Università degli Studi Roma Tre

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Magistrale in Matematica

The Typing Problem for Light Logics with Second Order

il Relatore
prof. Marco Pedicini

il Candidato
Paolo Tranquilli

Anno Accademico 2004-2005

Keywords: Type Inference, Polymorphic Lambda-Calculus, Light Linear Logic, Implicit Complexity

AMS classification: 03B15, 03B70 (03F52, 68Q15, 68N18)

a mamma e papà
se c'è qualcosa di buono in me
è merito loro

Acknowledgments

There are a bunch of people I would like to thank. I will salomonically list them in rigorous alphabetical order, and I'm sure I'm forgetting someone. If so I prostrate myself begging for an unmerited pardon.

Thanks to Antonella of the secretariate, both of them. They know why[1].

Thanks to Archive. Their music has accompanied me in the frantic days of this work, and soothed me. They are not the only ones to have done so, but... well, I wanted to say *You all look the same to me* is a masterpiece, and *Again* is the best song ever made. In the end I just wanted to know how does it feel to thank a musical group in a math thesis.

Thanks to my friends. All of them, apart from Natascia. Thanks. I can only wonder what would have become of me in these last months without them, and the thought makes me shudder. Thanks. Now that I think of it... it's hard to miss someone with a first order quantifier like this...

Thanks to Lorenzo Tortora De Falco. His patience in spite of all has been precious. Moreover, if someone wants to blame somebody for my choice, he is one of the two. Without him I would hardly have appreciated a beautiful subject such as Logic.

Thanks to Marco Pedicini, my supervisor. Apart from being the second one in alphabetical order to be blamed, and the first one in the chronological one, he has opened my eyes on λ-calculus, and, you know, I indeed like λ-calculus very much.

Thanks to mum and dad. With their kid putting three rooms to total disorder (without speaking of the kitchen) while working, and also being somewhat rude at times, and keeping up the support, well they deserve all the thanks in the world.

Thanks to Natascia. I owe it to her because of a whole handful of help she has dispensed to me strictly relating to this last endevour of mine, apart from the more general suppport I thanked my other friends for. I gladly repay my debt, and thank her again sincerely. Thanks also for the crutches.

Thanks to Patrick Baillot, and all the guys I've met in Paris. Their welcome has been so nice, and discussions with them so stimulating, that I look forward to meeting and speaking with them again.

[1] a gentile richiesta traduco. Grazie a entrambe le Antonelle della segreteria. Loro sanno perché.

The abuse of structural rules
may have damaging complexity effects

Contents

Introduction

Since Church introduced λ-calculus back in 1936 [Chu36], this formal language has become a paradigm in studying computability, and a model for a new style of programming, the functional one.

We can point out two ways of dealing with the notion of deterministic computability. One is the theoretical abstraction of mechanical functioning: the Turing machine, introduced in the same year, 1936. Is as if we take a machine with really limited capacities: we employ a finite alphabet, a tape (potentially infinite) in which only the input is written, a finite set of *states* of the machine, and a head capable of reading and writing on a single cell at a time, moving left or right and of changing the state, all based exclusively on the state and the symbol being read. A program is then a table that assigns to every pair consisting of a symbol and a state a triple with the new state, the symbol to be written and the movement to be done. Up to now a rigorous concept of time or space needed for a computation is based on the number of steps one such simple machine needs to carry out the algorithm.

The other way around is an axiomatic approach to the subject. We precise what is computable with a mathematical definition of an algebra of functions: the least class containing some base functions and closed with respect to certain schemes. So rather than preoccupying ourselves with the actual way a function can be computed, we give a rigorous mathematical foundation to the functions.

We may regard λ-calculus as lying in the middle. It is highly formal, but it in fact it also shows how we have to effectively compute the function. Basically the two main ideas behind it are regarding all, programs and data, in a single class of objects, and then cut down to the two most basic constructs of functions the building of such objects: *application* of an object to another and *abstraction*, that is the definition of a function by stating that a *variable* component of an object has to be regarded as a parameter for the function. By the Church-Turing thesis, in fact all these notions are equivalent in defining what up to today is the accepted definition of what is computable.

This last approach gained more and more importance as various developments were made in it: type assignment, a way to give a tight discipline on otherwise "wild" applications, brings around the proofs-as-programs paradigm: the Curry-Howard isomorphism, for which a proof in a logic system can be translated into a program and viceversa, with cut-elimination being equivalent to the execution of the program.

In the meantime on both the sides of the approach to computable functions a notion of complexity classes has been developed. The two main classes we will interest ourselves in are the *Kalmar recursive* and the *polytime* functions. This notion regards both the sides of the approach to recursive functions: they can be viewed as classes of functions that require a time to be computed on Turing machines bounded by respectively a tower of exponentials of fixed height and a polynomial in the size of the input. Or else they also have an axiomatic definition: Kalmar recursive functions where in fact first introduced as a function algebra, while it is quite a recent result (1992, [BC92], due to Bellantoni and Cook) that also polytime functions have a definition that does not explicitly refer to polynomials, using a new notion of *safe* recursion.

We may say that recently the λ-calculus was brought up-to-date by a new approach starting from the Curry-Howard isomorphism. Using the tools provided by linear logic new systems were developed that capture the above complexity class from a proofs-as-programs point of view: LLL, light linear logic in 1998 due to Girard, and ELL, elementary linear logic in 2003 due to a germinal idea presented again by Girard and then developed in full by Danos and Joinet. With some accommodations this approach has been adapted in the form of a type assignment system to the λ-calculus.

This brings an exciting new possibility: of the three approaches here described to computability, λ-calculus is by far the one more suitable to be developed as the core of a programming language. ML and its object-oriented spawn OCAML are examples of it. Developing a type discipline that certificates good complexity bounds may lead to an effective control on the resources needed by a program, implicitly, without having to control the effective time of computation.

However this brings some complication: the type assignment disciplines are up to now far from being easy and intuitive, so we would like to leave to a machine the problem of finding the right type, while we occupy ourselves with just writing a program, eventually modifying it if we get a negative result for the typing. And in fact we also have a need to use polymorphism, without which we do not have enough expressive power. Unfortunately the two needs somewhat clash against each other: renouncing to polymorphism brings type inference to our reach, while exploiting the full power of polymorphism poses much problems to deciding automatically the type assignment.

Outline of the thesis

In chapter 1 we will give a survey on pure λ-calculus. We will go over the basic definitions and properties, and then show the range of its expressive power.

Chapter 2 is dedicated to introduce type assignment systems, starting from simply typed λ-calculus. Again definitions and properties are exposed, as well as the limits of its expressive power. A full account on its type inference is presented. The second part of the chapter consists in a brief outline of a first extension of simply typed calculus: **PCF**. Type inference is shown also for this system.

Chapter 3 is dedicated to introduce the concept of polymorphism. This is done by exposing system **F**. Its expressive power is assessed, and then its main undecidability results are shown: typability and type checking for this system are undecidable.

In chapter 4 then we start to speak of linear logic and its application to implicit complexity. After a brief outline of affine logic we get to know the type assignment systems for elementary affine logic, light linear logic and the recent dual light affine logic. Their expressive power in the sense of completeness for their respective complexity classes is depicted. Then the algorithms for typing of the propositional fragment are outlined, in order to arrive finally to speak of the problems arising with second order.

Notations and conventions

We denote the set of booleans $\mathbb{B} = \{\, \texttt{true}, \texttt{false} \,\}$.

When we speak of a function we mean a partial function, unless otherwise specified or clear from the context. Given a function $\varphi : X \to Y$ we denote its domain and range by

$$\mathrm{DOM}(\varphi) := \{\, x \in X \mid \varphi(x) \text{ is defined} \,\},$$
$$\mathrm{RAN}(\varphi) := \{ y \in Y \mid \exists x \in \mathrm{DOM}(\varphi) \text{ with } \varphi(x) = y \,\}.$$

When describing a function, every time there is no way to determine its value it is to be regarded as undefined in that case (so for example $g(f_1(n), f_2(n))$ is undefined at n either if $f_1(n)$ is undefined, or $f_2(n)$ is undefined, or at last if both are defined but $g(f_1(n), f_2(n))$ is not). We denote by $f|_X$ the restriction of f to some set X, i.e. the function defined on $X \cap \mathrm{DOM}(f)$ such that $f|_X(x) = f(x)$. Also we write $f(X)$ for $\mathrm{RAN}(f|_X)$. If a function has domain and range in the same set we define its support as

$$\mathrm{SUPP}(f) = \{\, x \in \mathrm{DOM}(f) \mid f(x) \neq x \,\}.$$

Whenever we have a sequence of objects written $X_1, X_2, \ldots, X_n$ we may denote it by $\vec{X}^n$, or simply $\vec{X}$ if there is no need to specify the length. We specify $\vec{X}_0^n$ or $\vec{X}_0$ if we want the

index to start from 0. We denote by $\vec{X} \setminus X_i$ the result of deleting X_i from the sequence, obtaining $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_N$. Given a function φ, by $\overrightarrow{\varphi(X)}$ we mean the sequence $\varphi(X_1), \ldots, \varphi(X_n)$. Whenever there is no fear of misunderstanding we will denote by $\vec{X}$ also the set $\{ X_1, \ldots, X_n \}$, and we will switch between the notations at will. Being this a subject where sequences of various objects are frequently encountered, we will heavily use this notation. The set of sequences (also called *words*) of arbitrary (yet finite) length with components in the set X is denoted by X^*, with the empty word denoted by ε:

$$X^* ::= \varepsilon \mid XX^*.$$

The length of a word w is denoted by $|w|$. The same notation is used for the cardinality of sets, in accordance with the notation depicted above.

Chapter 1

Pure λ-calculus

An extensive introduction to λ-calculus in general and the pure one in particular is given in [Kri90]. However we give a survey of the topic in this chapter.

Pure λ-calculus (also called untyped λ-calculus) consists of a set of *terms*, various equalities between these terms and some notions of *reduction* on terms.

1.1 Definition and α-equivalence

Let $\mathcal{V}$ be a countable set of *variables*, over which we range with letters such as x, y, z.

Definition 1.1.1 (Λ). The set Λ of λ-*terms* is built from the following grammar using the symbols λ, (,), . and x with $x \in \mathcal{V}$:

$$\Lambda ::= \mathcal{V} \mid (\lambda \mathcal{V}.\Lambda) \mid (\Lambda\, \Lambda).$$

A term is named a *variable*, an *abstraction* or an *application* depending on which of the three rules of construction is used last. In an application we say the first term is *applied* to the second.

We usually use letters such as M, N to range over terms. As a convention, parentheses are omitted wherever possible: a sequence of applications associates to the left, so that we denote $((\dots((M_1\, M_2)M_3)\dots)M_n) = (M_1 M_2 \dots M_n)$, and the scope of an abstraction is as large as possible, so for example $\lambda x.M\, N$ means $(\lambda x.(M\, N))$ and not $((\lambda x.M)N)$. Also dots between abstractions are omitted, so that $\lambda x \lambda y.M$ means $\lambda x.\lambda y.M$. We may abbreviate $\lambda x_1 \dots \lambda x_n.M$ with $\overrightarrow{\lambda x^n}.M$ and $(M\, N_1 \dots N_n)$ with $(M\, \vec{N}^n)$.

Given a term M we can recursively build its *construction tree* $\texttt{tree}(M)$:

$$\texttt{tree}(x) := x,$$

$$\texttt{tree}(\lambda x.M) := \begin{array}{c} \lambda x \\ | \\ \texttt{tree}(M) \end{array},$$

$$\texttt{tree}(M\,N) := \begin{array}{ccc} & @ & \\ \swarrow & & \searrow \\ \texttt{tree}(M) & & \texttt{tree}(N) \end{array}.$$

An *occurrence* of a variable is a "place" the variable occupies in the term or in its construction tree. Formally it can be defined as follows: let L, D and R be partial functions on Λ so that L and R are defined only on applications while D only on abstractions in the following way:

$$L(M\,N) = M, \quad R(M\,N) = N, \quad D(\lambda x.M) = M.$$

Given a sequence $\vec{f} \in \{\,L, D, R\,\}^*$ we denote $(f_1 \circ f_2 \circ \cdots \circ f_n)(M)$ by simply writing $\vec{f}(M)$. We call $\vec{f}$ a *path* in M if $\vec{f}(M)$ is defined. So, speaking more generally of subterms and occurrences of subterms, we have the following definition:

Definition 1.1.2 (subterm, subterm occurrence). A subterm of a given term M is the result of applying a path to M. An occurrence of a subterm is a path leading to it.

Often (if not always) we will speak of a variable or a subterm meaning instead one of its specific occurrences, or we will denote an occurrence of a variable x by x itself.

Some measures on terms are useful:

Definition 1.1.3 (term size, term depth). The size of a term M, denoted $|M|$, is the number of nodes (together with leaves) of its construction tree. So

$$|x| := 1,$$
$$|(M\,N)| := 1 + |M| + |N|\,,$$
$$|(\lambda x.M)| := 1 + |M|\,.$$

Its depth $\mathrm{d}(M)$ is the maximum length of the branches of its construction tree. So

$$\mathrm{d}(x) := 0,$$
$$\mathrm{d}(M\,N) := 1 + \max(\mathrm{d}(M), \mathrm{d}(N)),$$
$$\mathrm{d}(\lambda x.M) := 1 + \mathrm{d}(M).$$

Given a term M we define various (finite) subsets of $\mathcal{V}$. The *set of variables of* M, denoted by $\mathrm{V}(M)$, is the set of leaves of the construction tree of M. Its *free variables*, $\mathrm{FV}(M)$, are variables in $\mathrm{V}(M)$ not "captured" by an abstraction above in the tree. The *bounded variables* $\mathrm{BV}(M)$ are those bounded by some abstraction in the term. The following recursive definitions are given:

$$\begin{aligned}\mathrm{V}(x) &:= \{x\},\\ \mathrm{V}(\lambda x.M) &:= \mathrm{V}(M),\\ \mathrm{V}(M\,N) &:= \mathrm{V}(M) \cup \mathrm{V}(N);\end{aligned}$$

$$\begin{aligned}\mathrm{FV}(x) &:= \{x\}, & \mathrm{BV}(x) &:= \emptyset,\\ \mathrm{FV}(\lambda x.M) &:= \mathrm{FV}(M) \setminus \{x\}, & \mathrm{BV}(\lambda x.M) &:= \mathrm{BV}(M) \cup \{x\},\\ \mathrm{FV}(M\,N) &:= \mathrm{FV}(M) \cup \mathrm{FV}(N); & \mathrm{BV}(M\,N) &:= \mathrm{BV}(M) \cup \mathrm{BV}(N).\end{aligned}$$

A term with no free variables is called *closed*. We distinguish between free and bounded occurrences of variables by checking if the given occurrence is within the scope of an abstraction of its variable. We trivially extend the above definitions to more than one term:

$$*\mathrm{V}(M_1, M_2, \ldots, M_n) := \bigcup_{i=1}^{n} *\mathrm{V}(M_i)$$

where $*\mathrm{V}$ is either V, FV or BV.

Having also the notion of subterm we may define the *scope* of a bounded variable.

Definition 1.1.4 (scope). Given M and $x \in \mathrm{BV}(M)$, for every subterm of M with the form $\lambda x.N$ we say that N and all its subterms are in the *scope* of x.

At the basis of any process involving the λ-terms is the notion of *substituting a variable with a term*. It is designed to avoid two problems occurring when substituting "blindly": nothing is done when trying to substitute a term for a bounded variable if we are in the scope of its abstraction, and there is no free variable capture, i.e. if I want to substitute x with N in M, there is a renaming of bounded variables in M so that a free variable in N is not captured by an abstraction whose scope contains x. In order to obtain such a notion, we first introduce one that circumvents only the first (and easiest) problem: the *simple substitution*. We more generally give a notion of simultaneous substitution.

Definition 1.1.5 (simple substitution). The operation of simple substitution consists of replacing free occurrences of variables $x_1, x_2, \ldots x_n$ in M with given terms $N_1, N_2, \ldots N_n$ and is denoted

by $M\langle N_1/x_1, \dots N_n/x_n\rangle$, abbreviated by $M\langle\overrightarrow{N/x}\rangle$:

$$y\langle\overrightarrow{N/x}\rangle := \begin{cases} N_i & \text{if } y = x_i \text{ for some } i, \\ y & \text{otherwise,} \end{cases}$$

$$(M_1\, M_2)\langle\overrightarrow{N/x}\rangle := (M_1\langle\overrightarrow{N/x}\rangle\, M_2\langle\overrightarrow{N/x}\rangle),$$

$$(\lambda y.M)\langle\overrightarrow{N/x}\rangle := \begin{cases} \lambda y.M\langle(\overrightarrow{N/x}) \setminus (N_i/x_i)\rangle & \text{if } y = x_i \text{ for some } i, \\ \lambda y.M\langle\overrightarrow{N/x}\rangle & \text{otherwise.} \end{cases}$$

We are now ready to move on to a more suitable notion of substitution.

Definition 1.1.6 (substitution). We denote by $M[N_1/x_1, \dots, N_n/x_n]$ the result of substituting $x_1, \dots, x_n$ with $N_1, \dots, N_n$ in M. We abbreviate it by $M[\overrightarrow{N/x}]$. Formally:

$$y[\overrightarrow{N/x}] := y\langle\overrightarrow{N/x}\rangle,$$
$$(M_1\, M_2)[\overrightarrow{N/x}] := (M_1[\overrightarrow{N/x}]\, M_2[\overrightarrow{N/x}]),$$
$$(\lambda y.M)[\overrightarrow{N/x}] := \lambda z.M\langle z/y\rangle[\overrightarrow{N/x}],$$

where in the last case z is a variable not in $\mathrm{FV}(\vec{N})$. Note that in this case, the most delicate one, the definition is possible because substituting a variable for a variable does not change the term structure (thus keeping the recursive definition possible). Any ambiguity in the same step can be overcome with an enumeration of $\mathcal{V}$ (for example by leaving $z = y$ when y is not free in $\vec{N}$, and by taking z as the first variable not free in $\vec{N}$ otherwise). However any such ambiguity will be vanquished by the definition of α-equivalence.

If the N_is are all equal to N we will use the abbreviation $M[N/\vec{x}]$.

In order to reason about substitution on a purely syntactic level we introduce the concept of *context*.

Definition 1.1.7 (context). A context is a λ-term with "holes" into which any term may be plugged. Formally the set of contexts $\mathcal{C}[\ ]$ ranged over by the meta-variable $C[\ ]$ is defined by the grammar

$$\mathcal{C}[\ ] ::= \Box \mid \mathcal{V} \mid (\lambda\mathcal{V}.\mathcal{C}[\ ]) \mid (\mathcal{C}[\ ]\,\mathcal{C}[\ ]).$$

Given a context $C[\ ]$ and a term, we denote by $C[M]$ the result of substituting every occurrence of the hole $\Box$ with M, regardless of variable capture. We call a context *simple* if $\Box$ occurs exactly once in it. We trivially extend to contexts basic definitions on terms such as bounded variables or size. We additionally define a set $\mathrm{BHV}(C[\ ])$ as the set of bounded variables in $C[\ ]$ whose scope contains a hole.

Definition 1.1.8 (relation that passes to context). We say a relation $\sim$ *passes to context* if given any context $C[\ ]$, we have

$$M \sim N \implies C[M] \sim C[N].$$

This condition is equivalent to saying:

$$M \sim N \implies \lambda x.M \sim \lambda x.N,\ (M\,P) \sim (N\,P),\ (P\,M) \sim (P\,N).$$

We go on defining a syntactic equivalence between terms. We want to achieve a system where the names of bounded variables do not count (so that an occurrence of a bounded variables may be seen as a pointer to its abstraction, where the label really does not count).

Definition 1.1.9 (α-equivalence). α-equivalence, denoted by $\equiv_\alpha$, is the least equivalence relation that passes to context for which:

$$\lambda x.M \equiv_\alpha \lambda y.M\langle y/x\rangle, \quad \text{given that } y \notin \mathrm{V}(M).$$

More precisely:

$$\begin{aligned} x \equiv_\alpha N &\iff N = x, \\ (M_1\,M_2) \equiv_\alpha N &\iff N = (N_1\,N_2) \text{ with } M_1 \equiv_\alpha N_1, M_2 \equiv_\alpha N_2, \\ \lambda x M \equiv_\alpha N &\iff N = \lambda y.N' \text{ with } \forall z \notin \mathrm{V}(M, N') : M\langle z/x\rangle \equiv_\alpha N'\langle z/y\rangle. \end{aligned}$$

We will now always regard two α-equivalent terms to be identical, so that we are now working with $\Lambda/\equiv_\alpha$ rather than Λ, though we will keep using the same name. We may now see as the result of the substitution $M[\overrightarrow{N/x}]$ is really an α-equivalence class.

We now have the freedom to rename bounded variables at will. To simplify things, we may use this freedom so that whenever we deal with a set of terms we rename bounded variables so that they are different from the free variables and also between each other. Under this convention we may use simple substitution without checking variable capture.

1.2 Reduction

We are now ready to introduce a concept central to λ-calculus, the core of its computational significance: *β-reduction*.

Definition 1.2.1 (β-reduction, β-equivalence, β-normal). The binary relation on terms $\xrightarrow{\beta}$ (β-reduction step) is the least relation that passes to context such that

$$((\lambda x.M)N) \xrightarrow{\beta} M[N/x].$$

Given a term M, we call a subterm of the form $((\lambda x.N_1)N_2)$ a *β-redex*. We say we contract it if we apply a β-reduction step that changes it into $N_1[N_2/x]$, and we call such subterm *β-contractum*.

We denote the transitive and reflexive closure of $\xrightarrow{\beta}$ by $\overset{\beta}{\twoheadrightarrow}$ (β-reduction). Clearly:

$$M \overset{\beta}{\twoheadrightarrow} N \iff \exists \vec{M}_0^n \mid M_0 = M, M_n = N, \forall i : M_i \xrightarrow{\beta} M_{i+1}$$

where possibly $n = 0$. The least equivalence relation containing β-reduction is called β-equivalence and is denoted by $\equiv_\beta$. We have:

$$M \equiv_\beta N \iff \exists \vec{M}_0^n \mid M_0 = M, M_n = N, \forall i : M_i \xrightarrow{\beta} M_{i+1} \text{ or } M_{i+1} \xrightarrow{\beta} M_i.$$

A term with no redexes, that is a term M for which there is no term N such that $M \xrightarrow{\beta} N$, is said to be β-normal, or just normal, or also in normal form.

Remark 1.2.2. Every term can be uniquely written in the form

$$\overrightarrow{\lambda y}^n (M \vec{N}^m),$$

where M is either an abstraction (and $m \geq 1$) or a variable. In the former case we call $(M\,N_1)$ the *head redex*.

Looking at this form we see that a term is in normal form if and only if M is a variable and all the N_is are normal. If we drop the condition on the N_is then the term is said to be in *head normal form*.

We may regard a β-reduction step as a step of a computation (possibly started by applying the program M to the input N and thus trying to reduce $M\,N$), and a β-normal form as its output. So questions arise:

- a β-reduction depends on the redexes I choose to contract. Does it change the result? Closely related to it: is the result unique?
- does a result always exist?
- if a result exists, how can I get to it? Do I always reach it no matter what reduction steps I take?
- can I check if a result exists? In other words, is there an algorithm that terminates always and tells whether there is a result or not?

Definition 1.2.3 (weak normalization, strong normalization). We say a term M normalizes weakly, or simply say it normalizes, if there is a normal term M' such that $M \overset{\beta}{\twoheadrightarrow} M'$. We say M' is a *normal form* of M. A term M instead normalizes strongly when every reduction is finite, or said

differently there is no infinite sequence $M_0, M_1, M_2, \ldots$ such that $M_0 = M$ and $M_i \xrightarrow{\beta} M_{i+1}$ (such a chain, whether infinite or not, is called *reduction chain*). In the first case we write $M \in WN$, while in the second one $M \in SN$.

Definition 1.2.4 (normalization tally). The *normalization tally* of a term M, written $||M||$ is defined to be the maximum length of a reduction chain starting from M, eventually ∞.

Remark 1.2.5. A term is strongly normalizing if and only if its tally is finite. This is because from the axiom of choice comes that every finitely branching tree with finite branches is finite (König's lemma), and so there is a maximum length of its branches.

We may already see that there are non trivial (i.e. non normal) strongly normalizing terms, terms that are normalizing but not strongly and terms that are not even normalizing. This is also an occasion to see reduction and bounded variable renaming at work.

Example 1.2.6. Let I be the term $\lambda x.x$ (the *identity*), and D the term $\lambda x.x\,x$. There is only one way to reduce $(D\,I)$:

$$(\lambda x.x\,x)(\lambda y.y) \xrightarrow{\beta} (\lambda y.y)(\lambda z.z) \xrightarrow{\beta} \lambda z.z,$$

so it is strongly normalizing.

On the other hand $(D\,D)$ has only one redex, and

$$(\lambda x.x\,x)(\lambda y.y\,y) \xrightarrow{\beta} (\lambda y.y\,y)(\lambda z.z\,z) = (D\,D)$$

so every reduction step leaves it unchanged and thus it is not normalizing.

Finally $(\lambda x.y)(D\,D)$ has two possibilities: one leads directly to the free (and β-normal) variable y, while the other as seen before leaves the term unchanged. So the term is normalizing but not strongly.

Remark 1.2.7. A normal term is minimal for β-reduction: if a term M is normal and $M \xrightarrow{\beta} M'$ then $M' = M$. However the converse does not hold, as was shown by the example of $(D\,D)$.

To answear the first of the questions comes the notion of *confluence*.

Definition 1.2.8 (Church-Rosser property). A binary relation $\sim$ over any given set is said to be *confluent*, or to have the *Church-Rosser property*, if

$$x \sim y \text{ and } x \sim y' \implies \exists z \mid y' \sim z \text{ and } y'' \sim z.$$

Theorem 1.2.9 (Church-Rosser). *β-reduction on terms has the Church-Rosser property.*

Proof. See [Kri90, §I.3]. □

Immediately follows

Corollary 1.2.10. *If a term has a normal form it is unique. We denote the (eventual) normal form of a term M by M^*.*

Proof. Suppose $M \overset{\beta}{\twoheadrightarrow} M'$ and $M \overset{\beta}{\twoheadrightarrow} M''$, and both M' and M'' are normal. From the Church-Rosser property there is M''' such that $M' \overset{\beta}{\twoheadrightarrow} M'''$ and $M'' \overset{\beta}{\twoheadrightarrow} M'''$. But then, they are normal, so $M' = M''' = M''$. □

Remark 1.2.11. With confluence we may see that

$$M \equiv_\beta N \iff \exists M' \mid M \overset{\beta}{\twoheadrightarrow} M', N \overset{\beta}{\twoheadrightarrow} M',$$

and also that if $M \equiv_\beta N$ and N is normal then $N = M^*$ (and in particular $M \overset{\beta}{\twoheadrightarrow} N$).

As for the question about how reaching a normal form if there is one, the answear is the so called *lazy reduction*, also called *left reduction*.

Definition 1.2.12 (deterministic strategy). A *deterministic strategy* is a computable function s on terms such that $s(M)$ gives either a redex of M or a "STOP" signal (note that if M is normal s must give the STOP signal). Applying a strategy to a term $M = M_0$ means yielding M_{i+1} obtained with a reduction step done contracting $s(M_i)$ in M_i, while $s(M_i)$ is not the STOP signal. In the latter case we say M_i is the (unique because of determinism) normal form of M with respect to s, and write it as M^*_s. We write $M \to_s N$ for a single step of the reduction given by the strategy, and $M \twoheadrightarrow_s N$ for multiple (possibly none) steps.

We also define a tally $||M||_s$ as the length of the (unique) chain leading to M^*_s if it exists, ∞ otherwise.

Definition 1.2.13 (head reduction, lazy reduction). Define the following strategies:

- the *head reduction strategy* h gives the head redex of a term if there is one, STOP otherwise. Note that a term is in normal form for this strategy if and only if it is in head normal form.
- the *lazy* or *left reduction strategy* ℓ gives the redex most to the left of the term, if the term has any redex, STOP otherwise. Here normality is equivalent to β-normality.

Note that the lazy reduction begins with head reduction, and if this terminates to a head normal form

$$\overrightarrow{\lambda y}^n.(x\,\vec{N}^m)$$

it goes on by processing N_1 in the same manner. If it finds the β-normal form of N_1 it begins working on N_2 and so on.

Theorem 1.2.14. *Given a term M, if the normal form M^* exists then*

$$M \twoheadrightarrow_\ell M^*.$$

A somewhat weaker notion of normalizability is that of solvability.

Definition 1.2.15 (solvable term). A term M is said to be solvable if one of the following equivalent properties hold:

1. $\forall N : \exists \vec{P}^h, \vec{x}^k, \vec{Q}^k \mid (M[\vec{P}/\vec{x}]\,\vec{Q}) \equiv_\beta N$;
2. $\exists \vec{P}^h, \vec{x}^k, \vec{Q}^k \mid (M[\vec{P}/\vec{x}]\,\vec{Q}) \equiv_\beta I$, where I is the identity $\lambda x.x$;
3. $\exists \vec{P}^h, \vec{x}^k, \vec{Q}^k \mid (M[\vec{P}/\vec{x}]\,\vec{Q}) \equiv_\beta y$ with $y \notin \mathrm{FV}(M)$.

Remark 1.2.16. We may note that for closed terms all the part about substitution is not needed: a closed term M is solvable if and only if there exist terms $\vec{Q}^k$ so that $(M\,\vec{Q}) \equiv_\beta I$. Also trivially if $(M\,N)$ is solvable so is M (and so if M is not solvable neither is $M\,N$).

Theorem 1.2.17. *The following properties about a λ-term M are equivalent:*

1. *M is solvable;*
2. *M is β-equivalent to a term in head normal form;*
3. *head reduction on M terminates.*

From this theorem follows that a normalizable term is also a solvable term.

1.3 Representation of recursive functions

Now let's get down to what λ-calculus is about. In representing function λ-calculus is halfway between Turing machines, as it gives proper instruction to compute the function in question, and the axiomatic approach to recursive functions, as it is purely formal.

In order to talk about representing functions we must first of all have a way to represent integers in $\mathbb{N}$ and the boolean values true and false. Let us write

$$(M^n\,N) \text{ for } (\underbrace{M(M\ldots(M}_{n \text{ times}}\,N)\ldots)),$$

that is the result of applying n times M to N.

Definition 1.3.1 (Church integers, booleans). Given $n \in \mathbb{N}$ we define the following term as the *Church representation* of n:

$$\underline{n} := \lambda f \lambda x.(f^n x).$$

We also call $\underline{n}$ an *iterator*: it can be seen as taking a function and an object as arguments and give (with lazy reduction) the function iterated n times on the object as output.

Booleans `true` and `false` are represented by the following terms:

$$\underline{\texttt{true}} := \lambda x \lambda y.x, \quad \underline{\texttt{false}} := \lambda x \lambda y.y.$$

They are a particular case of *selectors* as we will see afterwards. We can regard them as an IF...THEN...ELSE construct in a λ-term: if $b \in \mathbb{B}$ then $(\underline{b}\, M\, N)$ can be read as IF(b) THEN M ELSE N:

$$(\underline{b}\, M\, N) \to_h \begin{cases} M & \text{if } b \text{ is true,} \\ N & \text{otherwise.} \end{cases}$$

Note that all these representations are closed and normal. Note also that $\underline{0} = \underline{\texttt{false}}$, but $\underline{1} \neq \underline{\texttt{true}}$.

Definition 1.3.2 (representation of functions). A function $\varphi : \mathbb{N}^k \to \mathbb{N}$ or $\varphi : \mathbb{N}^k \to \mathbb{B}$ is said to be represented (resp. strongly represented) by a closed term Φ (notation $\Phi = \underline{\varphi}$, though Φ is not unique) if and only if for every $\vec{n} \in \mathbb{N}^k$:

- if $\vec{n} \in \mathrm{DOM}(\varphi)$ then $(\Phi\, \underline{\vec{n}}) \equiv_\beta \underline{\varphi(\vec{n})}$;
- if $\vec{n} \notin \mathrm{DOM}(\varphi)$ then $(\Phi\, \underline{\vec{n}}) \notin WN$ (resp. not solvable).

Note that in the first case, as $\underline{\varphi(\vec{n})}$ is normal, we have the equivalent condition

$$(\Phi\, \underline{\vec{n}}) \twoheadrightarrow_\ell \underline{\varphi(\vec{n})},$$

while the latter case is equivalent to the lazy strategy yielding an infinite reduction (resp. the head reduction strategy). Clearly a strong representation is also a weak representation.

We recall here one of the many definitions used to characterize recursive functions.

Definition 1.3.3 (recursive functions). Given functions $g : \mathbb{N}^h \to \mathbb{N}$ and $\vec{f^h}$ with $f_i : \mathbb{N}^k \to \mathbb{N}$ the *composition scheme* COMP gives a function $\mathrm{COMP}(g, \vec{f}) : \mathbb{N}^k \to \mathbb{N}$ defined by

$$\mathrm{COMP}(g, \vec{f})(\vec{n}) := g(f_1(\vec{n}), \ldots, f_h(\vec{n})).$$

Given a function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ the *minimization scheme* μ gives a function $\mu f : \mathbb{N}^k \to \mathbb{N}$ so that

$$\mu f(\vec{n}) = m \iff f(\vec{n}, i) = 0, \forall i < m : f(\vec{n}, i) \text{ is defined and } \neq 0.$$

The set of recursive (partial) functions is the least set of functions closed under the composition and minimization schemes and containing the following functions called *base functions*:

- the constant function $\mathbf{0} : \mathbb{N} \to \mathbb{N}$, defined by $\mathbf{0}(n) = 0$;
- the successor $\mathtt{succ} : \mathbb{N} \to \mathbb{N}$, defined by $\mathtt{succ}(n) = n + 1$;
- the addition $\mathtt{add} : \mathbb{N}^2 \to \mathbb{N}$, with $\mathtt{add}(m, n) = m + n$;
- the multiplication $\mathtt{mult} : \mathbb{N}^2 \to \mathbb{N}$, with $\mathtt{mult}(m.n) = mn$;
- the characteristic function of $\leq$, $\chi_{\leq} : \mathbb{N}^2 \to \mathbb{N}$, defined by $\chi_{\leq}(m, n) = 1$ if $m \leq n$, 0 otherwise;
- the projections $\pi^i_k : \mathbb{N}^k \to \mathbb{N}$ defined by $\pi^i_k(\vec{n}) = n_i$.

We are leaving out functions to booleans on purpose, as the $0 - 1$-valued functions can take their place. However we will keep using the λ-terms representing them mainly for commodity of the IF...THEN...ELSE construct.

We now want to show that λ-calculus represents all partial recursive functions.

Lemma 1.3.4 (representation of base functions). *The base functions are strongly represented by the following terms:*

- $\underline{\mathbf{0}} := \lambda d.0$;
- $\underline{\mathtt{succ}} := \lambda n \lambda f \lambda x.(n\, f\, (f\, x))$ *or* $\lambda n \lambda f \lambda x.(f\, (n\, f\, x))$;
- $\underline{\mathtt{add}} := \lambda m \lambda n \lambda f \lambda x (m\, f\, (n\, f\, x))$;
- $\underline{\mathtt{mult}} := \lambda m \lambda n \lambda f (m\, (n\, f))$;
- $\underline{\chi_{\leq}} := \lambda m \lambda n.(m\, A\, (\lambda d.\underline{1})\, (n\, A\, (\lambda d.\underline{0})))$ *with* $A := \lambda f \lambda g.(g\, f)$; *we may note we can make this function output representations booleans simply replacing* $\underline{1}$ *with* $\underline{\mathtt{true}}$ *and* $\underline{0}$ *with* $\underline{\mathtt{false}}$;
- $\underline{\pi^i_k} := \overrightarrow{\lambda x^k}.x_i$ *(those terms are called* selectors*).*

Proof. Constant and projections are trivial.

$$(\underline{\mathtt{succ}}\, \underline{n}) \to_h \lambda f \lambda x.(\underline{n}\, f\, (f\, x)) \twoheadrightarrow_h \lambda f \lambda x.(f^n\, (f\, x)) = \underline{n+1}.$$

In a similar way we see $(\underline{\mathtt{add}}\, \underline{m}\, \underline{n}) \overset{\beta}{\twoheadrightarrow} \underline{m+n}$ and $(\underline{\mathtt{mult}}\, \underline{m}\, \underline{n}) \overset{\beta}{\twoheadrightarrow} \underline{mn}$.

$$\begin{aligned}(\underline{\chi_{\leq}}\, \underline{m}\, \underline{n}) \overset{\beta}{\twoheadrightarrow} (A^m\, (\lambda d.\underline{1})\, (A^n\, (\lambda d.\underline{0}))) &\to_h \\ &\to_h (A^n\, (\lambda d.\underline{0})\, (A^{m-1}\, (\lambda d.\underline{1}))) \to_h \\ &\to_h (A^{m-1}\, (\lambda d.\underline{1})\, (A^{n-1}\, (\lambda d.\underline{0}))) \twoheadrightarrow_h \\ &\twoheadrightarrow_h (A^{m-k}\, (\lambda d.\underline{1})\, (A^{n-k}\, (\lambda d.\underline{0}))),\end{aligned}$$

Now if $m \leq n$ when $k = m$ we have

$$((\lambda d.\underline{1})(A^{n-m}\,(\lambda d.\underline{0}))) \to_h \underline{1}.$$

Otherwise when $k = n$ and we take another step

$$((\lambda d.\underline{0})(A^{m-n-1}\,(\lambda d.\underline{1}))) \to_h \underline{0}.$$

□

We will now denote $\hat{n} := (\underline{\mathtt{succ}}^n\,\underline{0})$. Clearly $\hat{n} \overset{\beta}{\twoheadrightarrow} \underline{n}$ and $(\underline{\mathtt{succ}}\,\hat{n}) = \widehat{n+1}$.

Now, to represent the composition scheme one would be tempted, when having the representations $\underline{f_i}$ and $\underline{g}$ of $\vec{f}$ and g, to simply use the term

$$\overrightarrow{\lambda x}.(\underline{g}\,(\underline{f_1}\,\vec{x})\ldots(\underline{f_h}\,\vec{x})).$$

The fact is that it could be normalizable even if one of the f_i is not defined at some point. For example, if f is the function never defined (represented by the term $\lambda d.(D\,D)$, with D like in example 1.2.6), and $g = \mathbf{0}$, then for any n

$$((\lambda x.((\lambda d.\underline{0})\,(\underline{f}\,x)))\underline{n}) \overset{\beta}{\twoheadrightarrow} \underline{0}$$

when we want it to be undefined.

Let us first define that naïve composition between two terms M and N as $M \circ N := \lambda x(M\,(N\,x))$. We may see by induction that if $k \geq 1$ then $((\lambda m.(m \circ N))^k\,P) \twoheadrightarrow_h \lambda x(P\,(N^k x))$.

Now if Φ and ν are two terms let $\langle \Phi, \nu\rangle$ be the term $(\nu\,(\lambda g.(g \circ \underline{\mathtt{succ}}))\,\Phi\,\underline{0})$.

Lemma 1.3.5. *If ν is not solvable then neither $\langle \Phi, \nu\rangle$ is. If $\nu \equiv_\beta \underline{n}$ for some $n \in \mathbb{N}$, then $\langle \Phi, \nu\rangle \equiv_\beta (\Phi\,\underline{n})$ (and so it is solvable if and only if $\Phi\,\underline{n}$ is, and is not solvable if Φ is not).*

Proof. From remark 1.2.16 we get the first claim. If $\nu \equiv_\beta \underline{n}$ and $n \geq 1$:

$$\langle \Phi, \nu\rangle \equiv_\beta (\underline{n}\,(\lambda g.(g \circ \underline{\mathtt{succ}}))\,\Phi\,\underline{0}) \twoheadrightarrow_h ((\lambda g.(g \circ \underline{\mathtt{succ}}))^n\,\Phi\,\underline{0}) \twoheadrightarrow_h$$
$$\twoheadrightarrow_h ((\lambda x.\Phi\,(\underline{\mathtt{succ}}^n\,x))\,\underline{0}) \to_h (\Phi\,\hat{n}) \equiv_\beta (\Phi\,\underline{n}).$$

If $n = 0$ more simply we have

$$\langle \Phi, \nu\rangle \overset{\beta}{\twoheadrightarrow} (\underline{0}\,(\lambda g.(g \circ \underline{\mathtt{succ}}))\,\Phi\,\underline{0}) \twoheadrightarrow_h (\Phi\,\underline{0}).$$

□

Let $\langle \Phi, \vec{\nu}^k\rangle$ be inductively defined by $\langle\langle \Phi, \vec{\nu}^{k-1}\rangle, \nu_k\rangle$.

