdash a$ and $y\mathsf{tt} \Vdash l$ and $y\mathsf{ff} \Vdash r$ then $T\mathsf{ff}\ (Tay) \Vdash \mathsf{node}(a, l, r)$.

It is then clear that the constructors give rise to morphisms

$$\begin{aligned}\mathsf{leaf}' &: A \longrightarrow \mathsf{T}'(A)\\ \mathsf{node}' &: A \multimap (\mathsf{T}'(A) \times \mathsf{T}'(A)) \longrightarrow \mathsf{T}'(A)\end{aligned}$$

and that we can define an operator for case distinction

$$\mathsf{case}^{\mathsf{T}'} : (A \multimap X) \times (A \multimap (\mathsf{T}'(A) \times \mathsf{T}'(A)) \multimap X) \multimap \mathsf{T}'(A) \multimap X$$

Notice that due to the cartesian product in the second branch of a case distinction on Leivant trees we can either refer to the left subtree or to the right subtree, but not to both.

Unfortunately, we did not manage to show that recursion on Leivant trees is possible in quite the same generality as for the ordinary trees. What we can offer is the following iteration principle which does not provide access to the recursion variable.

Proposition 4.3.8 *Let $P, X \in \mathcal{H}$ and*

$$\begin{aligned}h_{\mathsf{leaf}} &: P \xrightarrow{PTIME} A \multimap X\\ h_{\mathsf{node}} &: P \xrightarrow{PTIME} A \multimap (X \times X) \multimap X\end{aligned}$$

be PTIME-functions. Then the function $f : |P \otimes \mathsf{T}'(A)| \longrightarrow |X|$ defined by

$$\begin{aligned}f(p, \mathsf{leaf}(a)) &= h_{\mathsf{leaf}}(p)(a)\\ f(p, \mathsf{node}(a, l, r)) &= h_{\mathsf{node}}(p)(a)(f(p, l), f(p, r))\end{aligned}$$

is a PTIME-function.

Proof. We first note that using safe recursion on notation (Thm. 4.3.4) with result type $\mathsf{T}'(A) \multimap X$ and $\mathsf{case}^{\mathsf{T}'}$ we can define a *PTIME*-function $f' : P \otimes \mathsf{N} \xrightarrow{PTIME} \mathsf{T}'(A) \multimap X$ with the property that

$$f'(p, n)(t) = f(p, t)$$

provided that $|n|$ exceeds the depth of t.

Now notice that by Prop. 4.1.6 we have $\ell(et) \leq e + \max(\ell(\mathsf{ff}), \ell(\mathsf{tt}))$ for $t \in \{\mathsf{tt}, \mathsf{ff}\}$. We may thus assume without essential loss of generality that if $t \Vdash_{\mathsf{T}'(A)} \mathsf{node}(a, l, r)$ then

$$\ell(t) > \max(\ell(P_2(P_2\ t)\ \mathsf{tt}), \ell(P_2(P_2\ t)\ \mathsf{ff}))$$

and thus it follows by (external) tree induction that if $e \Vdash_{\mathsf{T}'(A)} t$ then $\ell(e)$, hence in particular $|e|$ is an upper bound on the depth of t. Thus, if $k'(p, n)$ is a realiser for f' then

$$k(p, t) = k'(p, \mathrm{num}(t))t$$

is a realiser for f. □

4.4 Recursion operators as higher-typed constants

In this section we show how to embed the category of H-sets as well as the *PTIME*-functions between them into a single functor category in which the recursion patterns identified in the preceding four propositions take the form of higher-order constants involving modalities.

The strategy is to first combine H-set morphisms and *PTIME*-functions into a single category $\mathcal{H}^{\Box}$ which is structurally similar to the category $\mathcal{B}$ of "polymax-bounded" functions described in Section 2.6.12.3

Definition 4.4.1 *The category $\mathcal{H}^{\Box}$ has as objects pairs $X = (X_0, X_1)$ where both X_0 and X_1 are H-sets. A morphism from X to Y consists of a PTIME-function $f_0 : X_0 \longrightarrow Y_0$ and a PTIME-function $f_1 : X_0 \longrightarrow (X_1 \multimap Y_1)$. The identity morphism is given by the identity function at X_0 and the constant function yielding the identity morphism at X_1. The composition of (f_0, f_1) and (g_0, g_1) is given by $g_0 \circ f_0$ and $x \mapsto g_1(g_0(x)) \circ f_0(x)$.*

Proposition 4.4.2 *The following data endow $\mathcal{H}^{\Box}$ with the structure of a well-pointed ALC.*

The tensor product of $X = (X_0, X_1)$ and $Y = (Y_0, Y_1)$ is given by

$$(X_0, X_1) \otimes (Y_0, Y_1) = (X_0 \otimes Y_0, X_1 \otimes Y_1)$$

The first projection $\pi : X \otimes Y \longrightarrow X$ is given by the obvious canonical maps

$$\begin{aligned} \pi_0 &: X_0 \otimes Y_0 \xrightarrow{PTIME} X_0 \\ \pi_1 &: X_0 \otimes Y_0 \xrightarrow{PTIME} (X_1 \otimes Y_1) \multimap X_1 \end{aligned}$$

The second projection is defined analogously.

The terminal object is given by $\top = (\top, \top)$.

Proof. Routine verification. □

The category $\mathcal{H}^\Box$ also has cartesian products given by $(X_0, X_1) \times (Y_0, Y_1) = (X_0 \times Y_0, X_1 \times Y_1)$ where $\times$ is the cartesian product in $\mathcal{H}$.

An object of the form $(X_0, \top)$ is called *normal*; an object of the form $(\top, X_1)$ is called *safe.*

We notice that an $\mathcal{H}^\Box$-map from a safe object to a normal one must be constant.

The category $\mathcal{H}$ can be embedded fully and faithfully into $\mathcal{H}^\Box$ via $X \mapsto (\top, X)$ and $\mathcal{H}(X, Y) \ni f \mapsto (\mathrm{id}_\top, \hat{f})$ where $\hat{f} : \top \longrightarrow X \multimap Y$ is obtained as the transpose of

$$\top \otimes X \xrightarrow{\cong} X \xrightarrow{f} Y$$

This embedding preserves tensor product and cartesian product up to equality and we will therefore treat it as an inclusion thus identifying $\mathcal{H}$ with the full subcategory of $\mathcal{H}^\Box$ consisting of the safe objects.

Proposition 4.4.3 *If $A \in \mathcal{H}^\Box$ is arbitrary and B is normal then*

$$A \otimes B \cong A \times B$$

Proof. Suppose that $f : Z \longrightarrow A$ and $g : Z \longrightarrow B$. This means that we have *PTIME*-functions $f_0 : Z_0 \xrightarrow{PTIME} A_0$ and $f_1 : Z_0 \xrightarrow{PTIME} (Z_1 \multimap A_1)$, as well as $g_0 : Z_0 \xrightarrow{PTIME} B$ as indicated. The component $g_1 : Z_0 \xrightarrow{PTIME} Z_1 \multimap \top$ is trivial. Now the function sending $z \in |Z_0|$ to $(f_0(z), g_0(z)) \in |A_0 \otimes B_0|$ is a *PTIME*-function, too. Together with f_1 it furnishes the desired $\mathcal{H}^\Box$-morphism $\langle f, g \rangle : Z \longrightarrow A \otimes B$. □

Lemma 4.4.4 *An object $D = (D_0, D_1)$ is duplicable in $\mathcal{H}^\Box$ if either D_0 is empty or D_1 is duplicable in $\mathcal{H}$.*

Proof. Immediate calculation. □

Proposition 4.4.5 *The category $\mathcal{H}^\Box$ has linear exponentials $X \multimap Y$ if X and Y are safe. In this case $X \multimap Y$ is also safe and explicitly given by $(\top, X_1 \multimap Y_1)$.*

Proof. If X, Y are safe and $P = (P_0, P_1)$ is arbitrary then a $\mathcal{H}^\Box$ morphism from $P \otimes X$ to Y is given by a *PTIME*-function from P_0 to $(P_1 \otimes X) \multimap Y$. But $(P_1 \otimes X) \multimap Y$ is isomorphic to $P_1 \multimap (X \multimap Y)$ whence we obtain a *PTIME*-function from P_0 to $P_1 \multimap X \multimap Y$ which gives a $\mathcal{H}^\Box$-morphism from P to $X \multimap Y$. Inverting this process gives the other direction of the required natural isomorphism. □

We note that $\mathcal{H}^\Box$ is not an ALCC; in particular the linear function space $(\mathsf{N},\top){\multimap}(\top,\mathsf{N})$ does not exist in $\mathcal{H}^\Box$. Suppose for a contradiction that $A = (A_0, A_1)$ was such function space. Then, in particular, we would have an evaluation map $\mathrm{ev} : A_0 \otimes \mathsf{N} \xrightarrow{PTIME} A_1{\multimap}\mathsf{N}$ which has the property that for every "true" *PTIME*-function f there exist elements $a_0 \in |A_0|$ and $a_1 \in |A_1|$ such that $f(x) = \mathrm{ev}(a_0, x)(a_1)$. But this would mean that ev is a universal polynomial time computable function which is impossible by diagonalisation.

The lacking function spaces can be added to $\mathcal{H}^\Box$ by moving to the functor category $\mathrm{Ext}(\mathcal{H}^\Box)$ described in Chapter 2. In order that this functor category exists we must make sure that the category $\mathcal{H}^\Box$ is small.