Lemma 1.3.6. *If any of the ν_is is not solvable so is $\langle \Phi, \vec{\nu}^k \rangle$. If $\nu_i \equiv_\beta \underline{n_i}$ for all i and for some n_is, then $\langle \Phi, \vec{\nu}^k \rangle \equiv_\beta (\Phi\, \underline{\vec{n_i}})$, and thus it is solvable if and only if $(\Phi\, \underline{\vec{n_i}})$ is.*

Proof. By induction using the above lemma. □

Proposition 1.3.7. *Given functions $g : \mathbb{N}^h \to \mathbb{N}$ and $f_i : \mathbb{N}^k \to \mathbb{N}$ strongly represented by terms $\underline{g}$ and $\underline{f_i}$ then $\mathrm{COMP}(g, \vec{f})$ is strongly represented by*

$$\overrightarrow{\lambda x}^k.\langle \underline{g}, (\underline{f_1}\, \vec{x}), \ldots, (\underline{f_h}\, \vec{x}) \rangle.$$

Proof. By a simple application of the above lemma. □

To introduce the minimization scheme, we need the so called *fixed point combinators*. One may see that the minimization scheme is what makes the recursive functions go out of the total functions. When we will study a discipline on λ-terms to have strong normalization, or even normalization within some complexity boundary, there will be no fixed points and minimization.

Definition 1.3.8 (Curry's and Turing's fixed point combinators). A fixed point combinator is a closed normal term M such that $(M\, F) \equiv_\beta (F\, (M\, F))$ for all F. So for we may regard $(M\, F)$ as a fixed point of the function F for any F. The two most known fixed point combinators are

- $Y := \lambda f.(X[f]\, X[f])$ with $X[\ ] := \lambda x.(\square\, (x\, x))$. This is called *Curry's fixed point combinator*. We see that
$$(Y\, F) \to_h (X[F]\, X[F]) \to_h (F\, (X[F]\, X[F])) \equiv_\beta (F\, (Y\, F)).$$
However it is not true that $(Y\, F) \overset{\beta}{\twoheadrightarrow} (F\, (F\, Y))$.
- $\Theta := (A\, A)$ with $A := \lambda a \lambda f.(f\, (a\, a\, f))$. This is *Turing's fixed point combinator*. We have with two head reduction steps:
$$(A\, A\, F) \twoheadrightarrow_h (F\, (A\, A\, F)).$$

Remark 1.3.9. Fixed point combinators are a potent yet dangerous tool in λ-calculus programming. In practice we may design a term M which, apart from the input the program has to process, is made to accept itself as an argument. So the complete program will be $(M\, M)$, M will begin with λm and anywhere I want to replicate the program I may put $(m\, m)$. But like a virus, a self-replicating λ-term may go out of control and have ill complexity effects.

Let T be the term

$$T := \lambda f \lambda g \lambda n (g\, n\, (f\, g\, (\underline{\mathtt{succ}}\, n))\, n)$$

and Δ a fixed point of T, for example $\Delta := (X[T]\, X[T])$. Then

$$(\Delta\, \Phi\, \nu) \to_h (T\, \Delta\, \Phi\, \nu) \twoheadrightarrow_h (\Phi\, \nu\, (\Delta\, \Phi\, (\underline{\mathtt{succ}}\, \nu))\, \nu).$$

Lemma 1.3.10. *If $B \equiv_\beta \underline{\mathtt{true}}$ then for any terms M_1 and M_2 we have*

$$(B\, M_1\, M_2) \twoheadrightarrow_h M_1.$$

Proof. See [Kri90, §II.3]. □

Lemma 1.3.11. *Let Φ be any term, and $n \in \mathbb{N}$. If $(\Phi\, \underline{n})$ is not solvable then neither is $(\Delta\, \Phi\, \underline{n})$. Moreover, if $(\Phi\, \underline{n}) \equiv_\beta \underline{\mathtt{false}}$ then*

$$(\Delta\, \Phi\, \underline{n}) \equiv_\beta (\Phi\, \nu\, (\Delta\, \Phi\, (\underline{\mathtt{succ}}\, \nu))\, \nu) \equiv_\beta \underline{n},$$

while if $(\Phi\, \underline{n}) \equiv_\beta \underline{\mathtt{true}}$ then $(\Delta\, \Phi\, \hat{n}) \twoheadrightarrow_h (\Delta\, \Phi\, \widehat{n+1})$.

Proof. Being $(\Delta\, \Phi\, \underline{n}) \twoheadrightarrow_h ((\Phi\, \underline{n})\, (\Delta\, \Phi\, (\underline{\mathtt{succ}}\, \underline{n}))\, \underline{n})$ we get the first claim from remark 1.2.16. If $(\Phi\, \underline{n}) \equiv_\beta \underline{\mathtt{true}}$ then also $(\Delta\, \Phi\, \underline{n}) \equiv_\beta \underline{n}$. The last claim follows from the previous lemma:

$$((\Phi\, \hat{n})\, (\Delta\, \Phi\, (\underline{\mathtt{succ}}\, \hat{n}))\, \hat{n}) \twoheadrightarrow_h (\Delta\, \Phi\, (\underline{\mathtt{succ}}\, \hat{n}))$$

which is the desired result. □

Let now $\Gamma := \lambda n.(n\, (\lambda d.\underline{\mathtt{true}})\, \underline{\mathtt{false}})$: it is the term that represents the characteristic function of $\mathbb{N} \setminus \{0\}$.

Proposition 1.3.12. *Given a function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ strongly represented by the term $\underline{f}$. The minimization μf is strongly represented by*

$$G = \overrightarrow{\lambda n}^k (\Delta\, (\lambda m.(\Gamma\, (\underline{f}\, \vec{n}\, m)))\, \underline{0}).$$

Proof. From the above lemma. Let $\vec{n} \in \mathbb{N}^k$. Let us write $N = \lambda m.(\Gamma\, (\underline{f}\, \underline{\vec{n}}\, m))$. Suppose now $\mu f(\vec{n}) = p$ is defined. So $f(\vec{n}, p) = 0$ and for all $i < p : f(\vec{n}, i) \neq 0$ and is defined. So trivially $(N\underline{p}) \equiv_\beta \underline{\mathtt{false}}$ and $(N\underline{i}) \equiv_\beta \underline{\mathtt{true}}$ for all $i < p$. Applying the above lemma yields the reduction:

$$(G\, \vec{n}) \twoheadrightarrow_h (\Delta\, N\, \hat{0}) \twoheadrightarrow_h (\Delta\, N\, \hat{1}) \twoheadrightarrow_h \ldots \twoheadrightarrow_h (\Delta\, N\, \hat{p}) \equiv_\beta \underline{p}.$$

$\mu f(\vec{n})$ may be undefined for two reasons. Either $f(\vec{n}, p)$ is undefined and $f(\vec{n}, i) \neq 0$ for all $i < p$, and then we have $(G\, \vec{n}) \equiv_\beta (\Delta\, N\, \underline{p})$ which is unsolvable from the above lemma; or else $f(\vec{n}, i) \neq 0$ for all $i \in \mathbb{N}$, in which case proceding as above:

$$(G\, \vec{n}) \twoheadrightarrow_h (\Delta\, N\, \hat{0}) \twoheadrightarrow_h (\Delta\, N\, \hat{1}) \twoheadrightarrow_h \ldots$$

and so on, giving an infinite head reduction which by theorem 1.2.17 gives unsolvability. □

Remark 1.3.13. The classic *recursion scheme*, which from functions

$$g : \mathbb{N}^{k-1} \to \mathbb{N} \quad \text{and} \quad h : \mathbb{N}^{k+1} \to \mathbb{N}$$

gives a function $\mathrm{REC}(g,h) : \mathbb{N}^k \to \mathbb{N}$ defined so that

$$\mathrm{REC}(g,h)(m,\vec{n}^{k-1}) := \begin{cases} g(\vec{n}) & \text{if } m = 0, \\ h(m-1,\vec{n},\mathrm{REC}(g,h)(m-1,\vec{n})) & \text{otherwise,} \end{cases}$$

can be simulated using the minimization scheme and thus it is not shown how to represent it. However we note as also for recursion we may use more directly the fixed point operators. Take a function r from functions to functions, such that

$$r(f)(m,\vec{n}^{k-1}) := \begin{cases} g(\vec{n}) & \text{if } m = 0, \\ h(m-1,\vec{n},f(m-1,\vec{n})) & \text{otherwise,} \end{cases}$$

This is easily representable in λ-calculus if g, h and `pred`, the predecessor function, are representable, for example by the term

$$R = \lambda f \lambda m \overrightarrow{\lambda x}.(\underline{\chi_{\leq}}\,(\underline{\texttt{pred}}\,m)\,\underline{0}\,(\underline{g}\,\vec{n})\,((\underline{h}\,\vec{n})\,(f\,(\underline{\texttt{pred}}\,m)\,\vec{n}))).$$

We will see how to express the predecessor function inside system **F** framework, see example 3.2.4. Then the terms which represents $\mathrm{REC}(g,h)$ should be a fixed point of R, as $\mathrm{REC}(g,h)$ is the (unique) f such that $f = r(f)$. The only thing to be done is manipulating the terms involved so that also the condition on points where the function is not defined is satisfied, just like we had to check for the other schemes.

So, at last:

Theorem 1.3.14 (λ-calculus represents recursive functions). *Every recursive (partial) function is strongly represented by a λ-term.*

Proof. Now it is trivial: λ-calculus strongly represents the base functions, and composition and minimization schemes are strongly representable. □

Remark 1.3.15. The converse is also true, as, however long to do, we can design a Turing machine that emulates λ-calculus. So the functions represented (not necessarily strongly) by λ-calculus are those T-computable and thus recursive. This is the Church-Kleene theorem.

Now we are ready to reply to the last unanswered question: is there a terminating algorithm that tells me whether a term is normalizable or not? In other words, is WN recursive? The answear is no.

Let M_n be a (computable) enumeration of Λ. Given a term M let us denote by $[\![M]\!]$ the Church integer $\underline{n}$ such that $M = M_n$ (we are embedding the encoding of terms into integers inside the system).

Theorem 1.3.16. *For every term F, there is a term A such that $A \equiv_\beta (F\,[\![A]\!])$.*

Proof. Let us consider the function from $\mathbb{N}$ to itself that taken n gives m such that $M_m = (M_n\,\underline{n})$. We see it is computable. This means, from the above theorem, that there is a term U that represents this function: applied to $\underline{n}$ it β-reduces to the integer $\underline{m}$. That is:

$$\forall n : (U\,\underline{n}) \equiv_\beta [\![(M_n\,\underline{n})]\!].$$

Let

$$B[\ ] := \lambda n.\Box\,(U\,n)$$

and let us consider $B[F]$. For any $n \in \mathbb{N}$ we have

$$(B[F]\,\underline{n}) \xrightarrow{\beta} (F\,(U\,\underline{n})) \xrightarrow{\beta} (F\,[\![(M_n\,\underline{n})]\!]).$$

We may take $\underline{n} = [\![B[F]]\!]$, so that $(M_n\,\underline{n}) = (B[F]\,[\![B[F]]\!])$ and chose A as this last term:

$$A := (B[F]\,[\![B[F]]\!]) \xrightarrow{\beta} (F\,[\![(B[F]\,[\![B[F]]\!])]\!]) = (F\,[\![A]\!]).$$

□

So we are now ready for the following theorem. We recall that a subset of $\mathbb{N}^k$ is said to be *decidable* or *recursive* if its characteristic function is recursive (that is if we can algorithmically decide whether a particular $\vec{n} \in \mathbb{N}^k$ belongs to it or not). We extend this notion to Λ using the above enumeration: $\mathcal{X} \subseteq \Lambda$ is decidable if and only if $\{\, n \mid M_n \in \mathcal{X} \,\}$ is decidable. Two subsets $\mathcal{X}$ and $\mathcal{Y}$ (of any set for which recursive sets has been defined) are said to be *recursively separable* if there is a recursive subset $\mathcal{A}$ such that $\mathcal{X} \subseteq \mathcal{A}$ and $\mathcal{Y} \subseteq A^c$; in this case we say $\mathcal{A}$ recursively separates $\mathcal{X}$ and $\mathcal{Y}$. A subset $\mathcal{X}$ of a certain set is said to be *saturated* with respect to a certain equivalence relation $\sim$ if whenever $x \in \mathcal{X}$ and $x \sim y$ then also $y \in \mathcal{X}$.

Theorem 1.3.17. *Let $\mathcal{X}, \mathcal{Y}$ be two subsets of Λ such that:*

- $\mathcal{X} \cap \mathcal{Y} = \emptyset$;
- $\mathcal{X}, \mathcal{Y} \neq \emptyset$;
- *$\mathcal{X}$ and $\mathcal{Y}$ are saturated with respect to $\equiv_\beta$.*

Then $\mathcal{X}$ and $\mathcal{Y}$ are recursively inseparable.

Proof. Suppose $\mathcal{A}$ is recursive and separates the two sets: $\mathcal{X} \subseteq \mathcal{A}$ and $\mathcal{Y} \subseteq \mathcal{A}^c$. Let A be the term that represents the characteristic function of $\mathcal{A}$, i.e.

$$(A\,\underline{n}) \equiv_\beta \begin{cases} \underline{\texttt{true}} & \text{if } M_n \in \mathcal{A}, \\ \underline{\texttt{false}} & \text{otherwise.} \end{cases}$$

The two sets are non-void, so let us take $M \in \mathcal{X}$ and $N \in \mathcal{Y}$. Let us apply the above theorem to the term $F = \lambda n.A\,n\,N\,M$. There is a term B such that

$$B \equiv_\beta (F\,[\![B]\!]) \xrightarrow{\beta} (A\,[\![B]\!]\,N\,M).$$

Now suppose $B \in \mathcal{A}$: we have $(A\,[\![B]\!]) \equiv_\beta \texttt{true}$ and so the above term is equivalent to N. So $B \equiv_\beta N$ and from $\equiv_\beta$-saturation of $\mathcal{Y}$ we get $B \in \mathcal{Y} \subseteq \mathcal{A}^c$ and we get a contradiction. However if we suppose $B \notin \mathcal{A}$ we get in the same way $B \in \mathcal{X} \subseteq \mathcal{A}$. □

Corollary 1.3.18. *WN is undecidable. So is the set of solvable terms.*

Proof. We take WN (resp. the set of solvable terms) as $\mathcal{X}$ (resp. the set of solvable terms), and $\mathcal{Y} = \mathcal{X}^c$. The two sets respect all of the hypotheses of the above theorem, and $\mathcal{X}$ separates them, so $\mathcal{X}$ is not recursive. □

Chapter 2

Typed λ-calculus

At the end of the previous chapter we have seen that pure λ-calculus has much expressive power, but lacks good computational properties: terms can cause infinite loops and (like all computation models) there is no way to tell if a certain term will terminate apart from trying to run it and wait. A first way to fix these problems is a discipline that assigns *types* to λ-terms and that limits the freedom one had with pure λ-calculus. We can see it as a formal way to tell where we can plug a given term. We may regard it as a programming language discipline: if we have a function that accepts objects of type τ and returns something of type σ we know what to expect when plugging a certain object of type τ. We are prevented from giving as input an object of type different from τ because it could have unexpected (and unwanted) results. A type practically *certificates* the formal correctness of a term.

First approaches to this topics were based to the so called *Church-style* typing. Types were formally introduced as "ontological": λ-calculus is rebuilt from scratch taking variables that have each their specific type embedded in them (the variable x^τ of type τ) and then giving explicit conditions under which the other rules for constructing terms may be applied. With such an approach there is no such thing as an untypable λ-term: we practically cannot write terms that do not have a type.

Later, and this is the approach used in this text, came the *Curry-style* typing. Rather than controlling the way a term is built we try to see if a type can be *assigned* to a pure λ-term (or, in some systems, terms with some kind of decorations added to the grammar that defines the terms).

We may say that the Curry approach separates the algorithmic content (the pure λ-term) from the certificate of correctness (the proof of the term having a particular type) that was embedded in the term in Church-style. The two are surely interchangeable and equivalent, if in Curry-style we take the pair λ-term–certificate to recuperate properties given for granted in Church style.

Let us first introduce these concepts generally.

2.1 An introduction to type systems

Let us call $\mathbb{T}_\Sigma$ a certain set of objects called *types*, ranged over by Greek letters such us σ, τ, ρ, and which contains objects of the form $\sigma \to \tau$ (not necessarily with this symbol). A *type environment* is a function A from λ-term variables to types with finite domain, ranged over by uppercase letters such as A, B. The statement $x : \tau$ is called a *type assumption*. We may freely regard a type environment as a function or as a set of type assumptions so that $x : \tau \in A \iff A(x) = \tau$. In particular a finite set of type assumptions is an environment only if there are no two type assumptions on the same term variable. We may also write a type environment in the form of a sequence of type assumptions $\overrightarrow{x : \tau} = x_1 : \tau_1, \ldots, x_n : \tau_n$. Two type environments A and B are said to be *compatible* if $A \cup B$ is a valid type environment, and in such case we write A, B for $A \cup B$. When we write or use A, B we always assume that they are compatible: eventually this may set an implicit condition on what we are defining.

A statement of the form $A \vdash M : \tau$ where A is a type environment, M a λ-term and τ a type is called a *sequent*, it is read "A induces the type τ on M" and it is ranged over by uppercase Greek letters such as Γ,Δ. We simply write $\vdash M : \tau$ if $\mathrm{DOM}(A) = \emptyset$. The part on the right, i.e. $M : \tau$, is called a *type judgement*.

Definition 2.1.1 (type system). A *type system* Σ is determined by its set of types $\mathbb{T}_\Sigma$ and by a set of rules to derive sequents from other sequents. This set contains at least rules that reflects the rules of λ-term construction: they are rules of the form

$$\frac{}{x : \tau \vdash x : \tau}\ (\mathrm{var}) \qquad \frac{A, x : \sigma \vdash M : \tau}{A \vdash (\lambda x.M) : \sigma \to \tau}\ (\mathrm{abs})$$

$$\frac{A \vdash M : \sigma \to \tau \quad B \vdash N : \sigma}{A, B \vdash (M\,N) : \tau}\ (\mathrm{app})$$

eventually decorated in some manner specific to the system. A system is said to be *syntax-driven* if every rule reflects a rule of λ-term construction. A tree whose nodes are sequents and whose links are rules of the system is called a *type derivation*, or simply a *derivation*. A meta-variable for derivations is $\mathcal{D}$. If a sequent $A \vdash M : \tau$ is the root of some derivation $\mathcal{D}$ we write $\mathcal{D} \rightsquigarrow A \vdash M : \tau$ and say that in Σ M *has type* τ *within the environment* A, written $A \vdash_\Sigma M : \tau$; we call $\mathcal{D}$ a *typing* (in Σ) of M, and we will say that M is *typable*.

Eventually a system may require some changes to the way λ-terms are defined. If it is not the case we say Σ is built upon pure λ-calculus.

We note that, by definition (we work with α-equivalence classes), a typing for a term must be a valid typing for an α-equivalent term, so that a renaming of bounded variables in a term may imply a renaming of type assumption in its type derivation.

We expect from a type system some kind of integration with substitution and reduction. The former is present in practically all the type systems, the latter has to be distinguished between different forms, two weaker than the first.

Definition 2.1.2 (closed under substituion, subterm typing, subject reduction). A type system Σ is *closed under substitution* if whenever

$$x : \sigma, A \vdash_\Sigma M : \tau, \quad B \vdash_\Sigma N : \sigma$$

we have $A, B \vdash_\Sigma M[N/x] : \tau$.

Σ has the *subterm typing* property if whenever M is typable within the environment A, then every subterm N of M is typable within an environment which assigns the same types to variables in $\mathrm{FV}(N) \cap \mathrm{FV}(M)$.

Σ has the *subject reduction* property if whenever $A \vdash_\Sigma M : \tau$ and $M \overset{\beta}{\twoheadrightarrow} N$ then $A \vdash_\Sigma N : \tau$.

Given a restriction of β-reduction $\twoheadrightarrow \subseteq \overset{\beta}{\twoheadrightarrow}$ (eventually, but not necessarily, a certain reduction strategy) we say the system enjoys subject reduction with respect to $\twoheadrightarrow$, or shortly $\twoheadrightarrow$-subject reduction, if the condition above is satisfied with $\twoheadrightarrow$ substituting $\overset{\beta}{\twoheadrightarrow}$.

Σ has the *weak subject reduction property* if whenever $A \vdash_\Sigma M : \tau$ and $M \overset{\beta}{\twoheadrightarrow} N$ and N is normal then $A \vdash_\Sigma N : \tau$.

Given a type system Σ, some questions arise:

- what do I get from adopting Σ?
- what do I lose adopting Σ?
- is there a way to check if a given term is of certain given type in Σ?
- is there a way to check if a given term is typable in Σ?
- is there a way to find a type, or even better, all the types of a given typable term?

We call the problems related to the third and fourth questions *type checking* and *typability* respectively.

Definition 2.1.3 (TC_Σ and TYP_Σ). An instance of TC_Σ is a sequent $A \vdash M : \tau$ defined inside Σ. The type checking problem TC_Σ is the problem of determining if an instance is derivable in Σ, i.e. is $A \vdash_\Sigma M : \tau$.

An instance of TYP_Σ is a term M. The typability problem is the problem of determining whether there exist an environment A and a type τ such that $A \vdash_\Sigma M : \tau$.

We call a type τ *principal* for a term M if τ is a type of M (within a certain environment) and every other type of M can be computed from it with a sequence of certain (finite) operations.

Every time we will present a system we will point out those questions and try to answear to them.

2.2 System S: simple types

The first type system we will encounter is *simply typed λ-calculus*, which we call **S**. It is the most basic one: it employs the minimum required for a type system.

2.2.1 Definition

Definition 2.2.1 (types of S). Given a countable set of *type variables* $\mathbb{V}$, ranged over by Greek letters such as α, β, γ, we define $\mathbb{T}_\mathbf{S}$ with the following grammar:

$$\mathbb{T}_\mathbf{S} ::= \mathbb{V} \mid (\mathbb{T}_\mathbf{S} \to \mathbb{T}_\mathbf{S}).$$

We call $\to$-types (*implication types*) those for which the second rule was used last.

As a convention we omit the parenthes whenever possible, associating implications to the right, so that

$$(\tau_1 \to (\tau_2 \to (\dots (\tau_{n-1} \to \tau_n) \dots))) = \tau_1 \to \dots \to \tau_n.$$

Moreover, we will abbreviate

$$\tau^n \to \sigma := \underbrace{\tau \to \tau \to \dots \to \tau}_{n \text{ times}} \to \sigma.$$

We define $\mathrm{TV}(\tau)$ as the set of type variables occurring in $\mathbb{T}_\mathbf{S}$:

$$\mathrm{TV}(\alpha) := \{\, \alpha \,\}, \qquad \mathrm{TV}(\sigma \to \tau) := \mathrm{TV}(\sigma) \cup \mathrm{TV}(\tau).$$

We define subtypes and subtype occurrences as we did for λ-terms (giving functions to navigate the construction tree of the type). So let L and R be functions defined on $\to$-types that give the left or right part of an implication. $\vec{f} \in \{\, L, R \,\}^*$ is a path in τ if $\vec{f}(\tau)$ is defined, and we call subtype the result, for which $\vec{f}$ itself is an occurrence.

As for terms, we define a notion *length* $|\tau|$ and *depth* $\mathrm{d}(\tau)$:

$$\begin{aligned} |\alpha| &:= 1, & \mathrm{d}(\alpha) &:= 0, \\ |\sigma \to \tau| &:= 1 + |\sigma| + |\tau|\,, & \mathrm{d}(\sigma \to \tau) &:= 1 + \max(\mathrm{d}(\sigma), \mathrm{d}(\tau)). \end{aligned}$$

We also define a notion of *substitution*, which for now will be more basic than the one for terms as there are no bound variables. So we define $\tau[\sigma_1/\alpha_1,\dots,\sigma_n/\alpha_n]$ abbreviated by $\tau[\overrightarrow{\sigma/\alpha}]$:

$$\alpha[\overrightarrow{\sigma/\alpha}] := \begin{cases} \sigma_i & \text{if } \beta=\alpha_i,\\ \beta & \text{otherwise,}\end{cases} \qquad (\tau_1\to\tau_2)[\overrightarrow{\sigma/\alpha}] := \tau_1[\overrightarrow{\sigma/\alpha}]\to\tau_2[\overrightarrow{\sigma/\alpha}]. \tag{2.1}$$

We will denote by letters such as S the functions on types induced by a substitution depicted as above. So S will be a $\to$-*homomorphism* with finite support when restricted to variables. We will assume the notation $S=[\overrightarrow{\sigma/\alpha}]$ if we want to specify what actually does the substitution. We denote by [ ] the empty substitution, i.e. the identity on types.

Definition 2.2.2 (rules of S). Simply typed λ-calculus is given by the following rules:

$$\frac{}{A,x:\tau\vdash x:\tau}\ (\text{var}) \qquad \frac{A,x:\sigma\vdash M:\tau}{A\vdash(\lambda x.M):\sigma\to\tau}\ (\text{abs})$$

$$\frac{A\vdash M:\sigma\to\tau \quad B\vdash N:\sigma}{A,B\vdash(M\,N):\tau}\ (\text{app})$$

So clearly **S** is syntax driven, and whenever $A\vdash_{\mathbf{S}} M:\tau$ then $\mathrm{FV}(M)\subseteq\mathrm{DOM}(A)$.

Example 2.2.3. Every Church integer is typable. For example we can assign to it the type $\mathtt{Int}_\alpha := (\alpha\to\alpha)\to\alpha\to\alpha$:

$$\dfrac{\dfrac{\dfrac{}{f:\alpha\to\alpha\vdash f:\alpha\to\alpha}\ (\text{var}) \quad \dfrac{}{x:\alpha\vdash x:\alpha}\ (\text{var})}{f:\alpha\to\alpha,x:\alpha\vdash(f\,x):\alpha}\ (\text{app})}{\vdots}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f:\alpha\to\alpha\vdash f:\alpha\to\alpha}\ (\text{var}) \quad f:\alpha\to\alpha,x:\alpha\vdash(f^{n-1}\,x):\alpha}{f:\alpha\to\alpha,x:\alpha\vdash(f^n\,x):\alpha}\ (\text{app})}{f:\alpha\to\alpha\vdash\lambda x.(f^n\,x):\alpha\to\alpha}\ (\text{abs})}{\vdash\lambda f\lambda x.(f^n\,x):\mathtt{Int}_\alpha}\ (\text{abs})$$

Also $\underline{\mathtt{true}}$ and $\underline{\mathtt{false}}$ are typable, both for example of type $\mathtt{Bool}_\alpha := \alpha\to\alpha\to\alpha$.

$$\dfrac{\dfrac{\dfrac{}{x:\alpha,y:\alpha\vdash x:\alpha}\ (\text{var})}{x:\alpha\vdash\lambda y.x\alpha\to\alpha}\ (\text{abs})}{\vdash\lambda x\lambda y.x:\alpha\to\alpha\to\alpha}\ (\text{abs})$$

$$\dfrac{\dfrac{\dfrac{}{x:\alpha,y:\alpha\vdash y:\alpha}\ (\text{var})}{x:\alpha\vdash\lambda y.y\alpha\to\alpha\alpha}\ (\text{abs})}{\vdash\lambda x\lambda y.y:\alpha\to\alpha\to\alpha}\ (\text{abs})$$

Remark 2.2.4. We may say more about integers and booleans. Let us put upside down the question, and let us take a-priori the type $\mathtt{Int}_\alpha$, or $\mathtt{Bool}_\alpha$. What can we say about the normal closed terms typable with those types? We will answear to this question when we have taken more confidence with types (see 2.2.11).

Remark 2.2.5. We may see that if we erase every reference to λ-terms we obtain proofs in the implication fragment of classic intuitionistic logic, if we perform the necessary contractions when joining environments and the necessary weakenings after the axioms. Also the converse is true if we once again move around weakenings and contractions. This is an example of the so called *Curry-Howard isomorphism*, which extends also to a dynamical point of view. and we will see as most (but not all) systems have an underlying intuitionistic logic in the same way. We will see for example what is brought to λ-calculus by a more active control on contractions as seen in linear (light) logics.

We could have defined the system with rules that reflect weakening and contraction, but apart from not being useful it would spoil the system from being syntax-driven, which is a good property for resolving TC and TYP.

2.2.2 First properties

A first basic property (related to weakening being integrated in the rules), is that if $A \vdash_{\mathbf{S}} M : \tau$ and $A \subseteq B$ then also $B \vdash_{\mathbf{S}} M : \tau$. It can be proved by first renaming all the bound variables in M so that $\mathrm{BV}(M) \cap \mathrm{DOM}(B) = \emptyset$, and then by appending $B \setminus A$ to all the type environments in the derivation. We can do this because $B \setminus A$ is compatible with every subset of A and with every $x : \sigma$ that disappears in (abs)-rules during the derivation. A (var)-rule so treated is still valid and so the derivation remains valid. Unsurprisingly we call such an operation weakening.

Conversely we have:

Proposition 2.2.6 (weakening on reverse). *We have*

$$A \vdash_{\mathbf{S}} M : \tau \implies A|_{\mathrm{FV}(M)} \vdash_{\mathbf{S}} M : \tau.$$

Proof. By induction on a derivation of $A \vdash_{\mathbf{S}} M : \tau$, depending on the last rule used. We are basically trimming off all unused hypotheses in the environment, i.e. those that neither are necessary for a (var) nor get used and deleted in an (abs).

var: $M = x$ and trivially $A_{\mathrm{FV}(x)} = x : \tau$, so we get by another (var) $x : \tau \vdash_{\mathbf{S}} x : \tau$.

app: $M = (M_1\, M_2)$ and applying the induction hypothesis on the premises $A_1 \vdash M_1 : \tau' \to \tau$ and $A_2 \vdash M_2 : \tau'$ and then an (app) yields:

$$\frac{A_1|_{\mathrm{FV}(M_1)} \vdash_{\mathbf{S}} M_1 : \tau' \to \tau \quad A_2|_{\mathrm{FV}(M_2)} \vdash_{\mathbf{S}} M_2 : \tau'}{A_1|_{\mathrm{FV}(M_1)}, A_2|_{\mathrm{FV}(M_2)} \vdash (M_1\, M_2) : \tau}\ (\mathrm{app})$$

and $A_1|_{\mathrm{FV}(M_1)}, A_2|_{\mathrm{FV}(M_2)} = (A_1, A_2)|_{\mathrm{FV}(M_1) \cup \mathrm{FV}(M_2)} = A|_{\mathrm{FV}(M)}$.

abs: $M = \lambda x.M'$, $\tau = \tau_1 \to \tau_2$, and we apply induction hypothesis on the premise $A', x : \tau_1 \vdash M' : \tau_2$ getting $(A', x : \tau_1)|_{\mathrm{FV}(M')} \vdash_{\mathbf{S}} M' : \tau_2$. If eventually $x \notin \mathrm{FV}(M')$ we add it by weakening, so in any case by pointing out $x : \tau_1$ from A' we get $A'|_{\mathrm{FV}(M')\setminus\{x\}}, x : \tau_1 \vdash_{\mathbf{S}} M' : \tau_2$. An application of the (abs)-rule gives the result as $\mathrm{FV}(M') \setminus \{x\} = \mathrm{FV}(M)$.

□

Then we may see properties like closure under substitution and subterm typing:

Lemma 2.2.7 (substitution lemma). S *is closed under substitution.*

Proof. Let $A, x : \sigma \vdash_{\mathbf{S}} M : \tau$ and $B \vdash_{\mathbf{S}} N : \sigma$. Let us reason by induction on the size of a derivation of M (or equivalently on the size of M), depending on its last rule. The basic idea is that whenever x is introduced by a (var)-rule we replace it with the type derivation of N.

var: In this case either $M = x$ and $\sigma = \tau$ or $M = y \in \mathrm{DOM}(A)$. In the first case the proposition goes down to stating $A, B \vdash_{\mathbf{S}} N : \tau$ which is true by weakening on the derivation of the type for N. In the second one we have $A, B \vdash_{\mathbf{S}} y : \tau$ which is true by weakening on $M = y$'s derivation.

app: $M = (M'\, M'')$. If $x : \sigma$ is not present in one of the premises we don't need to apply induction hypothesis on that premise because we know then x is not free in the term relative to that premise, and so substitution is harmless. Let's presume the two premises are $A', x : \sigma \vdash_{\mathbf{S}} M' : \tau' \to \tau$ and $A'', x : \sigma \vdash_{\mathbf{S}} M'' : \tau'$ with $A = A', A''$. Then by induction hypothesis:

$$\frac{A', B \vdash_{\mathbf{S}} M'[N/x] : \tau' \to \tau \quad A'', B \vdash_{\mathbf{S}} M''[N/x] : \tau'}{A, B \vdash (M'[N/x]\, M''[N/x]) : \tau}\ (\mathrm{app})$$

and $(M'[N/x]\, M''[N/x]) = M[N/x]$.

abs: $M = \lambda y.M'$, $\tau = \tau' \to \tau''$ and $A, y : \tau', x : \sigma \vdash_{\mathbf{S}} M' : \tau''$. We eventually rename y so that it doesn't appear in $\mathrm{DOM}(B)$. So by induction hypothesis:

$$\frac{A, y : \tau', B \vdash_{\mathbf{S}} M'[N/x]}{A, B \vdash \lambda y.M'[N/x]}\ (\mathrm{abs})$$

and $\lambda y.M'[N/x] = M[N/x]$.

□

Proposition 2.2.8 (subterm typing). S *enjoys subterm typing. In particular, if N is a subterm occurrence of M, every $\mathcal{D}$ such that $\mathcal{D} \rightsquigarrow A \vdash M : \tau$ contains a subderivation $\mathcal{D}' \rightsquigarrow A' \vdash N : \sigma$, with $A|_{\mathrm{FV}(M)\cap\mathrm{FV}(N)} = A'|_{\mathrm{FV}(M)\cap\mathrm{FV}(N)}$. We say that $\mathcal{D}$ assigns type σ to N.*

Proof. This clearly comes from the fact that type derivations reflect term construction trees. So if $\vec{f}$ is a path in M leading to N, then following the same path in a typing of M yields a subderivation for N. The differences between the environments come from variables in A' which later get bounded (and so they are not in $\mathrm{FV}(M)$), and variables which are added later (which therefore are not in $\mathrm{FV}(N)$). □

Finally we will prove subject reduction property.

Theorem 2.2.9 (subject reduction). S *enjoys subject reduction.*

Proof. Clearly subject reduction is equivalent to seeing it for one step reductions. So let $\mathcal{D} \rightsquigarrow A \vdash_{\mathbf{S}} M : \tau$ and N any term such that $M \xrightarrow{\beta} N$. Let us reason by induction on $\mathcal{D}$.

var: For this case the statement is true as a variable has no redexes.

app: Here $M = (M_1\, M_2)$ and $\mathcal{D}_1 \rightsquigarrow A_1 \vdash M_1 : \tau' \to \tau$ and $\mathcal{D}_2 \rightsquigarrow A_2 \vdash M_2 : \tau'$. We must distinguish between three cases: either we are reducing a redex in M_1, or else in M_2, or else M itself is the redex contracted.

In the first case, $M_1 \xrightarrow{\beta} N_1$ and $N = (N_1\, M_2)$. By induction hypothesis $A_1 \vdash_{\mathbf{S}} N_1 : \tau' \to \tau$ and with an (app) we get what needed. The second case is treated in the same manner.

If M is the redex contracted then $M_1 = \lambda x.M_1'$ and $N = M_1'[M_2/x]$. Then going up in the subderivation for M_1 gives

$$\mathcal{D}_1' \rightsquigarrow A_1, x : \tau' \vdash M_1' : \tau,$$

and by substitution lemma $A_1, A_2 \vdash_{\mathbf{S}} M_1'[M_2/x] : \tau$ that is what we were looking for.

abs: $M = \lambda x.M'$, $\tau = \tau_1 \to \tau_2$ and $\mathcal{D}' \rightsquigarrow A, x : \tau_1 \vdash M' : \tau_2$. We are necessarily reducing a redex in M', so that $M' \xrightarrow{\beta} N'$ and $N = \lambda x.N'$. By induction hypothesis

$$\frac{A, x : \tau_1 \vdash_{\mathbf{S}} N' : \tau_2}{A \vdash \lambda x.N' : \tau_1 \to \tau_2}\ (\mathrm{abs})$$

and that is all.

□

Remark 2.2.10. We may note in the proof that we are saying slightly more than simple subject reduction. Not only the descendant N gets the same type within the same environment, but given a certain typing $\mathcal{D}$ of M it induces a particular typing in N such that subterms in N left unchanged by reduction retain the same type, and also contracta retain the same type of the redexes they come from. This is useful because there can be different typings within the same environments,

and so types of subterms could go wild during a reduction. For example the term $(\lambda x.d)\,(I\,I)$ is typed with type α within the environment $d : \alpha$. This type however does not depend on which type is assigned to the subterm $(I\,I)$, and keeps not depending if we contract $(I\,I) \xrightarrow{\beta} I$, so that we may assign to I a completely different type.

Remark 2.2.11. Let us recover the topic about what closed normal terms are typed with type $\mathtt{Int}_\alpha$ and $\mathtt{Bool}_\alpha$. Let us take a look at $\mathtt{Int}_\alpha$: first of all by weakening on reverse we can take a derivation with an empty environment in the final sequent without loosing generality. So the last rule of such a derivation must be an abstraction, which is the only one that can lead to empty environment. So the sequent immediately preceding the end must be

$$f : \alpha \to \alpha \vdash \lambda f.M : \alpha \to \alpha$$

with M normal, and its only free variable can be y. So if M is a variable it must be y, that yields the term $\lambda f.f$. It is not a Church integer, but we will explain it later. If $M = (M_1\,M_2)$, then M_1 cannot be an abstraction, but cannot either be the variable y because of type mismatch. So it also should be an application but then we could iterate the reasoning in an infinite chain of applications. So the only possibility left is for it to be an abstraction. Going up one rule in the derivation we have the sequent:

$$f : \alpha \to \alpha, x : \alpha \vdash N : \alpha.$$

No more abstractions are possible (here we are using the fact that $\mathtt{Int}_\alpha$ was built upon a type variable). So either N is a variable and in such case it must be x (the only one in the environment with the same type), or an application $(N_1\,N_2)$. The first case gives $\lambda f \lambda x.x = \underline{0}$. In the second case N_1 must be assigned a type $\alpha \to \alpha$: if it is a variable it must be f (the only one with implication type); otherwise (as it cannot be an abstraction) we should have $(N_{11}\,N_{12})$, and N_{11} cannot be any variable because its type is too complex. Reasoning as before we get another infinite chain of applications, so $N_1 = f$. Reasoning on the normal term N_2 of type α as we have done for N, and going on like this until we finally find the x (there are no infinite terms!) we see that all other terms of this type are $\lambda f \lambda x.(f^n\,x) = \underline{n}$.

How does $\lambda f.f$ fall into this? This is a slight imperfection of the representation of integers, as it is just an alternative representation of 1, in the sense that using both instead of only the classic $\underline{1}$ gives the same results. For example

$$\underline{\mathtt{succ}}\,\lambda f.f \xrightarrow{\beta} \lambda f' \lambda x.((\lambda f.f)\,f'\,(f'\,x)) \xrightarrow{\beta} \lambda f' \lambda x.(f'\,(f'\,x)) = \underline{2}.$$

This is consistent with the idea that integers are iterators: a 1-iterator is practically the identity on functions[1]. If we do not want this ambiguity we can leave the classic representation by simply

[1] putting in η-equivalence, we have $\underline{1} \xrightarrow{\eta} I$.

exchanging the abstractions and use instead $\underline{n} := \lambda x\lambda f.(f^n\, x)$, and the corresponding type $\mathtt{Int}_\alpha := \alpha \to (\alpha \to \alpha) \to \alpha$. Or else we can make sure that a given result is indeed a Church integer by putting a term of type $\mathtt{Int}_\alpha$ i the context $\lambda f\lambda x.\Box\, f\, x$. We will soon see that every typable term is SN: so we may say that every normal closed term typable with $\mathtt{Int}_\alpha$ normalizes to an integer, and so every term typable with $\mathtt{Int}_\alpha^k \to \mathtt{Int}_\alpha$ represents a function from $\mathbb{N}^k \to \mathbb{N}$. We may say that $\mathtt{Int}_\alpha$ gives a faithful representation of $\mathbb{N}$. Alas, we will see in 2.2.22 that though we are now sure the terms typable with $\mathtt{Int}_\alpha^k \to \mathtt{Int}_\alpha$ represent a class of functions, they represent a really poor class of functions. If we want to extend the representation we must allow complex types to appear in $\mathtt{Int}_\tau$, which in turn may give normal terms that are not integers but have the type of integers, in any of the two representation of integers we have given. For example

$$\vdash_{\mathbf{S}} \lambda f\lambda x\lambda z.z : \mathtt{Int}_{\alpha\to\alpha}$$

in both the ways we may define $\mathtt{Int}_\tau$ (it suffices to exchange the roles of f and x).

For $\mathtt{Bool}_\alpha$ we may apply the same reasoning and see that without ambiguities normal closed terms of this form are either $\underline{\mathtt{true}}$ or $\underline{\mathtt{false}}$. Still if we allow complex types to appear the type ceases to give a faithful representation of $\mathbb{B}$.

We will continue this topic in the framework of system **F** which gives much more satisfaction in this sense.

2.2.3 What do we get from S?

The best thing I get from adopting **S** is that every typable term is not only normalizable, but even strongly normalizable. Because strong normalization implies the weak one we could show only the second result. There are two main ways to do this. One is by indeed showing first WN, and then, aided by a complicated translation of terms into other terms, show that WN implies SN. For example, see [Gan80]. Another shows directly and independently SN using techniques that may be then expanded to other systems, and this is the one presented here. However we will also briefly give the proof of WN, because it highlights an interesting bound on the depth of the normal form of a term (and thus also to its size).

Theorem 2.2.12. *If M is typable in* **S** *then $M \in WN$.*

Proof. Let M be a typable with a typing $\mathcal{D}$. Because of subterm typing every redex has a type. Given a redex $R = (\lambda x.P)\, Q$ where $\lambda x.P$ is given type $\sigma_1 \to \sigma_2$ by $\mathcal{D}$, we define the *degree* of the redex by $\partial(R) := \mathrm{d}(\sigma_1 \to \sigma_2)$.

Note that the definition on redexes is not independent from $\mathcal{D}$ as we have seen in remark 2.2.10.

We go on defining a *redex degree* of M as

$$\mathrm{rd}(M) := \max\{\, \partial(R) \mid R \text{ redex in } M \,\},$$

and setting it to 0 if there are no redexes. We may see that in general $\mathrm{rd}(M) \leq \mathrm{rd}(M')$ if $M \overset{\beta}{\twoheadrightarrow} M'$.

Now, we see that for every term M with $\mathrm{rd}(M) \leq d$ with $d > 0$ there is a term M' such that $M \overset{\beta}{\twoheadrightarrow} M'$ and $\mathrm{rd}(M') < d$. By the way, we will also show that $\mathrm{d}(M') \leq 2^{\mathrm{d}(M)}$. Practically we are contracting redexes bottom-up in the construction tree, so that we do not contract redexes that were not visible at the beginning of the step. So, by induction on M:

$M = x$**:** Nothing to show, $M' = x$.

$M = \lambda x.N$**:** $\mathrm{rd}(N) = \mathrm{rd}(M)$, so by induction hypothesis $\lambda x.N \overset{\beta}{\twoheadrightarrow} \lambda x.N'$, with $\mathrm{rd}(\lambda x.N') = \mathrm{rd}(N') < d$ and

$$\mathrm{d}(\lambda x.N') = 1 + \mathrm{d}(N') \leq 1 + 2^{\mathrm{d}(N)} < 2^{\mathrm{d}(M)}.$$

$M = (M_1\, M_2)$**:** By induction hypothesis $M \overset{\beta}{\twoheadrightarrow} (M_1'\, M_2')$, with M_1' and M_2' satisfying the properties. Now, if M_1' is not an abstraction, or if it is an abstraction but the degree of the redex $(M_1'\, M_2')$ is already less than $d - 1$:

$$\mathrm{rd}(M_1'\, M_2') = \max(\mathrm{rd}(M_1'), \mathrm{rd}(M_2'), \partial(M_1'\, M_2')) < d,$$

and

$$\mathrm{d}(M_1'\, M_2') = 1 + \max(\mathrm{d}(M_1'), \mathrm{d}(M_2')) \leq 1 + 2^{\max(\mathrm{d}(M_1), \mathrm{d}(M_2))} < 2^{\mathrm{d}(M)}.$$

If on the other hand $M_1' = \lambda x.P$ and the type assigned to $\lambda x.P$ by the initial typing is $\tau_1 \to \tau_2$ with $\mathrm{d}(\tau_1 \to \tau_2) = d$, we may take one more reduction step and obtain $P[M_2'/x]$. Here all the redexes are those in P (that are those in M_1'), those in M_2' (if x appears in P) and those new, created if M_2' is an abstraction and gets in the left part of an application. As the degrees of the redexes already present does not change with substitution, of those redexes the only ones that might have degree higher than $d - 1$ are the last ones. However they have all degree equal to the depth of the type of M_2', i.e. $\mathrm{d}(\tau_1) < \mathrm{d}(\tau_1 \to \tau_2) = d$. As for the depth of the term, it is easily seen that

$$\mathrm{d}(P[M_2'/x]) \leq \mathrm{d}(P) + \mathrm{d}(M_2') < \mathrm{d}(M_1') + \mathrm{d}(M_2') \leq 2^{\mathrm{d}(M_1)} + 2^{\mathrm{d}(M_2)} \leq 2^{\mathrm{d}(M)}.$$

Weak normalization is then obtained by applying the reduction depicted above until $\mathrm{rd}(M) = 0$ and so M is normal.

□

Corollary 2.2.13. *The depth of the normal form of a term M is bounded by $e_2(\mathrm{d}(M), \mathrm{rd}(M))$ where $e_a(m,n)$ is the tower of exponentials of base a, height n and argument m, that is:*

$$e_a(m,n) := \begin{cases} m & \text{if } n = 0, \\ a^{e_a(m,n-1)} & \text{otherwise.} \end{cases}$$

Its size is bounded by $2e_2(\mathrm{d}(M), \mathrm{rd}(M)+1)$.

Moreover there is a reduction strategy that makes these bounds valid for every term N in the reduction chain starting from M.

Proof. It follows directly from the two bounds of the reduction that gives WN in the proof above. The bound on size is from the easy relation $|M| \leq 2^{\mathrm{d}(M)+1}$. □

Before moving on to strong normalization, we give the definition of *reducibility candidate for type τ*. This definition is valid for all the systems Σ built upon pure λ-calculus, and is easily extendable otherwise. It is central to many strong normalization theorems. Rather than being useful here (as we will simply state that a certain fixed set called the *reducible* terms of type τ is a reducibility candidate), they will be useful in the future when a definition of reducible terms is not possible. Having chosen a Church-style approach we must take into account also the type environments in which the typings are carried out.

Definition 2.2.14 (reducibility candidate). A set

$$\mathcal{X} \subseteq \{\, A \text{ type environment} \,\} \times \Lambda$$

of pairs (A, M) such that $A \vdash_\Sigma M : \tau$ holds, is a *reducibility candidate for τ* if the following three properties hold, where we denote by π_2 the usual projection ($\pi_2\mathcal{X} = \mathrm{RAN}(\pi_2|_{\mathcal{X}})$, with $\pi_2(a,b) := b$):

1. $\pi_2\mathcal{X} \subseteq SN$;
2. if $(A, M) \in \mathcal{X}$ and $M \overset{\beta}{\twoheadrightarrow} N$ then $(A, N) \in \mathcal{X}$.
3. if M typeable with type τ within A is not an abstraction and for every M' such that $M \overset{\beta}{\rightarrow} M'$ (single step) we have $(A, M') \in \mathcal{X}$ then $(A, M) \in \mathcal{X}$.

Note that the last clause implies that a normal typable term that is not an abstraction also is in $\pi_2\mathcal{X}$. In particular $(A, x) \in \mathcal{X}$ for every (A, x) such that $A(x) = \tau$.

Note also that if subject reduction holds then we must not check whether the type is preserved in the second property.

Definition 2.2.15 (reducible terms). The set RED_τ of reducible terms of type τ together with environments is defined inductively on τ as follows, assuming that all pairs (A, M) in RED_τ must be such that $A \vdash_{\mathbf{S}} M : \tau$:

$$(A, M) \in \mathrm{RED}_\alpha \iff M \in SN,$$
$$(A, M) \in \mathrm{RED}_{\tau_1 \to \tau_2} \iff \forall (B, N) \in \mathrm{RED}_{\tau_1} : ((A, B), (M\,N)) \in \mathrm{RED}_{\tau_2}\,.$$

As usual in order to use $A, B = A \cup B$ we implicitly impose in the quantifier that A and B are compatible. We will switch here to the insiemistic notation $A \cup B$ for ease of notation. We define the set of reducible terms as the union of the projections $\pi_2\,\mathrm{RED}_\tau$ with τ ranging over the types. Note that if $(A, M) \in \mathrm{RED}_\tau$ and $A \subseteq A'$ then $(A', M) \in \mathrm{RED}_\tau$.

Lemma 2.2.16. *RED_τ is a reducibility candidate for every τ.*

Proof. By induction on τ, checking the three properties of candidates:

$\tau = \alpha$: 1. the first property is satisfied by definition.

2. SN is closed under $\overset{\beta}{\twoheadrightarrow}$, and so the property holds.

3. Every reduction chain must have its first step in one of the M's, which in turn by definition are in SN, so the chain must be finite. In fact $||M|| = 1 + \max(||M_i||)$.

$\tau = \tau_1 \to \tau_2$: 1. Let $(A, M) \in \mathrm{RED}_{\tau_1 \to \tau_2}$. Let us take any variable x not present in A and consider $(x : \tau_1, x) \in \mathcal{X}$ by induction hypothesis 3. By definition and induction hypothesis 1, we have

$$(M\,x) = \pi_2(A \cup \{x : \tau_1\}, (M\,x)) \in \pi_2\,\mathrm{RED}_{\tau_2} \subseteq SN.$$

Moreover for every reduction chain M_i starting from M we have a corresponding reduction chain $(M_i\,x)$ starting from $(M\,x)$, so that $||M|| \leq ||(M\,x)|| < \infty$.

2. Let $M \overset{\beta}{\twoheadrightarrow} N$ with $(A, M) \in \mathrm{RED}_{\tau_1 \to \tau_2}$. Let (B, P) be in RED_{τ_1} with A and B compatible. Then

$$(A \cup B, (M\,P)) \in \mathrm{RED}_{\tau_2}, \quad (M\,P) \overset{\beta}{\twoheadrightarrow} (N\,P),$$

so by induction hypothesis 2 we get $(A \cup B, (N\,P)) \in \mathrm{RED}_{\tau_2}$ which was what needed.

3. Let (B, P) be any term in RED_{τ_1}, with B compatible with A: we want to show that $(A \cup B, (M\,P)) \in \mathrm{RED}_{\tau_2}$, and we show it by another induction on $||P||$ (defined because by induction hypothesis 1 $P \in SN$). As M is not an abstraction each redex in $(M\,P)$ is either in M or in P. In the first case reducing it leads to $(M'\,P)$ for one of the M's, and so by definition $(A \cup B, (M'\,P)) \in \mathrm{RED}_{\tau_2}$. In the second case the one step reduction leads to $(M\,P')$ with $P \overset{\beta}{\to} P'$: $(B, P') \in \mathrm{RED}_{\tau_1}$ by induction hypothesis 2, and so by

the other induction hypothesis ($||P'|| = ||P|| - 1$) we also get $(A \cup B, (M\,P')) \in \mathrm{RED}_{\tau_1}$. Now using induction hypothesis 3 we have $(A, (M\,N)) \in \mathrm{RED}_{\tau_2}$.

□

We now bring down to reducibility the abstraction that eludes the properties of reducibility candidates in the third clause.

Lemma 2.2.17 (abstraction). *Given A and M, if for every $(B, N) \in \mathrm{RED}_{\tau_1}$ such that A and B are compatible we have $(A \cup B, M[N/x]) \in \mathrm{RED}_{\tau_2}$ then $(A, \lambda x.M) \in \mathrm{RED}_{\tau_1 \to \tau_2}$.*

Proof. We obviously have $(A \cup \{x : \tau_1\}, M) \in \mathrm{RED}_{\tau_2}$, so M itself is reducible. Let (B, N) be any pair in RED_{τ_1} such that A and B are compatible, and let us show that $(A \cup B, (\lambda x.M)\,N) \in \mathrm{RED}_{\tau_2}$ by induction on $||M|| + ||N||$ (defined because both reducible). We have that every one step reduction that contracts a redex in M or in N makes the pair end within RED_{τ_2} by induction hypothesis. The only other redex left is the term itself, and by reducing it we remain in RED_{τ_2} by hypothesis. So, by the third property of reducibility candidates, we get what needed. □

Now what remains to be done is proving that all terms are reducible, then SN follows from the first property of reducibility candidates. Something stronger needs to be proved:

Theorem 2.2.18. *Let (A, M) be such that $\mathcal{D} \rightsquigarrow A \vdash_{\mathbf{S}} M : \tau$, $\mathrm{FV}(M) \subseteq \vec{x}^n$ and $\vec{N}^n$ are terms such that $\forall i : (B, N_i) \in \mathrm{RED}_{A(x_i)}$ with B compatible with A. Then $(A \cup B, M[\overrightarrow{N/x}]) \in \mathrm{RED}_\tau$.*

Proof. By induction on M.

$M = x_i$**:** Tautological.

$M = (M_1\,M_2)$**:** We have subderivations of $\mathcal{D}$ that by weakening we may consider within the same environment and that give types $\tau' \to \tau$ to M_1 and τ' to M_2. By induction hypothesis $(A \cup B, M_1[\overrightarrow{N/x}]) \in \mathrm{RED}_{\tau' \to \tau}$ and $(A \cup B, M_2[\overrightarrow{N/x}]) \in \mathrm{RED}_{\tau'}$, so by definition we get what needed.

$M = \lambda y.M'$**:** We rename y so that it does not appear anywhere else. $\tau = \tau_1 \to \tau_2$ and $y : \tau_1, A \vdash_{\mathbf{S}} M' : \tau_2$. Let us take any $(C, P) \in \mathrm{RED}_{\tau_1}$ such that C is compatible with $A \cup B$. In particular $(B \cup C, N_i)$ remains in $\mathrm{RED}_{A(x_i)}$. By induction hypothesis we have

$$(A \cup B \cup C, M'[\overrightarrow{N/x}][P/y]) = (A \cup B \cup C, M'[P/y, \overrightarrow{N/x}]) \in \mathrm{RED}_{\tau_2},$$

so by the lemma above we get $(A \cup B, \lambda y.M'[\overrightarrow{N/x}]) \in \mathrm{RED}_{\tau_1 \to \tau_2}$.

□

Corollary 2.2.19. *All typable terms are reducible, and so all typable terms are in SN.*

Proof. Just take $\vec{N} = \vec{x}$ and $B = A|_{\vec{x}}$. □

2.2.4 What do we lose with S?

What actually is missing in simply typed λ-calculus is expressive power. First of all, we can't use fixed points, so we lose the possibility to easily express function defined recursively[2].

Proposition 2.2.20. *Church's and Curry's fixed point combinators are not typable in* **S**.

Proof. This follows directly from the fact that there is no way to type self application of variables, i.e. $(x\,x)$ is not typable no matter what environment one sets. Note that in Church-style there are some self applications that are typable, for example $(I\,I)$. □

Proposition 2.2.21. *There is no term Y typable with $(\sigma \to \sigma) \to \sigma$ such that given any term typable with $\sigma \to \sigma$ we have $Y\,M \overset{\beta}{\twoheadrightarrow} M\,(Y\,M)$.*

Proof. Trivial: it would contradict strong normalization theorem. □

There is also another problem regarding what functions are representable. As we saw in example 2.2.3, Church integrals are typable of type $\mathtt{Int}_\alpha$. We will see that every type for Church integrals is $\mathtt{Int}_\tau$ for some τ (thus $\mathtt{Int}_\alpha$ is a principal type), if we take them as a whole[3], so it depends on the choice of τ. The first choice would be to take τ as simple as possible, i.e. a variable, and then in order to be coherent we should stick to $\mathtt{Int}_\alpha$, so that we say a closed term F represents a function $f : \mathbb{N}^k \to \mathbb{N}$ if and only if it is typable with type $\mathtt{Int}_\alpha^k \to \mathtt{Int}\,\alpha$ and for every $\vec{n} \in \mathbb{N}^k$ we have $(F\,\underline{\vec{n}}) \equiv_\beta \underline{f(\vec{n})}$. However the resulting representable functions are quite a poor class.

Theorem 2.2.22. *The functions representable by terms of type $\mathtt{Int}_\alpha \to \mathtt{Int}_\alpha \to \ldots \to \mathtt{Int}_\alpha$ for some α are the least class closed under composition containing* **0**, `succ`, `add`, `mult`, χ_0 *(the characteristic function of $\{0\}$) and the projections. This class is practically that of multivariate polynomials extended with conditional clause.*

Proof. One direction is trivial, as it suffices to see that the representations of base functions here listed, as seen in 1.3.4, are typable with the appropriate type. We can represent χ_0 with the typable term:

$$\underline{\chi_0} := \lambda n \lambda f \lambda x.(n\,(\lambda d.x)\,(f\,x)).$$

The converse has been proved in [Sch76]. □

[2] unrestricted minimization scheme is out of the question as it leads out of the class of total functions, while by SN we must remain inside of it. We may also note how if we stick to typable λ-terms there is no distinction between strong and weak representation.

[3] $\underline{0}$ is in fact of principal type $\alpha \to \beta \to \beta$, while $\underline{1}$ has principal type $(\alpha \to \beta) \to \alpha \to \beta$. Starting from 2 the principal type must be the one stated above.

On the other hand we may take $\mathtt{Int}_\tau$ with τ exchangeable. A typical example is taking $\alpha_0 := \alpha$ and inductively $\alpha_{i+1} := \alpha_i \to \alpha_i$, and then taking integers of $\mathtt{Int}_i := \mathtt{Int}_{\alpha_i}$. With this convention we may easily, for example, represent the exponential of type $\mathtt{Int}_{i+1} \to \mathtt{Int}_i \to \mathtt{Int}_i$:

$$\underline{\mathtt{exp}} := \lambda m \lambda n.(m\, n),$$

that represents $\mathtt{exp}(m,n) := n^m$. Note that the type of the exponent is more complex than that of input and output. We can now represent a great deal of functions, but still we cannot for example represent such simple functions as $\chi_=$ or $\chi_\leq$, even if we let the type of integers be built from any type, i.e. if we let integers to be of type $\mathtt{Int}_\tau$ for any τ. By corollary 2.2.13 we know we remain inside the scope of elementary space functions. This by a known result means we are inside the elementary functions (see 4.2.1). However the inclusion is strict.

And then again, in order to go beyond polynomial functions, we have to "cheat": we *change* the types of the integers we are dealing with, so that there is no single type for integers but rather a *scheme* the types for integers must follow. This is clearly a prelude to polymorphism, where a single type will do the trick. A first solution can be however to adopt some external terms (*constants*) that will have reduction rules to emulate that which simply typed λ-calculus cannot achieve: for example system **PCF**, shown in the next section.

2.2.5 Type checking, typability and type inference

Adopting Curry-style saves us form cumbersome programming, by splitting the problem of finding an algorithm that computes what we want to compute (by designing the pure term) from the problem of certificating that the algorithm is correct from the point of view of termination (by finding a typing for the pure term). It is known that the first problem is completely up to the programmer. We would like to leave the second one to a machine.

We will now see as there are no problems regarding the assignment of simple types to terms. The problems in fact goes down to finding *unifiers* for types, that is substitutions that make two types equal.

Definition 2.2.23 (unifiers). A *unifier* for type σ and τ is a substitution S such that $S(\sigma) = S(\tau)$. We then say that this common value is a *unified type* for σ and τ.

We will say that a unifier S for σ and τ is a *most general unifier* if for every unifier T_1 of the same types there exist a substitution T_2 such that $T_1 = T_2 \circ S$.

If E is a set of equations on types $\{\, \sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n \,\}$, we say that S is a unifier for E if it is a unifier for each pair (σ_i, τ_i), and we say it is a most general one as we did above.

Example 2.2.24. For example $(\alpha \to \beta) \to \beta$ and $\gamma \to \delta \to \delta$ have the unifiers

$$S = [\delta \to \delta/\beta, \alpha \to \delta \to \delta/\gamma],$$
$$S_1 = [\delta \to \delta/\beta, \varepsilon \to \varepsilon/\alpha, (\varepsilon \to \varepsilon) \to \delta \to \delta/\gamma].$$

The first is a most general one, the second is not.

We will say that a type σ is a *variant* of type τ if there exist substitutions S_1 and S_2 such that $S_1(\sigma) = \tau$ and $S_2(\tau) = \sigma$. Types unified by a most general unifier are always a variant of each other.

In this case with simple types unification is decidable.

Proposition 2.2.25. *There is a computable function that accepts a pair of types and outputs either a substitution or a value* `fail` *such that*

- *if σ and τ have a unifier then $U(\sigma, \tau)$ is a most general unifier for them,*
- *otherwise $U(\sigma, \tau)$ returns* `fail`.

Moreover such function is polynomial on the size of the two types.

We can extend this function to systems of equations E, so that U gives a most general unifier if there is a unifier and `fail` *if there is none.*

Proof. Recall R and L are the functions giving the right and left part of an implication, and are defined only on implications.

U on types is defined recursively by the following algorithm:

Require: σ and τ types;
1: **if** $\sigma = \tau$ **then return** [ ];
2: **else**
3: **if** $\sigma \in \mathbb{V}$ **then**
4: **if** $\sigma \notin \mathrm{TV}(\tau)$ **then return** $[\tau/\sigma]$;
5: **else return** `fail`;
6: **else**
7: **if** $\tau \in \mathbb{V}$ **then return** $U(\tau, \sigma)$;
8: **else**
9: $S_2 \leftarrow U(R(\sigma), R(\tau))$;
10: **if** $S_2 =$ `fail` **then return** `fail`;
11: **else**
12: $S_1 \leftarrow U(S_2(L(\sigma)), S_2(L(\tau)))$;

13: **if** $S_1 = \texttt{fail}$ **then return** `fail`;

14: **else return** $S_1 \circ S_2$;

The algorithm terminates because each time the recursive call is made at most two times on strictly smaller types, and a binary tree with finite branches is finite. Moreover it gives a most general unifier if there is one, which can be seen by induction.

In fact the base is that α and τ cannot be unified if and only if $\tau \neq \alpha$ and $\alpha \in \mathrm{TV}(\tau)$, because in such a case for every substitution S we have $|S(\tau)| > |S(\alpha)|$. Otherwise $[\tau/\alpha]$ is a most general unifier, as we see that if $S_1(\alpha) = S_1(\tau)$ then in fact $S_1 = S_1 \circ [\tau/\alpha]$:

$$S_1 \circ [\tau/\alpha](\beta) = \begin{cases} S_1(\tau) = S_1(\alpha) & \text{if } \beta = \alpha, \\ S_1(\beta) & \text{otherwise.} \end{cases}$$

Now supppose $\sigma = \sigma_1 \to \sigma_2$ and $\tau = \tau_1 \to \tau_2$. Every unifier here is automatically a unifier for both the left and the right part of the implications, so if there is no unifier for σ_2, τ_2 (and by induction hypothesis $U(\sigma_2, \tau_2)$ returns `fail`), there is no unifier for σ, τ either. Otherwise by induction hypothesis S_2 is a most general unifier. Now if there is a unifier S' for σ, τ we have necessarily that $S' = S'_1 \circ S_2$, because S' is a unifier for σ_2, τ_2. So given S_2 it is justified to say that there exist a unifier for σ, τ if and only if there exist one for $S_2(\sigma_1), S_2(\tau_1)$. If moreover S_1 is a most general unifier for $S_2(\sigma_1), S_2(\tau_1)$ (and it is by induction hypothesis) and S' is a unifier for σ, τ, then we already know that $S' = S'_1 \circ S_2$, so that S'_1 is a unifier for $S_2(\sigma_1), S_2(\tau_1)$ and thus $S' = S'' \circ S_1 \circ S_2$: $S_1 \circ S_2$ is a most general unifier.

To extend U to $E = \{ \overrightarrow{\sigma = \tau} \}$ it suffices to define

$$U(E) := U(\sigma_1 \to \ldots \to \sigma_n, \tau_1 \to \ldots \to \tau_n)$$

as

$$S(\sigma_1 \to \ldots \to \sigma_n) = S(\tau_1 \to \ldots \to \tau_n) \iff \forall i : S(\sigma_i) = S(\tau_i).$$

Now for the computational cost: if we chose to represent the substitution in the form of the string $[\overrightarrow{\sigma/\alpha}]$, and accept that applying a substitution on a type is a polynomial operation in both the size of the type and in that of the string representing S, we see that the composition is

$$[\overrightarrow{\sigma/\alpha}] \circ [\overrightarrow{\tau/\beta}] = \left[\overrightarrow{\tau[\overrightarrow{\sigma/\alpha}] \,/\, \beta}, \overrightarrow{\sigma/\alpha}^*\right],$$

where $\overrightarrow{\sigma/\alpha}^*$ is $\overrightarrow{\sigma/\alpha}^* \setminus \{\sigma/\alpha \mid \alpha = \beta_i\}$, which is polynomial. Now by inspection of the algorithm we may see that the length of the string describing the substitutions involved is bounded by the size of the types, and that also the number of recursive calls of U is bounded by two times the size of the two terms. If we have a polynomial way of checking the condition $\alpha \notin \mathrm{TV}(\tau)$ (which can be taken for granted), we obtain the polynomiality. □

Now there is a way to build standard equations on types that permits to solve our problems.

Proposition 2.2.26. *There is a polynomially computable function E that accepts an environment A, a term M with $\mathrm{FV}(M) \subseteq \mathrm{DOM}(A)$ and a type τ and outputs a system of equation on type so that:*

- *if S is a unifier for $E(A, M, \tau)$ then $S(A) \vdash_{\mathbf{S}} M : S(\tau)$,*
- *if S is such that $S(A) \vdash_{\mathbf{S}} M : S_1(\tau)$ then there exist S_1 which unifies $E(A, M, \tau)$ and such that $S_1|_{\mathrm{TV}(A,\sigma)} = S|_{\mathrm{TV}(A,\sigma)}$.*

In particular $E(A; M; \sigma)$ has no unifier if and only if there is no S with $S(A) \vdash_{\mathbf{S}} M : S(\sigma)$.

Proof. Let E be defined by

Require: A, M with $\mathrm{FV}(M) \subseteq \mathrm{DOM}(A)$, σ;
1: **if** $M = x \in \mathcal{V}$ **then return** $\{A(x) = \sigma\}$;
2: **else**
3: **if** $M = M_1\, M_2$ **then**
4: choose α fresh;
5: $(E_1, E_2) \leftarrow (E(A, M_1, \alpha \to \sigma), E(A, M_2, \alpha))$;
6: in E_2 rename $\mathrm{TV}(E_2) \setminus (\mathrm{TV}(A) \cup \{\alpha\})$ to variables not in A, σ and E_1;
7: **return** $E_1 \cup E_2$;
8: **else**
9: **if** $M = \lambda x.M'$ **then**
10: choose α, β fresh;
11: $E' \leftarrow E(A \cup \{x : \alpha\}, M', \beta)$;
12: in E' rename $\mathrm{TV}(E') \setminus (\mathrm{TV}(A) \cup \{\alpha, \beta\})$ to variables not in A and σ;
13: **return** $E' \cup \{\sigma = \alpha \to \beta\}$;

The algorithm always terminates because each recursive call is made on a term of strictly smaller size.

The rest is seen by induction.

If $M = x$, and $S(A(x)) = S(\sigma)$ then clearly $S(A) \vdash_{\mathbf{S}} x : S(\sigma)$. If on the other hand $S(A) \vdash x : S(\sigma)$, then necessarily $S(A(x)) = S(\sigma)$, because the (var) rule must be applied, and so S is a unifier for $E(A, x, \sigma)$.

If $M = M_1\, M_2$ and S unifies $E(A, M_1, \alpha \to \sigma) \cup E(A, M_2, \alpha)$ then by induction hypothesis $S(A) \vdash_{\mathbf{S}} M_1 : S(\alpha) \to S(\sigma)$ and $S(A) \vdash_{\mathbf{S}} M_2 : S(\alpha)$. So applying (app) yields the result. On the converse if S is such that $S(A) \vdash_{\mathbf{S}} M : S(\sigma)$, then climbing up the derivation gives us

$$S(A) \vdash_{\mathbf{S}} M_1 : \rho \to S(\sigma), \quad S(A) \vdash_{\mathbf{S}} M_2 : \rho,$$

with ρ some type. Now take α, E_1 and E_2 as chosen by the algorithm, and define $S'(\alpha) = \rho$ and equal to S in other cases. α was fresh, so it does not appear in A and σ, and we have $S'(A) \vdash_{\mathbf{S}} M_1 : S'(\alpha \to \sigma)$ and $S'(A) \vdash_{\mathbf{S}} M_2 : S'(\alpha)$. By induction hypothesis we have S''_1 and S''_2 that both act like S' on A, σ and α (and thus like S on A and σ). Define

$$S''(\beta) := \begin{cases} S''_1(\beta) & \text{if } \beta \in \mathrm{TV}(E_1), \\ S''_2(\beta) & \text{if } \beta \in \mathrm{TV}(E_2), \\ S(\beta) & \text{otherwise.} \end{cases}$$

Because of the renaming internal to the algorithm we have that $\mathrm{TV}(E_1) \cap \mathrm{TV}(E_2) \subseteq \mathrm{TV}(A) \cup \{\alpha\}$, so that S''_1 and S''_2 yield the same values in such a set, and thus S'' is defined. Moreover $S''|_{\mathrm{TV}(A,\sigma)} = S|_{\mathrm{TV}(A,\sigma)}$. Finally by definition $S''|_{E_i} = S''_i|_{E_i}$, so it unifies both E_1 and E_2, and thus unifies E.

If $M = \lambda x.M'$ and S unifies

$$E(A \cup \{x : \alpha\}, M', \beta) \cup \{\sigma = \alpha \to \beta\}$$

then by induction hypothesis

$$\frac{S(A), x : S(\alpha) \vdash_{\mathbf{S}} M' : S(\beta)}{S(A) \vdash \lambda x.M' : S(\alpha \to \beta)} \text{ (abs)}$$

and then $S(\alpha \to \beta) = S(\sigma)$. On the converse suppose $S(A) \vdash_{\mathbf{S}} M : S(\sigma)$, and let's go up one rule to $S(A), x : \sigma_1 \vdash M' : \sigma_2$ where necessarily $S(\sigma) = \sigma_1 \to \sigma_2$. Take α and β as chosen by the algorithm and $E = E(A \cup \{x : \alpha\}, M', \beta)$. Chose S' such that it acts like S on $\mathrm{TV}(A)$ and σ while $S'(\alpha) = \sigma_1$ and $S'(\beta) = \sigma_2$. Then

$$S'(A, x : \alpha) \vdash_{\mathbf{S}} M' : S'(\beta)$$

and thus by induction hypothesis there is S'' that acts like S' on the set $\mathrm{TV}(A) \cup \{\alpha, \beta\}$ (and so it is like S on A) and is a unifier of E. Define

$$S'''(\gamma) := \begin{cases} S''(\beta) & \text{if } \beta \in \mathrm{TV}(E), \\ S(\beta) & \text{otherwise.} \end{cases}$$

Clearly $S'''(A) = S(A)$. As we have renamed variables so that $\mathrm{TV}(\sigma) \cap \mathrm{TV}(E) \subseteq \mathrm{TV}(A) \cup \{\alpha, \beta\}$ we have

$$S'''(\sigma) = S(\sigma) = \sigma_1 \to \sigma_2 = S'(\alpha \to \beta) = S'''(\alpha \to \beta)$$

so S''' is a unifier with the required properties for $E \cup \{\sigma = \alpha \to \beta\}$.

The polynomial bound is trivial, provided we have a polynomial way to choose fresh variables and rename variables in environments, which can be taken for granted. □

The equivalent of the most general unifier regarding a typing of a term is the so called *principal pair*.

Definition 2.2.27 (principal pair, principal type in system S). Let M be a term. We say (A, σ) is a *principal pair* for M if

- $A \vdash_{\mathbf{S}} M : \sigma$,
- if $A' \vdash_{\mathbf{S}} M : \sigma'$ then there exist S such that $S(A) \subseteq A'$ and $S(\sigma) = \sigma'$.

If M is closed we say that σ is a *principal type* for M if $(\emptyset, \sigma)$ is a principal pair for M. In other words if

- $\vdash_{\mathbf{S}} M : \sigma$,
- if $\vdash_{\mathbf{S}} M : \sigma'$ then there is S such that $S(\sigma) = \sigma'$.

Note that this notion of principal type is in line with the one given for type system in general. Note also that if (A, σ) is a principal pair for M then $\mathrm{DOM}(A) = \mathrm{FV}(M)$.

Now we may see that in fact there is a way to compute a principal pair.

Proposition 2.2.28. *There is a polynomially computable function pp that given a term M returns a principal pair for it if it is typable within some environment and return* `fail` *otherwise. Moreover there is a polynomially computable function pt that given a closed term M returns principal type for it if it is typable, and* `fail` *otherwise.*

Proof. Let *pp* be defined as follows.

Require: M;

1: choose $\vec{\alpha}^n, \beta$ distinct variables, where $n = |\mathrm{FV}(M)|$;
2: $A' \leftarrow \{ x_1 : \alpha_1, \ldots, x_n : \alpha_n \}$;
3: $E \leftarrow E(A', M, \beta)$;
4: $S \leftarrow U(E)$;
5: **if** $S =$ `fail` **then return** `fail`;
6: **else return** $(S(A'), S(\beta))$;

We see that *pp* returns `fail` if and only if U returns it, and it happens if and only if $E(A', M, \beta)$ has no unifier. We have seen in the above proposition this happens if and only if there is no S such that $S(A') \vdash_{\mathbf{S}} M : S(\beta)$. As all the types involved are distinct type variables, it is equivalent to saying that there are no types $\vec{\sigma}$ and τ such that $\overrightarrow{x : \sigma} \vdash_{\mathbf{S}} M : \tau$, i.e. M is not typable.

Now suppose $pp(M) = (A, \tau)$. We know that $A = S(A')$ and $\tau = S(\beta)$, and S is a most general unifier of $E = E(A', M, \beta)$. As S is a unifier for E, it follows that $S(A') \vdash M : S(\beta)$.

Now take B and σ such that $B \vdash_{\mathbf{S}} M : \sigma$. Take $S' = [\overrightarrow{B(x)/\alpha}, \sigma/\beta]$, so that $S'(A') \vdash_{\mathbf{S}} M : S'(\beta)$, where we have used that $B|_{\mathrm{FV}(M)} \vdash_{\mathbf{S}} M : \sigma$. So by the proposition above we know there is S'' such that $S''(A') = S'(A') \subseteq B$, $S''(\beta) = S'(\beta) = \sigma$ and S'' is a unifier for E. As S is most general, we have a substitution S''' so that $S'' = S''' \circ S$. So $S'''(A) \subseteq B$ and $S'''(\tau) = \sigma$, which was what needed.

pp is polynomial as both U and E are polynomial.

pt can be easily defined from *pp*: $pt(M) := \pi_2(pp(M))$. □

So we have already solved $\mathrm{TYP}_{\mathbf{S}}$.

What about $\mathrm{TC}_{\mathbf{S}}$? First we need a slightly modified version of the unification function.

Lemma 2.2.29. *There is a polynomially computable function U' that given two types σ and τ outputs S such that $\tau = S(\sigma)$ and $\mathrm{SUPP}(S) \subseteq \mathrm{TV}(\sigma)$, and if $S'(\sigma) = \tau$ then $S'|_{\mathrm{TV}(\sigma)} = S$, or* `fail` *if there is no substitution of σ into τ.*

U' is extendable to sets $\{\overrightarrow{(\sigma, \tau)}\}$.

Proof. Define U' with the following algorithm.

Require: τ and σ;

1: **if** $\tau = \sigma$ **then return** [ ];
2: **else**
3: **if** $\sigma \in \mathbb{V}$ **then return** $[\tau/\sigma]$;
4: **else**
5: **if** $\tau \in \mathbb{V}$ **then return** `fail`;
6: **else**
7: $S_2 \leftarrow U'(R(\sigma), R(\tau))$;
8: **if** $S_2 =$ `fail` **then return** `fail`;
9: **else**
10: $S_1 \leftarrow U'(L(\sigma), L(\tau))$;
11: **if** $S_1 =$ `fail` **then return** `fail`;
12: **else**
13: **if** $S_1 \neq S_2$ on $\mathrm{SUPP}(S_1) \cap \mathrm{SUPP}(S_2)$ **then return** `fail`;
14: **else return** $S_1 + S_2$;

where $S_1 + S_2$ means S_1 on $\mathrm{SUPP}(S_1)$ and S_2 otherwise (or viceversa).

By quick induction: if $\sigma = \alpha$ then clearly $S = [\tau/\alpha]$ is the right substitution, and its support is the strictly necessary (is contained in $\mathrm{TV}(\sigma)$).

If $\sigma = \sigma_1 \to \sigma_2$ and $\tau = \alpha$ then there is no way of substituting σ into τ. If on the other hand $\tau = \tau_1 \to \tau_2$, then $S(\sigma) = \tau$ if and only if $S(\sigma_i) = \tau_i$. If one of the two substitutions are not

possible then neither S is. If both are possible, then by induction hypothesis $S|_{\mathrm{TV}(\sigma_i)} = S_i$, where S_1, S_2 are the substitutions given by the algorithm. Moreover $\mathrm{SUPP}(S_i) \subseteq \mathrm{TV}(\sigma_i)$, and if there is $\beta \in \mathrm{SUPP}(S_1) \cap \mathrm{SUPP}(S_2)$ on which S_1 and S_2 discord then we would have a contradiction $S(\beta) \neq S(\beta)$. If they are consistent with each other then we may easily see that $(S_1 + S_2)(\sigma_1 \to \sigma_2) = \tau$, $\mathrm{SUPP}(S_1 + S_2) = \mathrm{SUPP}(S_1) \cup \mathrm{SUPP}(S_2) \subseteq \mathrm{TV}(\sigma)$ and if $S'(\sigma) = \tau$ we have already seen that $S'|_{\mathrm{SUPP}(S_1+S_2)} = S_1 + S_2$.

Polynomiality is carried out like in U.

Likewise the extension is

$$U'(\overrightarrow{(\sigma, \tau)}) := U'(\sigma_1 \to \ldots \to \sigma_n, \tau_1 \to \ldots \tau_n).$$

□

Proposition 2.2.30 ($\mathrm{TYP}_{\mathbf{S}}$ and $\mathrm{TC}_{\mathbf{S}}$). $\mathrm{TYP}_{\mathbf{S}}$ *and* $\mathrm{TC}_{\mathbf{S}}$ *are polynomially decidable.*

Proof. Given an instance $A \vdash M : \tau$ of $\mathrm{TC}_{\mathbf{S}}$ suppose $\mathrm{FV}(M) \subseteq \mathrm{DOM}(A)$ (otherwise we may already answear there is no typing), compute $pp(M)$. If it gives `fail` then surely there is no typing ending in our instance. So suppose $pp(M) = (B, \sigma)$. If $\vec{x} = \mathrm{FV}(M)$, we compute

$$U'(\{\, (B(x_1), A(x_1)), \ldots, (B(x_n), A(x_n)), (\sigma, \tau) \,\})$$

and check whether it gives `fail`.

If it does then there is no substitution S such that $S(B) \subseteq A$ and $S(\sigma) = \tau$, so by the properties of the principal pair $A \vdash M : \tau$ is not derivable. Otherwise A and τ can be obtained by B and σ with a substitution, and so indeed $A \vdash_{\mathbf{S}} M : \tau$. □

We will now give another version of the principal pair algorithm that does not rely on E and builds its own constraint on the way. This is a version more suitable for implementation. So let pp' be the following algorithm:

Require: M term;
  if $M = x$ **then return** $(\{x : \alpha\}, \alpha)$;
  else
    if $M = \lambda x.M'$ **then**
      $(A, \tau) \leftarrow pp'(M')$;
      if $(A, \tau) =$ `fail` **then return** `fail`;
      else
        if $x \in \mathrm{DOM}(A)$ **then return** $(A \setminus \{x : A(x)\}, A(x) \to \tau)$;
        else
          choose α fresh; **return** $(A, \alpha \to \tau)$;

else
if $M = (M_1\, M_2)$ **then**
$(A_1, \tau_1) \leftarrow pp'(M_1)$;
$(A_2, \tau_2) \leftarrow pp'(M_2)$;
if (A_1, τ_1) or (A_2, τ_2) is `fail` **then return** `fail`;
else
in (A_1, τ_1) rename variables so that $\mathrm{FTV}(A_1, \tau_1) \cap \mathrm{FTV}(A_2, \tau_2) = \emptyset$;
choose α fresh;
$S \leftarrow U(\tau_1, \tau_2 \to \alpha)$;
if $S =$ `fail` **then return** `fail`;
else return $(S(A_1 \cup A_2), S(\alpha))$;

Using the same techniques applied in the preceding algorithm we can prove its correctness and completeness.

2.3 System PCF: easier programming

System **PCF** is a system that fills in some characteristic system **S** lacks, by a simple principle: if we miss something, we add it. The core of the system is that we add the fix point combinator we did not have in **S**. Not only: we embed directly in the system naturals and booleans and some simple functions on them. We may regard it as something closer to real computer programming: we do not care about how the base functions and types are intrinsically made, we just use them as black boxes. So here we will not deal anymore with pure λ-terms, as we will extend them.

System **PCF** was introduced by Scott [Sco93] in 1969, in a manuscript published only in 1993, and was later developed by Plotkin in [Plo77]. This section is meant to be only a swift presentation of the topic, in order to give a more complete look at the scene of typing disciplines.

2.3.1 Definition and first properties

Definition 2.3.1 (terms of PCF). Terms of **PCF** are built from the set $\mathcal{V}$ of term variables extended with new constants, denoted by the set $\mathcal{L}$:

- `n` for each $n \in \mathbb{N}$,
- `true` and `false`,
- `Y`, the fixed point combinator,
- `if`, the if then else construct,
- `succ` and `pred`,

- `zero`?, the characteristic function of 0,

Note that the terms are not underlined to distinguish them from the usual representations within pure λ-calculus. The rules are the usual ones. Clearly abstraction can only be made with variables.

$$\Lambda_{\mathcal{L}} ::= \mathcal{L} \mid \mathcal{V} \mid (\Lambda_{\mathcal{L}}\ \Lambda_{\mathcal{L}}) \mid \lambda\mathcal{V}.\Lambda_{\mathcal{L}}.$$

Note that the above grammar depends only on $\mathcal{L}$. We will here deal with the definition given above, but in fact we can extend or restrict it depending on the particular choice of $\mathcal{L}$.