This can be achieved by requiring that the underlying sets of H-sets be taken from a suitably chosen universe $\mathcal{U}$ closed under all set-theoretic operations required to form the H-sets of interest. Note that such universe can be defined by a simple inductive process and in particular no "large cardinal assumption" is needed for its existence.

It now follows from the results presented in Section 2.7.6 that $\mathrm{Ext}(\mathcal{H}^\Box)$ is an ALCC and that the Yoneda embedding $\mathcal{Y} : \mathcal{H}^\Box \longrightarrow \mathrm{Ext}(\mathcal{H}^\Box)$ preserves the ALC structure as well as existing linear function spaces and cartesian products. In particular, the linear function spaces between safe objects are preserved by the embedding.

Moreover, $\mathrm{Ext}(\mathcal{H}^\Box)$ supports a comonad ! with the property that whenever $D \in \mathcal{H}^\Box$ is duplicable then $!\,D = D$ in $\mathrm{Ext}(\mathcal{H}^\Box)$ and for arbitrary presheaf $F \in \mathrm{Ext}(\mathcal{H}^\Box)$ the presheaf $!\,F$ is duplicable.

4.4.1 Polynomial-time functions via a comonad

In this section we identify a comonad $\Box$ on $\mathrm{Ext}(\mathcal{H}^\Box)$ which has the property that if X is safe then $\Box(X) \cong (X, \top)$ so that by Prop. 2.7.8 we have $\Box(X){\multimap}F_{(Y_0,Y_1)} \cong F_{(Y_0 \otimes X, Y_1)}$.

For presheaf F we define $\Box F$ by

$$\Box F_{(Z_0,Z_1)} = F_{(Z_0,\top)}$$

Notice that if X is safe then $\Box(\mathcal{Y}(X)) \cong \mathcal{Y}(\Box X)$: At argument Z both presheaves consist of *PTIME*-functions from Z_0 to X.

If $Z \in \mathcal{H}^\Box$ write Z_0 for the *normal* object $(Z_0, \top)$ and $p_Z : Y \longrightarrow Z_0$ for the obvious projection arising from $Z \cong Z_0 \otimes Z_1$. The *counit* $\mathsf{unbox}_F : \Box F \longrightarrow F$ is then defined by $(\mathsf{unbox}_F)_Z = F_{p_Z}$.

Like in Section 2.6.12.3 we define the comultiplication as the identity in view of $\Box\Box F = \Box F$.

If $f : \Box F \longrightarrow G$ then the lifted morphism $f^\Box : \Box F \longrightarrow \Box G$ is given concretely by $f^\Box_{(Z_0,Z_1)} = f_{(Z_0,\top)}$.

Proposition 4.4.6 *For presheaves F, G we have $F \otimes \Box G = F \times \Box G$ and $\Box F{\multimap}G = \Box F{\rightarrow}G$ where $\times, \rightarrow$ are cartesian product and ordinary function space of presheaves. Moreover, $\Box(F \otimes G) = \Box F \otimes \Box G = \Box F \times \Box G$.*

Proof. Suppose that $X, Y \in \mathcal{H}^\Box$, write Y_0 for the normal object $(Y_0, \top)$. Suppose, furthermore, that $f \in F_X$ and $g \in \Box G_Y$. We have $(f, g) \in (F \otimes \Box G)_{(X,Y)}$ by

$$\begin{tikzcd}(X,Y) \arrow[r] & X \times Y_0 \arrow[r, "\text{Prop 4.4.3}"] & X \otimes Y_0\end{tikzcd}$$

and $\bar{f} = f, \bar{g} = g$.

The equality of the function spaces then is a direct consequence:

$$\begin{aligned} & \Box F \multimap G_{(X,Y)} \\ = \; & \mathrm{Ext}(\mathcal{H}^\Box)((X,Y) \otimes \Box F, G) \\ = \; & \mathrm{Ext}(\mathcal{H}^\Box)((X,Y) \times \Box F, G) \\ = \; & \Box F \to G_{(X,Y)} \end{aligned}$$

The last part is similar to the first. □

Proposition 4.4.7 *Let $F \in \mathrm{Ext}(\mathcal{H}^\Box)$. We have $!\Box F = \Box ! F = \Box F$.*

Proof. Clearly, $\Box ! F \subseteq \Box F$ and $!\Box F \subseteq \Box F$. To show $\Box F \subseteq !\Box F$ we notice that whenever $X = (X_0, X_1) \in \mathcal{H}^\Box$ then $D = (X_0, \top)$ is duplicable and so, if $f \in \Box F_X$ then $f \in !\Box F_X$ can be witnessed by the projection $X \longrightarrow D$ and f itself. Since D is normal this also shows the other inclusion. □

Finally, we notice that $\mathcal{G}(\Box F) \cong \mathcal{G}(F)$ as $\mathcal{G}(\Box F) \cong F_\top = F_{(\top,\top)} = \Box F_{(\top,\top)} \cong \mathcal{G}(\Box F)$. Also recall from Prop. 2.7.9 that $\mathcal{G}(! F) = \mathcal{G}(F)$. We will now see how the recursion patterns defined in Theorems 4.3.4 and 4.3.6 can be lifted to $\mathrm{Ext}(\mathcal{H}^\Box)$.

Theorem 4.4.8 *Let X be a safe presheaf. There exists a global element*

$$\mathsf{rec}^{\mathsf{N}} : !(\Box\mathsf{N} \multimap X \multimap X) \multimap \Box\mathsf{N} \multimap X \multimap X$$

such that the following equation holds for global elements $h : \Box\mathsf{N} \multimap X \multimap X$, $g : X$, $x \in \mathcal{G}(\mathsf{N}) \setminus \{0\}$.

$$\begin{aligned} &\mathsf{rec}^{\mathsf{N}}\ h\ 0\ g = g \\ &\mathsf{rec}^{\mathsf{N}}\ h\ x\ g = h\ x\ (\mathsf{rec}^{\mathsf{N}}\ h\ \left\lfloor \tfrac{x}{2} \right\rfloor\ g) \end{aligned}$$

Proof. We must define a natural transformation from $!(\Box\mathsf{N} \multimap X \multimap X)$ to $\Box\mathsf{N} \multimap X \multimap X$.

Suppose that $Z \in \mathcal{H}^\Box$ and assume

$$h \in !(\Box\mathsf{N} \multimap X \multimap X)_Z$$

witnessed by $t : Z \longrightarrow D$ and $\bar{h} \in (\Box\mathsf{N}\multimap X\multimap X)_D$ where $D = (D_0, D_1)$ is duplicable.

Now, in view of Prop. 2.7.8 we have

$$(\Box\mathsf{N}\multimap X\multimap X)_Z \cong \mathcal{H}^{\Box}((Z_0 \otimes \mathsf{N}, Z_1)\,,\,(\top, X\multimap X))$$
$$(\Box\mathsf{N}\multimap X\multimap X)_D \cong \mathcal{H}^{\Box}((D_0 \otimes \mathsf{N}, D_1)\,,\,(\top, X\multimap X))$$

$$\bar{h} : D_0 \otimes \mathsf{N} \xrightarrow{PTIME} (D_1 \otimes X)\multimap X$$

and

$$h : Z_0 \otimes \mathsf{N} \xrightarrow{PTIME} (Z_1 \otimes X)\multimap X$$

and $t_0 : Z_0 \xrightarrow{PTIME} D_0$, $t_1 : Z_0 \xrightarrow{PTIME} Z_1\multimap D_1$ and

$$h(z_0, x)(z_1, g) = \bar{h}(t_0(z_0), x)(t_1(z_0, z_1), g)$$

We must define (again, neglecting isomorphism) a function

$$f = \mathsf{rec}^{\mathsf{N}}_Z(h) : Z_0 \otimes \mathsf{N} \xrightarrow{PTIME} Z_1\multimap X\multimap X$$

such that

$$f(0)(g) = g$$
$$f(x)(g) = h(z_0, x)(f(\left\lfloor \tfrac{x}{2} \right\rfloor)(g))$$

Since $D = (D_0, D_1)$ is duplicable we know by Lemma 4.4.4 that either D_0 is empty or D_1 is a duplicable H-set. In the former case, Z_0 must also be empty and so f will be the empty function. Otherwise, Corollary 4.3.5 gives us a function $f' : D_0 \otimes \mathsf{N} \xrightarrow{PTIME} D_1\multimap X\multimap X$ such that

$$f'(d_0, 0)(d_1, g) = g$$
$$f'(d_0, x)(d_1, g) = \bar{h}(d_0, x)(f'(d_0, \left\lfloor \tfrac{x}{2} \right\rfloor)(g))$$

Now we define

$$f(z_0, x) = \lambda z_1 g. f'(t_0(z_0), x)\ (t_1(z_0)\ z_1)\ g$$

and the desired equations follow by induction on x. Since the thus defined f is uniquely determined by the recursive equations it does not depend on the witness $\bar{h}$.

Naturality of the assignment $f \mapsto \mathsf{rec}^{\mathsf{N}}(h)$ means that recursive definition respects substitution of parameters and is also readily established by induction on x. □

Similarly, we can lift tree recursion to $\mathrm{Ext}(\mathcal{H}^{\Box})$:

Theorem 4.4.9 *Let A, X be safe presheaves. There exists a global element*

$$\mathsf{rec}^{\mathsf{T}(A)} : !(\Box A\multimap X)\multimap\,!(\Box A\multimap\Box\mathsf{T}(A)\multimap\Box\mathsf{T}(A)\multimap X\multimap X\multimap X)\multimap\Box\mathsf{T}(A)\multimap X$$

such that the following equations are valid for appropriate global elements.

$$\mathsf{rec}^{\mathsf{T}(A)}(g, h, \mathsf{leaf}(a)) = g$$
$$\mathsf{rec}^{\mathsf{T}(A)}(g, h, \mathsf{node}(a, l, r)) = h(a, l, r, \mathsf{rec}^{\mathsf{T}(A)}(g, h, l), \mathsf{rec}^{\mathsf{T}(A)}(g, h, r))$$

Proof. Analogous to the previous one, this time using Cor. 4.3.7. □

In this way, other recursion patterns we might be interested in can also be lifted to $\mathrm{Ext}(\mathcal{H}^{\Box})$.

4.4.2 Interpretation of SLR

We are now ready to define an interpretation of SLR without rule S-Ax in $\mathrm{Ext}(\mathbb{C})$. We show later in Section 4.5 how to encompass that rule if so desired.

To each aspect a we associate a functor

$$\begin{aligned}&F_a(X) = X, \text{ if } a = (\text{nonmodal}, \text{linear})\\ &F_a(X) = {!}\,X, \text{ if } a = (\text{nonmodal}, \text{nonlinear})\\ &F_a(X) = \Box X, \text{ if } a = (\text{modal}, \text{nonlinear})\end{aligned}$$

We also define a natural transformation $\varepsilon_a : F_a \longrightarrow Id$ as either the identity, $\mathsf{derelict}$, or unbox.

If $f : F_a(X) \longrightarrow Y$ then we write $f^a : F_a(X) \longrightarrow F_a(X)$ for the Kleisli lifting of f w.r.t. F_a according to Section 2.6.12.1. If $a <: a'$ then we define a natural transformation $\iota_{a,a'} : F_a(X) \longrightarrow F_{a'}(X)$ by

$$\begin{aligned}&\iota_{a,a'} = \varepsilon_a && \text{if } a = (\text{nonmodal}, \text{linear})\\ &\iota_{a,a'} = \mathrm{id} && \text{if } a = a'\\ &\iota_{a,a'} = !(\mathsf{unbox}) = \mathsf{unbox}^! && \text{if } a = (\text{modal}, \text{nonlinear}), a' = (\text{nonmodal}, \text{nonlinear})\end{aligned}$$

Let η be a partial function mapping type variables to objects of $\mathcal{H}$ and A be a type. The presheaf $[\![A]\!]\eta$ is defined by

$$\begin{aligned}&[\![X]\!]\eta = \eta(X)\\ &[\![\mathsf{N}]\!]\eta = \mathsf{N}\\ &[\![\mathsf{L}(A)]\!]\eta = \mathsf{L}([\![A]\!]\eta)\\ &[\![\mathsf{T}(A)]\!]\eta = \mathsf{T}([\![A]\!]\eta)\\ &[\![A \xrightarrow{a} B]\!]\eta = F_a([\![A]\!]\eta) \multimap [\![B]\!]\eta\\ &[\![\forall X.A]\!]\eta = \textstyle\prod_{B \in \mathcal{H}} [\![A]\!]\eta[X \mapsto B]\\ &[\![A \times B]\!]\eta = [\![A]\!]\eta \times [\![B]\!]\eta\\ &[\![A \otimes B]\!]\eta = [\![A]\!]\eta \otimes [\![B]\!]\eta\end{aligned}$$

Here $\prod_{B \in \mathcal{H}}$ is a $|\mathcal{H}|$-indexed cartesian product of presheaves, see Section 2.6.10.2.

We notice that if A is safe then $[\![A]\!]\eta$ is a safe object so that the defining clauses for $\mathsf{L}(A)$ and $\mathsf{T}(A)$ make sense.

A context $\Gamma = x_1 \overset{a_1}{:} A_1, \dots, x_n \overset{a_n}{:} A_n$ gets interpreted as the tensor product

$$[\![\Gamma]\!]\eta =_{\mathrm{def}} F_{a_1}([\![A_1]\!]\eta) \otimes \cdots \otimes F_{a_n}([\![A_n]\!]\eta)$$

A *derivation* of a judgement $\Gamma \vdash e : A$ gets interpreted as a morphism

$$[\![\Gamma \vdash e : A]\!]\eta : [\![\Gamma]\!]\eta \longrightarrow [\![A]\!]\eta$$

Before actually defining this interpretation let us warn the reader that we will *not* prove that the interpretation is independent of the chosen typing derivation. Neither will we prove that it enjoys one or the other substitution property and neither will we prove that it validates whatsoever equational theory between terms. We are confident that such properties could be established if so desired, but they are not needed for the present development.

4.4.2.1 Constants and variables

A variable gets interpreted as the corresponding projection morphism possibly followed by a counit ε_a. For example, if $a = (\text{modal}, \text{nonlinear})$ then $\Gamma, x \overset{a}{:} A \vdash x : A$ gets interpreted as the morphism

$$[\![\Gamma, x \overset{a}{:} A]\!]\eta = [\![\Gamma]\!]\eta \otimes \Box[\![A]\!]\eta \xrightarrow{\textsf{projection}} \Box[\![A]\!]\eta \xrightarrow{\textsf{unbox}} [\![A]\!]\eta$$

The first order constants such as $\mathsf{S}_0, \mathsf{S}_1, \mathsf{node}$, etc. get interpreted by applying the Yoneda embedding to their interpretations in $\mathcal{H}^\Box$. The recursors are interpreted as the terminal projection $[\![\Gamma]\!] \longrightarrow \top$ followed by the global elements defined in Theorems 4.4.8, 4.4.9.

4.4.2.2 Application and abstraction

It follows from Propositions 2.7.9, 4.4.7, and Lemma 2.7.3 that $[\![\Gamma]\!]\eta$ is duplicable whenever Γ is nonlinear.

More generally, if Γ is nonlinear and Δ_1, Δ_2 are arbitrary as in rules T-Arr-E and T-Tens-E then we can define a map

$$v_{\Gamma,\Delta_1,\Delta_2} : [\![\Gamma, \Delta_1, \Delta_2]\!]\eta \longrightarrow [\![\Gamma, \Delta_1]\!]\eta \otimes [\![\Gamma, \Delta_2]\!]\eta$$

as

$$v_{\Gamma,\Delta_1,\Delta_2} =_{\text{def}} w_1 \circ (\delta \otimes ([\![\Delta_1]\!]\eta \otimes [\![\Delta_2]\!]\eta)) \circ w_2$$

where δ is the diagonal on $[\![\Gamma]\!]\eta$ and w_1, w_2 are wiring maps.

Suppose now that $\Gamma, x \overset{a}{:} A \vdash e : B$ and let $f : [\![\Gamma]\!]\eta \otimes F_a([\![A]\!]\eta) \longrightarrow [\![B]\!]\eta$ be the interpretation of e. The currying or exponential transpose of this morphism yields a map $[\![\Gamma]\!] \longrightarrow F_a([\![A]\!]\eta) \multimap [\![B]\!]\eta$ which serves as the interpretation of $\lambda x{:}\, A.e : A \xrightarrow{a} B$.

Now suppose that

$$\begin{array}{l} \Gamma, \Delta_1 \vdash e_1 : A \xrightarrow{a} B \\ \Gamma, \Delta_2 \vdash e_2 : A \\ \Gamma, \Delta_2 <: a \\ \Gamma \text{ nonlinear} \end{array}$$

as in the premises to rule T-Arr-E and let

$$\begin{array}{l} f_1 : [\![\Gamma]\!]\eta \otimes [\![\Delta_1]\!]\eta \longrightarrow F_a([\![A]\!]\eta) \multimap [\![B]\!]\eta \\ f_2 : [\![\Gamma]\!]\eta \otimes [\![\Delta_2]\!]\eta \longrightarrow [\![A]\!]\eta \end{array}$$

be the interpretations of e_1 and e_2. The side condition on the aspects in Γ, Δ_2 together with the fact that $\Box$ commutes with $\otimes$ allows us to "raise" f_2 to a morphism $f^{\Box} : [\![\Gamma]\!]\eta \otimes [\![\Delta_2]\!]\eta \longrightarrow F_a([\![A]\!]\eta)$.

Now the interpretation of $e_1 e_2$ is obtained as

$$\mathsf{ev} \circ (f_1 \otimes f_2^{\Box}) \circ v_{\Gamma,\Delta_1,\Delta_2}$$

where

$$\mathsf{ev} : (F_a([\![A]\!]\eta) \multimap [\![B]\!]\eta) \otimes [\![A]\!]\eta \longrightarrow [\![B]\!]\eta$$

is the evaluation morphism and $v_{\Gamma,\Delta_1,\Delta_2}$ is defined as described above using the fact that Γ is nonlinear.

4.4.2.3 Polymorphic abstraction and application

Suppose that $\Gamma \vdash e : A$ and that X does not occur in Γ. Then the interpretation of $\Gamma \vdash \Lambda X.e : \forall X.A$ is defined by

$$[\![\Gamma \vdash \Lambda X.e : \forall X.A]\!]\eta = \langle [\![\Gamma \vdash e : A]\!]\eta[X \mapsto B] \mid B \in \mathcal{H} \rangle$$

If $\Gamma \vdash e : \forall X.A$ and B is safe then we define

$$[\![\Gamma \vdash e[B] : A[B/X]]\!]\eta = \pi_{[\![B]\!]\eta} \circ [\![\Gamma \vdash e : \forall X.A]\!]\eta$$

4.4.2.4 Cartesian products

Suppose that $\Gamma \vdash e_1 : A_1$, $\Gamma \vdash e_2 : A_2$. Then we define

$$[\![\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2]\!]\eta = \langle [\![\Gamma \vdash e_1 : A_1]\!]\eta, [\![\Gamma \vdash e_2 : A_2]\!]\eta \rangle$$

where $\langle -, - \rangle$ is the pairing operation associated with cartesian products in $\mathcal{H}^{\Box}$. If $\Gamma \vdash e : A_1 \times A_2$ then we define

$$[\![\Gamma \vdash e.i : A_i]\!]\eta = \pi_i \circ [\![\Gamma \vdash e : A_1 \times A_2]\!]\eta$$

where $\pi_i : [\![A_1 \times A_2]\!]\eta \longrightarrow [\![A_i]\!]\eta$ is the projection morphism.