We define the usual notions of bounded and free variables and of substitution ignoring completely the constants. α-equivalent on constants is the equality, and is thus extended as usual.

Definition 2.3.2 (types of PCF). Types are constructed with the same rules of system **S**, but starting from a base set of only two types called *base types*: $\mathbb{V}_{\mathbf{PCF}} := \{o, \imath\}$, where o represents the boolean type, and $\imath$ represents the integer type. The set of types, still ranged over by letters such as τ or σ, is then defined as usual:

$$\mathbb{T}_{\mathbf{PCF}} ::= \mathbb{V}_{\mathbf{PCF}} \mid \mathbb{T}_{\mathbf{PCF}} \to \mathbb{T}_{\mathbf{PCF}}.$$

Definition 2.3.3 (rules of PCF). The rules of the system are the following ones:

$$\frac{}{A, x:\tau \vdash x:\tau}\ (\text{var}) \qquad \frac{A, x:\sigma \vdash M:\tau}{A \vdash (\lambda x.M):\sigma\to\tau}\ (\text{abs})$$

$$\frac{A \vdash M:\sigma\to\tau \quad B \vdash N:\sigma}{A, B \vdash (M\,N):\tau}\ (\text{app}) \qquad \frac{}{A \vdash \mathtt{Y}:(\sigma\to\sigma)\to\sigma}\ (\text{Y})$$

$$\frac{}{A \vdash \mathtt{true}:o}\ (\text{true}) \qquad \frac{}{A \vdash \mathtt{false}:o}\ (\text{false})$$

$$\frac{}{A \vdash \mathtt{if}:o\to o\to o\to o}\ (\text{bool if}) \qquad \frac{}{A \vdash \mathtt{if}:o\to\imath\to\imath\to\imath}\ (\text{nat if})$$

$$\frac{}{A \vdash \mathtt{n}:\imath}\ (\text{nat}) \qquad \frac{}{A \vdash \mathtt{succ}:\imath\to\imath}\ (\text{succ})$$

$$\frac{}{A \vdash \mathtt{pred}:\imath\to\imath}\ (\text{pred}) \qquad \frac{}{A \vdash \mathtt{zero?}:\imath\to o}\ (\text{zero})$$

So all the constants we have introduced gets assigned some type a-priori. Note that the if then else construct gets typed in two possible ways, but it strictly requires base types as input. On the converse the Y-combinator accepts all terms that can be typed as a function that brings a type to itself.

We say that a term is a *program* if it is closed and typable with base type. These in fact are the main objects **PCF** is concerned with. They can be seen as a particular instance of an algorithm, after it has been applied to its input and we expect it to return an integer or a boolean.

Now, having introduced special constant with special type assignment rules means that we must also give special rules for their reduction.

Definition 2.3.4 (operational reduction). *One step operational reduction* is the relation $\to_{op}$ given by the following rules:

$$(\lambda x.M)\,N \to_{op} M[N/x] \qquad \mathtt{Y}\ M \to_{op} M\,(\mathtt{Y}\ M)$$

$$\mathtt{if\ true}\ M_1\,M_2 \to_{op} M_1 \qquad \mathtt{if\ false}\ M_1\,M_2 \to_{op} M_2$$

$$\mathtt{succ\ n} \to_{op} \mathtt{n}+\mathtt{1} \qquad \mathtt{pred}(\mathtt{n}+\mathtt{1}) \to_{op} \mathtt{n}$$

$$\mathtt{zero?}\ \mathtt{0} \to_{op} \mathtt{true} \qquad \mathtt{zero?}\ \mathtt{n}+\mathtt{1} \to_{op} \mathtt{false}$$

$$M \to_{op} M' \implies (M\,N) \to_{op} (M'\,N)$$
$$N \to_{op} N' \implies (Z\,N) \to_{op} (Z\,N'), \quad \text{if } Z \in \{\,\mathtt{if}, \mathtt{succ}, \mathtt{pred}, \mathtt{zero?}\,\}.$$

Operational reduction is the transitive and reflexive closure of $\to_{op}$.

Note that in fact $\to_{op}$ is a deterministic strategy rather than a classic reduction: given a term there can be at most one step that can be done. The rules are designed so that before going on one tries to reduce the function rather than the argument. This is also done in order to "give a chance" to the fixed point combinator: once we reduce $\mathtt{Y}\ M$ to $M\,(\mathtt{Y}\ M)$ the operator duplicates again only if it goes back to head position.

All the static properties of typings in **S** still hold here. We may reread each of the proofs by substituting to the base types two distinct type variables α and β and replacing the constants with dummy variables being assigned the desired types in the environment. If we require for these dummy variables to be left unchanged the rest of the proofs carries on normally. Then we easily translate back.

Also subject reduction holds, by simply inspecting the new cases of reduction and confronting with the ad-hoc types designed for the constants.

We *evaluate* a program by applying to it operational reduction until we find a constant. But does it terminate? As the reduction being considered is deterministic there is no distinction between SN and WN. In any case, we may see that there is no universal normalization property, not even for programs. Consider the term $\mathtt{Y\ succ}$: it can be typed with $\imath$, but following the main strategy

leads to an ever increasing term. This should not surprise, that term should represent the fixed point of the successor function. And in any case, this language has been developed to study the abstract properties of programs, including eventually non-termination.

2.3.2 What do we get from PCF?

What is the expressive power of **PCF**? In fact the same of pure λ-calculus.

Example 2.3.5. First of all let us show how recursion is easily programmable here. We have already told in example 1.3.13 how it can be regarded as a fixed point. In fact if $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is defined by:

$$f(m, \vec{n}) = \begin{cases} g(\vec{n}) & \text{if } m = 0, \\ h(m-1, \vec{n}, f(m-1, \vec{n})) \end{cases}$$

then we can represent f by

$$\underline{f} := \mathtt{Y}\,(\lambda x \lambda m \overrightarrow{\lambda n}\ \mathtt{if}\,(\mathtt{zero?}\ m)\,(\underline{g}\,\vec{m})\,(\underline{h}\,(\mathtt{pred}\ m)\,\vec{n}\,(x\,(\mathtt{pred}\ m)\,\vec{n}))).$$

It can be shown that if $\underline{g}$ and $\underline{h}$ are of the appropriate types so is $\underline{f}$, if before abstraction we assign types $x : \iota^{k+1} \to \iota$, $m : \iota$, $n_i : \iota$ and we type $\mathtt{Y}$ with $(\iota^{k+1} \to \iota) \to (\iota^{k+1} \to \iota)$.

In fact there are some problems if we want to be sure that this is defined if and only if f is defined. We will put aside those problems by writing the minimization scheme instead. However recursion is a good tool for programming.

For example we may program `add` and `mult` as

$$\underline{\mathtt{add}} := \mathtt{Y}\,(\lambda f \lambda m \lambda n.\,\mathtt{if}\,(\mathtt{zero?}\ m)\,n\,(\mathtt{succ}\,(f\,(\mathtt{pred}\ m)\,n)))$$

$$\underline{\mathtt{mult}} := \mathtt{Y}\,(\lambda f \lambda m \lambda n.\,\mathtt{if}\,(\mathtt{zero?}\ m)\,0\,(\mathtt{add}\ n\,(f\,(\mathtt{pred}\ m)\,n)))$$

Also $\underline{\chi_{\leq}}$ can now easily be represented:

$$\underline{\chi_{\leq}} := \mathtt{Y}\,(\lambda f \lambda m \lambda n.\,\mathtt{if}\,(\mathtt{zero?}\ m)\ \mathtt{true}\ (\mathtt{if}\,(\mathtt{zero?}\ n)\ \mathtt{false}\,(f\,(\mathtt{pred}\ m)\,(\mathtt{pred}\ n))))$$

Theorem 2.3.6 (PCF represents recursive functions). *The functions on integers representable by* **PCF** *are those and only those recursive.*

Proof. With the example above we have all the base functions.

Let $\mathtt{check}(M, N)$ be the term $\mathtt{if}\,(\mathtt{zero?}\ M)\,N\,N$. It can be used to make sure we go on only if M terminates, regardless of M's value. We have to require that M is of type integer and N is of base type. We define

$$\mathtt{check}^n(\vec{M}^n, N) := \mathtt{check}(M_1, \mathtt{check}(M_2, \ldots, \mathtt{check}(M_n, N)\ldots));$$

it checks one at a time if every M_i terminates, and then evaluates N.

Composition of $g : \mathbb{N}^h \to \mathbb{N}$ and $\vec{f} : \mathbb{N}^k \to \mathbb{N}$ given $\underline{g}$ and $\underline{\vec{f}}$ can be represented by

$$\underline{g \circ \vec{f}} = \overrightarrow{\lambda n}.\, \mathtt{check}^h((\underline{f_1}\,\vec{n}), \ldots, (\underline{f_h}\,\vec{n}), \\ (\underline{g}\,(\underline{f_1}\,\vec{n}) \ldots (\underline{f_1}\,\vec{n}))).$$

It is built so that if any of the f are not defined (so that they do not terminate on their input) neither does the term, regardless of what g might do. The term is clearly well typed.

Minimization is easier. Given $f : \mathbb{N}^{k+1} \to \mathbb{N}$ we have

$$\underline{\mu f} = \overrightarrow{\lambda n}.\, \mathtt{Y}(\lambda g \lambda m.\, \mathtt{if}\,(\mathtt{zero?}(\underline{f}\,m\,\vec{n}))\,m\,(g\,(\mathtt{succ}\,m)))\,\mathtt{0}.$$

We may see that each value before the eventual result gets computed, and so minimization does not terminate when it does not have to.

The converse holds because we can map **PCF** into pure λ-calculus. □

PCF was introduced to make an abstract study of programs in a framework closer to the object of study. In particular much study has been done on the interpretation of **PCF** in *domains*; however one has to abandon the idea of a logical framework behind the language.

2.3.3 TC, TYP and type inference

We can easily adapt the algorithms described for **S** to **PCF**. Thus we will here make heavy references to subsection 2.2.5. First we extend $\mathbb{T}_{\mathbf{PCF}}$ to the set $\mathbb{T}^*_{\mathbf{PCF}}$ of types with variables in order to formulate type parameters:

$$\mathbb{T}^*_{\mathbf{PCF}} ::= \mathbb{V}_{\mathbf{PCF}} \mid \mathbb{V} \mid \mathbb{T}^*_{\mathbf{PCF}} \to \mathbb{T}^*_{\mathbf{PCF}}.$$

A substitution on $\mathbb{T}^*_{\mathbf{PCF}}$ has the usual meaning: it substitutes variables in $\mathbb{V}$, leaving the base types unaltered. We define unifiers for types and for sets of equations, principal pairs and principal types in the same manner. We may then temporarily work inside $\mathbf{PCF}^*$, the system obtained with identical rules but with types in $\mathbb{T}^*_{\mathbf{PCF}}$.

Proposition 2.3.7. *There are polynomially computable functions $U_{\mathbf{PCF}}$, $E_{\mathbf{PCF}}$, $pp_{\mathbf{PCF}}$, $U'_{\mathbf{PCF}}$ and $pt_{\mathbf{PCF}}$ that carry out in $\mathbf{PCF}^*$ the same operations of U, E, pp, and pt in **S**.*

Proof. Take the algorithms described in subsection 2.2.5.

To define $U_{\mathbf{PCF}}$, insert after line 2 the following ones:

if $\sigma \in \mathbb{V}_{\mathbf{PCF}}$ or $\tau \in \mathbb{V}_{\mathbf{PCF}}$ **then return** `fail`;

else

as if one of the two is a base type the other must be equal in order for the two to be unified, and this case is already solved in the preceding conditional clause. The rest of the algorithm works as usual.

For $E_{\mathbf{PCF}}$ there is more to add as we have to encode all the new type inference rules. In order to handle the `if` term we have to be able to add a new type of condition to the set of equations, namely $\tau \in \mathbb{V}_{\mathbf{PCF}}$: we will then say that S unifies E containing such condition if also $S(\tau) \in \mathbb{V}_{\mathbf{PCF}}$. Note that only variables and base types can be unified to satisfy such a condition. For each set of constants C with fixed type τ_C insert a line at the beginning of the algorithm of the form:

if $M \in C$ **then return** $\{\sigma = \tau_C\}$;
else

For `if` and `Y` insert the following lines:

if $M =$ `if` **then**
    choose α fresh;
    return $\{\, \sigma = o \to \alpha \to \alpha \to \alpha, \alpha \in \mathbb{V}_{\mathbf{PCF}} \,\}$;
else
    if $M =$ `Y` **then**
        choose α fresh;
        return $\{\sigma = (\alpha \to \alpha) \to \alpha\}$;
    else

The rest is carried out as usual. It is not hard to check that the new lines added complete he induction basis of the proof of correctness for this system.

$pp_{\mathbf{PCF}}$ and $pt_{\mathbf{PCF}}$ remain exactly the same, if we simply replace the appropriate functions with the ones depicted above.

$U'_{\mathbf{PCF}}$ is obtained adding exactly the same line we have added in $U_{\mathbf{PCF}}$. □

Note that the notion of principal pair and principal type is somewhat modified: they do not anymore valid typings in the system as it is, but we can obtain all the valid typings if we restrict the substitutions used to the ones that maps the type variables involved in types of $\mathbb{T}_{\mathbf{PCF}}$. In any case:

Theorem 2.3.8. $\mathrm{TC}_{\mathbf{PCF}}$, $\mathrm{TYP}_{\mathbf{PCF}}$, *and checking whether a term can be a program are all polynomially decidable.*

Proof. TYP is resolved by checking if pp returns `fail`. If it does not we can substitute to the type variable any valid **PCF** type and obtain a typing. TC is solved by applying pp and then U' like in **S**. The last one goes down to checking if the term is closed, and if it is applying pt and checking if the resulting type is not a $\to$-type. □

Chapter 3

Polymorphic λ-calculus

We have seen how **PCF** has much expressive power, but it loses, with respect to simply typed λ-calculus, the correspondence with logic. Such correspondace is recuperated with system **F**, also called *polymorphic λ-calculus*. In fact we are taking the intuitionistic logic framework that we noted is behind system **S** and we are adding to it the second-order quantification, i.e. we permit the *quantification of types*, so that a single type may represent multiple ones with a common form. We may regard it as the passage from simple type programming language such as Pascal or C to object-oriented programming languages such as Obj-C and Java. We are now capable of defining classes that represent many different types, from which we just require some common characteristics.

System **F** was introduced in 1971 by Jean-Yves Girard. A survey may be found in [GTL89]. We present a common variant to the system first introduced by Girard: by switching to Church style we do not define the Λ binder that takes types as input. Some minor changes are taken from Wells (see, for example, [Wel99]).

3.1 Definition and first properties

Definition 3.1.1 (types of F). The set $\mathbb{T}_{\mathbf{F}}$ is defined from the set of type variables $\mathbb{V}$ by the following grammar:

$$\mathbb{T}_{\mathbf{F}} ::= \mathbb{V} \mid \mathbb{T}_{\mathbf{F}} \to \mathbb{T}_{\mathbf{F}} \mid \forall \mathbb{V}.\mathbb{T}_{\mathbf{F}}.$$

Clearly $\mathbb{T}_{\mathbf{S}} \subseteq \mathbb{T}_{\mathbf{F}}$. We will call $\forall$-types (*quantified types*) the ones where the third rule was used last. Otherwise (variable or implication) we will call it a *non quantified type*. The type $\forall\alpha.\alpha$ may be denoted as $\bot$ (*bottom*)[1].

[1] in logic $\bot$ corresponds to incoherence, as all formulas can be derived from it. In fact there is no closed term of

Apart from the notations already mentioned for system **S** we also adopt the convention of omitting the dot between quantifiers and of abbreviating

$$\overrightarrow{\forall\alpha}^n.\tau := \forall\alpha_1 \ldots \forall\alpha_n.\tau,$$

eventually omitting n if we don't need to specify.

We may regard the quantifier $\forall$ the same way we see the λ in terms: it binds the variable, and can be instantiated (in this case without having to specify an input) with a type that has to be substituted for every (free) occurrence of the variable bounded by the quantifier itself. To do this we need the same machinery we have for λ-terms, and we define it in practically the same way.

Given a term τ we define, apart from the set of variables already mentioned for system **S** (whose definition we extend by simply ignoring the quantifiers), the set of *free type variables* and the set of *bounded type variables*:

$$\begin{aligned} \mathrm{FTV}(\alpha) &:= \{\alpha\}, & \mathrm{BTV}(\alpha) &:= \emptyset, \\ \mathrm{FTV}(\sigma \to \tau) &:= \mathrm{FTV}(\sigma) \cup \mathrm{FTV}(\tau), & \mathrm{BTV}(\sigma \to \tau) &:= \mathrm{BTV}(\sigma) \cup \mathrm{BTV}(\tau), \\ \mathrm{FTV}(\forall\alpha.\tau) &:= \mathrm{FTV}(\tau) \setminus \{\alpha\}, & \mathrm{BTV}(\forall\alpha.\tau) &:= \mathrm{BTV}(\tau) \cup \{\alpha\}. \end{aligned}$$

We extend the definitions above to sets of types by means of union, and also to type environments by applying them to the range of the environment. We will abbreviate by

$$\forall.\tau := \overrightarrow{\forall\alpha}.\tau$$

where $\vec{\alpha} = \mathrm{FTV}(\tau)$.

We can now define simple substitution $\sigma\langle\overrightarrow{\tau/\alpha}\rangle$, usual substitution $\sigma[\overrightarrow{\tau/\alpha}]$ and α-equivalence $\equiv_\alpha$ in the same way we did for λ-terms, using TV, FTV, BTV and $\forall$ in place of V, FV, BV and λ. We define substitutions S as we did for simple types, but we will see (lemma 3.1.7) when we are able to apply them to type derivations.

We will also use *renamings* ranged over by R, which are bijective substitutions of both type and term variables with other type and term variables. More precisely a renaming R is defined on terms (and contexts) as $R(M) = M[\vec{y}/\vec{x}]$ where $\vec{y}$ is $\vec{x}$ permuted, and on types as $R(\tau) = \tau[\vec{\beta}/\vec{\alpha}]$, where $\vec{\beta}$ is $\vec{\alpha}$. For commodity we let renamings act also on bounded variables, i.e. a renaming induces a specific α-conversion.

Subtypes are defined like subterms: we add to R and L a third function D defined only on $\forall$-types, which gives the type being quantified.

Moreover we introduce another equivalence relation that comes from the fact that we no longer require that the binder $\forall$ accepts input, so that order in quantifiers and unneeded binding of

type $\bot$.

variables are no longer needed taken into account. So define $\sim$ as the least equivalence relation such that

$$\begin{aligned} \sigma_1 \sim \tau_1, \quad \sigma_2 \sim \tau_2 &\implies \sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2. \\ \sigma \sim \tau &\implies \forall\alpha.\sigma \sim \forall\alpha.\tau, \\ \sigma \sim \tau, \quad \alpha \notin \mathrm{FTV}(\tau) &\implies \sigma \sim \forall\alpha.\tau, \\ \sigma \sim \tau &\implies \forall\alpha\forall\beta.\sigma \sim \forall\beta\forall\alpha.\tau. \end{aligned}$$

We now work with $\mathbb{T}_{\mathbf{F}}/(\equiv_\alpha, \sim)$ rather than $\mathbb{T}_{\mathbf{F}}$, though we will keep calling it the same way. Note that by pruning out unnecessary bindings we have also left out bindings on the same variable. As with λ-terms, we will freely rename bounded variables to ensure there are no repetitions both between bounded variables and free variable, and between bounded variables themselves. From now on when we write $\forall\alpha.\tau$ we implicitly take for granted that $\alpha \in \mathrm{FTV}(\tau)$. Note that this means that no (gen) rule can follow a (var) rule that introduces a non quantified type, as all the free type variables are also in the environment.

We define also a relation that regulates how types can be directly derived from each other using the quantification.

Definition 3.1.2 (direct containment relation). Let X be a type, a set of types or a type environment. Then we define

$$\tau \preceq \sigma \iff \tau = \overrightarrow{\forall\alpha}.\rho, \quad \sigma = \overrightarrow{\forall\gamma}.S(\rho)$$

where $\vec{\gamma} \cap \mathrm{FTV}(X) = \emptyset$ and S is a substitution such that

$$\mathrm{SUPP}(S) \subseteq \{\vec{\alpha}\} \cup (\mathrm{FTV}(\rho) \setminus \mathrm{FTV}(X)).$$

We define $\tau \preccurlyeq_X \sigma$ in the same manner, but with

$$\mathrm{SUPP}(S) \subseteq \{\vec{\alpha}\}.$$

Both relations are read τ is *directly contained* in σ with respect to X. If we need to specify, we sill read the second relation as *(ins) before (gen) direct containment*, for reasons we will see later. The relation should be understood as the fact that a term typable with τ in the environment A such that $\tau \preceq_A \sigma$ is typable also with σ within the same environment. We omit X if $X = \emptyset$.

The first relation can be checked to be transitive. The second one is not. For example $\alpha \to \beta \preccurlyeq \forall\alpha.\alpha \to \beta$ and $\forall\alpha\alpha \to \beta \preccurlyeq \gamma \to \beta$, but we should bring $\alpha \to \beta$ to $\gamma \to \beta$ with a substitution with $\mathrm{SUPP}(S) \subseteq \emptyset$.

As we have already said we have the inclusion $\mathbb{T}_{\mathbf{S}} \subseteq \mathbb{T}_{\mathbf{F}}$. On the converse we also have a function from $\mathbb{T}_{\mathbf{F}}$ to $\mathbb{T}_{\mathbf{S}}$ that erases quantifiers. It is defined by

$$(\alpha)_{\mathbf{S}} := \alpha,$$
$$(\sigma \to \tau)_{\mathbf{S}} := (\sigma)_{\mathbf{S}} \to (\tau)_{\mathbf{S}},$$
$$(\forall\alpha.\tau)_{\mathbf{S}} := (\tau)_{\mathbf{S}}.$$

We note that really it is not well defined over α-equivalence classes, but we will keep it that way, making it dependant on the representant we choose.

Definition 3.1.3 (rules of F). System **F** is given by the following rules that extend those for system **S**:

$$\frac{}{A, x : \tau \vdash x : \tau}\ (\text{var}) \qquad \frac{A, x : \sigma \vdash M : \tau}{A \vdash (\lambda x.M) : \sigma \to \tau}\ (\text{abs})$$

$$\frac{A \vdash M : \sigma \to \tau \quad B \vdash N : \sigma}{A, B \vdash (M\,N) : \tau}\ (\text{app}) \qquad \frac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\tau/\alpha]}\ (\text{ins})$$

$$\frac{A \vdash M : \tau}{A \vdash M : \forall\alpha.\tau}\ (\text{gen}) \quad \text{where } \alpha \notin \mathrm{FTV}(A).$$

The last two rules are called *instantiation* and *generalization*. We will abbreviate multiple consecutive uses of those rules by

$$\frac{A \vdash M : \overrightarrow{\forall\alpha.}\tau}{A \vdash M : \sigma[\overrightarrow{\tau/\alpha}]}\ (\text{ins}) \qquad \frac{A \vdash M : \tau}{A \vdash M : \overrightarrow{\forall\alpha.}\tau}\ (\text{gen})$$

where clearly in the latter we require $\vec{\alpha} \cap \mathrm{FTV}(A) = \emptyset$.

We may immediately see that **F** is not syntax-driven. We will call (ins) and (gen) rule *term-invariant* rules, as the term being typed does not change between premise and conclusion. Note that they even do not depend on the term M. There exist (and we will consider them later) syntax-driven variations that integrate the term-invariant rules in the other ones. It must be noted however that this does not make the type inference problem any easier, and we will see it later.

System **F** trivially enjoys properties like weakening, weakening on reverse (if $A \vdash_{\mathbf{F}} M : \tau$ then $A|_{\mathrm{FV}(M)} \vdash_{\mathbf{F}} M : \tau$) and substitution closure, as the induction proofs for system **S** are easily extended to the two new rules. The only thing one has to check is that in case of (gen) the environment eventually added does not contain free the bounded variable, but this is easily achieved renaming the bound variable.

Proposition 3.1.4 (subterm typing). **F** *enjoys subterm typing. In particular, if N is a subterm occurrence of M, every $\mathcal{D}$ such that $\mathcal{D} \rightsquigarrow A \vdash M : \tau$ contains a subderivation $\mathcal{D}' \rightsquigarrow A' \vdash N : \sigma$.*

Proof. Though not syntax-driven anymore, a derivation still reflects the construction tree of the term, though now decorated by new rules that do not change the term. Still an abstraction must be introduced by an (abs)-rule and an application by an (app) one. So the proposition follows easily from an induction on $\mathcal{D}$. □

So we can now define $\mathrm{IDT}(\mathcal{D}, N)$ (*initial derived type for N in $\mathcal{D}$*) as the type τ such that $A' \vdash N : \tau$ is the first sequent appearing in $\mathcal{D}$ regarding N (so it is the conclusion of a (var), (abs) or (app) generating the occurrence N). In the same way we define $\mathrm{FDT}(\mathcal{D}, N)$, the *final derived type*, where $A' \vdash N : \mathrm{FDT}(\mathcal{D}, N)$ is the premise of a (var), (abs) or (app), or the last sequent if the subterm being analyzed is M itself. So between the sequent corresponding to $\mathrm{IDT}(\mathcal{D}, N)$ and that corresponding to $\mathrm{FDT}(\mathcal{D}, N)$ there are only the term-invariant rules (gen) and (ins). In particular the environment does not change, so we define $\mathrm{DE}(\mathcal{D}, N)$ (*derived environment*) as this unique environment. In particular:

Proposition 3.1.5. *For every subterm occurrence of M, with $\mathcal{D} \rightsquigarrow A \vdash M : \tau$, let*

$$\mathcal{D}' \rightsquigarrow A' \vdash N : \sigma$$

be any subderivation corresponding to N. Then $\mathrm{IDT}(\mathcal{D}, N) \preceq_{A'} \mathrm{FDT}(\mathcal{D}, N)$.

Proof. By inspection of the effect of a chain of term-invariant rules on the type, where the environment A' does not change. (gen) brings from a type σ' to $\forall\alpha.\sigma'$ where $\alpha \notin \mathrm{FTV}(A')$, so $\sigma' \preceq_{A'} \forall\alpha.\sigma'$. (ins) brings $\forall\alpha.\sigma'$ to $\sigma'[\rho/\alpha]$, and by taking $S = [\rho/\alpha]$ we see that $\forall\alpha.\sigma' \preceq_{A'} S(\sigma')$. Then by transitivity follows the result. □

In order to have a derivation slightly better to manipulate we introduce a property regulating consecutive uses of (ins) and (gen).

Definition 3.1.6 ((ins) before (gen) property). A derivation $\mathcal{D}$ is said to have the *(ins) before (gen) property* if every (gen) rule is never immediately followed by an (ins) rule.

Lemma 3.1.7 (type substitution). *Let $\mathcal{D} \rightsquigarrow A \vdash M : \tau$, as usual with the convention that there are no clashes between bounded and free variable. If $\vec{\alpha}$ are variables that either occur free in $A \vdash M : \tau$ or do not occur at all, then $\mathcal{D}[\overrightarrow{\tau/\alpha}]$ is a valid derivation, where the substitution $[\overrightarrow{\tau/\alpha}]$ is made on every sequent in the derivation. Note that the new derivation has the same structure, i.e. the rules used are the same and in the same order.*

Said in other words, if $A \vdash M : \sigma$ is derivable, so is $A[\overrightarrow{\tau/\alpha}] \vdash M : \sigma[\overrightarrow{\tau/\alpha}]$, using a derivation with the same structure.

Proof. The renaming convention here is essential, as otherwise one could substitute a complex type for a variable yet to be bounded, making the (gen) impossible to apply. Choosing a variable appearing free at the end guarantees that it is not bounded by any (gen) in the derivation, so all rules get preserved. We speak also about variables not occurring at all for induction's sake. The idea is that we are substituting at the source of each variable, where it was introduced: (var) or eventually (ins). So by induction on $\mathcal{D}$:

var: Trivial: $A'[\overrightarrow{\sigma/\alpha}], x : \tau[\overrightarrow{\sigma/\alpha}] \vdash_{\mathbf{F}} x : \tau[\overrightarrow{\sigma/\alpha}]$.

abs: $M = \lambda x.M'$, $\tau = \tau_1 \to \tau_2$ and we have the subderivation

$$\mathcal{D}' \rightsquigarrow A, x : \tau_1 \vdash M' : \tau_2.$$

$\vec{\alpha}$ must still appear free at the end of $\mathcal{D}'$ if they appear at all, so by induction hypothesis $\mathcal{D}'[\overrightarrow{\sigma/\alpha}]$ is a valid derivation and so applying back (abs) gives the result.

app: As before, apart from the fact that in the case one of the two subderivations does not contain one (or more) of the $\vec{\alpha}$ free in the conclusion then by the renaming convention it does not contain it anywhere else above. So we can still apply induction hypothesis.

ins: $\tau = \tau'[\rho/\beta]$ and there is a subderivation $\mathcal{D}' \rightsquigarrow A \vdash M : \forall\beta.\tau'$. Note that $\beta \neq \alpha_i$ for every i. If some α_is do not appear free at the end of $\mathcal{D}'$, we still can apply the induction hypothesis:

$$\frac{\mathcal{D}'[\overrightarrow{\sigma/\alpha}] \rightsquigarrow A[\overrightarrow{\sigma/\alpha}] \vdash M : \forall\beta.\tau'[\overrightarrow{\sigma/\alpha}]}{A[\overrightarrow{\sigma/\alpha}] \vdash M : \tau'[\overrightarrow{\sigma/\alpha}]\big[\rho[\overrightarrow{\sigma/\alpha}]/\beta\big]}\ (\text{ins})$$

and the last type is indeed $\tau[\overrightarrow{\sigma/\alpha}]$.

gen: $\tau = \forall\beta.\tau'$ and there is a subderivation $\mathcal{D}' \rightsquigarrow A \vdash M : \tau'$. Note that as $\beta \notin \mathrm{FTV}(A)$ necessarily $\beta \neq \alpha_i$ for all i, and moreover by renaming we may take β so that it does not appear in σ. By induction hypothesis $\mathcal{D}'[\overrightarrow{\sigma/\alpha}]$ is valid, β is still not free in $A[\overrightarrow{\sigma/\alpha}]$, and so applying back (gen) gives the sequent

$$A[\overrightarrow{\sigma/\alpha}] \vdash M : \forall\beta.\tau'[\overrightarrow{\sigma/\alpha}],$$

and as $\beta \notin \mathrm{FTV}(\vec{\sigma})$ we have that the last type is $\tau[\overrightarrow{\sigma/\alpha}]$.

□

Proposition 3.1.8 ((ins) before (gen)). *For every derivation*

$$\mathcal{D} \rightsquigarrow A \vdash M : \tau$$

there exist a derivation $\mathcal{D}'$ of the same sequent which enjoys the (ins) before (gen) property. The size of $\mathcal{D}'$ (number of rules) is less than the size of $\mathcal{D}$, and equal only if $\mathcal{D}$ already enjoys the property. Moreover $\mathcal{D}'$ inherits its structure from $\mathcal{D}$, in the sense that the rules used are the same and in the same order, apart from some (gen) and (ins) in $\mathcal{D}$ which disappear in the new derivation.

Proof. Suppose there is a (gen)-(ins) sequence somewhere in $\mathcal{D}$. We take the subderivation immediately preceding it, say

$$\dfrac{\dfrac{\tilde{\mathcal{D}} \rightsquigarrow A \vdash M : \tau}{A \vdash M : \forall\alpha.\tau}\ (\text{gen})}{A \vdash M : \tau[\rho/\alpha]}\ (\text{ins})$$

where $\alpha \notin \mathrm{FTV}(A)$.

α must appear free in τ because of the convention with $\sim$. We then apply the lemma above to $\tilde{\mathcal{D}}$ giving a valid derivation

$$\tilde{\mathcal{D}}[\rho/\alpha] \rightsquigarrow A \vdash M : \tau[\rho/\alpha]$$

and so we can completely substitute the subderivation depicted above with $\tilde{\mathcal{D}}[\rho/\alpha]$, which has the same structure of $\tilde{\mathcal{D}}$, so we are erasing the last two rules.

If the new derivation still does not have the (ins) before (gen) property we restart and go on. As every step reduces the size of the derivation by two we must in the end arrive to an (ins) before (gen) form. We can even show that this one step "reduction" is confluent[2], and so there is only one (ins) before (gen) form of a given derivation. □

Remark 3.1.9. (gen)-(ins) elimination corresponds to reducing all universal redexes in Church-style system **F**.

Proposition 3.1.10. *If $\mathcal{D} \rightsquigarrow A \vdash M : \tau$ is an (ins) before (gen) derivation and N is a subterm with a subderivation $\mathcal{D}' \rightsquigarrow A' \vdash N : \sigma$ then* $\mathrm{IDT}(\mathcal{D}, N) \preccurlyeq_{A'} \mathrm{FDT}(\mathcal{D}, N)$.

Proof. In an (ins) before (gen) derivation a chain of term-invariant rules is always a chain of (ins) followed by a chain of (gen). So if $\vec{\alpha}$ are all the variables getting instantiated by the (ins) rules we have $\mathrm{IDT}(\mathcal{D}, N) = \overrightarrow{\forall\alpha}.\sigma'$, then after the (ins) rules we have $\sigma'[\overrightarrow{\rho/\alpha}]$ and then after the (gen) rules

$$\mathrm{FDT}(\mathcal{D}, N) = \overrightarrow{\forall\gamma}.\sigma'[\overrightarrow{\rho/\alpha}]$$

with $\gamma \cap \mathrm{FTV}(A') = \emptyset$, and $\mathrm{SUPP}([\overrightarrow{\rho/\alpha}]) \subseteq \{\vec{\alpha}\}$. □

Corollary 3.1.11. *If $\mathcal{D} \rightsquigarrow A \vdash M : \tau$ is an (ins) before (gen) derivation then*

- *if $M = x$ then $A(x) \preccurlyeq_A \tau$;*

[2] again the renaming convention is essential, as it gives commutativity to the substitution operation in this case.

- *if* $M = \lambda x.M'$ *then there is a subderivation* $\mathcal{D}'$ *ending with an (abs) in* $A \vdash M : \sigma \to \rho$ *with* $\tau = \overrightarrow{\forall\alpha}.\sigma \to \rho$, $\vec{\alpha} \cap \mathrm{FTV}(A) = \emptyset$.
- *if* $M = (M_1\, M_2)$ *then there is a subderivation* $\mathcal{D}'$ *ending with an (app) in* $A \vdash M : \tau'$ *with* $\tau' \preccurlyeq \tau$.

Proof. The application of the proposition is straightforward. Note that in the case of abstraction there cannot be any (ins) rules because there is no $\forall$ to instantiate. □

Inherited from system **S** we have also subject reduction.

Theorem 3.1.12 (subject reduction). F *enjoys subject reduction.*

Proof. The proof extends the one given in system **S** (2.2.9). The cases for (var) and (abs) are exactly identical. Also the case for (app)-rule is almost identical to the one in system **S**. The only difference is the case in which $M = (M_1\, M_2)$ is itself the redex reduced, as it is not anymore sure that the rule immediately preceding (app) on the left is (abs). However by applying the above corollary to the subderivation ending with M_1, and noting how there cannot be any quantifiers on the type or else (app) would be impossible to apply, we see the rule preceding (app) on the left must be (abs). So the proof proceeds as in system **S** (we use substitution lemma).

As for the term-invariant rules, they pose no problem: the term in the premise is the same, so applying the induction hypothesis is smooth, and as those rules do not depend on the term being typed we can easily apply them back. □

3.2 What do we get from F?

One of the answears to this question could be: self-application, without renouncing to strong normalization.

First of all, system **F** gives a nice (and modular) way to represent so called *free structures*. We will somewhat restrict the definitions for sake of clarity. For a more general survey see [GTL89, Chapter 11].

3.2.1 Representation of free structures

Definition 3.2.1 (free structures). A free structure Θ is a set of formal expressions generated by some finite symbols $f_1, \ldots, f_k$ called *constructors* and by elements in some sets $U_1, \ldots, U_h$, following rules specified for every constructor. By this we mean that every constructor has a *type* written $f_i : S_i$ with

$$S_i = T_1^i \to T_2^i \to \ldots \to T_{n_i}^i \to \Theta$$

where every T^i_j is either Θ or one of the U_is[3]. We require that at least one of the constructors f_i is an *atom*, by which we mean its type is such that T^i_j is not Θ for all the js. So if the constructors are defined as above we have the grammar defining Θ as

$$\Theta ::= f_1(T^1_1, \ldots, T^1_{n_1}) \mid \cdots \mid f_k(T^k_1, \ldots, T^k_{n_k}).$$

A *function defined by induction* on Θ is a function $g : \Theta \to X$ determined by k functions (or eventually constants) $g_i : S_i[X/\Theta]$ with the relation

$$g(f_i(\overrightarrow{t^i})) := g_i(\overrightarrow{x^i}),$$

where $x^i_j = g(t^i_j)$ if $T^i_j = \Theta$ and $x^i_j = t^i_j$ otherwise.

Example 3.2.2. $\mathbb{N}$ is (or we may say can be represented by) a free structure: it does not depend on any external set and it has two constructors, the atom $0 : \mathbb{N}$ and the successor $\texttt{succ} : \mathbb{N} \to \mathbb{N}$. Every integer is $\texttt{succ}^n(0)$.

$\mathbb{B}$ (and in general any finite set) is a free structures with only two atoms `true` and `false`.

$U \times V$ is a free structure with the atom $\langle \,.\,, \,.\, \rangle : U \to V \to U \times V$.

U^* is a free structure based on the set U: it has two constructors, the atom ε and the append function $\cdot : U \to U^* \to U^*$.

As a particular case $\{0,1\}^*$ is a free structure built from three constructors, the atom ε and the two successors functions $\texttt{succ}_i : \{0,1\}^* \to \{0,1\}^*$ which append i to the string.

The set U^{b} of binary trees with nodes in U is a free structure with two constructors: the atom $\mathrm{leaf} : U \to \Theta$ which gives a single node equal to the argument, and $\texttt{branch} : U \to U^{\mathrm{b}} \to U^{\mathrm{b}} \to U^{\mathrm{b}}$, so that $\texttt{branch}(u, x, y)$ means the tree with u as root and x and y as the two trees branching from u.

Λ itself, before quotienting it with $\equiv_\alpha$, is a free structure based on the set $\mathcal{V}$: its constructors are the variable $\texttt{var} : \mathcal{V} \to \Lambda$, the abstraction $\lambda : \mathcal{V} \to \Lambda \to \Lambda$ and the application $\cdot : \Lambda \to \Lambda \to \Lambda$.

Now we will represent free structures with a type. Given that $\vec{U}$ are representend each by a type ν_j (we will denote it by $\nu_j = \underline{U_j}$), and denoting by $\sigma_i := S_i[\overrightarrow{\nu/U}, \alpha/\Theta]$ with the obvious meaning, and choosing α "fresh", we are now ready to use as a representation of Θ the type

$$\underline{\Theta} := \forall\alpha.\sigma_1 \to \ldots \to \sigma_k \to \alpha.$$

Note that we are choosing a particular order of the constructors.

[3] this is where we restrict more than is really necessary, with respect to [GTL89], where T^i_j is allowed to be built with the usual implication rule from Θ and the U_is, provided Θ appears *positively*.

Now in order to represent a given element of the free structure what we do is we chose distinct variables $y_1, \ldots, y_k$ to represent each a constructor. What actual variables we choose is unimportant, as later they will all be bounded. They act as marks to show which constructor is used where. Now we get to work in the environment $\{y_i : \sigma_i\}$, and we simply translate every constructor $f_i(\overrightarrow{t^i})$ in y_i applied to the translations of the t^i_js. So, supposing the translation function is already defined on elements $u \in U_i$ giving closed terms $\underline{u}$ typable with ν_i, we define by induction $\mathtt{trans} : \Theta \to \Lambda$:

$$\mathtt{trans}(f_i(\overrightarrow{t^i})) := (y_i \overrightarrow{\mathtt{trans}(t^i)}),$$

where we let `trans` work as the known translation on elements in one of the U_is.

It is easy to show that given any $\theta \in \Theta$:

$$\overrightarrow{y : \sigma} \vdash_{\mathbf{F}} \mathtt{trans}(\theta) : \alpha$$

.

We finally define $\underline{\theta} := \overrightarrow{\lambda y}.\mathtt{trans}(\theta)$. Applying consecutive (abs) rules and a final (gen) on the typing depicted above we get

$$\vdash_{\mathbf{F}} \underline{\theta} : \underline{\Theta}.$$

Also it is now easy to represent the constructors as functions: we take the necessary arguments, we then recreate the abstracted variables that have to go at the beginning of the term and then eventually pass them to the arguments in Θ so that they are inherited by the subterms. So:

$$\underline{f_i} := \overrightarrow{\lambda t^i}.\overrightarrow{\lambda y}.(y_i\, \vec{P}^{n_i}),$$

where P_j is $(t^i_j\, \vec{y})$ if $T^i_j = \Theta$ and t^i_j itself otherwise.