4.4.2.5 Tensor products

Suppose that $\Gamma, \Delta_i \vdash e_i : A_i$ and that Γ is nonlinear as in the premise to rule T-Tens-I. Using the diagonal on $[\![\Gamma]\!]\eta$ we can define a map

$$v : [\![\Gamma, \Delta_1, \Delta_2]\!]\eta \longrightarrow [\![\Gamma, \Delta_1]\!]\eta \otimes [\![\Gamma, \Delta_2]\!]\eta$$

as in the previous case. Then we define,

$$[\![\Gamma, \Delta_1, \Delta_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2]\!]\eta = ([\![\Gamma, \Delta_1 \vdash e_1 : A_1]\!]\eta \otimes [\![\Gamma, \Delta_2 \vdash e_2 : A_2]\!]\eta) \circ v$$

Now suppose that

$$\begin{array}{l} \Gamma, \Delta_1 \vdash e_1 : A_1 \otimes A_2 \\ \Gamma, \Delta_2, x \overset{a_1}{:} A_1, x \overset{a_2}{:} A_2 \vdash e_2 : B \\ \Gamma, \Delta_1 <: a_1 \wedge a_2 \\ \Gamma \text{ nonlinear} \end{array}$$

as in the premise to T-Tens-E.

Let $f_1 : [\![\Gamma, \Delta_{\vdash} e_1]\!]\eta \longrightarrow [\![A_1 \otimes A_2]\!]$ and $f_2 : [\![\Gamma, \Delta_2, x \overset{a_1}{:} A_1, x \overset{a_2}{:} A_2]\!] \longrightarrow [\![B]\!]$ be the interpretations of e_1 and e_2.

Since $\Gamma, \Delta_1 <: a_1 \wedge a_2$ we can raise f_1 to form a morphism

$$f_1^{a_1 \wedge a_2} : [\![\Gamma, \Delta_1]\!] \longrightarrow F_a([\![A_1]\!]) \otimes F_a([\![A_2]\!])$$

from which we obtain

$$g =_{\text{def}} (\iota_{a_1 \wedge a_2, a_2} \otimes \iota_{a_1 \wedge a_2, a_2}) \circ f_1^{a_1 \wedge a_2} : [\![\Gamma, \Delta_1]\!] \longrightarrow F_{a_1}([\![A_1]\!]) \otimes F_{a_2}([\![A_2]\!])$$

We then define the interpretation of $\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1 = x \otimes y \,\mathsf{in}\, e_2 : B$ as

$$f_2 \circ ([\![\Gamma, \Delta_2]\!] \otimes g) \circ v_{\Gamma, \Delta_1, \Delta_2}$$

where $v_{\Gamma, \Delta_1, \Delta_2}$ is as above in Section 4.4.2.2

4.4.2.6 Subtyping and subsumption

By induction on the definition of subtyping we define a coercion map $\iota_{A,B} : [\![A]\!]\eta \longrightarrow [\![B]\!]\eta$ when $A <: B$ using identity, composition, the mappings $\iota_{a,a'}$, and functoriality of the semantic type formers $\multimap, \otimes, \times, \prod$.

If $\Gamma \vdash e : B$ was obtained from $\Gamma \vdash e : A$ and $A <: B$ by rule T-Sub then we define

$$[\![\Gamma \vdash e : B]\!]\eta = \iota_{A,B} \circ [\![\Gamma \vdash e : A]\!]\eta$$

4.4.3 Main result

Let us temporarily write $[\![-]\!]^S$ for the set-theoretic interpretation and $[\![-]\!]^F$ for the interpretation in $\mathrm{Ext}(\mathcal{H}^\Box)$.

If $f \in \mathrm{Ext}(\mathcal{H}^\Box)(\Box\mathsf{N}, \mathsf{N})$ then it follows from the Yoneda Lemma and the discussion in Section 2.6.10.4 that $\mathcal{G}(f) : \mathbb{N} \cong \mathcal{G}(\Box\mathsf{N}) \longrightarrow \mathcal{G}(\mathsf{N}) \cong \mathbb{N}$ is a polynomial time computable function. This means that if $e : \Box\mathsf{N} \to \mathsf{N}$ is a closed term then $\mathcal{G}([\![f]\!]^F)$ is a *PTIME*-function and the same goes for functions with several arguments. It remains to show that the thus obtained function coincides with the intended set-theoretic meaning. To do this, we use again a logical relation.

Let η, ζ be partial functions mapping type variables to sets in $\mathcal{U}$ and safe presheaves, respectively. Let ρ be a partial function which assigns to each type variable X a binary relation $\rho(X) \subseteq \eta(X) \times \mathcal{G}(\zeta(X))$. We define a relation $R_{\eta,\zeta,\rho}(A) \subseteq [\![A]\!]^S\eta \times \mathcal{G}([\![A]\!]^F\zeta)$ by

$$\begin{array}{rcl}
x\, R_{\eta,\zeta,\rho}(X)\, y & \iff & x\, \rho(X)\, y \\
x\, R_{\eta,\zeta,\rho}(\mathsf{N})\, y & \iff & x = y \\
u\, R_{\eta,\zeta,\rho}(A \xrightarrow{a} B)\, v & \iff & \\
\multicolumn{3}{c}{\forall x \in [\![A]\!]^S\eta. \forall y \in \mathcal{G}([\![A]\!]^F\rho). x\, R_{\eta,\zeta,\rho}(A)\, y \Rightarrow u(x)\, R_{\eta,\zeta,\rho}(B)\, v(y)} \\
u\, R_{\eta,\zeta,\rho}(\forall X.A)\, v & \iff & \\
\multicolumn{3}{c}{\forall U, V \in \mathcal{H}. \forall R \subseteq |U| \times |V|. \pi_U(u)\, R_{\eta[X\mapsto U],\zeta[X\mapsto V],\rho[X\mapsto R]}(A)\, \pi_V(v)} \\
(u_1, u_2)\, R_{\eta,\zeta,\rho}(A_1 \times A_2)\, (v_1, v_2) & \iff & u_i\, R_{\eta,\zeta,\rho}(A_i)\, v_i \ i = 1, 2 \\
(u_1, u_2)\, R_{\eta,\zeta,\rho}(A_1 \otimes A_2)\, (v_1, v_2) & \iff & u_i\, R_{\eta,\zeta,\rho}(A_i)\, v_i \ i = 1, 2
\end{array}$$

The relations $R_{\eta,\zeta,\rho}(\mathsf{L}(A))$ and $R_{\eta,\zeta,\rho}(\mathsf{T}(A))$ are defined inductively by

$$\begin{array}{l}
\mathsf{nil}\, R_{\eta,\zeta,\rho}(\mathsf{L}(A))\, \mathsf{nil} \quad \text{always} \\
\mathsf{cons}(a, l)\, R_{\eta,\zeta,\rho}\, \mathsf{cons}(a', l') \iff \\
\qquad a\, R_{\eta,\zeta,\rho}(A)\, a' \wedge l\, R_{\eta,\zeta,\rho}(\mathsf{L}(A))\, l'
\end{array}$$

$$\begin{array}{l}
\mathsf{leaf}(a)\, R_{\eta,\zeta,\rho}(\mathsf{T}(A))\, \mathsf{leaf}(a') \iff a\, R_{\eta,\zeta,\rho}(A)\, a' \\
\mathsf{node}(a, l, r)\, R_{\eta,\zeta,\rho}(\mathsf{T}(A))\, \mathsf{node}(a', l', r') \iff \\
\qquad a\, R_{\eta,\zeta,\rho}(A)\, a' \wedge l\, R_{\eta,\zeta,\rho}(\mathsf{T}(A))\, l' \wedge r\, R_{\eta,\zeta,\rho}(\mathsf{T}(A))\, r'
\end{array}$$

Now, a direct inspection of the definitions, and a reasoning similar to the one in Section 2.6.10.6 shows that we have

$$[\![c]\!]^S R_A \mathcal{G}([\![c]\!]^F)$$

for every SLR-constant $c : A$.

Next we show by induction on subtyping derivations that whenever $A <: B$ then

$$\forall x \in [\![A]\!]^S\eta. \forall y \in \mathcal{G}([\![A]\!]^F\rho). x\ R_{\eta,\zeta,\rho}(A)\ y \Rightarrow x\ R_{\eta,\zeta,\rho}(B)\ \iota_{A,B}(y)$$

Notice here that $[\![A]\!]^S = [\![B]\!]^S$ whenever $A <: B$. Finally, by induction on typing derivations we prove the following extension of Theorem 2.6.2

Proposition 4.4.10 *Suppose that $\Gamma \vdash e : A$ and that η, ζ, ρ are assignments as above. Suppose, further that $\gamma^S \in \mathcal{G}([\![\Gamma]\!]^S)$ and $\gamma^F \in \mathcal{G}([\![\Gamma]\!]^F)$ are such that $\gamma^S(x)\ R_{\eta,\zeta,\rho}(\Gamma(x))\ \gamma^F(x)$ for each $x \in \mathrm{dom}(\Gamma)$. Then*

$$[\![\Gamma \vdash e : A]\!]\eta^S(\gamma^S)\ R_{\eta,\zeta,\rho}(A)\ [\![\Gamma \vdash e : A]\!]\rho^F$$

Corollary 4.4.11 *The set-theoretic interpretation of a closed term $f : \Box\mathsf{N} \to \mathsf{N}$ is a polynomial time computable function.*

Proof. By specialising the above proposition to $\Gamma = \emptyset, A = \Box\mathsf{N}\to\mathsf{N}$ and expanding the definitions. $\Box$

4.4.4 Interpretation of $\lambda^{\Box !}$

We can use $\mathrm{Ext}(\mathcal{H}^\Box)$ also in order to give meaning to the alternative system $\lambda^{\Box !}$ thus providing a proof that also the definable functions of the latter system are *PTIME*. Spelling this out in detail would be a rather boring exercise in typesetting so we will only set out a few of the important points. Types are interpreted as objects of $\mathrm{Ext}(\mathcal{H}^\Box)$. The new type formers $\Box, !$ are interpreted by the eponymous comonads on $\mathrm{Ext}(\mathcal{H}^\Box)$. The associated operations on morphisms provide meaning for the term formers associated with the modalities. The defining clauses for cartesian and tensor product follow the interpretation of SLR.

It seems plausible that any other reasonable formulation of modal/linear lambda calculus including the one in [2] can be interpreted in $\mathrm{Ext}(\mathcal{H}^\Box)$ in a similar fashion.

4.5 Duplicable numerals

The algebra H has the disadvantage that even values of type N may not be duplicated, i.e, there does not exist an element $\delta \in H$ such that $\delta\mathsf{num}(x) = \lambda f.f\mathsf{num}(x)\mathsf{num}(x)$. Indeed, assuming such diagonal element would contradict Theorem 4.3.4 for the following reason. Multiplication is easily seen to be a morphism from $\mathsf{N}\otimes\mathsf{N}$ to N. Using the hypothetical δ we could realise the diagonal function from N to $\mathsf{N}\otimes\mathsf{N}$ and thus by composition the squaring function would be a morphism from N to N. Iterating it using safe recursion on notation would allow us to define a function of exponential growth.

In Bellantoni-Cook's original system and in SLR duplication of values of ground type is, however, permitted and sound. The reason is that multiplication is not among the basic functions of these systems and as an invariant it is maintained that a function depending on several safe arguments of ground type is bounded by the maximum of these arguments plus a constant. Obviously, multiplication does not have this property.

In order to obtain an analogue of the algebra H we need to get a handle on the maximum of the lengths of the two components of a pair. This motivates the following definitions. First, like in Lemma 2.1.1 we fix disjoint injections

$$\begin{aligned}\mathsf{num} &: \mathbb{N} \longrightarrow \mathbb{N}\\ \mathsf{pad} &: \mathbb{N} \longrightarrow \mathbb{N}\\ \langle\cdot,\cdot\rangle &: \mathbb{N}\times\mathbb{N}\to\mathbb{N}\end{aligned}$$

together with test functions isnum, ispad, ispair and inverses getnum, getpad, $.1$, $.2$ computable in linear time and satisfying specifications and size restrictions analogous to the ones in Lemma 2.1.1. We need two bits now to distinguish the ranges of the injections, so we will have $|\mathsf{num}(x)| = |\mathsf{pad}(x)| = 2$ and $|\langle u, v\rangle| = |u| + |v| + 2||v|| + 4$. (Recall that two bits are needed to separate the $|v|$ block from u, v.)

Definition 4.5.1 (Linear length, maximum length) *The length functions $\ell_{\rm lin}(x)$ (linear length) and $\ell_{\rm max}(x)$ (maximum length) are defined recursively as follows.*

$$\begin{aligned}
&\ell_{\rm lin}(\mathsf{num}(x)) = 1\\
&\ell_{\rm max}(\mathsf{num}(x)) = |x|\\
&\ell_{\rm lin}(\mathsf{pad}(x)) = |x| + 1\\
&\ell_{\rm max}(\mathsf{pad}(x)) = 0\\
&\ell_{\rm lin}(\langle x, y\rangle) = 4 + \ell_{\rm lin}(x) + \ell_{\rm lin}(y)\\
&\ell_{\rm max}(\langle x, y\rangle) = \max(\ell_{\rm max}(x), \ell_{\rm max}(y))\\
&\ell_{\rm lin}(x) = |x|,\ \textit{otherwise}\\
&\ell_{\rm max}(x) = 0,\ \textit{otherwise}
\end{aligned}$$

We may assume that $\ell_{\rm lin}(0) = \ell_{\rm max}(0) = 0$.

Lemma 4.5.2 *The following inequalities hold for every $x \in \mathbb{N}$.*

$$\begin{aligned}
|x| &\geq \ell_{\rm lin}(x) + \ell_{\rm max}(x)\\
|x| &\leq \ell_{\rm lin}(x)(1 + ||x|| + \ell_{\rm max}(x))
\end{aligned}$$

Proof. By course of values induction on x. If $x = \mathsf{num}(y)$ then

$$|x| = |y| + 2 \geq 1 + |y| = \ell_{\rm lin}(x) + \ell_{\rm max}(x)$$

and

$$|x| = |y| + 2 \leq 1 \cdot (1 + 2 + \ell_{\rm max}(x)) \leq \ell_{\rm lin}(x)(1 + ||x|| + \ell_{\rm max}(x))$$

since $||2|| = 2$.

If $x = \mathsf{pad}(y)$ then

$$|x| = |y| + 2 \geq |y| + 1 = \ell_{\rm lin}(x) + \ell_{\rm max}(x)$$

and

$$|x| = |y| + 2 \leq (|y| + 1)(1 + 2) \leq \ell_{\rm lin}(x)(1 + ||x|| + \ell_{\rm max}(x))$$

The interesting case is when $x = \langle u, v\rangle$. In this case, we have

$$\begin{aligned}
|\langle u, v\rangle| \geq |u| + |v| + 4 &\overset{\mathsf{IH}}{\geq} \ell_{\rm lin}(u) + \ell_{\rm lin}(v) + \ell_{\rm max}(u) + \ell_{\rm max}(v) + 4 \geq\\
&\geq \ell_{\rm lin}(\langle u, v\rangle) + \ell_{\rm max}(\langle u, v\rangle)
\end{aligned}$$

thus establishing the first inequality. For the second we calculate as follows.

$$\begin{aligned}
& \ell_{\mathrm{lin}}(\langle u,v\rangle)(1+||\langle u,v\rangle||+\ell_{\max}(\langle u,v\rangle) \\
= \; & (\ell_{\mathrm{lin}}(u)+\ell_{\mathrm{lin}}(v)+4)(1+||\langle u,v\rangle||+\max(\ell_{\max}(u),\ell_{\max}(v))) \\
\geq \; & \ell_{\mathrm{lin}}(u)(1+||u||+\ell_{\max}(u))+\ell_{\mathrm{lin}}(v)(1+||v||+\ell_{\max}(v))+4(1+||\langle u,v\rangle||) \\
\overset{\mathsf{IH}}{\geq} \; & |u|+|v|+2||v||+4 \\
= \; & |\langle u,v\rangle|
\end{aligned}$$

In the remaining case we have

$$|x| \geq |x|+0=\ell_{\mathrm{lin}}(x)+\ell_{\max}(x)$$

and

$$|x| \leq |x|(1+||x||)=\ell_{\mathrm{lin}}(x)(1+||x||+\ell_{\max}(x))$$

□

Corollary 4.5.3 *There exists a quadratic polynomial p such that $|x| \leq p(\ell_{\mathrm{lin}}(x)+\ell_{\max}(x))$.*

Proof. Writing u for $\sqrt{|x|}$ we obtain for

$$u^2 \leq \ell_{\mathrm{lin}}(x)(1+u+\ell_{\max}(x))$$

for x large enough from $||x|| \leq \sqrt{|x|}$. This gives

$$u \leq \ell_{\mathrm{lin}}(x)+\sqrt{\ell_{\mathrm{lin}}(x)^2+4\ell_{\mathrm{lin}}(x)(1+\ell_{\max}(x))} \leq \ell_{\mathrm{lin}}(x)+(\ell_{\mathrm{lin}}(x)+2+2\ell_{\max}(x))$$

hence the result by squaring. □

Accordingly, any function computable in time $O(|x|)$ is computable in time $O((\ell_{\mathrm{lin}}(x)+\ell_{\max}(x))^2)$.

We assume an encoding of computations such that for each algorithm e and integer $N \geq \ell_{\mathrm{lin}}(e)$ we can find an algorithm e' such that the runtime of $\{e'\}(x)$ is not greater than $|N|$ plus the time needed to compute $\{e\}(x)$ and such that $\ell_{\mathrm{lin}}(e') \geq N$. Similarly, we assume that we can arbitrarily increase the maximum length of an algorithm. That this is in principle possible hinges on the two injections num and pad.

Let $p > 2$ be a fixed integer.

Definition 4.5.4 *A computation $\{e\}(x)$ is called* short *(w.r.t. p) if it needs less than $d(\ell_{\mathrm{lin}}(e)+\ell_{\mathrm{lin}}(x)+\max(\ell_{\max}(e),\ell_{\max}(x)))^p$ steps where $d=\ell_{\mathrm{lin}}(e)+\ell_{\mathrm{lin}}(x)-\ell_{\mathrm{lin}}(\{e\}(x))$ and in addition $\ell_{\max}(\{e\}(x)) \leq d+\max(\ell_{\max}(e),\ell_{\max}(x))$. The difference d between $\ell_{\mathrm{lin}}(e)+\ell_{\mathrm{lin}}(x)$ and $\ell_{\mathrm{lin}}(\{e\}x)$ is again called the defect of the computation $\{e\}(x)$.*

An algorithm e is called short if $\{e\}(x)$ is short for all x.