Suppose we now want to represent the function $g : \Theta \to W$, defined by induction, with a (closed) term typable with type $\underline{\Theta} \to \underline{W}$: we suppose we already know how to represent elements of W, and how to represent the functions g_i defining g, so that we have terms $\underline{g_i}$ typable with $\sigma_i[\underline{W}/\alpha]$. In order to get $\underline{g(\theta)}$ all we have to do is just feed the $\underline{g_i}$s as input to $\underline{\theta}$, as every g_i will go in the variable marking the position of its corresponding constructor and will be thus applied to the arguments of the constructor which will be meanwhile treated in the same manner. So

$$\underline{g} := \lambda t.(t\, \underline{g_1} \ldots \underline{g_k}).$$

Eventually some simplifications may be carried out on the term given above. In any case it is always typable with $\underline{\Theta} \to \underline{W}$: the key fact is that we can instantiate the quantifier of $\underline{\Theta}$ in $t : \underline{\Theta} \vdash t : \underline{\Theta}$ to give

$$t : \underline{\Theta} \vdash_{\mathbf{F}} t : \sigma_1[\underline{W}/\alpha] \to \ldots \to \sigma_k[\underline{W}/\alpha] \to \underline{W}.$$

Let us see some examples to clarify.

Example 3.2.3. We may define $\sigma \times \tau$ using the definition of $U \times V$ as a free structure with its single constructor, so that $\sigma \times \tau := \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha$, and

$$\langle M, N\rangle = \lambda z.(z\, M\, N).$$

The constructor seen as a function is represented by $\lambda x\lambda y.\lambda z.(z\, x\, y)$, while the functions defined by induction are none other than the passage from a function defined on two inputs to one defined over a pair. In particular from $\underline{\pi_i^2} = \lambda x_1\lambda x_2.x_i$ comes

$$\underline{\pi_i} = \lambda c.(c\, \underline{\pi_i^2}).$$

From a logical point of view we are defining the disjunction $\wedge$ using $\forall$ and $\to$. In fact all the connectives and constants can be represented by solely these two formula constructors. We present here how to represent other logical connectives and constants and what kind of data they represent in λ-calculus.

$\sigma + \tau$ (the counterpart of $\vee$) is made up of two constructors, the two *injections*, left and right. So we represent it by $\forall\alpha.(\sigma \to \alpha) \to (\tau \to \alpha) \to \alpha$. So

$$\mathtt{injl}\, M = \lambda x\lambda y.x\, M, \quad \mathtt{injr}\, N = \lambda x\lambda y.y\, N.$$

A function defined by induction is practically a conditional clause on the effective content of a term of type $\sigma + \tau$.

$\exists\alpha.\tau$ is defined by $\forall\beta(\forall\alpha.\tau \to \beta) \to \beta$ (this is a little astray of our definition of free structure). If a term M is typable with $\tau[\sigma/\alpha]$ then $\lambda x.x\, M$ is typable with $\exists\alpha.\tau$ if we assume type $\forall\alpha.\tau \to \beta$ for x and then instantiate with σ. Induction here means having a function defined so that it can take arguments of type $\tau[\sigma/\alpha]$ for any σ.

$\bot$ is defined by having no constructors: $\bot = \forall\alpha.\alpha$. Clearly there is no term of type $\bot$.

Example 3.2.4. $\mathbb{N}$ is given by the constructors `succ` and 0. If we take them in this order we thus have $S_1 = \mathbb{N} \to \mathbb{N}$ and $S_2 = \mathbb{N}$. So

$$\underline{\mathbb{N}} = \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha = \forall\alpha.\, \mathtt{Int}_\alpha =: \mathtt{Int}\,.$$

So, choosing f for `succ` and x for 0 we have:

$$\mathtt{trans}(\mathtt{succ}^n(0)) = (f\ \mathtt{trans}(\mathtt{succ}^{n-1}(0))) = \cdots = (f^n\ \mathtt{trans}(0)) = (f^n\, x),$$

and $\underline{n} = \lambda f\lambda x.(f^n\, x)$, which is the classic representation. The representation of the constructor 0 is just $\underline{0}$ (it is not a function), while taking `succ` gives

$$\underline{\mathtt{succ}} = \lambda n.\lambda f\lambda x.(f\, (n\, f\, x))$$

which is one of the two representations we already gave. A function $g : \mathbb{N} \to W$ defined by induction on $\mathbb{N}$ is (in the sense given for free structures) given by $g_1 : W \to W$ and $g_2 \in W$, so that

$$g(0) := g_2$$
$$g(\mathtt{succ}(n)) := g_1(g(n)).$$

So it is an iteration of g_1 on g_2: $g(n) = g_1^n(g_2)$. In fact

$$(\lambda n.(n\,\underline{g_1}\,\underline{g_2}))\,\underline{n} \overset{\beta}{\twoheadrightarrow} (\underline{g_1}^n\,\underline{g_2}).$$

As we have seen before for system **S** (remark 2.2.11), also this representation in system **F** is such that every closed normal term typable with $\mathtt{Int}$ is a Church integer (apart from the ambiguity between I and $\underline{1}$). The proof is begun by noting that at the end of the derivation the only possibility is a (gen) that has premise of type $\mathtt{Int}_\alpha$. From then on the proof is identical to that for system **S**, with some minor changes. As for system **S** we may take the other representation of integers: it corresponds to reversing the order of the two constructors, so that there is no ambiguity about $\underline{1}$.

Here the result is much more meaningful than in system **S**: we will see that the class of the functions representable in system **F** is enormous; here we are saying that any given term typable with type $\mathtt{Int}^k \to \mathtt{Int}$ is actually a representation of a certain function. We are saying that the type $\mathtt{Int}^k \to \mathtt{Int}$ represents completely that enormous class of functions. In system **S** we couldn't go beyond polynomial functions...

So we have iteration. What about recursion? Given an element $u \in U$ and a function $h : U \times \mathbb{N} \to U$ we want to represent $\mathrm{REC}(u,h) : \mathbb{N} \to U$ defined by

$$\mathrm{REC}(u,h)(0) = u,$$
$$\mathrm{REC}(u,h)(\mathtt{succ}(n)) = h(\mathrm{REC}(u,h)(n), n).$$

Let us first define a function $\mathrm{REC}'(h) : U \times \mathbb{N} \to U \times \mathbb{N}$ by

$$\mathrm{REC}'(h)(v,n) := (h(v,n), n+1).$$

We have transformed a recursion into an iteration, as $\mathrm{REC}(u,h)(n)$ is the first component of $(\mathrm{REC}'(h))^n(u,0)$. Now, $\mathrm{REC}'(h)$ is represented by

$$\underline{\mathrm{REC}'_h} = \lambda p.\langle \underline{h}\,(p\,\mathtt{true})\,(p\,\mathtt{false}), \mathtt{succ}\,(p\,\mathtt{false})\rangle,$$

typable with type $\underline{U} \times \mathtt{Int} \to \underline{U} \times \mathtt{Int}$. Now we can use the argument of the recursion as an iterator, and then project the first component of the result:

$$\underline{\mathrm{REC}_{u,h}} := \lambda n.(n\,\underline{\mathrm{REC}'_h}\,\langle \underline{u}, \underline{0}\rangle\ \mathtt{false}),$$

and it can have type $\texttt{Int} \to \underline{U}$. More in general if M is typable with type τ and H with type $\tau \to \texttt{Int} \to \tau$ we have a term $\underline{\mathrm{REC}}_{M,H}$ of type $\texttt{Int} \to \tau$ such that

$$\underline{\mathrm{REC}}_{M,H}\,\underline{0} \equiv_\beta M, \qquad \underline{\mathrm{REC}}_{M,H}\,\underline{n+1} \equiv_\beta H\,(\underline{\mathrm{REC}}_{M,H}\,\underline{n})\,\underline{n}.$$

Though cumbersome, this construct can be used to formulate the predecessor function:

$$\underline{\texttt{pred}} = \mathrm{REC}_{\underline{0},\underline{\texttt{false}}},$$

so that

$$\underline{\texttt{pred}}\,\underline{0} \equiv_\beta \underline{0}, \qquad \underline{\texttt{pred}}\,\underline{n+1} \equiv_\beta \underline{\texttt{false}}\,(\underline{\texttt{pred}}\,\underline{n})\,\underline{n} \xrightarrow{\beta} \underline{n}.$$

Example 3.2.5. We can say the same things said above for

$$\underline{\mathbb{B}} = \forall\alpha.\,\texttt{Bool}_\alpha =: \texttt{Bool}\,.$$

The representations correspond to the classic ones and functions defined by induction is the IF...THEN...ELSE construct. By the way, if we consider a finite set as a free structure, we get as representants the projections $\underline{\pi_i^k}$.

Example 3.2.6. We here introduce the type linked to the structure $\{0,1\}^*$ calling it `BInt`. In fact it can be used to encode the integers, given we deal with the trailing zeros. So, given that we have three constructors, the empty string and the two successors, we have that the type is

$$\texttt{BInt} := \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha \to \alpha.$$

A string $(b_k, \ldots, b_0)$ is represented by

$$\underline{(b_k, \ldots, b_0)} = \lambda f_0 \lambda f_1 \lambda x.f_{b_0}(f_{b_1} \ldots (f_{b_k}\,x)\ldots).$$

The functions representing the two active constructors are

$$\underline{\texttt{succ}_i} = \lambda s \lambda f_0 \lambda f_1 \lambda x.(f_0\,(s\,f_0\,f_1\,x)).$$

If we choose to use this type to represent integers we put $\underline{0} := \underline{\varepsilon}$, and $\underline{n} := \underline{(b_k, \ldots, b_0)}$ if $n = (b_k, \ldots, b_0)_2$ in base two. We will see when we will extensively use this representation (chapter 4) how to deal with trailing zeros. The two successors represent the operation $\texttt{succ}_i(n) = 2n + i$.

Example 3.2.7. Let us take Λ, and suppose we represent $\mathcal{V}$ with `Int` using an enumeration of $\mathcal{V}$: the corresponding type is

$$\underline{\Lambda} = \forall\alpha.(\texttt{Int} \to \alpha) \to (\texttt{Int} \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha.$$

We chose as dummy names for the three constructors v, ℓ and a respectively. Then:

$$\begin{aligned}\texttt{trans}(x_n) &= v\,\underline{n},\\ \texttt{trans}(\lambda x_n.M) &= \ell\,\underline{n}\ \texttt{trans}(M),\\ \texttt{trans}(M\,N) &= a\ \texttt{trans}(M)\ \texttt{trans}(N).\end{aligned}$$

So for example:

$$\underline{\lambda x_0.x_0\,x_0} = \lambda v\lambda\ell\lambda a.(\ell\,\underline{0}\,(a\,(v\,\underline{0})\,(v\,\underline{0}))).$$

The constructors as functions are

$$\begin{aligned}\underline{\texttt{var}} &= \lambda x.\lambda v\lambda\ell\lambda a.(v\,x),\\ \underline{\lambda} &= \lambda x\lambda m.\lambda v\lambda\ell\lambda a.(\ell\,x\,(m\,v\,\ell\,a)),\\ \underline{\cdot} &= \lambda m\lambda n.\lambda v\lambda\ell\lambda a.(a\,(m\,v\,\ell\,a)\,(n\,v\,\ell\,a)).\end{aligned}$$

As a *divertissement* we may see that the representation of the length function is

$$\underline{\texttt{length}} = \lambda m.(m\,(\lambda d.\underline{1})\,(\lambda d.\underline{\texttt{succ}})\,(\lambda x\lambda y.\underline{\texttt{succ}}\,(\underline{\texttt{add}}\,x\,y))),$$

or even more complex functions such as simple substitution

$$\underline{\texttt{ssubs}} : \underline{\Lambda} \to \underline{\Lambda} \to \texttt{Int} \to \underline{\Lambda},$$

with $\texttt{ssubs}(M,N,n) = M\langle N/x_n\rangle$, where we use pairs to keep track of the term in case an abstraction aborts the substitution:

$$\begin{aligned}\underline{\texttt{ssubs}} = \lambda m\lambda n\lambda x.(m\\ &(\lambda y.\langle\underline{\texttt{var}}\,y, (\underline{\chi_=}\,y\,x)\,n\,(\underline{\texttt{var}}\,y)\rangle)\\ &(\lambda y\lambda p.\langle\underline{\lambda}\,y\,(p\,\underline{\texttt{true}}), (\underline{\chi_=}\,y\,x)\,(\underline{\lambda}\,y\,(p\,\underline{\texttt{true}}))\,(\underline{\lambda}\,y\,(p\,\underline{\texttt{false}}))\rangle)\\ &(\lambda p\lambda q.\langle\underline{\cdot}\,(p\ \texttt{true})\,(q\ \texttt{true}), \underline{\cdot}\,(p\ \texttt{false})\,(q\ \texttt{false})\rangle)\\ &\underline{\texttt{false}}).\end{aligned}$$

Where $\underline{\chi_=}$ represents the characteristic function of equality between integers. We can simplify the above term with

$$\begin{aligned}\underline{\texttt{ssubs}} = \lambda m\lambda n\lambda x.\lambda v\lambda\ell\lambda a.(m\\ &(\lambda y.\langle v\,y, (\underline{\chi_=}\,y\,x)\,(n\,v\,\ell\,a)\,(v\,y)\rangle)\\ &(\lambda y\lambda p.\langle\ell\,y\,(p\,\underline{\texttt{true}}), (\underline{\chi_=}\,y\,x)\,(\ell\,y\,(p\,\underline{\texttt{true}}))\,(\ell\,y\,(p\,\underline{\texttt{false}}))\rangle)\\ &(\lambda p\lambda q.\langle a\,(p\ \texttt{true})\,(q\ \texttt{true}), a\,(p\ \texttt{false})\,(q\ \texttt{false})\rangle)\\ &\underline{\texttt{false}}).\end{aligned}$$

3.2.2 Strong normalization

Now we will show that though we have dramatically increased expressiveness, we are still dealing with a strongly normalizing system.

The ideal would be to define reducible terms as we did for system **S**. Fact is, for a quantified type we should have to define something like $(A, M) \in \mathrm{RED}_{\forall\alpha.\tau}$ if and only if for every type σ we have $(A, M) \in \mathrm{RED}_\tau[\sigma/\alpha]$. We clearly can't do that, as the complexity of $\tau[\sigma/\alpha]$ is higher than that of $\forall\alpha.\tau$ and so recursiveness fails. So, as we are not able to define sets of reducible terms of a certain type we take all the sets that *could* be a set of reducible term. From here comes the definition of reducibility candidate we have seen in system **S** (2.2.14). Note that the definition does not use in any way the type it is designed upon, if not by simply stating that all environment-term pairs (A, M) are such that A induces the given type on M. We will use letters as R_τ, S_τ as meta-variables for reducibility candidates for type τ, omitting the type if it is unimportant, and we will denote by $\mathfrak{R}$ the set of all candidates of reducibility, where each retains information on the type it was designed for. $\mathfrak{R}_\tau$ will denote the reducibility candidates for type τ.

Given two candidates R_τ and S_σ we define $R_\tau \to S_\sigma$ made by pairs (A, M) such that $A \vdash_{\mathbf{F}} M : \tau \to \sigma$ by the same relation with which we defined $\mathrm{RED}_{\tau\to\sigma}$ for system **S**:

$$(A, M) \in R_\tau \to S_\sigma \iff \forall (B, N) \in R_\tau : ((A, B), (M\,N)) \in S_\sigma.$$

As usual we mean any (B, N) such that A and B are compatible.

Using the proof that $\mathrm{RED}_{\tau\to\sigma}$ is a reducibility candidate in system **S** we see that $R \to S$ is actually still a candidate of reducibility.

Now in order to capture the quantifier one *interprets* types with reducibility candidates. Let e range over the functions $e : \mathbb{V} \to \mathfrak{R}$ (no assumption on the types of the reducibility candidates). Let us denote by τ_e the result of substituting all free variables in τ with the corresponding type assigned by e, in the following sense: if $\vec{\beta} = \mathrm{FTV}(\tau)$ and for all i we have $e(\beta_i) \in \mathfrak{R}_{\sigma_i}$ then

$$\tau_e := \tau[\overrightarrow{\sigma/\beta}].$$

Then we extend this definition to A_e by applying it on the images of A. We denote by $e[R/\alpha]$ the function on variables defined by

$$e[R/\alpha](\beta) := \begin{cases} R & \text{if } \beta = \alpha, \\ S & \text{otherwise.} \end{cases}$$

Now we consistently extend any e to a function on all types so that $e(\tau)$ is a candidate of reducibility for τ_e. The definition is recursive, and exploits the fact that we may consider $e[S/\alpha]$ defined every time e already is. Clearly for variables the claim that the type is α_e is satisfied. Note that in order

to show the definition is well defined on $\equiv_\alpha$ and $\sim$ equivalence classes we are implicitly showing by following the induction that if $\alpha \notin \mathrm{FTV}(\tau)$ then $e[S/\alpha](\tau) = e(\tau)$, that $e[S/\alpha](\tau) = e[S/\beta](\tau[\beta/\alpha])$, and also that if $\alpha \neq \beta$ then $e[R/\alpha][S/\beta](\tau) = e[S/\beta][R/\alpha](\tau)$.

$\tau = \tau' \to \tau''$**:** $e(\tau' \to \tau'') := e(\tau') \to e(\tau'')$, and the type is $\tau'_e \to \tau''_e = \tau_e$.

$\tau = \forall\alpha.\tau'$**:** We define $e(\forall\alpha.\tau')$ as the set of all (A, M) with $A \vdash_{\mathbf{F}} M : (\forall\alpha.\tau')_e$ such that for all $S \in \mathcal{R}$ we have $(A, M) \in e[S/\alpha](\tau')$. The type is the right one by definition. We may see this case as stating

$$e(\forall\alpha.\tau') = \bigcap_{S\in\mathcal{R}} e[S/\alpha](\tau'),$$

given the condition on type, that by lemma 3.2.10 we will see we may drop.

In order to prove by induction that $e(\tau)$ is a reducibility candidate the only case remaining to be shown is the quantifier. Let us quickly check the properties. Note that the induction hypothesis is valid *for every* e. The trick is that as sets, if we chose some $S \in \mathcal{R}$, we have $e(\forall\alpha.\tau') \subseteq e[S/\alpha](\tau')$, and so by induction hypothesis on τ' all the properties (which depend on M and not on the type) trivially hold.

A function e built in this manner is called an *interpretation*.

Now we see that this definition behaves well with substitution. We use the fact that if $\alpha \notin \mathrm{FTV}(\tau)$ then $e[S/\alpha](\tau) = e(\tau)$.

Lemma 3.2.8.

$$e(\tau[\sigma/\alpha]) = e[e(\sigma)/\alpha](\tau).$$

Proof. By easy induction:

$\tau = \beta$**:** Trivial for the definition of $e[R/\alpha]$ on variables.

$\tau = \tau' \to \tau''$**:**

$$e((\tau' \to \tau'')[\sigma/\alpha]) = e[e(\sigma)/\alpha](\tau') \to e[e(\sigma)/\alpha](\tau'') = e[e(\sigma)/\alpha](\tau).$$

$\tau = \forall\beta.\tau'$**:** If $\beta \neq \alpha$ then substitution goes to τ' after doing the necessary renaming to avoid variable capture, so that also for any $\mathbf{S}$ we have $e[S/\beta](\sigma) = e(\sigma)$, and thus

$$\begin{aligned} e(\forall\beta.\tau'[\sigma/\alpha]) = \bigcap_{S\in\mathcal{R}} e[S/\beta](\tau'[\sigma/\alpha]) = \\ = \bigcap_{S\in\mathcal{R}} e[S/\beta][e[S/\beta](\sigma)/\alpha](\tau') = \bigcap_{S\in\mathcal{R}} e[e(\sigma)/\alpha][S/\beta](\tau') = \\ = e[e(\sigma)/\alpha](\forall\beta.\tau'). \end{aligned}$$

If on the other hand $\alpha = \beta$ nothing happens both on the left and on the right of the equality.

□

We recall the result shown for system **S** (2.2.17), the proof is identical.

Lemma 3.2.9 (abstraction). *Given e, A and M, if for every $(B, N) \in e(\tau_1)$ such that A and B are compatible we have $(A \cup B, M[N/x]) \in e(\tau_2)$ then $(A, \lambda x.M) \in e(\tau_1 \to \tau_2)$.*

We now bring down the new cases brought by universal quantification.

Lemma 3.2.10 (generalization). *If $(A, M) \in e[S/\alpha](\tau)$ for every $S \in \mathcal{R}$ then $(A, M) \in e(\forall\alpha.\tau)$.*

Proof. There is indeed something to prove: we have to show that $A_e \vdash_{\mathbf{F}} M : (\forall\alpha.\tau)_e$. We just take **S** a reducibility candidate for type β where β does not appear free in A_e. This gives $A_e \vdash_{\mathbf{F}} M : \tau_{e[S/\alpha]}$ and

$$\tau_{e[S/\alpha]} = \tau[\beta/\alpha, \overrightarrow{\sigma/\beta}],$$

where $\vec{\beta} = \mathrm{FTV}(\tau)\backslash\{\alpha\} = \mathrm{FTV}(\forall\alpha.\tau)$. As β does not appear free in A we can apply generalization and obtain

$$A \vdash_{\mathbf{F}} \forall\beta.\tau[\beta/\alpha, \overrightarrow{\sigma/\beta}].$$

Clearly the above type is $(\forall\alpha.\tau)_e$. □

Lemma 3.2.11 (instantiation). *If $(A, M) \in e(\forall\alpha.\tau)$ then for any ρ:*

$$(A, M) \in e(\tau[\rho/\alpha]).$$

Proof. By hypothesis $(A, M) \in e[S/\alpha](\tau)$ for any candidate **S**. If we choose $S = e(\rho)$ and apply lemma 3.2.8 we get $(A, M) \in e[e(\rho)/\alpha](\tau) = e(\tau[\rho/\alpha])$. □

Now the main result is ready. We say a term M is *reducible* of type τ if there is an environment A for which $(A, M) \in sn(\tau)$, where sn is the interpretation defined by

$$sn(\alpha) = \{\, (B, N) \mid B \vdash_{\mathbf{F}} N : \alpha, \quad N \in SN \,\}$$

which is clearly a candidate of reducibility for α, moreover for type α itself.

Theorem 3.2.12. *Let e by* any *interpretation. Let (A, M) be such that $\mathcal{D} \rightsquigarrow A \vdash_{\mathbf{S}} M : \tau$, $\mathrm{FV}(M) \subseteq \vec{x}^n$ and $\vec{N}^n$ are terms such that $\forall i : (B, N_i) \in e(A(x_i))$ with B compatible with A_e. Then $(A_e \cup B, M[\overrightarrow{N/x}]) \in e(\tau)$.*

Proof. First of all we may see that applying lemma 3.1.7 we can consider the derivation

$$\mathcal{D}_e \rightsquigarrow A_e \vdash M : \tau_e$$

obtained substituting all the free variable present only at the end with the corresponding types given by e. Then we reason by induction on $\mathcal{D}$. We may see that practically also in system **S** the induction was on the derivation, though in that case induction on the term was equivalent. The proofs given there for (var), (app) and (abs) following from the lemma specific to abstraction still hold here. So let us see the last two cases.

In case (gen) was the last rule used, so that $\tau = \forall\alpha.\tau'$, and we have a subderivation $\mathcal{D}' \rightsquigarrow A \vdash M : \tau'$ where α does not appear free in A. We take any $S \in \mathcal{R}$ and apply induction hypothesis using $e[S/\alpha]$ as interpretation, and clearly $A_{e[S/\alpha]} = A_e$. So

$$(A_e \cup B, M[\overrightarrow{N/x}]) \in e[S/\alpha](\tau')$$

and (having chosen **S** arbitrary) by lemma 3.2.10 we get the desired result.

If (ins) was the last rule instead we have $\tau = \tau'[\rho/\alpha]$ and a subderivation $\mathcal{D}' \rightsquigarrow A \vdash M : \forall\alpha.\tau'$. Then by inducing hypothesis $(A_e \cup B, M[\overrightarrow{N/x}]) \in e(\forall\alpha.\tau')$, and so by lemma 3.2.11 we get $(A, M) \in e(\tau'[\rho/\alpha])$. □

Corollary 3.2.13. *Every term typable in system* **F** *is reducible and thus* SN.

Proof. We choose $e = sn$ so that $\tau_e = \tau$ for any τ. Then chose $\vec{N} = \vec{x}$ and $B = A$. By applying the theorem we obtain $(A, M) \in sn(\tau)$, and in particular by the first property of reducibility candidates $M \in SN$. □

3.3 Functions representable in F

Now on to what is the expressive power of the system. The aim of this section is to give a quick sketch of the proof that closed terms typable with $\texttt{Int}^k \to \texttt{Int}$ are all those representing functions provably total in PA_2, where PA_2 is the theory of Peano arithmetics with second order quantification. This is by no means intended to be exhaustive on the topic.

We will content ourselves with showing the result for terms of type $\texttt{Int} \to \texttt{Int}$. The total result then follows easily with an encoding of $\mathbb{N}^k$ by $\mathbb{N}$.

We say that a computable function is *provably total* in a system for arithmetic T if T proves that some program representing f terminates on every input. How all this is formalized depends on the theory and how we set out to encode programs and inputs. For example for PA_2 we can take a λ-term that represents f (possible, as we have already seen), encode λ-terms as integers, and then write in (primitive and thus quantifier free) formulas the operation of reduction. Then we can create a primitive formula $P(e, n, m, p)$ that holds true if and only if the term coded by e applied on input $\underline{n}$ and then processed with a computation (we may see it as a list of reductions

to be done) coded by p gives the term $\underline{m}$. In all other cases (such as integers not corresponding to any coding, or a computation which does not correspond to a possible one, or a term which is not the normal $\underline{m}$) it is false. So we can express "the program coded by e applied on input $\underline{n}$ terminates with output $\underline{m}$" with the formula $T_1(e,n,m) = \exists p.P(e,n,m,p)$. Then we can express "the program coded by e terminates with a valid output for every input" with the formula

$$\forall n.\exists m.\exists p.P(e,n,m,p).$$

Using an encoding of $\mathbb{N}\times\mathbb{N}$ into $\mathbb{N}$ we can merge the two existentials, obtaining a formula in the Π^0_2 logical complexity class[4].

If M is closed and typable with $\mathtt{Int}^k \to \mathtt{Int}$ we may define a function $f_M : \mathbb{N}^k \to \mathbb{N}$ by $f_M(\vec{n}) = m$ if and only if $(M\,\vec{\underline{n}}) \equiv_\beta \underline{m}$, and undefined if $(M\,\underline{n})$ has a normal form which is not a Church numeral or worse if it does not have a normal form. Strong normalization theorem however tells us that that $(M\,\underline{n})$ will always have a normal form, and because this normal form is typable with $\mathtt{Int}$ it will be necessarily a Church integer. So indeed f_M is provably total, but the proof as it is is outside PA_2. In fact strong normalization theorem implies the consistency of PA_2, while we know by Gödel's second incompleteness theorem that PA_2 cannot prove its own consistency, so apart by direct inspection of the specific proof we are sure that any strong normalization proof has to be outside PA_2.

However the need to go beyond PA_2 is due to proving for *all* typable terms that they are reducible, while here we are interested in just the numerals $\underline{n}$ (which are immediately shown to be always reducible, as they are normal) and M alone. So we need only induction on the reducibility predicates for the types involved in the typing of M, and then by comprehension scheme and second order quantification (which are all principles contained in PA_2) we get normalization for M and so totality for f_M. The proof thus obtained depends on M, but at least we are inside PA_2.

The converse is practically due to the Curry-Howard isomorphism. The idea is to take a proof of the totality of the function and translate it into the typing of a term that in fact computes the value.

First of all, we must have a way to express our proof of totality in an intuitionistic framework, or else it will be impossible to translate it into a typing. So we will work in HA_2 (Heyting arithmetic with second order), and we will not loose anything because HA_2 is as strong as PA_2 in proving totality of funcitons.

[4] Π^0_k is defined to be the set of primitive formulas preceded by k alternating quantifiers beginning with $\forall$: $\forall x_1.\exists x_2.\ldots.\Diamond x_k.P$. Σ^0_k is defined the same way but the first quantifier is $\exists$ in this case. The definition is up to equivalence, so every formula is in a complexity class, and $\Pi^0_k \cup \Sigma^0_k \subseteq \Pi^0_{k+1} \cup \Sigma^0_{k+1}$.

3.3.1 HA_2

We will briefly outline HA_2.

Definition 3.3.1 (formulas of HA_2). The set $\mathcal{F}$ of formulas in HA_2 will be ranged over by letters such as F, G and are defined on two sets, one of variables for integers $\mathcal{V}_{HA_2} = \{\xi, \eta, \zeta, \dots\}$ and one of variables for sets, for which we will deliberately use $\mathbb{V}$ already chosen for type variables. *Terms* $\mathcal{T}_{HA_2}$ are defined by

$$\mathcal{T}_{HA_2} ::= 0 \mid \mathcal{V}_{HA_2} \mid \mathtt{succ}\,\mathcal{T}_{HA_2}$$

and are ranged over by letters such as a, b. The grammar defining $\mathcal{F}$ is

$$\mathcal{F} ::= \mathcal{T}_{HA_2} \in \mathbb{V} \mid \mathcal{T}_{HA_2} = \mathcal{T}_{HA_2} \mid \mathcal{F} \Rightarrow \mathcal{F} \mid \forall \mathcal{V}_{HA_2}.\mathcal{F} \mid \exists \mathcal{V}_{HA_2}.\mathcal{F} \mid \forall \mathbb{V}.\mathcal{F}.$$

One may note there are some logic symbols "missing": it is due to the fact that $\wedge$, $\vee$, $\bot$, $\exists\alpha$ and $\neg$ can be completely simulated by using $\Rightarrow$, $\forall\xi$ and $\forall\alpha$. Also $\exists\xi$ can be simulated, but we chose to retain it in the base formulas. So we define:

$$\begin{aligned}
F \wedge G &:= \forall\alpha\forall\xi.(F \Rightarrow G \Rightarrow \xi \in \alpha) \Rightarrow \xi \in \alpha,\\
F \vee G &:= \forall\alpha\forall\xi.(F \Rightarrow \xi \in \alpha) \Rightarrow (G \Rightarrow \xi \in \alpha) \Rightarrow \xi \in \alpha,\\
\bot &:= \forall\alpha\forall\xi.\xi \in \alpha,\\
\exists\alpha.F &:= \forall\beta\forall\eta.(\forall\alpha.(F \Rightarrow \eta \in \beta)) \Rightarrow \eta \in \beta,\\
\neg F &:= F \Rightarrow \bot.
\end{aligned}$$

Sequents are expressions of the form $F_1, \dots, F_n \vdash G$. We will denote by letters such as Γ, Δ multisets of formulas on the left of $\vdash$.

Definition 3.3.2 (rules of HA_2). HA_2 is defined by the following rules, apart from those regulating equality:

axioms:

$$\frac{}{F \vdash F}\ (\mathrm{ax}) \qquad \frac{}{\neg\,\mathtt{succ}\,\xi = 0}\ (\mathrm{a1})$$

$$\frac{}{\mathtt{succ}\,\xi = \mathtt{succ}\,\eta \Rightarrow \xi = \eta}\ (\mathrm{a2})$$

introductions:

$$\frac{F, \dots, F, \Gamma \vdash G}{\Gamma \vdash F \Rightarrow G}\ (\vdash\Rightarrow) \qquad \frac{\Gamma \vdash F[a/\xi]}{\Gamma \vdash \exists\xi.F}\ (\vdash \exists^1)$$

$$\frac{\Gamma \vdash F}{\Gamma \vdash \forall\xi.F}\ (\vdash \forall^1)(*) \qquad \frac{\Gamma \vdash F}{\Gamma \vdash \forall\alpha.F}\ (\vdash \forall^2)(*)$$

eliminations:

$$\frac{\Gamma \vdash F \quad \Delta \vdash F \Rightarrow G}{\Gamma, \Delta \vdash G}\ (\Rightarrow\vdash) \qquad \frac{\Gamma \vdash \exists\xi.F \quad F, \ldots, F, \Delta \vdash G}{\Gamma, \Delta \vdash G}\ (\exists^1\vdash)$$

$$\frac{\Gamma \vdash \forall\xi.F}{\Gamma \vdash F[a/\xi]}\ (\forall^1\vdash) \qquad \frac{\Gamma \vdash \forall\alpha.F}{\Gamma \vdash F[\{\xi.G\}/\alpha]}\ (\forall^2\vdash)$$

The $(*)$ means we are applying the usual condition that the bound variable is not free on the left of $\vdash$. The notation $F[\{\xi.G\}/\alpha]$ means that we are substituting (avoiding as always variable capture) $G[a/\xi]$ wherever we find $a \in \alpha$.

We call a set of occurrences of a formula that get deleted together in $(\vdash\Rightarrow)$ or in $(\exists^1\vdash)$ *parcels of hypotheses.*

We may give derived rules for the other derived symbols. Each of these rule is in fact a combination of those already given, and represents them faithfully, in the sense that the combination of base rules they hide is the unique way in which one can obtain those symbols. In fact we are purely interested in the rules regarding regarding $\wedge$:

$$\frac{\Gamma \vdash F \quad \Delta \vdash G}{\Gamma, \Delta \vdash F \wedge G}\ (\vdash \wedge)$$

$$\frac{\Gamma \vdash F \wedge G}{\Gamma \vdash F}\ (\wedge 1\vdash) \qquad \frac{\Gamma \vdash F \wedge G}{\Gamma \vdash G}\ (\wedge 2\vdash)$$

We may say that comprehension scheme and induction principle are somewhat already present. For the first one we may derive

$$\forall\alpha\exists\beta\forall\xi.(\xi \in \alpha \Leftrightarrow \xi \in \beta)$$

and then apply $(\forall^2\vdash)$ to obtain

$$\exists\beta\forall\xi.(C \Leftrightarrow \xi \in \beta).$$

The second is obtained just *defining* integers by the fact that they respect the induction principle. So we define

$$\mathtt{Nat}(\xi) := \forall\alpha.(0 \in \alpha \Rightarrow \forall\eta.(\eta \in \alpha \Rightarrow \mathtt{succ}\,\eta \in \alpha) \Rightarrow \xi \in \alpha).$$

Then it is easy to check that for any formula F we have:

$$\big(F[0/\xi] \wedge \forall\eta.(\mathtt{Nat}(\eta) \Rightarrow F[\eta/\xi] \Rightarrow F[\eta/\xi])\big) \Rightarrow \forall\eta.(\mathtt{Nat}(\eta) \Rightarrow F[\eta/\xi]).$$

So induction holds provided we relativize universal quantifiers to naturals.

3.3.2 Translation into F

We first translate every formula F of HA_2 into a type $[\![F]\!]$ of system **F**. So:

- $[\![a = b]\!] := \sigma$ with σ any type with at least one closed term typable with it, for example $\vdash_{\mathbf{F}} I : \forall\alpha.\alpha \to \alpha$;
- $[\![a \in \alpha]\!] := \alpha$;
- $[\![F \Rightarrow G]\!] := [\![F]\!] \to [\![G]\!]$;
- $[\![\forall\xi.F]\!] := [\![\exists\xi.F]\!] := [\![F]\!]$;
- $[\![\forall\alpha.F]\!] := \forall\alpha.\,[\![F]\!]$.

One may easily see then that $[\![F \wedge G]\!] = [\![F]\!] \times [\![G]\!]$ and $[\![\bot]\!] = \bot$.
Now we translate a proof π of HA_2 with conclusion $F_1, \ldots, F_n \vdash G$ into a type derivation

$$\mathcal{D}_\pi \rightsquigarrow x_1 : [\![F_1]\!], \ldots, x_n : [\![F_n]\!] \vdash_{\mathbf{F}} [\![\pi]\!] : [\![G]\!]\,.$$

We denote the final environment with $[\![\Gamma]\!]$ if Γ was the final multiset on the left of $\vdash$. Before going on let us note that there is a problem with the axiom $\neg\,\mathtt{succ}\,\xi = 0$, which is defined as $\mathtt{succ}\,\xi = 0 \Rightarrow \bot$: we should produce a term M such that $\vdash_{\mathbf{F}} M : \sigma \to \bot$, but no such term exists. A possible solution is temporarily extend system **F** with a "junk" term Ω and a "junk" rule $A \vdash_{\mathbf{F}} \Omega : \bot$. We will deal with this term will later. So, let us define the translation by induction.

axioms:

$$\frac{}{F \vdash F}\ (\text{ax}) \quad\longmapsto\quad \frac{}{x : [\![F]\!] \vdash x : [\![F]\!]}\ (\text{var})$$

with the convention that each (ax) introduces a different variable. Identifications will eventually be done afterwards.

$$\frac{}{\neg\,\mathtt{succ}\,\xi = 0}\ (\text{a1}) \quad\longmapsto\quad \frac{\dfrac{}{x : \sigma \vdash \Omega : \bot}}{\vdash \lambda x.\Omega : \sigma \to \bot}\ (\text{abs})$$

$$\frac{}{\mathtt{succ}\,\xi = \mathtt{succ}\,\eta \Rightarrow \xi = \eta}\ (\text{a2}) \quad\longmapsto\quad \frac{\dfrac{}{x : \sigma \vdash x : \sigma}\ (\text{var})}{\vdash \lambda x.x : \sigma \to \sigma}\ (\text{abs})$$

⇒:

$$\frac{\begin{array}{c}\pi\\ \vdots\\ F, \ldots, F, \Gamma \vdash G\end{array}}{\Gamma \vdash F \Rightarrow G}\ (\vdash\Rightarrow) \quad\longmapsto\quad \frac{\begin{array}{c}\mathcal{D}_\pi[x/\vec{y}]\\ \vdots\\ x : [\![F]\!], [\![\Gamma]\!] \vdash [\![\pi]\!]\,[x/\vec{y}]\,[\![G]\!]\end{array}}{[\![\Gamma]\!] \vdash \lambda x.\,[\![\pi]\!] : [\![F]\!] \to [\![G]\!]}\ (\text{abs})$$

where $\vec{y}$ are the variables corresponding to the parcel of hypotheses being deleted. If eventually $\vec{y}$ is empty we add the assumption $x : [\![F]\!]$ by weakening.

$$\dfrac{\begin{array}{cc}\pi_1 & \pi_2\\ \vdots & \vdots\\ \Gamma \vdash F & \Delta \vdash F \Rightarrow G\end{array}}{\Gamma, \Delta \vdash G}\ (\Rightarrow\vdash) \quad \longmapsto$$

$$\longmapsto \quad \dfrac{\begin{array}{cc}\mathcal{D}_{\pi_1} & \mathcal{D}_{\pi_2}\\ \vdots & \vdots\\ [\![\Gamma]\!] \vdash [\![\pi_1]\!] : [\![F]\!] & [\![\Delta]\!] \vdash [\![\pi_2]\!] : [\![F]\!] \to [\![G]\!]\end{array}}{[\![\Gamma]\!], [\![\Delta]\!] \vdash ([\![\pi_1]\!]\ [\![\pi_2]\!]) : [\![G]\!]}\ (\text{app})$$

Note that the environment are necessarily disjoint and thus compatible.

$\forall^1$ and $\exists^1$: $(\vdash \forall^1)$, $(\vdash \exists^1)$ and $(\forall^1 \vdash)$ does not get any translation, as interpretation of the first order quantifiers on types do not change. As for $(\exists^1 \vdash)$ suppose we have

$$\dfrac{\begin{array}{cc}\pi_1 & \pi_2\\ \vdots & \vdots\\ \Gamma \vdash \exists \xi.F & F, \ldots, F, \Delta \vdash G\end{array}}{\Gamma, \Delta \vdash G}\ (\exists^1 \vdash)$$

then we obtain

$$\mathcal{D}_{\pi_1} \rightsquigarrow [\![\Gamma]\!] \vdash [\![\pi_1]\!] : [\![F]\!],$$
$$\mathcal{D}_{\pi_2}[x/\vec{y}] \rightsquigarrow x : [\![F]\!], [\![\Delta]\!] \vdash [\![\pi_2]\!][x/\vec{y}] : [\![G]\!].$$

Then by substitution lemma we get

$$\mathcal{D}_{\pi} \rightsquigarrow [\![\Gamma]\!], [\![\Delta]\!] \vdash [\![\pi_2]\!]\left[[\![\pi_1]\!]/\vec{y}\right] : [\![G]\!].$$

$\forall^2$:

$$\dfrac{\begin{array}{c}\pi\\ \vdots\\ \Gamma \vdash F\end{array}}{\Gamma \vdash \forall \alpha.F}\ (\vdash \forall^2) \quad \longmapsto \quad \dfrac{\begin{array}{c}\mathcal{D}_{\pi}\\ \vdots\\ [\![\Gamma]\!] \vdash [\![\pi]\!] : [\![F]\!]\end{array}}{[\![\Gamma]\!] \vdash [\![\pi]\!] : \forall \alpha.\, [\![F]\!]}\ (\text{gen})$$

Clearly the condition on free variables is satisfied.

$$\dfrac{\begin{array}{c}\pi\\ \vdots\\ \Gamma \vdash \forall \alpha.F\end{array}}{\Gamma \vdash F[\{\xi.G\}/\alpha]}\ (\forall^2 \vdash) \quad \longmapsto \quad \dfrac{\begin{array}{c}\mathcal{D}_{\pi}\\ \vdots\\ [\![\Gamma]\!] \vdash [\![\pi]\!] : \forall \alpha.\, [\![F]\!]\end{array}}{[\![\Gamma]\!] \vdash [\![\pi]\!] : [\![F]\!]\left[[\![G]\!]/\alpha\right]}\ (\text{ins})$$

As for the derived rules for $\wedge$, we are here interested only in $(\wedge 1 \vdash)$:

$$\dfrac{\begin{array}{c}\pi\\ \vdots\\ \Gamma \vdash F \wedge G\end{array}}{\Gamma \vdash F}\ (\wedge 1 \vdash) \quad \longmapsto$$

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_\pi\\ \vdots\\ [\![\Gamma]\!] \vdash [\![\pi]\!] : \forall\alpha.([\![F]\!] \to [\![G]\!] \to [\![F]\!]) \to [\![F]\!]\end{array}}{[\![\Gamma]\!] \vdash [\![\pi]\!] : ([\![F]\!] \to [\![G]\!] \to [\![F]\!]) \to [\![F]\!]}\ (\text{ins}) \qquad \begin{array}{c}\vdots\\ \vdash \underline{\mathtt{true}} : [\![F]\!] \to [\![G]\!] \to [\![F]\!]\end{array}}{[\![\Gamma]\!] \vdash ([\![\pi]\!]\ \underline{\mathtt{true}}) : [\![F]\!]}\ (\text{app})$$

The interesting thing is that this translation maps cut-elimination passages in β-reduction: terms get substituted for abstracted variables the same way proofs get in the place of parcels of hypotheses. That is the core of Curry-Howard isomorphism. Next we may see that for every n there is a unique normal (in the sense of cut-elimination) proof $\check{n}$ of $\vdash \mathtt{Nat}(\mathtt{succ}^n\, 0)$ and in fact its translation is the derivation of $\vdash \underline{n} : \mathtt{Int}$footnotethe proof is really similar to the one done to show that every normal term typable with `Int` is a Church numeral. The only difference is one has to take into account the presence of the axiom $\neg\, \mathtt{succ}\, \xi = 0$. The consistency of HA_2, which gives that $\mathtt{succ}\, \xi = 0$ cannot be proved, needs to be exploited..