For what follows it is useful to recall the following basic rules of "maxplus-arithmetic".

- $x + \max(y, z) = \max(x + y, x + z)$
- more generally, $f(\max(x, y)) = \max(f(x), f(y))$ whenever f is monotone.
- $\max(x, y) \leq z \iff x \leq z \wedge y \leq z$.
- $\max(x, y) \leq x + y$.
- $\max(x, y + z) \leq y + \max(x, z)$.

The following shows that maximum-bounded number-theoretic functions are computable by short algorithms.

Lemma 4.5.5 *If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is computable in time $O(\max(|x_1|, \ldots, |x_n|)^p)$ and*

$$|f(x_1, \ldots, x_n)| = \max(|x_1|, \ldots, |x_n|) + O(1)$$

then there exists a short algorithm e such that

$$f(x_1, \ldots, x_n) = \{e\}(\langle \mathsf{num}(x_1), \langle \mathsf{num}(x_2), \ldots, \mathsf{num}(x_n)\rangle \ldots \rangle)$$

Proof. Immediate from the definitions. □

Proposition 4.5.6 *There exists a function $\mathsf{app} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ and a constant γ such that*

- *$\mathsf{app}(e, x)$ is computable in time $d(\ell_{\text{lin}}(e) + \ell_{\text{lin}}(x) + \gamma + \max(\ell_{\max}(e), \ell_{\max}(x)))^p$ where $d = \ell_{\text{lin}}(e) + \ell_{\text{lin}}(x) - \ell_{\text{lin}}(\mathsf{app}(e, x))$.*
- *$\ell_{\max}(\mathsf{app}(e, x)) \leq d + \max(\ell_{\max}(e), \ell_{\max}(x)))$*
- *$\mathsf{app}(e, x) = \{e\}(x)$ for every short computation $\{e\}(x)$.*

Proof. The following algorithm meets the required specification

```
read $e, x$;
$conf := \text{init}(e, x)$;
$t := 0$;
$T := (\ell_{\text{lin}}(e) + \ell_{\text{lin}}(x) + \max(\ell_{\max}(e) + \ell_{\max}(x)))^p$
while $t \leq (\ell_{\text{lin}}(e) + \ell_{\text{lin}}(x))T$ and $\text{term}(conf) \neq 0$ do
begin
  $conf := \text{step}(conf)$;
  $t := t + 1$;
end
$r := \mathsf{out}(conf)$;
$d := \ell_{\text{lin}}(e) + \ell_{\text{lin}}(x) - \ell_{\text{lin}}(r)$;
if
  $\text{term}(conf) = 0$ and
  $t \leq dT$ and
  $\ell_{\max}(r) \leq d + \max(\ell_{\max}(e), \ell_{\max}(x))$
then write $r$
else write $0$
```

The verification runs analogous to the one in Prop. 4.1.6. □

Again, we will abbreviate $\mathsf{app}(e, x)$ by $e \cdot x$ or ex.

Lemma 4.5.7 (Parametrisation) *For every e there exists an algorithm e' such that $e\langle x, y\rangle = e'xy$.*

Proof. Let e_0 be the following algorithm:

```
read $x$
write "read $y$; write app$(e, \langle `x, y\rangle)$"
```

Now, $\ell_{\text{lin}}(\{e_0\}(x)) \leq \ell_{\text{lin}}(e) + \ell_{\text{lin}}(x) + c$ and $\ell_{\max}(\{e_0\}(x)) \leq \max(\ell_{\max}(e), \ell_{\max}(x))$ where c is some fixed constant.

Sufficient padding of e_0 thus yields the desired algorithm e'. □

Theorem 4.5.8 *The set of natural numbers together with the above application function* app *is a BCK-algebra.*

Proof. The K-combinator is straightforward. For the composition combinator B we look again at the algorithm from the proof of Theorem 4.1.8.

```
B_0 ≡ read w
      write app(w.1.1, app(w.1.2, w.2))
```

We have $\{B_0\}(\langle\langle x,y\rangle,z\rangle) = x(yz)$ and the time t_{tot} needed to perform this computation is less than $t_1+t_2+t_b$ where

$$\begin{aligned}
&u = yz\\
&w = x(yz)\\
&d_1 = \ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)-\ell_{\mathrm{lin}}(u)\\
&d_2 = \ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(u)-\ell_{\mathrm{lin}}(w)\\
&t_1 = d_1(\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+\gamma+\max(\ell_{\max}(y),\ell_{\max}(z)))^p\\
&t_2 = d_2(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(u)+\gamma+\max(\ell_{\max}(x),\ell_{\max}(u)))^p
\end{aligned}$$

and where t_b—the time needed for shuffling around intermediate results—is linear in $|x|+|y|+|z|+|u|+|w|$ and thus quadratic in $\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z))$. Now

$$\ell_{\max}(u) \le d_1+\max(\ell_{\max}(y),\ell_{\max}(z))$$

So,

$$\begin{aligned}
t_2 \le\ & d_2(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(u)+\gamma+\\
& \quad \max(\ell_{\max}(x),(\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)-\ell_{\mathrm{lin}}(u))+\max(\ell_{\max}(y),\ell_{\max}(z))))^p\\
\le\ & d_2(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+\gamma+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z)))^p
\end{aligned}$$

This means that we can find a constant c such that

$$t_{\mathsf{tot}} \le (d_1+d_2)(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+c+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z)))^p$$

Now the defect of the computation $\{B_0\}(\langle\langle x,y\rangle,z\rangle)$ equals $\ell_{\mathrm{lin}}(B_0)+4+\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)-\ell_{\mathrm{lin}}(w)=\ell_{\mathrm{lin}}(B_0)+4+d_1+d_2$.

Moreover,

$$\begin{aligned}
& \ell_{\max}(w)\\
\le\ & d_2+\max(\ell_{\max}(x),\ell_{\max}(u))\\
\le\ & d_2+\max(\ell_{\max}(x),d_1+\max(\ell_{\max}(y),\ell_{\max}(z)))\\
\le\ & d_2+d_1+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z))
\end{aligned}$$

Therefore, by choosing $\ell_{\mathrm{lin}}(B_0)$ large enough we obtain

$$\mathsf{app}(B_0,\langle\langle x,y\rangle,z\rangle) = \{B_0\}(\langle\langle x,y\rangle,z\rangle) = x(yz)$$

The desired algorithm B is then obtained by applying Lemma 4.5.7 twice.

Notice that the coupling of defect and maximum length is essential for the definability of composition.

We come to the twisting combinator C. Again, we start with the algorithm

```
$C_0 \equiv$ read $w$
      write $\mathsf{app}(\mathsf{app}(w.1.1, w.2), w.1.2)$
```

Clearly,

$$\{C_0\}(\langle\langle x, y\rangle, z\rangle) = xzy$$

The total time t_{tot} needed for this computation is bounded by $t_1 + t_2 + t_b$ where

$$\begin{aligned}
&u = xz \\
&w = uy \\
&d_1 = \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(z) - \ell_{\mathrm{lin}}(u) \\
&d_2 = \ell_{\mathrm{lin}}(u) + \ell_{\mathrm{lin}}(y) - \ell_{\mathrm{lin}}(w) \\
&t_1 = d_1(\ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(z) + \gamma + \max(\ell_{\max}(x), \ell_{\max}(z))^p) \\
&t_2 = d_2(\ell_{\mathrm{lin}}(u) + \ell_{\mathrm{lin}}(y) + \gamma + \max(\ell_{\max}(u), \ell_{\max}(y))^p)
\end{aligned}$$

and the time t_b needed for administration is

$$O((\ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y) + \ell_{\mathrm{lin}}(z) + \max(\ell_{\max}(x), \ell_{\max}(y), \ell_{\max}(z)))^2)$$

Now,

$$\begin{aligned}
t_2 \leq\ & d_2(\ell_{\mathrm{lin}}(u) + \ell_{\mathrm{lin}}(y) + \gamma + \\
& \quad \max(\ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(z) - \ell_{\mathrm{lin}}(u) + \max(\ell_{\max}(x), \ell_{\max}(z)), \ell_{\max}(y)))^p \\
\leq\ & d_2(\ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y) + \ell_{\mathrm{lin}}(z) + \max(\ell_{\max}(x), \ell_{\max}(y), \ell_{\max}(z)))^p
\end{aligned}$$

and we can find a constant c such that

$$t_{\mathsf{tot}} \leq (d_1 + d_2)(\ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y) + \ell_{\mathrm{lin}}(z) + \max(\ell_{\max}(x), \ell_{\max}(y), \ell_{\max}(z)))$$

The defect of the computation $\{C_0\}(\langle\langle x, y\rangle, z\rangle)$ is

$$d = \ell_{\mathrm{lin}}(C_0) + 8 + \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y) + \ell_{\mathrm{lin}}(z) - \ell_{\mathrm{lin}}(w) = \ell_{\mathrm{lin}}(C_0) + 8 + d_1 + d_2$$

Therefore, by choosing $\ell_{\mathrm{lin}}(C_0)$ large enough we obtain the desired combinator as in the proof of Theorem 4.1.8. □

The thus obtained *BCK*-algebra will henceforth be called M_p. The next theorem shows that all the previous machinery about $\mathcal{H}^\Box$ can be applied to M_p as well.

Theorem 4.5.9 *The BCK-algebra M_p is polynomial with respect to the length measure $\ell(x) = \ell_{\mathrm{lin}}(x) + \ell_{\max}(x)$.*

Proof. It is clear that xy is computable in time polynomial in $\ell(x)+\ell(y)$. For the estimate $\ell(xy) \leq \ell(x)+\ell(y)$ we calculate as follows: If $x, y \in M_p$ then $\ell_{\text{lin}}(xy) \leq \ell_{\text{lin}}(x) + \ell_{\text{lin}}(y)$ and $\ell_{\max}(xy) \leq d + \max(\ell_{\max}(x), \ell_{\max}(y)$ where $d = \ell_{\text{lin}}(x) + \ell_{\text{lin}}(y) - \ell_{\text{lin}}(xy)$. Thus, $\ell(xy) = \ell_{\text{lin}}(xy) + \ell_{\max}(xy) \leq \ell_{\text{lin}}(x) + \ell_{\text{lin}}(y) - d + d + \max(\ell_{\max}(x), \ell_{\max}(y)) \leq \ell(x) + \ell(y)$.

The other estimates involving $|-|$ are direct from the definition. □

The following shows that in the category of M_p-sets natural numbers are duplicable.

Proposition 4.5.10 *Let A be an M_p-set such that there exists $k \in \mathbb{N}$ with $\ell_{\text{lin}}(x) \leq k$ for all $x \in \text{dom}(A)$. Then A is duplicable, i.e., the diagonal function $\delta_A : A \longrightarrow A \otimes A$ defined by $\delta_A(x) = (x, x)$ is realisable.*

Proof. Consider the following algorithm

```
dup ≡ read $x$
      write $\lambda f.f\ x\ x$
```

It is clear that *if* this algorithm is short then it is a realiser for the diagonal function $\delta : \mathsf{N} \longrightarrow \mathsf{N} \otimes \mathsf{N}$. Its running time is linear thus at most quadratic in $\ell_{\text{lin}}(x) + \ell_{\max}(x)$. In view of the characterisation in Lemma 4.5.5 it thus remains to show that it meets the growth restrictions

$$\begin{aligned}\ell_{\text{lin}}(\mathsf{dup}(x)) &= \ell_{\text{lin}}(x) + O(1)\\ \ell_{\max}(\mathsf{dup}(x)) &= \ell_{\max}(x) + O(1)\end{aligned}$$

If $x \in \text{dom}(A)$ then $\ell_{\text{lin}}(x) \leq k$ by assumption, so

$$\ell_{\text{lin}}(\mathsf{dup}(x)) = \ell_{\text{lin}}(\lambda f.fxx) = \ell_{\text{lin}}(x) + \ell_{\text{lin}}(x) + O(1) = O(1)$$

Next,

$$\ell_{\max}(\text{dup}(x)) = \ell_{\max}(\lambda f.fxx) = \max(\ell_{\max}(x), \ell_{\max}(x)) + O(1) = \ell_{\max}(x) + O(1)$$

□

It now follows from Prop. 2.7.9 that $\mathsf{N} = !\,\mathsf{N}$ in the category $\text{Ext}(\mathcal{H}^\square)$ constructed over $H = M_p$ thus providing an interpretation of SLR with the axiom SLR and hence a soundness proof for the latter system.

4.6 Computational interpretation

The interpretation in $\text{Ext}(\mathcal{H}^\square)$ although maybe complicated is entirely constructive and can be formalised for instance in extensional Martin-Löf type theory with quotient types, see [16]. Such formalisation gives rise to an algorithm which computes a polynomial time algorithm for every term of type $\Box\mathsf{N}\rightarrow\mathsf{N}$. The efficiency of such compilation algorithm and, more importantly, of the produced polynomial time algorithms hinges on the form of the soundness proof. A detailed analysis falls outside the scope of this thesis and must await further investigation.

We would like at this point, however, to comment on the apparent overhead involved with the runtime monitoring in the application function of the *BCK*-algebras involved. Intuitively, explicit runtime bounds could be omitted since we know anyway that all well-typed programs terminate within the allocated time. However, in doing so we change the behaviour of those programs which contain as a subcomputation an application which returns zero because of runtime exhaustion. In principle this might affect the results; in order to show that this is not so one can replace the category $\mathcal{H}$ by a category in which runtime bounds are built into the definition of morphisms.

Definition 4.6.1 *Let $p > 2$. The category $\mathcal{H}'$ has as objects pairs $X = (|X|, \Vdash_X)$ where $|X|$ is a set and $\Vdash \subseteq \mathbb{N} \times |X|$ is a surjective relation. A morphism from X to Y is a function $f : |X| \longrightarrow |Y|$ such that there exists an algorithm e with the property that whenever $t \Vdash_X x$ then $\{e\}(x)$ terminates with a result y such that $y \Vdash_Y f(x)$ and moreover the runtime of $\{e\}(x)$ is less than $(\ell(e) + \ell(x) - \ell(y))(\ell(e) + \ell(x))^p$.*

An analogous definition can be given relative to M_p.

One can now show that $\mathcal{H}'$ has essentially the same category-theoretic properties as $\mathcal{H}$, but no explicit runtime computations appear in $\mathcal{H}$. The big disadvantage of $\mathcal{H}'$ as opposed to $\mathcal{H}$ is that we have to carry out calculations on the level of algorithms for every single construction in $\mathcal{H}'$, whereas with $\mathcal{H}$ these calculations can be concentrated in the proof that H_p forms a polynomial time *BCK*-algebra. After that all the verifications can be carried out on the higher level of abstraction given by untyped linear lambda calculus.

4.7 More *BCK*-algebras

In this section we describe two more *BCK*-algebras. The first one is constructed from linear lambda terms rather than Turing machines and provides an alternative proof of the main result. The second has the property that exactly the *PTIME*-functions are represented in it as algorithms sending numerals to numerals. We do not have at present have a particular application for this algebra.

Finally, we mention that there exists a *BCK*-algebra similar to H_p but more general as in the ensuing category of realisability sets iteration patterns are available which allow one to define certain recursive functions without moving to the modal function space. In particular the insertion sort algorithm mentioned in Section 3.2.3 admits a natural representation in that category.

The details of this construction are worked out separately in [19].

4.7.1 Linear lambda terms

Rather than using H_p we can also work with untyped linear lambda terms directly. In order to represent numbers and booleans we augment these by constants $\mathsf{tt}, \mathsf{ff}, D, \mathsf{num}(n)$, for $n \in \mathbb{N}$ and constants for linearly bounded polynomial time numerical functions such as $\mathsf{S}_0, \mathsf{S}_1$, test for zero, etc. such lambda terms can be evaluated by performing beta reductions and reducing fully applied function constants according to their specification, i.e., if f is a constant for the function $f : \mathbb{N} \to \mathbb{N}$ then we evaluate $\mathsf{f}\mathsf{num}(n)$ in one step to $\mathsf{num}(f(n))$. By appropriately defining the length of all constants we can achieve that reduction steps do not increase the length which shows that evaluation to normal form can be done in polynomial time.

If we want to have duplicable numerals this syntactic approach becomes more difficult. It appears that by using a shared environment for N-valued variables similar to [2] one can do it, but details have not been worked out.

4.7.2 A *BCK*-algebra of all *PTIME*-functions

Next, there exists a *BCK*-algebra P in which all polynomial time functions and not only the linearly bounded ones are representable. Such algebra cannot be polynomial time computable in the sense of Definition 4.3.1, because if all polynomial time functions are represented then application cannot be polynomial time computable. However, there is an injection $\mathsf{num} : \mathbb{N} \longrightarrow P$ such that for each $e \in H$ the function $n \mapsto e\,\mathrm{num}(x)$ is polynomial time computable.

Recall the coding functions $\langle -, - \rangle$ and num from Lemma 2.1.1 and the length function ℓ.

The carrier of P is the set of integers of the form $\langle n, e \rangle$ where $n \geq 1$. The idea is that e is (a Gödel number of) an algorithm and that n is an "exponent" controlling the runtime of e.

The application function is given as follows.

$$\langle m, e\rangle\langle n, x\rangle = \begin{cases} \langle k, y\rangle, & \text{if}\{e\}(\langle m, x\rangle) = \langle k, y\rangle \\ & \text{and this computation takes} \leq (\ell(e) + \ell(x))^{mn} \text{ steps} \\ & \text{and } k \leq mn \text{ and } y \leq (\ell(e) + \ell(x))^{mn/k} \\ & \\ \langle 1, 0\rangle, & \text{otherwise} \end{cases}$$

Using an algorithm similar to the one for H_p it follows that there is a constant γ such that whenever find that $\langle m, e\rangle\langle n, x\rangle = \langle k, y\rangle$ then this application is computable in time $(\ell(e) + \ell(x) + \gamma)^{mn/k}$.

Proposition 4.7.1 (Composition) *For $\langle m, e\rangle \in P$ and $\langle m', e'\rangle \in P$ there exists $\langle r, f\rangle \in P$ such that*

$$\langle m, e\rangle(\langle m', e'\rangle\langle n, x\rangle) = \langle r, f\rangle\langle n, x\rangle$$

for all $\langle n, x\rangle \in P$. Moreover, $r \leq 2mm'$ and $\ell(f) = \ell(e) + \ell(e') + O(1)$.

Proof. Suppose that $\langle e', m'\rangle\langle n, x\rangle = \langle k, y\rangle$. We have $k \leq m'n$ and $y \leq (\ell(e')+\ell(x))^{mn/k}$. The time needed for this computation is $T_1 = (\ell(e') + \ell(x) + \gamma)^{m'n}$.