Now suppose we have the formula $F(n, m)$ which expresses that a given algorithm representing a function f terminates in output m if given input n. If we are able to prove

$$\forall n \in \mathbb{N} \exists m \in \mathbb{N}.F(n, m)$$

it means that in HA_2 we have a derivation π that proves

$$\vdash \forall\xi.(\mathtt{Nat}(\xi) \Rightarrow \exists\eta.(\mathtt{Nat}(\eta) \wedge F(\xi, \eta))).$$

Now applying the translation to the formula we get:

$$[\![\forall\xi.(\mathtt{Nat}(\xi) \Rightarrow \exists\eta.(\mathtt{Nat}(\eta) \wedge A(\xi, \eta)))]\!] = \mathtt{Int} \to (\mathtt{Int} \times [\![F]\!]).$$

So translating π gives

$$\vdash_{\mathbf{F}} [\![\pi]\!] : \mathtt{Int} \to (\mathtt{Int} \times [\![F]\!]).$$

Le us consider the term $M = \lambda x.([\![\pi]\!]\ x\, \underline{\mathtt{true}})$, which is typable with type $\mathtt{Int} \to \mathtt{Int}$. We have that M represents f. In fact, given any $n \in \mathbb{N}$:

$$\dfrac{\begin{array}{c}\check{n}\\ \vdots\\ \vdash \mathtt{Nat}(\mathtt{succ}^n\, 0)\end{array} \qquad \dfrac{\begin{array}{c}\pi\\ \vdots\\ \vdash \forall\xi.(\mathtt{Nat}(\xi) \Rightarrow \exists\eta.(\mathtt{Nat}(\eta) \wedge F(\xi, \eta)))\end{array}}{\vdash \mathtt{Nat}(\mathtt{succ}^n\, 0) \Rightarrow \exists\eta.(\mathtt{Nat}(\eta) \wedge F(\mathtt{succ}^n\, 0, \eta))}\ (\forall \vdash)}{\vdash \exists\eta.(\mathtt{Nat}(\eta) \wedge F(\mathtt{succ}^n\, 0, \eta))}\ (\Rightarrow \vdash)$$

is the proof which translates into $(\llbracket\pi\rrbracket\ \underline{n})$ typable with $\mathtt{Int} \times \llbracket F\rrbracket$. If we then proceed on this proof by cut-elimination (which corresponds to β-reduction steps) we obtain a cut-free one that must end with $(\vdash \exists^1)$, and this last rule must substitute a term which does not contain variables, i.e. a term of the form $\mathtt{succ}^m\, 0$. So we have a normal proof π_n which proves $\vdash \mathtt{Nat}(\mathtt{succ}^m\, 0) \wedge F(\mathtt{succ}^m\, 0, \mathtt{succ}^n\, 0)$, whose translation is β-equivalent to $(\llbracket\pi\rrbracket\ \underline{n})$. If we apply $(\wedge 1 \vdash)$ we obtain a normal proof of $\mathtt{succ}^m\, 0$ which therefore must be $\check{m}$: looking on translations we have that applying $(\wedge 1 \vdash)$ means having $(\llbracket\pi\rrbracket\ \underline{n}\,\mathtt{true}) \equiv_\beta \underline{m}$, and therefore $(M\,\underline{n}) \equiv_\beta \underline{m}$. Then again, if we apply to π_n $(\wedge 2 \vdash)$ instead, we get a proof of $F(\mathtt{succ}^n\, 0, \mathtt{succ}^m\, 0)$, which means that $F(n, m)$ is true, i.e. $f(n) = m$: M indeed represents f.

3.3.3 Removing the junk term

We map typable terms with junk to typable terms without junk.

First define a map on types:

$$\begin{aligned}\langle\!\langle\alpha\rangle\!\rangle &:= \alpha,\\ \langle\!\langle\tau \to \sigma\rangle\!\rangle &:= \langle\!\langle\tau\rangle\!\rangle \to \langle\!\langle\sigma\rangle\!\rangle\,,\\ \langle\!\langle\forall\alpha.\tau\rangle\!\rangle &:= \forall\alpha.(\alpha \to \langle\!\langle\tau\rangle\!\rangle).\end{aligned}$$

This map commutes with substitution, so that

$$\langle\!\langle\tau[\sigma/\alpha]\rangle\!\rangle = \langle\!\langle\tau\rangle\!\rangle\left[\langle\!\langle\sigma\rangle\!\rangle/\alpha\right]$$

.

Then we label some variables with type variables, so that we have variables x_α, and define for every type τ a term typable with $\langle\!\langle\tau\rangle\!\rangle$:

$$\begin{aligned}B_\alpha &:= x_\alpha,\\ B_{\tau\to\sigma} &:= \lambda y.B_\sigma,\\ B_{\forall\alpha.\tau} &:= \lambda x_\alpha.B_\tau.\end{aligned}$$

Every time we use B_τ we consider a typing $\mathcal{D}_\tau$ for it that ends in

$$x_{\alpha_1} : \alpha_1, \ldots, x_{\alpha_n} : \alpha_n \vdash B_\tau : \langle\!\langle\tau\rangle\!\rangle\,,$$

where $\vec{\alpha}^n = \mathrm{FTV}(\tau)$. Moreover we have that if τ is a closed type then $\langle\!\langle\tau\rangle\!\rangle$ is a closed type always such that there is a closed term typable with it.

Now, given a typing $\mathcal{D}$ for a term M with junk, we define by induction a translation into a typing $\langle\!\langle \mathcal{D} \rangle\!\rangle$ for a term $\langle\!\langle M \rangle\!\rangle$. (var) rule is affected only by applying $\langle\!\langle \, . \, \rangle\!\rangle$ on the types. (abs) and (app) simply carry on the induction without adding anything. For the remaining cases

$$\frac{\mathcal{D}' \rightsquigarrow A \vdash M : \tau}{A \vdash M : \forall\alpha.\tau} \text{ (gen)}$$

becomes, after eventually adding $x_\alpha : \alpha$ by weakening if it is not already in the environment:

$$\frac{\dfrac{\langle\!\langle \mathcal{D}' \rangle\!\rangle \rightsquigarrow \langle\!\langle A \rangle\!\rangle, x_\alpha : \alpha \vdash \langle\!\langle M \rangle\!\rangle : \langle\!\langle \tau \rangle\!\rangle}{\langle\!\langle A \rangle\!\rangle \setminus \{x_\alpha : \alpha\} \vdash \lambda x_\alpha. \langle\!\langle M \rangle\!\rangle : \alpha \to \langle\!\langle \tau \rangle\!\rangle} \text{ (abs)}}{\langle\!\langle A \rangle\!\rangle \setminus \{x_\alpha : \alpha\} \vdash \lambda x_\alpha. \langle\!\langle M \rangle\!\rangle : \forall\alpha.(\alpha \to \langle\!\langle \tau \rangle\!\rangle)} \text{ (gen)}$$

As for (ins):

$$\frac{\mathcal{D}' \rightsquigarrow M : \forall\alpha.\tau}{A \vdash M : \tau[\rho/\alpha]} \text{ (ins)}$$

becomes

$$\frac{\dfrac{\langle\!\langle \mathcal{D}' \rangle\!\rangle \rightsquigarrow \langle\!\langle A \rangle\!\rangle \vdash \langle\!\langle M \rangle\!\rangle : \forall\alpha.(\alpha \to \langle\!\langle \tau \rangle\!\rangle)}{\langle\!\langle A \rangle\!\rangle \vdash \langle\!\langle M \rangle\!\rangle : \langle\!\langle \rho \rangle\!\rangle \to \langle\!\langle \tau \rangle\!\rangle \left[\langle\!\langle \rho \rangle\!\rangle / \alpha\right]} \text{ (ins)} \quad \overrightarrow{x_\beta : \beta} \vdash B_\rho : \langle\!\langle \rho \rangle\!\rangle}{\langle\!\langle A \rangle\!\rangle, \overrightarrow{x_\beta : \beta} \vdash \langle\!\langle M \rangle\!\rangle \; B_\rho : \langle\!\langle \tau[\rho/\alpha] \rangle\!\rangle} \text{ (app)}$$

($\mathcal{D}_\rho$: $\vdots$)

where $\vec{\beta} = \mathrm{FTV}(\rho)$. The only rule left is the junk rule which gives $A \vdash \Omega : \bot$, and we replace it with $\mathcal{D}_\bot$ that types $B_\bot = I$, to which we add A by weakening, obtaining $\mathcal{D}_\bot \rightsquigarrow A \vdash I : \forall\alpha.(\alpha \to \alpha)$.

Note that applying elimination of (gen)-(ins) sequences as we had shown in 3.1.8 get translated in first doing the (gen)-(ins) erasing and then contracting a redex, this latter operation such that it changes only the terms B_τ. The translation is preserved if we take special attention so that when doing the substitution $B_\tau[B_\rho/x_\alpha]$ we make the necessary changes to the standard derivation accompanying B_τ so that we get the standard derivation of $B_{\tau[\rho/\alpha]}$ (which involves changing the type assigned to weakened variables).

Now we follow the reduction of a typable term with junk with an eye on its typing: if we always do (gen)-(ins) elimination on both $\mathcal{D}$ and $\langle\!\langle \mathcal{D} \rangle\!\rangle$ (which means reducing $\langle\!\langle M \rangle\!\rangle$) we have that every reduction step in $\mathcal{D}$ may be carried out in $\langle\!\langle \mathcal{D} \rangle\!\rangle$ preserving the fact that one is the translation of the other. Moreover the translation of an (ins) before (gen) derivation of a normal term is always such that the term being typed is normal. So, in short words, by keeping the typings with (ins) before (gen) property, if $M \overset{\beta}{\twoheadrightarrow} M^*$ then $\langle\!\langle M \rangle\!\rangle \overset{\beta}{\twoheadrightarrow} \langle\!\langle M^* \rangle\!\rangle$ and $\langle\!\langle M^* \rangle\!\rangle = \langle\!\langle M \rangle\!\rangle^*$.

In particular the translation of integers is

$$\langle\!\langle \mathtt{Int} \rangle\!\rangle = \forall\alpha.\alpha \to \alpha \to (\alpha \to \alpha) \to \alpha$$

and the translation corresponding to the unique (ins) before (gen) derivation of $\vdash \lambda x\lambda f.(f^n\, x) : \mathtt{Int}$ gives the term

$$\langle\!\langle \underline{n} \rangle\!\rangle = \lambda x_\alpha \lambda x \lambda f.(f^n\, x).$$

We can then easily design

$$\underline{\texttt{weak}} := \lambda n.\lambda x_\alpha.n, \qquad \vdash_{\mathbf{F}} \underline{\texttt{weak}} : \texttt{Int} \to \langle\!\langle \texttt{Int} \rangle\!\rangle\,,$$
$$\underline{\texttt{contr}} := \lambda n.\underline{\texttt{red}}(n\,I), \qquad \vdash_{\mathbf{F}} \underline{\texttt{contr}} : \langle\!\langle \texttt{Int} \rangle\!\rangle \to \texttt{Int}$$

that preserve the value. $\underline{\texttt{red}}$ is a term that reduces the type instantiated in $\texttt{Int}$ leaving the value equal to make the generalization possible again:

$$\underline{\texttt{red}} := \lambda n\lambda x\lambda f.(n\,I\,(\lambda h\lambda y.f\,(h\,y))\,x), \quad \vdash_{\mathbf{F}} \underline{\texttt{red}} : \forall\alpha.\,\texttt{Int}_{\alpha\to\alpha} \to \texttt{Int}_\alpha\,.$$

Then if M is a term with junk representing f then

$$M' := \lambda n.\underline{\texttt{contr}}\,(\langle\!\langle M \rangle\!\rangle\ (\underline{\texttt{weak}}\,n))$$

represents f and is typable with type $\texttt{Int} \to \texttt{Int}$.

3.3.4 An example of an unrepresented function

One may ask oneself: is there a computable total function that is not represented in system **F**? In other words, is there a recursive total function whose totality cannot be proved in PA_2? The answear is yes.

We may take a coding of typable terms together with their types into integers, say $[\![M, \tau]\!]$[5], and then define $N(n) = m$ if and only if $n = [\![N, \tau]\!]$, $m = [\![M, \tau]\!]$ and $M = N^*$, and $N(n) = 0$ if n is not the code of a term. It is computable: namely, the algorithm consists in taking the term coded by n and normalizing it. All this can even be encoded as recursive functions. And we have shown by SN that it is total. From what we know now we could already conclude it is not representable: otherwise it would be provably total in PA_2 and so PA_2 would prove SN for **F**. But let's briefly check directly on N.

We surely have the following functions, all representable in **F**:

- $\texttt{app}(n, m) := [\![(N\,M), \tau]\!]$, if $n = [\![N, \sigma \to \tau]\!]$ and $m = [\![M, \sigma]\!]$,
- $\sharp(n) := [\![\underline{n}, \texttt{Int}]\!]$,
- $\flat(n) := m$ if $m = [\![\underline{n}, \texttt{Int}]\!]$, $\flat(n) = 0$ otherwise.

Now we may define $D(n) := \flat(N(\texttt{app}(n, \sharp(n)))) + 1$. This function is clearly total. Suppose now P typable with type $\texttt{Int} \to \texttt{Int}$ represents D, and let's try to compute $D(n)$ with $n =$

[5] in fact we have to encode an entire derivation if we want for this encoding to certificate that M has type τ: we will see that type checking is not decidable, so if we want this encoding to be computable we must ourselves provide the prof.

$[\![P, \mathtt{Int} \to \mathtt{Int}]\!]$: we are applying N to the coding of the application $(P\,\underline{n})$, and so it will yield $\sharp(\underline{D(n)})$, and in the end $D(n) = D(n) + 1$, a contradiction. So neither D nor N are representable.

This result is a variant of the famous result by Turing for which there is no total recursive function enumerating all total recursive functions: from what we have said here we may see there is no *provably total* recursive function which enumerates all provably total recursive functions.

3.4 What do we loose with F?

So system **F** would seem perfect. In fact there are some problems concerning it.

First of all, it would seem that **F** is maybe *too* expressive. Given a term in **F** we are not able a-priori to tell anything about its computational cost. In fact even the Ackerman function is typable in system **F**.

Example 3.4.1. The Ackerman function $A : \mathbb{N}^2 \to \mathbb{N}$ is defined by:

$$\begin{aligned} A(0, n) &:= n + 1, \\ A(m + 1, 0) &:= A(m, 1), \\ A(m + 1, n + 1) &:= A(m, A(m + 1, n)). \end{aligned}$$

$A(n, n)$ is a function that bounds every elementary function, including all the towers of exponential of fixed height. To have an idea:

$$\begin{aligned} A(1, 1) &= 3, & A(2, 2) &= 7, \\ A(3, 3) &= 2^6 - 3 = 61 & A(4, 4) &= 2^{2^{65536}} - 3. \end{aligned}$$

The number of atoms in the universe can be bounded by 2^{266}, so for example $A(4, 4)$ is way more than the number of all possible sets of atoms in the universe.

It can be seen as a double recursion: one on functions $\mathbb{N} \to \mathbb{N}$ containing one on $\mathbb{N}$. The polymorphism of system **F** here suits well for defining the representation of A. In terms of recursion the inner one, supposing we know $A_m := A(m, \,.\,) : \mathbb{N} \to \mathbb{N}$, has $A_m(1)$ as base value and $h_n(i, j) := A_n(i)$ as step function. We may represent the latter with a context

$$H[\ ] := \lambda x \lambda y.\Box\, x,$$

so that $H[\underline{A_n}] = \underline{h_n}$ and is typable with $\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}$ if $\underline{A_n}$ is typable with $\mathtt{Int} \to \mathtt{Int}$. So if we have $\underline{A_n}$ the term representing A_{n+1} is

$$\underline{A_{n+1}} := \underline{\mathrm{REC}}_{(\underline{A_n}\,\underline{1}),H[\underline{A_n}]}.$$

See example 3.2.4 for the definition of $\underline{\text{REC}}$. Now if we see the outer recursion, we thus have that the base value is $A_0 = \texttt{succ}$, and the step function is the one depicted above, represented by

$$K := \lambda f \lambda n.\underline{\text{REC}}_{(f\,\underline{1}),H[f]}$$

typable with $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int} \to \texttt{Int}$. So in the end:

$$\underline{A} := \underline{\text{REC}}_{\underline{\texttt{succ}},K}.$$

This is not nice. In fact usually we are mainly interested in *feasible* functions, with some bound on complexity. System **F** provides no easy way of dealing with it.

Another problem regards type inference. We next show the result proved by Wells in [Wel99].

3.4.1 Undecidability of TC

First we show how type checking results to be undecidable, by reducing to it the following problem.

Definition 3.4.2 (SUP). Let $\vec{\sigma}^n$ and $\vec{\tau}^n$ be sequences of types in $\mathbb{T}_{\mathbf{S}}$ (so types *without* quantifiers). Then we call the sequence

$$(\sigma_1, \tau_1), \ldots, (\sigma_n, \tau_n) = \overrightarrow{(\sigma, \tau)}^n$$

an *instance of the* SUP *problem.*

Let **S** be an *open substitution*, i.e a substitution S with $\text{RAN}(S) \subseteq \mathbb{T}_{\mathbf{S}}$. We say S is a solution for $\overrightarrow{(\sigma, \tau)}$ if there exist open substitutions $\vec{S} = S_1, \ldots, S_n$ such that for every i we have $S_1(S(\sigma_i)) = S(\tau_i)$.

The *semi-unification problem*, denoted by SUP, is the problem of determining if an instance $\overrightarrow{(\sigma, \tau)}$ has a solution.

Kfoury, Tiuryn and Urzyczyn showed in 1990 that the immortality problem for a Turing machine (whether a given Turing machine which admits infinite tape transcription does not terminate from every possible initial configuration of status and tape) can be reduced to SUP with two pairs. (see [KTU93]), and in turn the immortality problem had been already shown undecidable in 1966 by Hooper in [Hoo66].

Theorem 3.4.3 (SUP $\leq$ TC$_{\mathbf{F}}$). SUP *with two pairs is reducible to* TC *in system* **F**.

Proof. Let $(\sigma_1, \tau_1), (\sigma_2, \tau_2)$ be any instance of SUP with two pairs. Let $\vec{\alpha}^m = \text{FTV}(\sigma_1, \sigma_2, \tau_1, \tau_2)$ and δ_1, δ_2 be variables not in $\vec{\alpha}$. Let's build an instance of TC$_{\mathbf{F}}$.

Let M be the term $b\,(\lambda x.c\,x\,x)$, A be the environment

$$A := \{\, b : \forall\gamma.(\gamma \to \gamma) \to \beta,\ c : \forall.(\tau_1 \to \delta_1) \to (\delta_2 \to \tau_2) \to (\sigma_1 \to \sigma_2) \,\}.$$

We will show that there is a solution of the SUP instance $(\sigma_1, \tau_1), (\sigma_2, \tau_2)$ if and only if $A \vdash_{\mathbf{F}} M : \beta$.

Suppose there is a solution of the SUP instance: there are S, S_1 and S_2 open substitutions such that $S_i(S(\sigma_i)) = S(\tau_i)$. Denote $\varphi := S(\sigma_1) \to S(\sigma_2)$ and $B := A, x : \forall.\varphi$. We have:

$$S_1(\varphi) = S(\tau_1) \to S_1(S(\sigma_2)), \quad S_2(\varphi) = S_2(S(\sigma_1) \to S(\tau_2),$$

so we can instantiate $A(c)$ by applying to it $\mathbf{S}$ and giving convenient types for δ_1 and δ_2 to obtain $S_1(\varphi) \to S_2(\varphi) \to \varphi$. We may also assume that β does not appear in $S(\mathrm{SUPP}(S))$ and $S_i(\mathrm{SUPP}(S_i))$.

Now we have the following derivation:

$$\dfrac{\dfrac{\dfrac{}{A \vdash b : \forall\gamma.(\gamma \to \gamma) \to \beta}\,(\mathrm{var})}{A \vdash b : ((\forall.\varphi) \to \forall.\varphi) \to \beta}\,(\mathrm{ins}) \qquad \begin{array}{c}\mathcal{D}\\ \vdots \\ A \vdash \lambda x.c\,x\,x : (\forall.\varphi) \to \forall.\varphi\end{array}}{A \vdash b\,(\lambda x.c\,x\,x) : \beta}\,(\mathrm{app})$$

where $\mathcal{D}$ is the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash c : A(c)}\,(\mathrm{var})}{B \vdash c : S_1(\varphi) \to S_2(\varphi) \to \varphi}\,(\mathrm{ins}) \quad \dfrac{\dfrac{}{B \vdash x : \forall.\varphi}\,(\mathrm{var})}{B \vdash x : S_1(\varphi)}\,(\mathrm{ins})}{B \vdash c\,x : S_2(\varphi) \to \varphi}\,(\mathrm{app}) \quad \dfrac{\dfrac{}{B \vdash x : \forall.\varphi}\,(\mathrm{var})}{B \vdash x : S_2(\varphi)}\,(\mathrm{ins})}{B \vdash c\,x\,x : \varphi}\,(\mathrm{app})}{B \vdash c\,x\,x : \forall.\varphi}\,(\mathrm{gen})}{A \vdash \lambda x.c\,x\,x : (\forall.\varphi) \to \forall.\varphi}\,(\mathrm{abs})$$

The other direction of the proof is more complicated. Let $\mathcal{D}$ be a typing $\mathcal{D} \rightsquigarrow A \vdash M : \beta$. We will basically "climb" it to get the information necessary to build a solution to the SUP problem. First of all we suppose $\mathcal{D}$ is in (ins) before (gen) form, and that by weakening on reverse all environments say the strictly necessary, i.e. their domain is the free variables of the term being typed. Let $B = \mathrm{DE}(\mathcal{D}, c\,x\,x)$ be the derived environment for $c\,x\,x$ in $\mathcal{D}$, and let ρ be the type assigned by it to x. So $B = \{\, x : \rho, c : A(c) \,\}$. Now let us climb the derivation up to the final derivation for c. Because it is applied to x and $\mathcal{D}$ is in (ins) before (gen) form there are only (ins) rules originating form the (var) rule. So if we add up all the substitutions we obtain a substitution T so that

$$\mathrm{FDT}(\mathcal{D}, c) = (T(\tau_1) \to T(\delta_1)) \to (T(\delta_2) \to T(\tau_2)) \to T(\sigma_1) \to T(\sigma_2).$$

We extend $\vec{\alpha}$ with the necessary variables so that $\mathrm{FTV}(T(\mathrm{SUPP}(T))) \subseteq \vec{\alpha}$. The type above means that the final derived type for x (the occurrence to which c is applied) must be $T(\tau_1) \to T(\delta_1)$, and then after (app) is applied:

$$\mathrm{IDT}(\mathcal{D}, c\,x) = (T(\delta_2) \to T(\tau_2)) \to T(\sigma_1) \to T(\sigma_2).$$

There are no (ins) following because there are no quantifiers, and there can't even be some (gen) because the result must be applied again to x. So the final derived type for $c\,x$ is the same as above, and thus the final derived type for the other occurrence of x must be $T(\delta_2) \to T(\tau_2)$, and the initial derived type for $c\,x\,x$ is $T(\sigma_1) \to T(\sigma_2)$. So afterwards there are no (ins) rules, and so there are only (gen) rules following, and the final derived type for $c\,x\,x$ is

$$\mathrm{FDT}(\mathcal{D}, c\,x\,x) = \overrightarrow{\forall\varepsilon}.T(\sigma_1) \to T(\sigma_2).$$

Then (abs) is applied and we obtain an initial derivation for $\lambda x.c\,x\,x$ with

$$A(c) \vdash \lambda x.c\,x\,x : \sigma \to \overrightarrow{\forall\varepsilon}.T(\sigma_1) \to T(\sigma_2).$$

If we now take a look on the other branch of the final (app), we see there can't be any external quantifiers, so no (gen) rules and one (ins) rule to delete the only quantifier of the type of b, so the final derived type for b must be $(\psi \to \psi) \to \beta$ for some ψ. So the type required as input by b has no quantifiers. So going back to the right branch, no (gen) rules, and no (ins) neither, and the application can be carried out only if:

$$\psi \to \psi = \sigma \to \overrightarrow{\forall\varepsilon}.T(\sigma_1) \to T(\sigma_2) \iff \psi = \sigma = \overrightarrow{\forall\varepsilon}.T(\sigma_1) \to T(\sigma_2).$$

So we have an expression for σ. If we now return up in the derivation to the branches regarding the two occurrences of x, we see the final types have no external quantifiers, so no (gen) rules and (ins) rules to delete all of $\overrightarrow{\forall\varepsilon}$. If we sum up the substitutions done for the first occurrence in T_1 and all those for the second in T_2, we get the equalities:

$$T_1(T(\sigma_1)) \to T_1(T(\sigma_2)) = T(\tau_1) \to T(\delta_1), \quad T(\delta_2) \to T(\tau_2).$$

Now if we split the two equalities in the left and right part of the implication, and take only the left one for the first and the right one for the other we get the equalities:

$$T_1(T(\sigma_i)) = T(\tau_i), \quad \text{for i=1,2.}$$

These are almost the solution to the semi-unification problem. Fact is, there could be quantifiers in the range of the substitutions. We have to delete them. To make things uniform, we α-convert all the bound variables to some arbitrary fresh variable δ, and then apply the erasing function $(\,.\,)_{\mathbf{S}}$. So we define the open substitutions

$$S := \big[\overrightarrow{(T(\alpha))_{\mathbf{S}}}/\alpha\big], \quad S_i := \big[\overrightarrow{(T_i(\alpha))_{\mathbf{S}}}/\alpha\big],$$

that act on the variables $\vec{\alpha}$, and leave the other as they are (including δ). So in particular

$$S_i(S(\sigma_i)) = (T_i((T(\sigma_i))_{\mathbf{S}}))_{\mathbf{S}} = (T_i(T(\sigma_i)))_{\mathbf{S}} = (T(\tau_i))_{\mathbf{S}} = S(\tau_i)$$

and S, S_1, S_2 is a solution of $(\sigma_1, \tau_1), (\sigma_2, \tau_2)$. □

Remark 3.4.4. We may think of some restriction we may impose on what types may be instantiated in bounded variables. One first idea would be to let instantiate only types without quantifiers. The proof above however shows that as far as type checking is concerned such an approach fails, because it comes from an undecidability result on types without quantifiers. However we can think of other forms of restriction, by controlling the way quantifiers are distributed in all types. Wells himself, together with Kfoury, describes in [KW94] an algorithm to solve typability and type inference in a fragment of system **F**: the rank 2 fragment. Rank is defined inductively so that rank 0 holds the types without quantifiers, and then rank $k+1$ is built with the usual rules using formulas of the same or inferior rank, except that on the left of an implication only types of rank strictly less can be used. In order to leave this structure invariant under instantiation we permit to instantiate only type of rank 0, i.e. open types. However apart from the positive result about rank 2, in the same paper typability is proved undecidable for every higher rank.

3.4.2 Undecidability of TYP

In the same work in which he proved undecidability of $\mathrm{TC}_{\mathbf{F}}$ Wells has proved that there is a reduction of $\mathrm{TC}_{\mathbf{F}}$ to $\mathrm{TYP}_{\mathbf{F}}$. As there is also a trivial reduction of $\mathrm{TYP}_{\mathbf{F}}$ to $\mathrm{TC}_{\mathbf{F}}$, we have that the two problems are equivalent and undecidable.

Proposition 3.4.5 ($\mathrm{TYP}_{\mathbf{F}} \leq \mathrm{TC}_{\mathbf{F}}$)**.** *There is a reduction of* $\mathrm{TYP}_{\mathbf{F}}$ *to* $\mathrm{TC}_{\mathbf{F}}$.

Proof. First observe that we can assume the term for which we want to decide typability is closed, as M is typable if and only if $\overrightarrow{\lambda x}.M$ is, where $\vec{x} = \mathrm{FV}(M)$: one direction is due to subterm typing, the other to multiple uses of (abs).

Now take the instance of $\mathrm{TC}_{\mathbf{F}}$ given by the sequent $\vdash (\lambda x \lambda y.y)\, M : \alpha \to \alpha$. Then if M is typable with σ, we can type $\lambda x \lambda y.y$ with $\sigma \to \alpha\alpha$ and so we solve $\mathrm{TC}_{\mathbf{F}}$. On the other hand if $\vdash_{\mathbf{F}} (\lambda y \lambda x.x)\, M : \alpha \to \alpha$ then by subterm typing we can conclude that M is typable. □

Now we must introduce a complicate machinery to prove reduction in the other direction. The main idea of the proof will be to construct, given A, M and σ, a simple contexts $C[\ ]$ such that when we plug M in the hole all the free variables get captured, and the existence of *any* valid typing of $C[M]$ induces type τ on M. So (apart from small tweaking at the end) we may say $A \vdash_{\mathbf{F}} M : \sigma$ if and only if $C[M]$ is typable.

Rather than build a specific context for every environment, term and type may come around we show a sort of induction that shows the problem solved for types of increasing complexity, building up a machinery independent of the various environments. The proof will however have an algorithmic content: given an instance of $\mathrm{TC}_{\mathbf{F}}$, we will be effectively able to build up the instance of $\mathrm{TYP}_{\mathbf{F}}$, though it may be quite hard.

At the core (or rather at the beginning) there is the notion of *invariant type assumption*: the first approach to induce given types to "places" in pure terms. Given a variable present in any place of a derivation $\mathcal{D}$, we define by $\mathcal{D}(x)$ the type σ such that $x : \sigma$ is present in some environment in $\mathcal{D}$. It is well defined, once we α-convert all bound variables so that there is no name collision. We also assume that the environment present at the end of any subderivation is present in every sequent of the same subderivation. Practically we are saying that we do not make use of the fact that (app) merges environments. We can make such an assumption because of weakening, and because we know there can't be any collisions with variables that get bounded later in the derivation.

Inducing invariant types

Definition 3.4.6 (invariant type assumption). Given a term M, an environment A and $x \in \mathrm{BV}(M)$ (which we will then consider fixed with respect to α-equivalence, and different from all other variables bound or not) we say A induces the *invariant type assumption* $x : \sigma$ if both of the following properties hold:

1. there is a derivation $\mathcal{D} \rightsquigarrow A \vdash M : \tau$ such that $\mathcal{D}(x) = \sigma$;
2. if $\mathcal{D}$ is a derivation leading to $A \vdash M : \tau$ for some τ, there exists a type variable renaming R such that $R(A) = A$ and $\mathcal{D}(x) = R(\sigma)$.

We will later define terms that induce some desired types. As for now, we will introduce a notion that extends that of invariant type assumption: it will allow us to put together various invariant assumptions. The defintion is designed so that we will be able to discard parts of the environment we have used to build invariant type assumption and which are not needed afterwards, and also in such a manner that such results will be chainable.

Definition 3.4.7 (inducing type environments). Let A and B be compatible environments and $C[\ ]$ a simple context such that

- $\mathrm{FV}(C[\ ]) \subseteq \mathrm{DOM}(A)$,
- $\mathrm{BV}(C) \cap \mathrm{DOM}(A) = \emptyset$,
- $\mathrm{DOM}(B) \subseteq \mathrm{BHV}(C[\ ]) \cup \mathrm{DOM}(A)$.

Recall that $\mathrm{BHV}(C[\ ])$ is the set of bounded variables whose scope contains the hole.

We say that the type environment A and the context C together induce the type environment B, written $A, C[\ ] \rhd B$ if for every term M and type environment A' such that

- A' is compatible with A, B,
- $\mathrm{FV}(M) \subseteq \mathrm{DOM}(A, B, A')$,

- $\mathrm{DOM}(A') \cap (\mathrm{DOM}(A) \cup \mathrm{BV}(C[M])) = \emptyset$,
- $\mathrm{FTV}(A') \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A)$,

both of the following properties hold:

1. If σ is such that $A', A, B \vdash_{\mathbf{F}} M : \sigma$ then there exist a type τ, an environment E with $\mathrm{DOM}(E) = \mathrm{BHV}(C[\ ]) \setminus \mathrm{DOM}(B)$ and a derivation $\mathcal{D}$ that contains the two sequents

$$A', A \vdash C[M] : \tau, \qquad A', A, B, E \vdash M : \sigma.$$

2. On the converse if $\mathcal{D}$ is a derivation containing $A, A' \vdash C[M] : \tau$ for some τ, then there exist a type σ, an environment E with $\mathrm{DOM}(E) = \mathrm{BHV}(C[\ ]) \setminus \mathrm{DOM}(B)$ and a variable renaming R invariant on A, A', i.e. such that $R(A, A') = A, A'$ and $\mathcal{D}$ contains also the sequent

$$A, A', R(B), E \vdash M : \sigma.$$

So confronting the two definitions the assumption B plays the role of $x : \sigma$, so that a variable $x \in \mathrm{DOM}(B) \cap \mathrm{BHV}(C[\ ])$ will have an invariant type assumption. A' covers all the extra free variables that are not handled either by A or by the B: we are not interested in those variables having an invariant type assumption with $C[\ ]$, and so we permit some variables to be handled instead by another context containing this one, and so we are able to use multiple contexts to induce more assumption at the same time.

The condition $\mathrm{FTV}(A') \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A)$ is needed because we will have to apply (gen) on the variables in $\mathrm{FTV}(B) \setminus \mathrm{FTV}(A)$. Finally, E is needed to treat those variables bounded by $C[\ ]$ but which need not to be invariant.

Now in order to be more abstract on the choice of of the context and the term variables we introduce a relation defined purely on sets of types.

Definition 3.4.8 (inducing types). Let $\mathbb{X}$ and $\mathbb{Y}$ be sets of types. We call an algorithm Ψ taking as input a type environment and giving as output a type environment and a context a *witness for* $\mathbb{X} \blacktriangleright \mathbb{Y}$ if and only if for every non-empty type environment B with $\mathrm{RAN}(B) \subseteq \mathbb{Y}$ we have $\Psi(B) = (A, C[\ ])$, where $\mathrm{RAN}(A) \subseteq \mathbb{X}$, $\mathrm{FTV}(B) \cap \mathrm{FTV}(\mathbb{X}) \subseteq \mathrm{FTV}(A)$ and $A, C[\ ] \rhd B$. We say $\mathbb{X} \blacktriangleright \mathbb{Y}$, to be read $\mathbb{X}$ *induces* $\mathbb{Y}$, if and only if we have a witness for it.

Lemma 3.4.9 (properties of $\rhd$). *$\rhd$ enjoys the following properties:*

1. *if A induces the invariant type assumption $x : \sigma$ in M then we can build $C[\ ]$ such that $A, C[\ ] \rhd \{x : \sigma\}$,*
2. *$A, C[\ ] \rhd B \implies \forall R$ renaming $: R(A), R(C[\ ]) \rhd R(B)$,*

3. $A, C[\ ] \rhd B \implies A, C[\ ] \rhd A, B$,
4. $A, C[\ ] \rhd B, \quad B' \subseteq B \implies A, C[\ ] \rhd B'$,
5. $A, C[\ ] \rhd B$, $\mathrm{FTV}(\hat{A}) \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A)$, *and* $\mathrm{DOM}(\hat{A}) \cap (\mathrm{DOM}(A) \cup \mathrm{BV}(B)) = \emptyset \implies (A \cup \hat{A}), C[\ ] \rhd B$.

Proof. The only ones that are not trivial are number 1 and 5.

Number 1 is the first part of the core tool used in the proof. As $x \in \mathrm{BV}(M)$ we have that $M = C'[\lambda x.N]$ for some context $C'[\ ]$ and some subterm N. Let $C[\ ] := C'[\lambda x.\underline{\mathtt{true}}\, N\, \Box]$. Now we check that $A, C[\ ] \rhd \{x : \tau\}$.

First: $\mathrm{FV}(C[\ ]) \subseteq \mathrm{FV}(M) \subseteq \mathrm{DOM}(A)$, $\mathrm{BV}(C[\ ]) = \mathrm{BV}(M)$ and by convention $\mathrm{BV}(M) \cap \mathrm{DOM}(A) = \emptyset$, and finally $\mathrm{DOM}(\{x : \sigma\}) = \{x\} \subseteq \mathrm{BHV}(C[\ ])$. Now take a term N' and an environment A' that satisfy the conditions given in the second part of the definition.

1. Suppose $A', A, x : \tau \vdash_{\mathbf{F}} N' : \tau'$. As we know that there is a derivation $\mathcal{D} \rightsquigarrow A \vdash C'[\lambda x.N] : \tau$ with $\mathcal{D}(x) = \sigma$, we may create a valid derivation $\mathcal{D}'$ from $\mathcal{D}$ by substituting the subderivation leading to $\lambda x.N$ with one (ending in the same type) leading to $\lambda x.\underline{\mathtt{true}}\, N\, N'$ and for which the final derived type of N' is τ', and then applying weakening to add A'. Then

$$\mathcal{D}' \rightsquigarrow A, A' \vdash C[N'] : \tau.$$

Moreover $\mathcal{D}'(x) = \mathcal{D}(x) = \sigma$, so by the convention of not using (app)-merging $\mathcal{D}'$ contains the following sequent on N':

$$A, A', x : \tau, E \vdash N' : \tau',$$

where E assigns types to all the variables that get bound later, all but x.

2. Suppose now that $\mathcal{D} \rightsquigarrow A, A' \vdash C[N'] : \tau$. There is a subderivation leading to $A, A', x : \sigma', E \vdash \underline{\mathtt{true}}\, N\, N' : \tau'$, and we can safely substitute it with one leading to $A, A', x : \sigma', E \vdash N : \tau'$, and then by weakening on reverse (A' is not needed anymore) and eventually weakening back to A we obtain

$$\mathcal{D}' \rightsquigarrow A \vdash M : \tau.$$

Then by hypothesis there is a renaming R such that $R(A) = A$ and $\sigma' = \mathcal{D}(x) = \mathcal{D}'(x) = R(\sigma)$. Let's restrict R to what is strictly necessary, i.e. we may take $\mathrm{SUPP}(R) \subseteq \mathrm{FTV}(\sigma)$. As $\mathrm{FTV}(A') \cap \mathrm{FTV}(\{x : \sigma\}) \subseteq \mathrm{FTV}(A)$ we then have that $R(A, A') = A, A'$. If we look up in $\mathcal{D}$ the subderivation ending in N' we thus finally obtain

$$A, A', R(\{x : \sigma\}), E \vdash N' : \tau'',$$

for some τ''.

Now let's check number 5. $\mathrm{DOM}(\hat{A}) \cap (\mathrm{DOM}(A) \cup \mathrm{BV}(B))$ implies both that A and $\hat{A}$ are compatible and that $\mathrm{BV}(B) \cap \mathrm{DOM}(A, \hat{A}) = \emptyset$. Take any M and A', with $\mathrm{FV}(M) \subseteq \mathrm{DOM}(A, \hat{A}, B, A')$, $\mathrm{DOM}(A') \cap (\mathrm{DOM}(A, \hat{A}) \cup \mathrm{BHV}(C[M])) = \emptyset$, and $\mathrm{FTV}(A') \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A, \hat{A})$. Let us check the two properties.

1. Let σ be a type such that $A, \hat{A}, A', B \vdash_{\mathbf{F}} M : \sigma$. Using the fact that $A, C[\ ] \rhd B$, we take A' in definition 3.4.8 as $A'\hat{A}$ here: in fact the hypotheses on $\hat{A}$ and A' together imply that

$$\mathrm{DOM}(A', \hat{A}) \cap (\mathrm{DOM}(A) \cup \mathrm{BHV}(C[\ ])) = \emptyset,$$
$$\mathrm{FTV}(A', \hat{A}) \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A).$$

So by definition there exist a type τ, an environment E with $\mathrm{DOM}(E) = \mathrm{BHV}(C[\ ]) \setminus \mathrm{DOM}(B)$ and a derivation containing both

$$A', \hat{A}, A \vdash C[M] : \tau, \quad A', \hat{A}, A, B, E \vdash M : \sigma.$$

2. Let $\mathcal{D}$ be a derivation that contains $A, \hat{A}, A' \vdash C[M] : \tau$. Again by taking A' in the definition to be $\hat{A}, A'$ here, there must be a type σ, an environment E with the right domain and a renaming R that fixes $A, \hat{A}, A'$ such that $\mathcal{D}$ contains

$$A, \hat{A}, A', R(B), E \vdash M : \sigma.$$

So we conclude $A \cup \hat{A}, C[\ ] \rhd B$. □

Lemma 3.4.10 (properties of ►). *► enjoys the following properties:*

1. *if A induces the invariant type assumption $x : \sigma$ in M and if σ is not a $\forall$-type, or else if* $\mathrm{FTV}(\sigma) \subseteq \mathrm{FTV}(A)$*, then* $\mathrm{RAN}(A) \blacktriangleright \{\sigma\}$,
2. $\mathbb{X} \blacktriangleright \mathbb{Y} \implies \forall R$ *renaming* $: R(\mathbb{X}) \blacktriangleright R(\mathbb{Y})$,
3. $\mathbb{X} \blacktriangleright \mathbb{Y} \implies \mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}$,
4. $\mathbb{X} \blacktriangleright \mathbb{Y}_1 \cup \mathbb{Y}_2 \implies \mathbb{X} \blacktriangleright \mathbb{Y}_1$,
5. $\mathbb{X}_1 \blacktriangleright \mathbb{Y}$, $\mathrm{FTV}(\mathbb{X}_2) \cap \mathrm{FTV}(\mathbb{Y}) \subseteq \mathrm{FTV}(\mathbb{X}_1) \implies \mathbb{X}_1 \cup \mathbb{X}_2 \blacktriangleright \mathbb{Y}$,
6. $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}$, $\mathbb{X} \cup \mathbb{Y} \blacktriangleright \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z} \implies \mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$,
7. $\mathbb{X} \blacktriangleright \mathbb{Y}_1$, $\mathbb{X} \blacktriangleright \mathbb{Y}_2$, $\mathrm{FTV}(\mathbb{Y}_1) \cap \mathrm{FTV}(\mathbb{Y}_2) \subseteq \mathrm{FTV}(\mathbb{X}) \implies \mathbb{X} \blacktriangleright \mathbb{Y}_1 \cup \mathbb{Y}_2$.

Proof. Let's first see the trivial ones.

2. Let Π_R, where R is a renaming, be the algorithm that taken a function Ψ' defined on environments returns the function defined by $\Pi_R(\Psi')(B) := R(\Psi(R^{-1}(B)))$. Then if Ψ' is a witness of $\mathbb{X} \blacktriangleright \mathbb{Y}$, we may easily see by property 2 of $\rhd$ that $\Psi = \Pi_R(\Psi')$ is a witness of $R(\mathbb{X}) \blacktriangleright R(\mathbb{Y})$.

4. The condition "any environment B with $\mathrm{RAN}(B) \subseteq \mathbb{Y}_1 \cup \mathbb{Y}_2$" clearly implies "any B with $\mathrm{RAN}(B) \subseteq \mathbb{Y}_1$". We can take the same witness.

5. If Ψ is a witness of $\mathbb{X}_1 \blacktriangleright \mathbb{Y}$. Take any B with $\mathrm{RAN}(B) \subseteq \mathbb{Y}$. Then $\Psi(B) = (A, C[\ ])$ is such that $\mathrm{RAN}(A) \subseteq \mathbb{X}_1 \subseteq \mathbb{X}_1 \cup \mathbb{X}_2$, $\mathrm{FTV}(B) \cap \mathrm{FTV}(\mathbb{X}_1 \cup \mathbb{X}_2) \subseteq \mathrm{FTV}(A)$ because

$$\mathrm{FTV}(\mathbb{X}_1) \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(B) \cap (\mathrm{FTV}(\mathbb{X}_1) \cap \mathrm{FTV}(\mathbb{Y})) \subseteq \mathrm{FTV}(B) \cap \mathrm{FTV}(\mathbb{X}_1),$$

and finally $A, C[\ ] \rhd B$. We can take the same witness for $\mathbb{X}_1 \cup \mathbb{X}_2 \blacktriangleright \mathbb{Y}$.

Now with the more complex ones. Given $X, Y, Z \subseteq \mathcal{V}$ subsets of term variables, we denote by $R_{X,Y,Z}$ a renaming built in some algorithmic manner with $\mathrm{SUPP}(R) = (X \cap Y) \setminus Z$ and with $R((X \cap Y) \setminus Z)$ completely outside $X \cup Y \cup Z$. Let $R_{X,Y} := R_{X,Y,\emptyset}$.

1. This completes the core tool we will use for the proof. By property 1 of $\rhd$ we know we can build $C[\ ]$ such that $A, C[\ ] \rhd \{x : \sigma\}$. Now in order to build the witness let's distinguish between the two cases. Note that a non empty environment B with $\mathrm{RAN}(B) \subseteq \{\sigma\}$ means that B is of the form $B = \{y_1 : \sigma, \ldots, y_n : \sigma\}$ with $n \geq 1$.
Suppose σ is not quantified. Then we define

$$\Psi(B) := (R(A), R(C)[C'[\ ]]),$$

where $R = R_{(\mathrm{DOM}(A) \cup \mathrm{V}(C[\ ])), \mathrm{DOM}(B)}$ and

$$C'[\ ] := (\lambda y_1.(\lambda y_2 \ldots (\lambda.y_n.\Box)R(x) \ldots)R(x))R(x).$$

Then Ψ is a valid witness. R is needed to avoid variable collision between $\vec{y}$ and the variables already mentioned in A and $C[\ ]$, so that the initial conditions of the definition of $R(A), R(C)[C'[\ ]] \rhd B$ are met. Let's take a term M' and an environment A' as usual, and use the fact that $R(A), R(C[\ ]) \rhd \{R(x) : \sigma\}$.

   (a) If σ' is such that $A', R(A), B \vdash_{\mathbf{F}} M' : \sigma'$, then from it we can derive

$$A', R(A), R(x) : \sigma \vdash C'[M'] : \sigma'.$$

   Now let us feed $C'[M]$ and A' to the definition of $R(A), R(C[\ ]) \rhd \{R(x) : \sigma\}$: we thus have τ, E with $\mathrm{DOM}(E) = \mathrm{BHV}(R(C)[\ ]) \setminus \{R(x)\}$ and a derivation containing

$$A', R(A) \vdash R(C)[C'[M']] : \tau$$

   and

$$A', R(A), R(x) : \sigma, E \vdash C'[M'] : \sigma'.$$

We can safely substitute the subderivation leading to the sequent above with the one we had derived from $A', R(A), B \vdash_{\mathbf{F}} M' : \sigma'$, weakened to add $E, R(x) : \sigma$, and indeed

$$\mathrm{DOM}(E, R(x) : \sigma) = \mathrm{BHV}(R(C)[C'[\ ]]) \setminus \mathrm{DOM}(B).$$

This part does not need to make use of the hypothesis on σ.

(b) Let there be a derivation $\mathcal{D}$ containing

$$R(A), A' \vdash R(C)[C'[M']] : \tau.$$

By applying $R(A), R(C)[\ ] \triangleright \{R(x) : \sigma\}$ as before we get σ', E with $\mathrm{DOM}(E) = \mathrm{BHV}(R(C)[\ ]) \setminus \{x\}$ and a type-variable renaming R' with $R'(R(A), A') = R(A), A'$ such that $\mathcal{D}$ contains also

$$R(A), A', R(x) : R'(\sigma), E \vdash C'[M'] : \sigma'.$$

Now if we go up in this derivation and look for the (var) rules that introduce all the $R(x)$, we see that they cannot be followed by any (ins) or (gen). The first one because $R'(\sigma)$ is not a $\forall$-type. The second one because of the $\sim$ convention. So the $R(x)$s arrive to (app) still typed with $R'(\sigma)$, and so the abstractions must be made to accept them, i.e. all the y_is must have final derived type $R'(\sigma)$, and so $\mathcal{D}$ contains also

$$R(A), A', R(x) : R'(\sigma), E, R'(B) \vdash M' : \sigma''$$

and as before

$$\mathrm{DOM}(E, R(x) : \sigma) = \mathrm{BHV}(R(C)[C'[\ ]]) \setminus \mathrm{DOM}(B).$$

Now suppose $\mathrm{FTV}(\tau) \subseteq \mathrm{FTV}(A)$. We then define

$$\Psi(B) := (R(A), C_1[C_2[\ldots C_n[\ ]\ldots]])$$

where, given R the same as in the previous case, R_i is such that it renames $\mathrm{BV}(R(C[\ ]))$ to fresh variables leaving all other as it is (remember that plugging in the hole of a context captures free variables), except for $R(x)$ which goes renamed with y_i, and $C_i[\ ] := R_i(R(C))[\ ]$. Now, as before, let's take a term M' and an environment A'. Let's denote $\vec{C}[\ ] := C_1[\ldots C_n[\ ]\ldots]$. For every i we have by applying R_i to $R(A), R(C)[\ ] \triangleright \{R(x) : \sigma\}$, that $R(A), C_i[\ ] \triangleright \{y_i : \sigma\}$.

(a) If σ' is such that $A', R(A), B \vdash_{\mathbf{F}} M' : \sigma'$. Then by induction, applying the hypotheses on C_i one at a time, we end up with a type τ, an environment $E_1, \ldots, E_n$ with $\mathrm{DOM}(E_i) = \mathrm{BHV}(C_i[\ ]) \setminus \mathrm{DOM}\{y_i\}$ (and thus $\mathrm{DOM}(\vec{E}) = \mathrm{BHV}(\vec{C}[\ ]) \setminus \mathrm{DOM}(B)$) and a derivation containing

$$A', R(A) \vdash \vec{C}[M] : \tau, \quad A', R(A), B, \vec{E} \vdash M' : \sigma'.$$

(b) Let there be a derivation $\mathcal{D}$ containing

$$R(A), A' \vdash \vec{C}[M'] : \tau.$$

We apply an induction in reversed order, so that there exists a type σ', $\vec{E}$ as before, and a type-variable renaming R' so that in the end we have a derivation containing

$$R(A), A', R'(B), \vec{E} \vdash M' : \sigma'.$$

R' is the effect of the first application of the hypotheses only (the one concerning $C_1[\ ]$): from then on the type σ becomes part of the A' of the definition, and so must be preserved.

In applying the induction hypotheses in to set as the A' of the definition the union of A' with what's left of B (and with what we have got as $\vec{E}$ so far). Here fits the hypothesis on $\mathrm{FTV}(\sigma)$, as given that, the hypothesis $\mathrm{FTV}(A') \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A)$ always holds true.

3. Let Ψ be a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}$. If A is an environment let's denote by $A \cap \mathbb{Y} := \{\, (x : \tau) \in A \mid \tau \in \mathbb{Y} \,\}$. Let Ψ' be the following algorithm:

Require: B with $\mathrm{RAN}(B) \subseteq \mathbb{X} \cup \mathbb{Y}$;

1: $(A, C[\ ]) \leftarrow \Psi(B \cap \mathbb{Y})$;
2: $B' \leftarrow B \setminus (B \cap \mathbb{Y})$;
3: $(Y, Z) \leftarrow (\mathrm{DOM}(A) \cup \mathrm{V}(C[\ ]), \mathrm{DOM}(B'))$;
4: $(A', C'[\ ]) \leftarrow (R_{Y,Z}(A), R_{Y,Z}(C)[\ ])$;
5: $A'' \leftarrow A' \cup B'$;
6: **return** $(A'', C'[\ ])$.

Note that this algorithm definition depends only on Ψ, so we can build an algorithm Π_{self} that taken a function Ψ from environments to environments and contexts returns the function defined by the above algorithm. So we have an effective computable way of bringing a witness of $\mathbb{X} \blacktriangleright \mathbb{Y}$ to what we will now see is a witness of $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}$. In particular we are applying $\Psi' = \Pi_{\mathrm{self}}$ to B with $\mathrm{RAN}(B) \subseteq \mathbb{X} \cup \mathbb{Y}$. We easily check that $\mathrm{RAN}(A'') \subseteq \mathbb{X}$ and $\mathrm{FTV}(B) \cap \mathrm{FTV}(\mathbb{X}) \subseteq \mathrm{FTV}(A'')$. We must show that $A'', C'[\ ] \rhd B$.
As $A, C[\ ] \rhd B \cap \mathbb{Y}$, and $R_{Y,Z}(B \cap \mathbb{Y}) = B \cap \mathbb{Y}$, we get using property 2 of $\rhd$ that $A', C'[\ ] \rhd B \cap \mathbb{Y}$. now by the fact that Ψ is a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}$ we have already that

$$\mathrm{FTV}(B \cap \mathbb{Y}) \cap \mathrm{FTV}(\mathbb{X}) \subseteq \mathrm{FTV}(A) = \mathrm{FTV}(A'),$$

but then again also $\mathrm{RAN}(B') \subseteq \mathbb{X}$, so that

$$\mathrm{FTV}(B \cap \mathbb{Y}) \cap \mathrm{FTV}(B') \subseteq \mathrm{FTV}(A')$$

and by property 5 of $\rhd$ (where we use also the way in which we have renamed the variables) we get $A' \cup B', C' \ \rhd B \cap \mathbb{Y}$. Then property 3 does the trick and gets us

$$A' \cup B', C' \ \rhd B \cap \mathbb{Y} \cup B'$$

which is exactly $A'', C'[\ ] \rhd B$.

6. Let Ψ_1 be a witness of the first statement, and Ψ_2 a witness of the second one. We define Ψ to be the following algorithm:

   Require: B with $\mathrm{RAN}(B) \subseteq \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$;

   1: $(A'_2, C_2[\ ]) \leftarrow \Psi_2(B)$;
   2: $(A'_1, C'_1[\ ]) \leftarrow \Psi_1(A'_2)$;
   3: $(X, Y, Z) \leftarrow (\mathrm{DOM}(A'_1) \cup \mathrm{BV}(C'_1[\ ]), \mathrm{BV}(C_2[\ ]), \mathrm{DOM}(A'_2))$;
   4: $(A_1, C_1[\ ]) \leftarrow (R_{X,Y,Z}(A'_1), R_{X,Y,Z}(C'_1[\ ]))$;
   5: **return** $(A_1, C_1[C_2[\ ]])$.

   Again we give a name to the algorithm which in turn yields the above algorithm from input Ψ_1, Ψ_2. We will call it Π_{chain}.

   The proof that $\Pi_{\mathrm{chain}}(\Psi_1, \Psi_2)$ is a witness for $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$ gets technical beyond the aim of this work. The proof can be found in the already mentioned article by Wells [Wel99].

7. Let $\mathbb{X} \blacktriangleright \mathbb{Y}_1$ and $\mathbb{X} \blacktriangleright \mathbb{Y}_2$ with $\mathrm{FTV}(\mathbb{Y}_1) \cap \mathrm{FTV}(\mathbb{Y}_2) \subseteq \mathrm{FTV}(\mathbb{X})$. By properties 5 and then 3 above we get $\mathbb{X} \cup \mathbb{Y}_1 \blacktriangleright \mathbb{Y}_2$ and then $\mathbb{X} \cup \mathbb{Y}_1 \blacktriangleright \mathbb{X} \cup \mathbb{Y}_1 \cup \mathbb{Y}_2$. note that if Ψ_1 was a witness of the initial statement, it is also of the second and then $\Pi_{\mathrm{self}}(\Psi)$ is a witness of the latter. Combining it with $\mathbb{X} \blacktriangleright \mathbb{Y}_1$ (which by property 3 again yields $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}_1$) and property 6 we get $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}_1 \cup \mathbb{Y}_2$. If Ψ_2 was a witness of the second initial statement, this one has witness $\Pi_{\mathrm{chain}}(\Pi_{\mathrm{self}}(\Psi_1), \Pi_{\mathrm{self}}(\Psi_2))$. The last statement reduces to $\mathbb{X} \blacktriangleright \mathbb{Y}_1 \cup \mathbb{Y}_2$ with the same witness after appplying property 4. Thus we define an algorithm that produces a witness for this statement given witnesses for the two premises, and it is:

$$\Pi_{\mathrm{alt}}(\Psi_1, \Psi_2) := \Pi_{\mathrm{chain}}(\Pi_{\mathrm{self}}(\Psi_1), \Pi_{\mathrm{self}}(\Psi_2)).$$

□

Now we need one more result to be able to chain together infinite chains of sets of types, without loosing the algorithmic content given by witnesses.

Lemma 3.4.11. *Let $\mathbb{X}, \mathbb{Y} \subseteq \mathbb{T}$, with $\mathbb{Y}$ approximated by the sequence $\mathbb{Y}_i$ of decidable sets, i.e. $\mathbb{Y}_i \subseteq \mathbb{Y}_{i+1}$ and $\mathbb{Y} = \bigcup_i \mathbb{Y}_i$. Let Θ be computable function that accepts as input a function and an integer so that if Ψ is a witness of $\mathbb{X} \blacktriangleright \mathbb{Y}_i$ then $\Theta(\Psi, i)$ is a witness of $\mathbb{X} \blacktriangleright \mathbb{Y}_{i+1}$. Then if $\mathbb{X} \blacktriangleright \mathbb{Y}_1$ we have $\mathbb{X} \blacktriangleright \mathbb{Y}$.*

Proof. Define the following recursive algorithm $\Psi'(\Phi, i, B)$:

Require: Φ function with input environment and output environment and context, i integer, B environment with $\mathrm{RAN}(B) \subseteq \mathbb{Y}$;

1: **if** $\mathrm{RAN}(B) \subseteq \mathbb{Y}_i$ **then return** $\Phi(B)$;

2: **else return** $\Psi'(\Theta(\Phi, i), i+1, B)$.

Then take a witness Ψ_1 of $\mathbb{X} \blacktriangleright \mathbb{Y}$ and define $\Psi(B) := \Psi'(\Psi_1, 1, B)$. Because $\mathbb{Y}$ is the limit of $\mathbb{Y}_i$ the algorithm always terminates at some step k with $\mathrm{RAN}(B) \subseteq \mathbb{Y}_k$. It is easy then to show that Φ at that step is a witness of $\mathbb{X} \blacktriangleright \mathbb{Y}_k$ and so the result follows. □

Exhausting the types

Now the aim is to build one step at a time the statement $\emptyset \blacktriangleright \mathbb{T}$.

Definition 3.4.12 (height and parheight). We define a measure on types that ignores quantifiers. The *height* $\mathrm{h}(\tau)$ is the depth of $(\tau)_{\mathbf{S}}$ plus one, i.e.

$$\begin{aligned} \mathrm{h}(\alpha) &:= 1, \\ \mathrm{h}(\tau_1 \to \tau_2) &:= 1 + \max(\mathrm{h}(\tau_1), \mathrm{h}(\tau_2)), \\ \mathrm{h}(\forall \alpha.\tau) &:= \mathrm{h}(\tau) \end{aligned}$$

If we write a type τ in the form

$$\tau = \overrightarrow{\forall \alpha}_1.\rho_1 \to \ldots \to \overrightarrow{\forall \alpha}_k.\rho_k \to \overrightarrow{\forall \alpha}_{k+1}.\beta$$

then we define the *parheight* of τ:

$$\mathrm{ph}(\tau) := \max\{\, \mathrm{h}(\rho_1), \ldots, \mathrm{h}(\rho_k), 0 \,\}.$$

Definition 3.4.13. We now will define the sets with which we will climb all the types.

$$\begin{aligned} \mathbb{B} &:= \{\bot\} \cup \{\, \alpha \to \alpha \mid \alpha \in \mathbb{V} \} \cup \mathbb{V}, \\ \mathbb{U} &:= \{\, \forall.\tau \mid \mathrm{BTV}(\tau) = \emptyset \,\}, \\ \mathbb{C} &:= \{\, \tau \mid \mathrm{FTV}(\tau) = \emptyset \,\}, \\ \mathbb{T}(k) &:= \{\, \tau \mid \mathrm{ph}(\tau) \leq k \,\}, \\ \mathbb{U}(k) &:= \mathbb{U} \cap \mathbb{T}(k), \\ \mathbb{C}(k) &:= \mathbb{C} \cap \mathbb{T}(k). \end{aligned}$$

Lemma 3.4.14. *There is a λ-term J such that the empty type environment induces the invariant type assumption $v : \bot$ and $x : \alpha \to \alpha$.*

Proof. (sketch) The term is

$$J = \lambda v.(\lambda y \lambda z.v\,(y\,y)\,(y\,z))(\lambda x.\underline{\texttt{true}}\,x\,(x\,(x\,v)))\,(\lambda w.w\,w).$$

This term is typable. Let's quickly check its three main components. We have

$$v : \bot, y : \texttt{Int}, z : \bot \to \bot \vdash v\,(y\,y)\,(y\,z) : \bot$$

if we instantiate the rightmost y as $\texttt{Int}_{\alpha\to\alpha}$, the central one as $\texttt{Int}_\alpha$, the rightmost as $\texttt{Int}_\bot$, and finally v as $\texttt{Int}_\alpha \to (\bot \to \bot) \to \bot$. We then abstract z and y. Then we have

$$v : \bot, x : \alpha \to \alpha \vdash \underline{\texttt{true}}\,x\,(x\,(x\,v)) : \alpha \to \alpha$$

if we let v be of type α. Next with an (abs) and a (gen) we have:

$$v : \bot \vdash \lambda x.\underline{\texttt{true}}\,x\,(x\,(x\,v)) : \texttt{Int}\,.$$

The last subterm is easily typable with

$$\vdash \lambda w.w\,w : \bot \to \bot).$$

So in the end we have

$$\mathcal{D} \rightsquigarrow \vdash J : \bot \to \bot$$

and in effect $\mathcal{D}(x) = \alpha \to \alpha$ and $\mathcal{D}(v) = \bot$.

The fact that every other typing gives the same type assumptions modulo renaming is shown by interaction of the various subterms. Both y and w must be quantified or they could not be applied to themselves, and in any case in order to be applied to themselves they need to have the quantified variable at the end of the left-going path in their type. Then because y must have the final type of the second subterm (from how we have divided them above) which is an abstraction, it must have depth at least one. Going on like this we arrive to the conclusion of having as invariant type assumptions for J the three depicted in the typing above for y, x and v. □

Corollary 3.4.15. $\emptyset \blacktriangleright \mathbb{B}$.

Proof. By property 1 we already have that $\emptyset \blacktriangleright \{\alpha \to \alpha\}$ and $\emptyset \blacktriangleright \{\bot\}$, and then by 7 $\emptyset \blacktriangleright \{\bot, \alpha \to \alpha\}$. Then we trivially get from a term an invariant type assumption using the types we have already induced. In fact the environment $v : \bot, x : \alpha \to \alpha$ trivially induces the invariant type assumption $y : \alpha$ in $(\lambda y.y)\,(x\,v)$. So reusing the first property we have $\{\bot, \alpha \to \alpha\} \blacktriangleright \{\alpha\}$, which by 4 becomes $\{\bot, \alpha \to \alpha\} \blacktriangleright \{\bot, \alpha, \alpha \to \alpha\}$. Combining by property 6

$$\emptyset \blacktriangleright \{\bot\} \cup \{\alpha \to \alpha\}, \quad \blacktriangleright \{\bot\} \cup \{\alpha \to \alpha\} \blacktriangleright \{\bot\} \cup \{\alpha \to \alpha\} \cup \{\alpha\}$$

we obtain $\emptyset \blacktriangleright \{\bot, \alpha, \alpha \to \alpha\}$.

We now have to extend the result to all $\mathbb{B}$. Let α_i be a decidable enumeration of $\mathbb{V}$. Then we denote

$$\mathbb{B}_i := \{\bot\} \cup \bigcup_{j=1}^{i} \{\, \alpha_j, \alpha_j \to \alpha_j \,\}.$$

Clearly $\bigcup_i \mathbb{B}_i = \mathbb{B}$. Say Ψ_1 is a witness of $\emptyset \blacktriangleright \mathbb{B}_1$, obtained by eventually renaming the above statement. Define $\Theta(\Psi, i) := \Pi_{\text{alt}}(\Psi, \Pi_{1,i+1}(\Psi_1))$ where $R_{i,j} := [\alpha_i/\alpha_j, \alpha_j/\alpha_i]$. Then clearly if Ψ is a witness of $\emptyset \blacktriangleright \mathbb{B}_i$ then $\Theta(\Psi, i)$ is a witness of $\emptyset \blacktriangleright \mathbb{B}_{i+1}$. So, using lemma 3.4.11, we have $\emptyset \blacktriangleright \mathbb{B}$. □

We now begin to cover up all the universal type, i.e. $\mathbb{U}$, using the following proposition of which we will not present the proof.

Lemma 3.4.16. *Given a type τ in $\mathbb{U}(k+1)$, we can build a term M_τ and an environment A_τ so that $\text{RAN}(A_\tau) \subseteq \mathbb{U}(k) \cup \mathbb{B}$ and they together induce the invariant type assumption $x : \tau$.*

Corollary 3.4.17. $\mathbb{B} \blacktriangleright \mathbb{U} \cup \mathbb{B}$.

Proof. Take $\tau \in \mathbb{U}(k+1)$: by property 1 we gain immediately $\text{RAN}(A_\tau) \blacktriangleright \{\tau\}$. Because $\mathbb{B}$ covers all free variables, we can then apply 5 and have $\mathbb{U}(k) \cup \mathbb{B} \blacktriangleright \{\tau\}$. Say $\Omega(\tau)$ is the witness we algorithmically build from A_τ and M_τ. $\Omega(\tau$ is independent of k: it is a witness for $\mathbb{U}(k) \cup \mathbb{B} \blacktriangleright \{\tau\}$ for any k such that $\tau \in \mathbb{U}(k)$. Consider an enumeration τ_j of $\mathbb{U}$ such that $\text{ph}(\tau_j) \leq \text{ph}(\tau_{j+1})$ and the sequence of sets $\mathbb{U}_i := \bigcup_{j \leq i} \{\tau_j\}$.

Given τ_{i+1} and k such that $\tau_{i+1} \in \mathbb{U}(k+1)$ we have that $\mathbb{U}(k) \subseteq \mathbb{U}_i$, and as $\text{FTV}(\mathbb{U}_i) = \emptyset$, we may apply 5 and have that $\Omega(\tau_{i+1})$ is a witness of $\mathbb{B} \cup \mathbb{U}_i \blacktriangleright \{\tau_{i+1}\}$. Note that necessarily $\tau_1 = \bot \in \mathbb{B}$, the unique closed type of parheight 0, so we have a witness Ψ_1 for $\mathbb{B} \blacktriangleright \mathbb{B} \cup \mathbb{U}_1$. As step algorithm we take $\Theta(\Psi, i) := \Pi_{\text{chain}}(\Psi, \Pi_{\text{self}}(\Omega(\tau_{i+1})))$. In fact if Ψ is a witness of $\mathbb{B} \blacktriangleright \mathbb{B} \cup \mathbb{U}_i$, being that $\Pi_{\text{self}}(\Omega(\tau_{i+1}))$ is a witness of $\mathbb{B} \cup \mathbb{U}_i \blacktriangleright \mathbb{B} \cup \mathbb{U}_i \cup \{\tau_{i+1}\}$ we have that $\Theta(\Psi, i)$ is indeed a witness of $\mathbb{B} \blacktriangleright \mathbb{B} \cup \mathbb{U}_{i+1}$. Applying lemma 3.4.11 we get what needed. □

Now it's time for the closed types, which have a similar lemma which gives rise to a similar corollary.

Lemma 3.4.18. *Given $\tau \in \mathbb{C}(k+1)$ there exist a term N_τ and a type environment B_τ with $\text{RAN}(B_\tau) \subseteq \mathbb{B} \cup \mathbb{U}(2) \cup \mathbb{C}(k)$ that induce the invariant type assumption $x : \tau$.*

Corollary 3.4.19. $\mathbb{B} \cup \mathbb{U} \blacktriangleright \mathbb{B} \cup \mathbb{C}$.

Now we have all we need to tackle down the whole set of types.

Lemma 3.4.20. *For every type τ there is an environment a term P_τ and an environment E_τ with* $\mathrm{RAN}(E) \subseteq \mathbb{B} \cup \mathbb{C}$ *such that they induce the invariant type assumption $x : \tau$.*

Proof. Take $\sigma = \forall.\alpha_1 \to \ldots \to \alpha_n \to \tau$ where $\vec{\alpha} = \mathrm{FTV}(\tau)$. Clearly we have $\sigma \in \mathbb{C}$. Now define

$$E_\tau := \overrightarrow{b : \alpha}, c : \sigma, d : \forall.(\gamma \to \delta_!) \to (\delta_2 \to \gamma) \to \delta_3$$

and $M_\tau := (d\,(\lambda x.x)\,(c\,\vec{b}))$. In order for c to accept all the b_is, its type must be instantiated with a substitution such that $T(\alpha_i) = \alpha_i$, and so $T(\tau) = \tau$, and thus $T(\sigma) = \alpha_1 \to \ldots \to \alpha_n \to T(\beta) \to \tau$. So $(c\,\vec{b})$ must be given type $T(\beta) \to \tau$. Checking the way $E_\tau(c)$ must be instantiated, we then see that γ needs to become τ. So the identity in which x is bound needs to be accepted by $\tau \to \rho$ for some ρ, and so the type assumed for x before it's abstracted away must necessarily be τ. □

Corollary 3.4.21. $\mathbb{B} \cup \mathbb{U} \blacktriangleright \mathbb{T}_{\mathbf{F}}$.

Proof. The proof proceeds like the two corollaries above. □

At last come the results we sought.

Theorem 3.4.22. $\emptyset \blacktriangleright \mathbb{T}_{\mathbf{F}}$.

Proof. It is given by subsequent uses of property 6: $\emptyset \blacktriangleright \mathbb{B}$ and $\mathbb{B} \blacktriangleright \mathbb{B} \cup \mathbb{U}$ together imply $\emptyset \blacktriangleright \mathbb{B} \cup \mathbb{U}$, this with $\mathbb{B} \cup \mathbb{U} \blacktriangleright \mathbb{B} \cup \mathbb{U} \cup \mathbb{C}$ yield $\emptyset \blacktriangleright \mathbb{B} \cup \mathbb{U} \cup \mathbb{C}$, and finally this with the last result above give $\emptyset \blacktriangleright \mathbb{T}_{\mathbf{F}}$. □

Theorem 3.4.23 (TC ≤ TYP). $\mathrm{TC}_{\mathbf{F}}$ *is reducible to* $\mathrm{TYP}_{\mathbf{F}}$.

Proof. Let $A \vdash M : \tau$ be an instance of TC. Choose z fresh and consider the environment $B = A \cup \{z : \tau \to \bot\}$, and build $C[\ ]$ from $\Psi(B) = (\emptyset, C[\ ])$ where Ψ is a witness of $\emptyset \blacktriangleright \mathbb{T}_{\mathbf{F}}$, so that $\emptyset, C[\ ] \rhd B$. The claim is that $A \vdash_{\mathbf{F}} M : \tau$ if and only if $C[z\,M]$ is typable. The preliminary hypotheses for $z\,M$ together with $\emptyset$ all trivially hold.

Suppose $A \vdash_{\mathbf{F}} M : \tau$. Then we can easily derive $B \vdash z\,M : \bot$. By definition 3.4.7 of $\rhd$, there is a derivation ending in $\emptyset \vdash C[zM] : \sigma$.

Suppose now that $C[z\,M]$ is typable, i.e. $A' \vdash C[z\,M] : \sigma$. We can assume A' is empty as by definition $\mathrm{FTV}(C[\ ]) \subseteq \emptyset$ and all the free variables of M get captured by C. So by definition of $\rhd$ the same derivation contains the sequent

$$R(B), E \vdash z\,M : \rho$$

for some ρ and some type variable renaming R. Then again, $\mathrm{FTV}(z\,M)$ is inside $R(B)$, so by weakening on reverse we have a derivation of $R(A), z : R(\tau) \to \bot \vdash z\,M$. Also take the (ins) before

(gen) form of it. Somewhere up in the derivation there must be an (app) rule of the form

$$\dfrac{\dfrac{}{R(B) \vdash z : R(\tau) \to \bot}\ (\text{var}) \qquad R(A), z : R(\tau) \to \bot \vdash M : R(\tau)}{R(A), z : R(\tau) \to \bot \vdash z, M : \bot}\ (\text{app})$$

where we see there can't be any (ins) after having introduced z, nor there can be any (gen) before applying it. Again we can apply weakening on reverse on the derivation on the right and have

$$R(A) \vdash M : R(\tau).$$

Now applying R^{-1} yields the desired result:

$$A \vdash M : R(\tau).$$

□

Corollary 3.4.24. $\mathrm{TC}_{\mathbf{F}}$ *and* $\mathrm{TYP}_{\mathbf{F}}$ *are equivalent problems and both undecidable.*

Chapter 4

Light logics and λ-calculus

Linear logic was discovered by Girard in 1987 [Gir87]. It has shed a completely new light on topics which were considered granitic.

What can be considered the main idea behind it? Say we have proved the formula $F \to G$ in classical logic. Fact is, we cannot say anything about the true use of the hypothesis F in proving the thesis G. We are used to the fact that for example if we have proved G, we may as well say we prove $F \to G$, though F had nothing to do with G. Or on the converse, we may have used the hypothesis F a hundred times in different places of the proof, but nevertheless we understately write just $F \to G$. The rules behind this are the so called *structural rules*: *weakening*, i.e. adding hypotheses, and *contraction*, i.e. merging two repetitions of the same hypothesis. But what if we do not allow these rules as they are formulated usually?

Because of its parallelism and application to λ-calculus, we will here always stick to the *intuitionistic* fragment of the logic systems we speak of. So Our sequents will be asymmetrical, with only one formula on the right.

4.1 An introduction to LL

We will now briefly outline linear logic. Structural rules are being restricted, and as the implication takes on a completely new meaning, it is replaced by a new symbol, *linear implication* $\multimap$. We want $F \multimap G$ to mean that we may prove G using exactly once, no more no less, occurrences of the hypothesis F. We may already see a functional meaning behind this connective: as $\tau \to \sigma$ meant vaguely "take an input of type τ and use it to compute something of type σ", here $\sigma \multimap \tau$ will mean "take an input of type τ and use it exactly once to compute σ".

Clearly having only this can hardly be much expressive. So in order to recuperate the power

structural rules gave to us, we reintroduce them in a controlled way. We will use *modalities*, which can be viewed as a status mark of the formula. $!A$ (*bang* A) will thus mark an hypothesis that acts like in the intuitionistic case: as it stands for a place that may be occupied by none, one or multiple copies of itself. In short words, we let weakening and contraction work only on hypotheses (i.e. formulas on the left of $\vdash$) marked in this way. And when can we say a formula may act in this way? Only if it was in turn obtained by hypotheses marked in the same manner, otherwise by transitivity of the implication it we still would not now how many times this unmarked hypothesis gets used. ! and its dual ? are called *modalities* or *exponential*. The breakthrough with linear logic can be brought down to having decomposed the implication in two distinct operations: one is raising the hypothesis to a new status, and the other of effectively using the hypothesis in deriving the thesis.

This brings a revolution in logic as it used to be. In fact the restriction of structural rules breaks down the symmetry of the usual connectives, and we discover that in fact there are two varsions of each classic connective and constant, different in the sense that they need completely different proofs to be proved. Thus $\wedge$ splits into $\otimes$ (*multiplicative disjunction*) and $\&$ (*additive disjunction*). Proving $F \otimes G$ means "we have used some of the hypotheses to prove F and all the other ones to prove G", while proving $F \,\&\, G$ is "we have used all the hypotheses in proving F and all the hypotheses again for G". The dual $\vee$ in the same manner splits in $⅋$ (*multiplicative conjunction*) and in $\oplus$ (*additive conjunction*). Again the two have a different meaning: $F ⅋ G$ is "we have proved under all the hypotheses that excluding one proves the other", while $F \oplus G$ means "we have used the hypotheses to prove F" or else "we have used the hypotheses to prove G". So for example we do not have $F \multimap F \otimes F$ but we have $F \multimap F \,\&\, F$, we have $F \multimap F \oplus G$ but we do not have $F \multimap F ⅋ G$. Also the logic constants undergo the same splitting. The truth value can be represented in two ways: $\mathbf{1}$ (*one*), which represents something that is provable without needing hypotheses, "it is true because it is provable", and $\top$ (*top*), which represents something we may always regard as being hypothesis, "it is true because everything that is provable comes from it". The opposites respectively are $\bot$ (*bottom*), from which we can derive any formula, "it is false because otherwise anything would be provable", and $\mathbf{0}$ (*zero*), which comes from proving two opposites, "it is false because it is a contradiction".

Another great revolution is the introduction by linear logic of a *parallel* concept of proof, where anywhere it's possible we transcend the particular order in which we add rules. This idea is carried out by *proof nets*. A proof is not a tree anymore, but a graph. Rules can be viewed locally, without necessarily looking at what's around... almost. Whenever we *have* to consider the environment, a so called *box* will be created. Proof nets enjoy the benefits of cut-elimination, which becomes almost local: the process can be done locally and in more points at the same time, in parallel.

Almost, because when it comes to boxes the operation must regard all the box.

When it comes to typing, linear logic has to be deployed with all its power to have much expressiveness: we have to express the additives in a distinct way from the multiplicatives we are used to (implication, but also pairing, which we have not presented but is easily implemented). This can be cumbersome. So, because it is easier to program, and it does not give rise to problems of expressiveness and completeness, usually a variant of linear logic is adopted: affine logic. The idea is that what we really don't want is not knowing how many times the hypothesis is used in a proof. We accept not knowing if it was used once or not at all. Basically, we have restricted contraction but unrestricted weakening. So $F \multimap G$ means "in proving F we have used G at most once". The main difference is that while retaining control over duplication of hypotheses, the multiplicatives become stronger that the additives: in type assignment it means that we can emulate an additive with a multiplicative.

Because of our interest mainly in λ-calculus, we will be in contact only with the implicational fragment, and in particular in its application to typing discipline. From now on we will identify types with formulas.

4.1.1 AL as a type system

LL poses various programming problems. It appears that additives are necessary to represent functions, so that we need to add to the syntax of the terms *with*, *first* and *second* constructs, that introduce a new type of pair with respect to the classical representation, that have the following typing rules:

$$\frac{A \vdash M_1 : \tau \quad A \vdash M_2 : \tau}{A \vdash M_1 \,\&\, M_2 : \tau \,\&\, \sigma}$$

$$\frac{A \vdash M : \tau \,\&\, \sigma}{A \vdash \mathtt{first}\, M : \tau} \qquad \frac{A \vdash M : \tau \,\&\, \sigma}{A \vdash \mathtt{second}\, M : \sigma}$$

The problem with the usual pair is that it represents $\otimes$, which means that if not banged both the components have to be used, and nothing can be discarded. The usual projection is not applyable anymore, whereas with the & pair we *must* use only one of the componets: the two pairs complements each other. In any case programming becomes harder.

To avoid those problems, we put aside our claim to control weakening, and concentrate on contraction: we permit *unrestricted* weakening. As a consequence multiplicatives become stronger than the additives, so that the additives can be completely represented by the multiplicatives, and thus, in the same way as we have seen in remark 3.2.3, all can be taken down to $\forall$ and $\multimap$. A system based on linear logic with unrestricted weakening is called *affine*. It has a drawback: namely the full logic system loses determinism in cut-elimination, because we can cut two formulas both introduced by weakening and we can choose which side we erase, so that we have two completely

different proofs. However this problem disappears in the intuitionistic case: the formula on the right cannot be obtained by weakening.

Definition 4.1.1 (types of AL). The set of *linear types* are built from $\mathbb{V}$ with the following grammar:

$$\mathbb{T}_{\mathbf{AL}} := \mathbb{V} \mid \mathbb{T}_{\mathbf{AL}} \multimap \mathbb{T}_{\mathbf{AL}} \mid !\mathbb{T}_{\mathbf{AL}}.$$

Depending on the last rule used we call τ a variable, a $\multimap$-*type* (or implication), and a !-type (*bang* or *exponential* or *modal type*). A type which is not exponential is called *linear*

$\mathrm{TV}(\tau)$ and substitutions are defined as for $\mathbb{T}_{\mathbf{S}}$, by ignoring ! and treating $\multimap$ as $\to$. Paths and subterms are defined similarly by introducing a function D on type defined on bang types which gives the argument of !.