Now, the time needed to compute $\langle m, e\rangle\langle k, y\rangle$ is $T_2 = (\ell(e) + \ell(y))^{mk} \leq (\ell(e) + (\ell(e') + \ell(x))^{m'n/k} + \gamma)^{mk} \leq (\ell(e) + \ell(e') + \ell(x) + \gamma)^{mm'n}$. Let $\langle q, z\rangle$ be the result of this computation. We have $q \leq mk \leq mm'n$ and also $\ell(z) \leq (\ell(e) + \ell(y))^{mk/q} \leq (\ell(e) + (\ell(e') + \ell(x))^{m'n/k})^{mk/q} \leq (\ell(e) + \ell(e') + \ell(x))^{mm'n/q}$. The total time to compute $\langle m, e\rangle(\langle m', e'\rangle\langle n, x\rangle)$ is therefore less than $(\ell(e)+\ell(e')+\ell(x)+\gamma)^{2mm'k}$ (the factor 2 is overly generous, but never mind), so we can find $\langle r, f\rangle$ computing the composition of $\langle m, e\rangle$ and $\langle m', e'\rangle$ such that $r \leq 2mm'$ and $\ell(f) = \ell(e) + \ell(e') + O(1)$. □

Notice how the tradeoff between exponent and size in the definition of application allows us to cancel the intermediate exponent q and thus to define composition.

Using an analogue of Lemma 4.1.7 (parametrisation) we can now show that P forms a *BCK*-algebra.

Numerical values are embedded into P by $\mathrm{num}'(x) = \langle 1, \mathrm{num}(x)\rangle$. Let us write $\mathrm{NUM} \subseteq P$ for the image of this embedding. The application of $\langle m, e\rangle$ to $\langle 1, \mathrm{num}(x)\rangle$ takes time $(\ell(e) + |x| + 1)^m$ thus is polynomial in $|x|$. On the other hand, every *PTIME*-function on integers is representable in P by appropriately padding a *PTIME* algorithm for it.

At present, we do not have an application for this algebra P, but it is interesting to note that it is indeed possible to organise *all PTIME*-functions into a *BCK*-algebra.

Chapter 5

Conclusions and further work

The ultimate goal behind this work is the development of a type system for functional programs which identifies polynomial time algorithms by using type-theoretic generalisations of primitive recursion.

In this thesis we have shown how to extend Bellantoni-Cook's notion of safe recursion to higher-order functions and inductively defined data structures. On the technical side we have developed a semantic method for proving correctness of such systems.

Like many model constructions in the field of type theory this semantic method can be seen as a distillation of the invariants needed to prove soundness by induction on the structure of typing derivations. The advantage is that it is now rather easy to extend the type system by other constructs as long as they can be interpreted in the model or—to reiterate this distillation metaphor—as long as the distilled invariants suffice for the desired extension.

Examples of such "cheap" extensions would be record types, coproduct types, variant types. Arbitrary inductive types can also be added if one merely aims for interpreting constructors and case distinction. If recursors are also desired then as we have seen in the case of the Leivant trees a more careful case-to-case analysis is necessary.

A number of other extensions have been considered and did not make it into the thesis for various reasons mostly for lack of time and space. One such further topic is the definition of a higher-order intuitionistic logic over SLR. As is well known, every presheaf category is a model of such logic and so is in particular $\widehat{\mathcal{H}^{\Box}}$. As shown in [15] it is necessary to restrict to the category of sheaves for a certain Grothendieck topology in order to validate decidability of atomic formulas. Alternatively, if one restricts attention to first-order logic, one can use a functional interpretation of proofs as SLR-terms and thereby avoid the use of sheaves.

In such a logic based on SLR[1] one would have a linear implication and modalities !, □ both on propositions and on individuals. Alternatively, one would have three kinds of implications and quantifiers. The induction schema would take the form

$$\varphi(0) \multimap\, !(\forall x{:}\,\Box\mathsf{N}.\varphi(\left\lfloor\frac{x}{2}\right\rfloor) \multimap \varphi(x)) \multimap \forall x{:}\,\Box\mathsf{N}.\varphi(x)$$

[1] We remark that in a recent talk S. Bellantoni has presented such a logic based on the system in [2].

Here φ must not contain the modalities ! and $\Box$ neither in formulas nor in quantifiers.

The interpretation of this logic would then show that Skolem functions of Π^0_2-statements are polynomial time computable functions. Using the Friedman-Dragalin translation one could even extend this to a classical variant of the logic.

So far this would merely be an exercise in spelling out definitions and writing down proof rules. The challenge would be to explore the strength of such logic, to see whether it is practically more convenient to use than e.g. Bounded Arithmetic, and to solve meta-theoretic questions which go beyond the provability of Π^0_2 statements.

The linearity restriction in the inductive step has some intuitive appeal since it is met by most of the so to speak "obvious" proofs by induction which use the induction hypothesis at most once. If more than one use is made of the induction hypothesis then this is often emphasised as something unusual and "ingenious". The modality restriction is less natural and could be perhaps be avoided in a logic based on Caseiro's LIN system.

Another strand of further development would be the extension with further modalities corresponding to the levels of the Grzegorczyk hierarchy [38]. As mentioned in Section 3.8 the SLR type system can easily be extended to this. Some work on extending the semantics will be necessary though. Similarly, one might consider an adaptation of the results to complexity classes below polynomial time. As far as recursion with basic result type is concerned this could be easily achieved using recursion-theoretic characterisations of such classes as can be found in [7]. For higher result types the construction of appropriate *BCK*-algebras would be required and it is not clear at present to what extent such is possible.

We emphasize that the restriction to extensional presheaves and accordingly the simplified construction of tensor product of presheaves which has been made in order to simplify the presentation precludes the use of non-well-pointed semantics. If such would be needed in the future then Day's original tensor product [10] and an appropriate definition of the !-modality must be employed. Indeed, the semantics given in [19] is not well-pointed, so in order to apply the described methods to that case one would have to follow that avenue.

On the practical side we would like to see an integration of SLR with a full scale programming language so that also non-polynomial time algorithms could be written down albeit with a different type. Partial evaluation (based on our semantic soundness proof) could then be used to compile a program into *PTIME*-code once all its arguments on which the runtime depends superpolynomially are known.

Bibliography

[1] Andrea Asperti. Light affine logic. In *Proc. Symp. Logic in Comp. Sci. (LICS)*. IEEE, 1998.

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

[5] 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.

[6] Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. In *Symposium on Logic in Computer Science (LICS'96)*, pages 264–275. IEEE, 1996.

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

[8] 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.

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

[10] Brian J. Day. An embedding theorem for closed categories. In *Category Seminar Sydney 1972–73, LNM 420*. Springer Verlag, 1974.

[11] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive Recursion for Higher-Order Abstract Syntax. Technical Report CMU-CS-96-172, Carnegie Mellon University, 1996.

[12] Thomas Ehrhard and Loïc Colson. On strong stability and higher-order sequentiality. In *Proc. 9th Symp. Logic in Comp. Sci. (LICS), Paris, France*, pages 103–109. IEEE, 1994.

[13] Jean-Yves Girard. Light Linear Logic. Manuscript. Available via http://hypatia.dcs.qmw.ac.uk/, 1995.

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

[15] 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.

[16] Martin Hofmann. *Extensional constructs in intensional type theory.* BCS distinguished dissertation series. Springer, 1997. 214 pp.

[17] Martin Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, 1997.

[18] Martin Hofmann. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *Proceedings of CSL '97, Aarhus. Springer LNCS 1414*, pages 275–294, 1998.

[19] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. Draft. Available from www.dcs.ed.ac.uk/home/mxh/papers/icfp.ps.gz, October 1998.

[20] Martin Hofmann. More results on modal/linear lambda calculus. Submitted. Available under www.dcs.ed.ac.uk/home/mxh, 1998.

[21] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS)*. IEEE, Computer Society Press, 1999. to appear.

[22] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In Giovanni Sambin and Jan Smith, editors, *Proceedings of the meeting* Twenty-five years of constructive type theory*, Venice, 1995.* Oxford University Press, 1998. Extended abstract in Proc. LICS '94.

[23] G.M. Kelly, editor. *Category Seminar Sydney 1972/73*, volume 420 of *Lecture Notes in Mathematics*. Springer, Berlin, Heidelberg, New York, 1973.

[24] Joachim Lambek and Philip Scott. *Introduction to Higher-Order Categorical Logic.* Cambridge University Press, 1986.

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

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

[27] 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.

[28] Daniel Leivant and Jean-Yves Marion. Ramified Recurrence and Computational Complexity IV: Predicative functionals and Poly-space. Manuscript, 1997.

[29] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, 1971.

[30] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[31] Jim Otto. *Complexity Doctrines.* PhD thesis, McGill University, 1995.

[32] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19, 1993.

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

[34] 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.

[35] Benjamin Pierce and David Turner. Local type inference. In *Conference record of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[36] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

[37] Gordon Plotkin. Type theory and recursion. Invited talk at the 8th Annual IEEE Symposium on Logic in Computer Science, Montreal, Canada, 1993.

[38] H. E. Rose. *Subrecursion.* Clarendon Press, Oxford, 1984.

[39] Vladimir Sazonov and A. Voronkov. A construction of typed lambda models related to feasible computability. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Proceedings of the Third Kurt Gödel Colloquium, KGC'93. (Lecture Notes in Computer Science; vol. 713)*, pages 301–313. Springer, 1993.

[40] R. Statman. The typed lambda calculus is not elementary recursive. *Theoretical Computer Science*, 9, 1979.

[41] 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.

[42] 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.