Definition 4.1.2 (rules of AL). AL is given by the following set of rules. An additional (and capital) condition is that every variable being introduced in the derivation is considered new. The only way a repetition can be achieved is through contraction. Because of this condition there is no need to check compatibility of environments when merging them: they come form different derivations and thus are disjoint. First the usual rules. As in system **F**, we build in weakening by letting additional environment in the (var) rule.

$$\frac{}{A, x : \tau \vdash x : \tau}\ (\text{var}) \qquad \frac{A, x : \sigma \vdash M : \tau}{A \vdash (\lambda x.M) : \sigma \multimap \tau}\ (\text{abs})$$

$$\frac{A \vdash M : \sigma \multimap \tau \quad B \vdash N : \sigma}{A, B \vdash (M\,N) : \tau}\ (\text{app})$$

Now the two structural rules, *contraction* and *weakening*:

$$\frac{\vec{x} : !\sigma : !\sigma, A \vdash M : \tau}{y : !\sigma, A \vdash M[y/\vec{x}]}\ (\text{con}) \qquad \frac{A \vdash M : \tau}{A, B \vdash M : \tau}\ (\text{weak})$$

where $!A$ is defined by $(!A)(x) := !A(x)$ and $\vec{x} : !\sigma$ means $x_1 : !\sigma, \ldots, x_n : !\sigma$. We require $n \geq 2$[1]. The rules used to handle the modality are *dereliction* and *promotion*:

$$\frac{A, B \vdash M : \tau}{!A, B \vdash M : \tau}\ (\text{der}) \qquad \frac{!A \vdash M : \tau}{!A \vdash M : !\tau}\ (\text{prom})$$

Finally, a rule that handles substitution, *cut*:

$$\frac{\overrightarrow{A \vdash N : \sigma} \quad \overrightarrow{x : \sigma}, B \vdash M : \tau}{\vec{A}, B \vdash M[\overrightarrow{N/x}] : \tau}\ (\text{cut})$$

where $\overrightarrow{A \vdash N : !\sigma}$ means having n derivations of form $A_i \vdash N_i : \sigma_i$.

[1] this requirement is for formal tidiness, rather than for correctness: in fact a 1-ary contraction would correspond to a rule that does nothing, while the 0-ary case would be a special case of weakening.

Remark 4.1.3. Let's discuss some issues about the rules.

Instead of saying that every variable is new, we may equivalently require that every time we merge environments from different derivations (so in the application and cut rules) they are disjoint, not only compatible.

Weakening, cut and contraction can be defined in a "slower" way, the first ones acting on only one assumption and the other on a single pair of variables. The two defintions are equivalent, and there is no complication in grouping the rules as one.

Cut rule is introduced to obtain substitution which is not granted by the other rules. In fact the problem is that we cannot guarantee that a banged type is obtained directly with a promotion: so we do not have any induction hypothesis valid for dereliction, we cannot apply back contraction on the environments after repeatedly applying induction hypothesis, and finally we cannot apply back promotion after substituting one of the type assumptions with a derivation. We can give an alternative system without the cut rule in which cut is integrated in those rules that pose a problem with substitution, so to give:

$$\frac{A \vdash N : !\sigma \quad \vec{x} : !\sigma, B \vdash M : \tau}{A, B \vdash M[N/\vec{x}] : \tau}\ (\text{con}) \qquad \frac{\overrightarrow{A_i \vdash N_i : !\sigma_i} \quad \overrightarrow{x_i : \sigma_i}, B \vdash M : \tau}{\vec{A}, B \vdash M[\overrightarrow{N/x}] : \tau}\ (\text{der})$$

$$\frac{\overrightarrow{A \vdash N : !\sigma} \quad \overrightarrow{x : \sigma} \vdash M\tau}{\vec{A} \vdash M[\overrightarrow{N/x}] : !\tau}\ (\text{prom})$$

where in contraction we mean that in $\overrightarrow{x : !\sigma}$ all the types are equal to $!\sigma$. With the two rule above the standard ones can be emulated by using (var) rules on the left, and substitution lemma (i.e. (cut) rule) can be proved.

Also we may split promotion into two different rules keeping the system equivalent. The two rules, one of which we continue to call promotion, while the other is called *bracket* or *digging*, would be (if we decide to leave out the cut rule):

$$\frac{\overrightarrow{B \vdash N : \sigma} \quad \overrightarrow{x : \sigma} \vdash M : \tau}{\vec{B}, A \vdash M[\overrightarrow{N/x}] : !\tau}\ (\text{prom}) \qquad \frac{!!A, B \vdash M : \tau}{!A, B \vdash M : \tau}\ (\text{brack})$$

This new promotion is obtained by applying first dereliction to all the environment and then the standard promotion. The bracket is obtained by applying substitution lemma (which because of the remark above is valid in this system) to the premise, cutting with

$$\frac{\dfrac{}{x : !\sigma \vdash x : !\sigma}\ (\text{var})}{x : !\sigma \vdash x : !!\sigma}\ (\text{prom})$$

On the converse the usual promotion can be obtained by the two rules above by first applying the new promotion (thus doubling the bangs in the environment) and then multiple uses of the

bracket. This decomposition is useful, as we will see that **ELL** is obtained from this system by adopting this new kind of prom rule and leaving out dereliction and bracket, and then **LLL** by tweaking more this new kind of promotion.

As we saw for simply typed λ-calculus this system lacks expressive power: the solution is the same, adding polymorphism (and so as we have said including additional structures). So we extend types and add the two rules to handle the quantifier.

Definition 4.1.4 (types and rules of AL2). Types $\mathbb{T}_{\mathbf{AL2}}$ are defined by the grammar

$$\mathbb{T}_{\mathbf{AL2}} ::= \mathbb{V} \mid \mathbb{T}_{\mathbf{AL2}} \multimap \mathbb{T}_{\mathbf{AL2}} \mid !\mathbb{T}_{\mathbf{AL2}} \mid \forall \mathbb{V}.\mathbb{T}_{\mathbf{AL2}}.$$

Free type variables, bounded type variables, substitutions are defined as for $\mathbb{T}_{\mathbf{F}}$.

Two rules are added to the ones presented for **AL**:

$$\frac{A \vdash M : \tau}{A \vdash M : \forall\alpha.\tau}\ (\text{gen}) \qquad \frac{A \vdash M : \forall\alpha.\tau}{A \vdash M : \tau[\rho/\alpha]}\ (\text{ins})$$

where in (gen) we require as usual that $\alpha \notin \mathrm{FTV}(A)$.

Example 4.1.5. We redefine the type for integers as $\mathtt{Int} := \forall\alpha.!(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$. Every Church integer is typable with this type. We cannot use anymore the classic `Int`, apart from $\underline{1}$ and $\underline{0}$ (not even this last one if we renounce to unrestricted weakening).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f_n : \alpha \multimap \alpha \vdash f_n : \alpha \multimap \alpha}\ (\text{var}) \quad \dfrac{}{x : \alpha \vdash x : \alpha}\ (\text{var})}{f_n : \alpha \multimap \alpha, x : \alpha \vdash (f_n\, x) : \alpha}\ (\text{app})}{\vdots}}{\dfrac{}{f_1 : \alpha \multimap \alpha \vdash f_1 : \alpha \multimap \alpha}\ (\text{var}) \quad \vec{f} : \alpha \multimap \alpha, x : \alpha \vdash (f_2\,(\dots(f_n\, x)\dots) : \alpha}\ (\text{app})}{\vec{f} : \alpha \multimap \alpha, x : \alpha \vdash (f_1(\dots(f_n\, x)\dots) : \alpha}\ (\text{abs})}{\vec{f} : \alpha \multimap \alpha \vdash \lambda x.(f_1(\dots(f_n\, x)\dots) : \alpha \multimap \alpha}\ (\text{der})}{\vec{f} : !(\alpha \multimap \alpha) \vdash \lambda x.(f_1(\dots(f_n\, x)\dots) : !\alpha}\ (\text{con})}{f : !\alpha \multimap \alpha \vdash \lambda x.(f^n\, x) : !(\alpha \multimap \alpha)}\ (\text{abs})}{\vdash \lambda f\lambda x.(f^n\, x) : !(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha}\ (\text{gen})}{\vdash \lambda f\lambda x.(f^n\, x) : \mathtt{Int}^{\mathbf{AL}}}$$

For $\underline{1}$ no contraction is needed (and in fact no dereliction either), and for $\underline{0}$ we have to have weakening to introduce f.Clearly it is not possible to type integers with the type used in **S**: no bang, no repetitions.

Definition 4.1.6 (forgetful function, embedding into linear). The *forgetful function* is a function bringing **AL**2 types, environments and sequents into their counterparts in system **F**. We

denote it with a bar, and define it on types by:

$$\begin{aligned}\overline{\alpha} &:= \alpha,\\ \overline{\tau_1 \multimap \tau_2} &:= \overline{\tau}_1 \to \overline{\tau}_2,\\ \overline{\forall\alpha.\tau} &:= \forall\alpha.\overline{\tau},\\ \overline{!\tau} &:= \overline{\tau}.\end{aligned}$$

Basically it erases the modalities. It is trivially extended to environments and sequents by acting on the types involved (and thus leaving the term unchanged). Clearly if applied on objects without quantifiers it gives objects in system **S**.

On the converse we embed simple types in linear ones with a function denoted by a hat, defined by

$$\begin{aligned}\hat{\alpha} &:= \alpha,\\ \widehat{\tau_1 \to \tau_2} &:= !\hat{\tau}_1 \multimap \hat{\tau}_2.\end{aligned}$$

We extend the defintion to environments prepending a bang, i.e. $\hat{A}(x) := !\widehat{A(x)}$, as environments are really hypotheses.

Note that $\bar{\hat{\tau}} = \tau$, and similarly for environments and sequents, while it is not true that $\hat{\bar{\tau}} = \tau$, as informations on the type get lost when reverting to simple types.

We may say that **AL** and **AL**2 give us the tools to study better the computation of a λ-term, but in fact the system itself relatively to the terms being typed is nothing new. In fact the following proposition tells us that the terms typable are exactly the same of system **F** (system **S** if we do not use quantifiers). So also the representable functions are the same.

Proposition 4.1.7. *For every typing $\mathcal{D} \rightsquigarrow \Gamma$ in* **AL**2 *there is a valid typing $\overline{\mathcal{D}} \rightsquigarrow \overline{\Gamma}$ in system* **F**

On the converse, for every typing $\mathcal{D} \rightsquigarrow \Gamma$ in **F** *there is a valid typing $\widehat{\mathcal{D}} \rightsquigarrow \widehat{\Gamma}$ in* **AL**2.

Proof. Reasoning by induction it suffices to see that every rule in one of the system can be emulated in the other after applying the right translation. For the first claim the validity of the (var), (app) and (abs) rules is trivially preserved by the forgetful function. (weak) is unsurprisingly emulated by applying weakening. For (con) and (der) we apply multiple times substitution lemma. The (prom) rule can be safely erased. (gen) and (ins) are identical.

Regarding the opposite direction there is some tweaking to be done about repeated variables. Note that it must be done only on application, as is the only rule that can duplicate variable occurrences. Let us reason by induction on the size of the derivation.

(var): A (var) yielding $A, x : \sigma \vdash x : \sigma$ becomes

$$\frac{\hat{A}, x : \hat{\sigma} \vdash x : \hat{\sigma}}{\hat{A}, x : !\hat{\sigma} \vdash x : \hat{\sigma}} \text{ (der)}$$

(abs): The premise of the rule is $A, x : \sigma \vdash M : \tau$ which by induction hypothesis becomes $\hat{A}, x : !\hat{\sigma} \vdash M : \hat{\tau}$, which in turn by (abs) rule becomes

$$\hat{A} \vdash \lambda x.M : !\hat{\sigma} \multimap \hat{\tau},$$

and in fact $!\hat{\sigma} \multimap \hat{\tau} = \widehat{\sigma \to \tau}$.

(app): We have $A \vdash_{\mathbf{S}} M : \sigma \to \tau$ and $B \vdash_{\mathbf{S}} N : \sigma$. For every $x \in \mathrm{FV}(M) \cap \mathrm{FV}(N)$ we rename it in one of the two with a fresh variable x'. Say that after this A, B, M and N become A', B', M' and N'. Then we apply induction hypothesis and obtain two derivations which yield:

$$\frac{\widehat{A'} \vdash_{\mathbf{LL}} M' : !\hat{\sigma} \multimap \tau \qquad \dfrac{\widehat{B'} \vdash_{\mathbf{LL}} N' : \hat{\sigma}}{\widehat{B'} \vdash N' : !\hat{\sigma}} \text{ (prom)}}{\widehat{A'}, \widehat{B'} \vdash (M'\,N') : \hat{\tau}} \text{ (app)}$$

Now we apply multiple contractions to bring back $\hat{A}$, $\hat{B}$, M and N.

(gen) and (ins): These pose no problem.

Again $\bar{\hat{\mathcal{D}}} = \mathcal{D}$, and not the converse. □

Corollary 4.1.8 (strong normalization). *Terms typable in* **AL** *are strongly normalazing.*

Let's unearth immediately a problem with **AL** (and its restrictions):

Proposition 4.1.9. AL *does not enjoy subject reduction.*

Proof. Take the typing $y : \alpha, z : \alpha \to !\alpha \vdash (\lambda x.z\,x)\,y : !\alpha$, and with a (cut) insert it in $x : !\alpha \vdash \lambda w.w\,x\,x : !((\alpha \to \alpha\alpha) \to \alpha)$, obtained with a final contraction. We have that a single β-step on only one of the two copies of the term brings to $\lambda w.w\,(z\,y)\,((\lambda x.z\,x)\,y)$ which evidently cannot be typed with the same environment above because y is of a type not banged.

The problem is that we have *shared* a derivation using contraction, but then we have applied reduction to only one of the copies thus breaking the sharing. In order to remain inside the type we have to apply the same step to all the copies cut with a contraction. This is not the only problem with sharing. One way of resolving them is by seeing the proof that hides behind the term and apply cut-elimination: the complexity results will be in general guaranteed if we adopt this strategy of elimination. In some way the typing is not anymore a simple certificate which we put aside after obtaining it: it also tells us *how* we have to reduce the term to use all the information contained in it. □

4.2 Light logics

So linear logic as a type assignment gives us new insight into the use we do of input in the term. These are tools that we need only to exploit.

In 1998 Girard presented such a way to use them, presenting **LLL**, *light linear logic*, in [Gir98]. It is a system that captures polynomial time. The sense of it is that if we bound the number of concatenated boxes we have a proof net that normalizes in polynomial time, not only with respect to the number of cut-elimination steps, but also to the steps that a Turing machine needs to do to normalize it. On the converse every polynomial function has a representation as proof net of **LLL**.

Expanding a germinal idea contained in that paper 3 years later Danos and Joinet introduced **ELL**, elementary linear logic, which captures the class of elementary functions.

The idea in the two is to again restrict the use of contraction, not by tampering directly with it, but by better controlling the way modalities are dispensed.

We here present in reversed order the type systems related to these two approaches to complexity, linked with the intuitionistic implicational versions of the two logic systems. Again we permit unrestricted weakening: the completeness results regarding the corresponding complexity classes remain sound, and programming becomes easier, without having to introduce additives.

4.2.1 EAL

Definition 4.2.1 (elementary functions). *Elementary recursive functions*, also called *Kalmar recursive functions* from the first who introduced them in 1943, are recursive functions for each of which there exists a Turing machine that computes it in a running time bounded by a tower of exponentials of fixed height, applied to the arguments.

Equivalently, they are the least class of functions containing the constant **0**, the successor `succ`, the addition `add`, the multiplication `mult`, the predecessor `pred`, the subtraction `sub`, the exponential `exp`, and closed under compostion and *bounded sum* and *product*, i.e. if we have elementary functions $f : \mathbb{N}^{k+1} \to \mathbb{N}$ we have that the following functions $h_1, h_2 : \mathbb{N}^{k+1} \to \mathbb{N}$ are elementary:

$$h_1(n, \vec{m}) = \sum_{i=0}^{n} f(i, \vec{m}), \qquad h_2(n, \vec{m}) = \prod_{i=0}^{n} f(i, \vec{m}).$$

EAL is obtained from **AL** by assuming the all-in-one promotion and not permitting anymore dereliction and bracket. The types being used are the same of **AL**. We here give rules without cut (that will be embedded in contraction and promotion).

Definition 4.2.2 (rules of EAL). Rules for **EAL** are the following:

$$\frac{}{A, x : \tau \vdash x : \tau} \text{ (var)}$$

$$\frac{A, x : \sigma \vdash M : \tau}{A \vdash (\lambda x.M) : \sigma \multimap \tau} \text{ (abs)} \qquad \frac{A \vdash M : \sigma \multimap \tau \quad B \vdash N : \sigma}{A, B \vdash (M\ N) : \tau} \text{ (app)}$$

$$\frac{A \vdash N : !\sigma \quad \vec{x} : !\sigma, B \vdash M : \tau}{A, B \vdash M[N/\vec{x}] : \tau} \text{ (con)} \qquad \frac{A \vdash M : \tau}{A, B \vdash M : \tau} \text{ (weak)}$$

$$\frac{\overrightarrow{A \vdash N : !\sigma} \quad \overrightarrow{x : \sigma}, B \vdash M : \tau}{\vec{A}, !B \vdash M[\overrightarrow{N/x}] : !\tau} \text{ (prom)}$$

Note that we allow some variables to remain named the same across promotion: this is for ease of notation, without having to rename them if they come directly from a (var).

Note also that every **EAL** derivation is trivially an **AL** derivation without need of translation if we take the definition of **AL** with the all-in-one promotion.

As we did for **AL** we can introduce polymorphism obtaining **EAL**2. The types being used are the same, and the two rules (gen) and (ins) apply too.

Example 4.2.3. Again we must change the type for integers. Throughout this section `Int` will denote

$$\texttt{Int} := \forall\alpha.!(\alpha \to \alpha) \to !(\alpha \to \alpha).$$

Type derivation for Church integers is obtained from that in **AL** (see example 4.1.5) by replacing the dereliction rule with an all-in-one promotion.

The new restriction indeed cuts down on the functions being represented.

Theorem 4.2.4. *A term typable in* **EAL**2 *can be reduced in elementary time, i.e. there exist a Turing machine that given any* **EAL**2*-typed term together with its derivation reduces it with a computational cost bounded by a tower of exponentials.*

Proof. (sketch) What we do is that we take the proof-net representation of the term and we check in that framework what happens. The restriction on rules permits to have a *stability* notion, for which we can carry out all cut-elimination steps one exponential level at a time, where by level we mean how many boxes (which are drawn when using the promotion rule) contains the cut we want to reduce. With the restrictions we have that reducing a cut in a certain level can create new cuts only at a higher level, and does not make any node change level (the total exponential depth of the proof does not change). So say we are reducing cuts at level d: we reduce at most s

cuts where s is the size of the proof at the moment. Each reduction produces at most s copies of a rule at a higher level, so the at the end of this steps the proof will have size at most $s^s + 1$, which is definitely under 2^{2^s}. So if we clear a level at a time starting from the lower one and proceding increasingly we have a bound on cut reductions by a tower of exponentials of height proportional to the total depth of the proof. Moreover every cut reduction is elementary for Turing machines and so the effective computational cost is elementary. □

Remark 4.2.5. We may note that we may speak of elementary bound only if the depth of the proof is fixed. In particular if we take a term of type $\sigma \multimap \tau$ it is not true that applied to *any* term of type σ the two reduce in elementary time with respect to the size of the argument, because we can take terms of exponential depth proportional to their size, and thus the tower of exponentials becomes of varied height. However this does not pose a problem as far as we are concerned with functions on integers, as we will see that the exponential depth of any integer is 1.

In order to have a better understanding on the way the type assignment rules work on pure terms, we will now introduce new terms that reflect them. Basically substitution due to promotion and sharing due to contraction will be made explicit. With pure λ-terms typing is no longer a certificate we can put aside once read: it contains information we want to use in reducing the term. These new terms make this mechanism explicit: we can define a reduction on this terms that in fact reflects the cut-elimination of the associated proof-net.

Definition 4.2.6 (Λ^{EA}). The set of *elementary affine* terms are built with the grammar

$$\Lambda^{\mathrm{EA}} ::= \mathcal{V} \mid (\Lambda^{\mathrm{EA}}\ \Lambda^{\mathrm{EA}}) \mid (\lambda\mathcal{V}.\Lambda^{\mathrm{EA}}) \mid \{\Lambda^{\mathrm{EA}}\}_{\Lambda^{\mathrm{EA}} \to \vec{\mathcal{V}}} \mid (!\Lambda^{\mathrm{EA}})\{\overrightarrow{\Lambda^{\mathrm{EA}}/\mathcal{V}}\}.$$

We will call the two new constructs *contracted term* and *boxed term*. Practically we are writing in the syntax the substitutions related to contraction and promotion. We use '{' and '}' to distinguish such an operation from the brackets of substitions. Brackets are a way of writing in brief an operation done on terms, while here the braces are coded in the syntax. We call *auxiliary* the variables appearing in the construction of these new terms, i.e. $\vec{x}$ in $\{M\}_{N \to \vec{x}}$ and $(!M)\{\overrightarrow{N/x}\}$. We require that all variables appear at most once in every term (appearing in the term and in the list of auxiliary variables of boxed and contracted terms does not count). In a boxed term we require that *all* free variables must become auxiliary. The definition of free and bounded variables

is extended to the new constructs by

$$\begin{aligned}
\mathrm{FV}(\{M\}_{N\to\vec{x}}) &:= \mathrm{FV}(M)\setminus\{\vec{x}\}\cup\mathrm{FV}(N),\\
\mathrm{FV}((!M)\{\overrightarrow{N/x}\} &:= \mathrm{FV}(M)\setminus\{\vec{x}\}\cup\mathrm{FV}(\vec{N}),\\
\mathrm{BV}(\{M\}_{N\to\vec{x}}) &:= \mathrm{BV}(M)\cup\mathrm{BV}(N),\\
\mathrm{BV}((!M)\{\overrightarrow{N/x}\} &:= \mathrm{BV}(M)\cup\mathrm{BV}(\vec{N}).
\end{aligned}$$

We extend α-equivalence to encompass also auxiliary variables, and substitution to avoid variable capture just like if auxiliary variables were bounded. We also identify boxed and contracted terms in which the corresponding list of auxiliary variables is in different order. If we α-convert variables so that no name clash ever happens we can apply substitution without worries. For ease of notation we may rewrite a term $(!M)\{\overrightarrow{N/x}, y'/y\}$ by $(!M[y'/y])\{\overrightarrow{N/x}\}$, contradicting the rule of all free variables appearing auxiliary: this is a notation, we have to keep track of the free variables and eventually put them again in the list. $!M$ is short for $(!M)\{\}$ (also regarding the above notation), $(!^q M)$ is short for M preceded by q modalities. If we write $(!^q M)\{\overrightarrow{N/x}\}$ we mean that the auxiliary information is for the last ! applied, while all the other ones follow the convention of hiding auxiliary variables.

We redefine the size of the term extending to the new cases by

$$\begin{aligned}
|\{M\}_{N\to\vec{x}}| &:= |M| + |N| + |\vec{x}| - 1,\\
\left|(!M)\{\overrightarrow{N/x}\}\right| &:= |M| + 1 + \sum_i (|N_i| + 1).
\end{aligned}$$

Clearly these new terms can be transformed into a usual term by applying the substitutions that are hard-coded into them. However there is also a subtler way of expanding an elementary affine term into a pure one.

Definition 4.2.7. Let's denote by $(M)^+$ the following function $(\,.\,)^+ : \Lambda^{\mathrm{EA}} \to \Lambda$:

$$\begin{aligned}
(x)^+ &:= x,\\
(\lambda x.M)^+ &:= \lambda x.(M)^+,\\
(M_1\, M_2)^+ &:= (M_1)^+\,(M_2)^+,\\
(\{M\}_{N\to\vec{x}})^+ &:= (M)^+[(N)^+/\vec{x}],\\
((!M)\{\overrightarrow{N/x}\})^+ &:= (M)^+[\overrightarrow{(N)^+/x}].
\end{aligned}$$

We define another translation $(\,.\,)^- : \Lambda^{\mathrm{EA}} \to \Lambda$:

$$\begin{aligned}
(x)^- &:= x,\\
(\lambda x.M)^- &:= \lambda x.(M)^-,\\
(M_1\, M_2)^- &:= (M_1)^-\,(M_2)^-,\\
(\{M\}_{N\to\vec{x}})^- &:= \begin{cases} (M)^-[N/\vec{x}], & \text{if } N \text{ is a variable,}\\ (\lambda z.(M)^-[z/\vec{x}])\, N & \text{otherwise,}\end{cases}\\
((!M))^- &:= (M)^-\\
((!M)\{\overrightarrow{N/x}, P/y\})^- &:= (\lambda y.((!M)\{\overrightarrow{N/x}])^-)\,(P)^-,
\end{aligned}$$

where we assumed all the terms substitute for the auxiliary variables of a boxed term are not variables.

Remark 4.2.8. We say the second one is subtler because it does not cause a heavy increase of the size. In fact $|(M)^-| \leq 2\,|M|$. Note that they are equivalent, in the sense that $(M)^- \overset{\beta}{\twoheadrightarrow} (M)^+$.

These new terms have the corresponding new rules:

Definition 4.2.9 (rules of $\mathbf{EAL}_s$). The typing system $\mathbf{EAL}_s$ (*elementary affine logic with sharing*) is given by the following rules over terms of Λ^{EA}.

$$\frac{}{A, x:\tau \vdash x:\tau}\ (\mathrm{var})$$

$$\frac{A, x:\sigma \vdash M:\tau}{A \vdash (\lambda x.M):\sigma \multimap \tau}\ (\mathrm{abs}) \qquad \frac{A \vdash M:\sigma \multimap \tau \quad B \vdash N:\sigma}{A, B \vdash (M\,N):\tau}\ (\mathrm{app})$$

$$\frac{A \vdash N:!\sigma \quad \vec{x}:!\sigma, B \vdash M:\tau}{A, B \vdash \{M\}_{N\to\vec{x}}:\tau}\ (\mathrm{con}) \qquad \frac{A \vdash M:\tau}{A, B \vdash M:\tau}\ (\mathrm{weak})$$

$$\frac{\overrightarrow{A \vdash N:!\sigma} \quad \overrightarrow{x:\sigma} \vdash M:\tau}{\vec{A}, \vdash (!M)\{\overrightarrow{N/x}]:!\tau}\ (\mathrm{prom})$$

Note that the definition of the promotion can in fact be adapted to the convention of not stating the explicit substitution of variables by omitting the relative (var) rules and simply writing

$$\frac{\overrightarrow{B \vdash N:!\sigma} \quad \overrightarrow{x:\sigma}, B \vdash M:\tau}{\vec{A}, !B \vdash (!M)\{\overrightarrow{N/x}]:!\tau}\ (\mathrm{prom})$$

We extend as usual to $\mathbf{EAL}2_s$ with second order adding (gen) and (ins). We may omit drawing an eventual (var) used in contraction.

Remark 4.2.10. Note that now $\mathbf{EAL}_s$ is syntax driven, and $\mathbf{EAL2}_s$ is almost syntax-driven in the same sense of system $\mathbf{F}$.

First of all we relate the new terms to the classic ones.

Proposition 4.2.11. *If* $A \vdash_{\mathbf{EAL2}_s} M : \tau$ *then*

$$A \vdash_{\mathbf{EAL2}} (M)^+ : \tau$$

and $A \vdash_{\mathbf{EAL2}} (M)^-$.

On the converse if $A \vdash_{\mathbf{EAL2}} M : \tau$ *there is a term* $N \in \Lambda^{\mathrm{EA}}$ *such that* $(N)^+ = M$ *and*

$$A \vdash_{\mathbf{EAL2}_s} N : \tau.$$

Proof. Regarding $(\,.\,)^+$ both directions are by simple induction on the derivation. Each rule simply translates into the corresponding one of the other system. In particular N is built from M by leaving explicit all the substitutions required by contraction and promotion. Note that a-priori the term N does *not* depend solely on M, or on the final sequent of the derivation: we have to use the entire derivation leading to $A \vdash M : \tau$ to build it.

As for $(\,.\,)^-$, where the definition differs we just have to replace the cut implemented in (prom) or (con) with abstraction followed by an application. □

Definition 4.2.12 (reduction on $\mathbb{T}^{\mathrm{EA}}$). We define the following one step relations, intending that the definition given should pass to context.

$$\begin{aligned}
(\lambda x.M)\, N &\xrightarrow{\beta} M[N/x],\\
\{M\}_{(!N)\{\overrightarrow{P/x}\}\to\vec{y}} &\to_{dup} \{\dots\{M[(!N)\{\overrightarrow{x^1/x}\}/y_1,\dots,(!N)\{\overrightarrow{x^k/x}\}/y_k]\}_{P_1\to\vec{x^1}}\cdots\}_{P_k\to\vec{x^k}},\\
(!M)\{\overrightarrow{N/x},(!P)\{\overrightarrow{Q/z}\}/y\} &\to_{!-!} (!M[N/y])\{\overrightarrow{N/x},\overrightarrow{Q/z}\},\\
(\{M\}_{P\to\vec{x}}\, N) &\to_{@-c} \{(M\,N)\}_{p\to\vec{x}},\\
(M\,\{N\}_{P\to\vec{x}}) &\to_{@-c} \{(M\,N)\}_{p\to\vec{x}},\\
(!M)\{\overrightarrow{N/x},\{P\}_{Q\to\vec{y}}/z\} &\to_{!-c} \{(!M)\{\overrightarrow{N/x},P/z\}\}_{Q\to\vec{y}},\\
\{M\}_{\{N\}_{P\to\vec{y}}\to\vec{x}} &\to_{c-c} \{\{M\}_{N\to\vec{x}}\}_{P\to\vec{y}},\\
\lambda x.\{M\}_{N\to\vec{y}} &\to_{\lambda-c} \{\lambda x.M\}_{N\to\vec{y}}, \quad \text{if } x\notin \mathrm{FV}(N).
\end{aligned}$$

In $\to_{dup}$ we explicitly ask that $(!N)\{\overrightarrow{P/x}\}$ is written with all the free variables appearing in the auxiliary list.

Note that with the condition on variables not appearing twice there is no capture in the above defintions. We define by $\twoheadrightarrow_{\mathrm{EA}}$ the transitive and reflexive closure of all the above one step reductions.

Theorem 4.2.13. *Every term in* Λ^{EA} *typable in* $\mathbf{EAL}_s$ *normalizes with* $\twoheadrightarrow_{\mathrm{EA}}$ *in elementary time.*

Proof. This is in fact a restatement of theorem 4.2.4. Practically the rules for $\rightarrow_{\mathrm{EA}}$ are the embedding into Λ^{EA} of the rules of cut-elimination in **ELL**, apart from the rules of interaction of contraction which are not present in the parallel notion of proofnets ($@ - c$, $c - c$, $\lambda - c$) or are a simple rewriting rule ($! - c$). However the presence of those can be seen to not alter the resulting complexity. □

So we will use this form for the terms, so that we have an easier understanding on how typing works. We will identify type derivation of the two system, writing $\vdash_{\mathbf{EAL}}$ indifferently for terms in one or the other system. In this framework the Church integers are the terms:

$$\underline{n} := \lambda f.\{(!\lambda x.f_1(f_2 \dots (f_n\, x) \dots))\}_{f \to \vec{f}}.$$

Again we will denote by $\underline{n}$ the classic pure form or the one given above depending on the context.

4.2.2 Representation theorem for EAL

From now on `Int` will denote the integers in **EAL**2, i.e.

$$\mathtt{Int} := \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \to \alpha),$$

and $\mathtt{Int}_\tau$ is `Int` instantiated with τ.

Definition 4.2.14 (representing functions in EAL2). We say a function $f : \mathbb{N}^k \to \mathbb{N}$ is represented in **EAL**2 if there is a pure term $\underline{f}$ typable in **EAL**2 with $\mathtt{Int}^k \multimap !^q\,\mathtt{Int}$ so that given $\vec{n} \in \mathbb{N}^k$ we have

$$\underline{f}\,\underline{\vec{n}} \overset{\beta}{\twoheadrightarrow} !^q \underline{f(\vec{n})}.$$

Moreover we say the representation $\underline{f}$ is *flat* if $q = 0$, *oblique* otherwise. We call q the *obliqueness* of the representation.

Flat functions will be the ones possible to iterate.

Theorem 4.2.15. *Every elementary function is representable by a term typable in* **EAL**2*. On the converse every term typable with* $\mathtt{Int}^k \multimap !^q\,\mathtt{Int}$ *represents an elementary function.*

One direction is practically ready. Given M typable with $\mathtt{Int}^k \multimap !^q\,\mathtt{Int}$, applying it to $\underline{\vec{n}}$ we have a term of type $!^q\,\mathtt{Int}$. If we apply the forgetful function we have that the term is typable as an integer in system **F**, so we already know its normal form is an integer (up to identification of $\underline{1}$ with the identity). By theorem 4.2.1 we know that computation time can be bounded by a tower

of exponentials with height proportional to the depth of the proof net associated with $M\,\vec{\underline{n}}$. This has a fixed depth independent of $\vec{n}$, so the computation time is elementary.

The proof of the other direction will be carried out with an axiomatic approach, i.e. we will use the axiomatic characterization of the class of elementary functions, rather than coding Turing machines in the system.

First let us introduce a tool of great use in programming with **EAL**2, the *iteration scheme*. clearly it will be restricted in some manner to prevent uncontrolled iteration.

Proposition 4.2.16 (iteration). *There is a scheme (a context with three holes in our case)* $\mathrm{IT}_{n,M,N}$ *so that given terms* $A \vdash_{\mathbf{EAL}} M : \tau \multimap \tau$ *and* $B \vdash_{\mathbf{EAL}} N : \tau$ *(A and B disjoint), we have the typing*

$$!A, !B, n : \mathtt{Int} \vdash \mathrm{IT}_{n,M,N} : !\tau$$

such that $\mathrm{IT}_{\underline{n},M,N}$ *represents M iterated n times on N.*

Proof. The scheme is $\mathrm{IT}_{n,M,N} := (!y\,N)\{(n\,(!M)/y\}$. We have $(\mathrm{IT}_{n,M,N})^+ = (n\,M\,N)$, so it is easy to see that indeed the scheme being represented is the iteration. Let's see how typing works.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{n : \mathtt{Int} \vdash n : \mathtt{Int}}\,(\text{var})}{n : \mathtt{Int} \vdash n : \mathtt{Int}_\tau}\,(\text{gen}) \quad \dfrac{A \vdash M : \tau \multimap \tau}{!A \vdash (!M) : !(\tau \multimap \tau)}\,(\text{prom})}{!A, n : \mathtt{Int} \vdash n\,(!M) : !(\tau \multimap \tau)}\,(\text{app}) \quad \dfrac{\dfrac{}{y : \tau \multimap \tau \vdash y : \tau \multimap \tau}\,(\text{var}) \qquad B \vdash N\tau}{B, y : \tau \multimap \tau \vdash y\,N : \tau}\,(\text{app})}{!A, !B, n : \mathtt{Int} \vdash (!y\,N)\{(n\,(!M)/y\} : !\tau}\,(\text{prom})$$

Note that

- we can apply an iteration only on a function which preserve the exponential depth, i.e. we can't apply it to a function $\sigma \multimap !\sigma$.
- the returned value has its exponential depth lifted. So, by the above point, we cannot iterate an iteration.

□

Now with the most basic base functions. **0** and projections pose no problems.

Lemma 4.2.17 (succ). succ *is representable in* **EAL**2 *by a flat term* $\underline{\mathtt{succ}}$.

Proof. We easily (and linearly) derive

$$f_1 : \alpha \multimap \alpha, z : \alpha \multimap \alpha, \vdash \lambda x.f_1\,(z\,x)$$

and

$$f_2 : !(\alpha \multimap \alpha), n : \mathtt{Int} \vdash n\,f_2 : !(\alpha \multimap \alpha).$$

Applying a promotion to the first one cutting it with the second we then have

$$\dfrac{\dfrac{\dfrac{f_1 : !(\alpha \multimap \alpha), f_2 : !(\alpha \multimap \alpha), n : \mathtt{Int} \vdash (!\lambda x.f_1\,(z\,x))\{(n\,f_2)/z\} : !(\alpha \multimap \alpha)}{f : !(\alpha \multimap \alpha), n : \mathtt{Int} \vdash \{(!\lambda x.f_1\,(z\,x))\{(n\,f_2)/z\}\}_{f \to f_1, f_2} : !(\alpha \multimap \alpha)}\ (\text{con})}{n : \mathtt{Int} \vdash \lambda f.\{(!\lambda x.f_1\,(z\,x))\{(n\,f_2)/z\}\}_{f \to f_1, f_2} : \mathtt{Int}_\alpha}\ (\text{abs})}{n : \mathtt{Int} \vdash \lambda f.\{(!\lambda x.f_1\,(z\,x))\{(n\,f_2)/z\}\}_{f \to f_1, f_2} : \mathtt{Int}}\ (\text{gen})$$

We take

$$\mathtt{succ}[\ ] := \lambda f.\{(!\lambda x.f_1\,(z\,x))\{(\Box\, f_2)/z\}\}_{f \to f_1, f_2}$$

and easily see that $(\mathtt{succ}[n])^+ = \lambda f \lambda x.(f\,(n\,f\,x))$, so that indeed $\underline{\mathtt{succ}} := \lambda n.\,\mathtt{succ}[n]$ is a term of type $\mathtt{Int} \multimap \mathtt{Int}$ representing succ. □

Now we have the tools necessary to make an integer change depth, but only making it grow (otherwise we could iterate any function).

Lemma 4.2.18 (coercion). *There is a context* $\mathtt{coerc}[\ ]$ *so that*

$$n : !^q\,\mathtt{Int} \vdash \mathtt{coerc}[n] : !^{q+1}\,\mathtt{Int}$$

is derivable for any q *and* $\mathtt{coerc}[\underline{n}] \equiv_\beta \underline{n}$ *for every* n. *We denote by* $\mathtt{coerc}_p[\ ]$ *the context* $\mathtt{coerc}[\ldots\mathtt{coerc}[n]\ldots]$ *with* p *copies of the context chained together, so that*

$$n : \mathtt{Int} \vdash \mathtt{coerc}_p[n] : !^q\,\mathtt{Int} \multimap !^{p+q}\,\mathtt{Int}\,.$$

Proof. We apply the iteration scheme to the flat term $\underline{\mathtt{succ}}$ with base value $\underline{0}$:

$$\mathtt{coerc}[n] := \mathrm{IT}_{n,\underline{\mathtt{succ}},\underline{0}}\,.$$

□

Remark 4.2.19. The presence of this term is why we can restrict ourselves to functions which do not have modalities on the arguments. If we obtain a representation of a function f as a term M typable with $!^p\,\mathtt{Int} \multimap !^q\,\mathtt{Int}$ (we suppose only one argument but it is not restrictive), we can define the term

$$\underline{f} := \lambda n.(M\ \mathtt{coerc}_p[n])$$

and we have a term representing f with the right typing. So we can take for represented also functions for which we find terms of type $!^{p_1}\,\mathtt{Int} \multimap \ldots \multimap !^{p_k}\,\mathtt{Int} \multimap !^q\,\mathtt{Int}$.

Also we can *lift* the obliqueness of a function to a desired value higher than the original obliqueness, by using $\lambda n.\,\mathtt{coerc}_p[\underline{f}\,n]$.

Instead we will say we *promote* a function $M : \mathtt{Int}^k \multimap !^q\,\mathtt{Int}$ by applying a certain number p of times promotion to $M\,\vec{n}$ and then abstracting the ns, so that we obtain a term representing the same function but with type $(!^p\,\mathtt{Int})^k \multimap !^{p+q}\,\mathtt{Int}$.

Lemma 4.2.20 (`add` and `mult`). `add` *and* `mult` *are representable by flat terms.*

Proof. Clearly we can't use iteration because they would not be flat, and we need them flat for bounded sum and product.

Surprisingly `mult` is easier. The usual term $\lambda n \lambda m.\lambda f.(m\,(n\,f))$ is easily typable with $\texttt{Int} \multimap \texttt{Int} \multimap \texttt{Int}$ without use of exponential rules. As for `add`, once we revert back to pure λ-calculus the term will be the usual one. However here we must contract the f. So we easily derive these three sequents

$$f_1 : !(\alpha \multimap \alpha), n : \texttt{Int} \vdash n\,f_1 : !(\alpha \multimap \alpha), \quad f_2 : !(\alpha \multimap \alpha), m : \texttt{Int} \vdash m\,f_2 : !(\alpha \multimap \alpha),$$

$$w : \alpha \multimap \alpha, z : \alpha \multimap \alpha \vdash \lambda x.w\,(z\,x) : \alpha \multimap \alpha.$$

Then with a promotion we plug the first two into z and w in the third one, and get

$$f_1 : !(\alpha \multimap \alpha), f_2 : !(\alpha \multimap \alpha), n : \texttt{Int}, m : \texttt{Int} \vdash (!\lambda x.w\,(z\,x))\{n\,f_1/w,\, m\,f_2/z\}.$$

Then we contract and abstract and get

$$\underline{\texttt{add}} := \lambda n \lambda m.\lambda f\{(!\lambda x.w\,(z\,x))\{n\,f_1/w,\, m\,f_2/z\}\}_{f \to f_1, f_2},$$

typable with $\texttt{Int} \multimap \texttt{Int} \multimap \texttt{Int}$. □

Lemma 4.2.21 (`exp`). `exp` *is representable in* **EAL**2.

Proof. Take $M := \lambda x.\underline{\texttt{mult}}\,m\,x$. We get $m : \texttt{Int} \vdash M : \texttt{Int} \multimap \texttt{Int}$ flat. So we can iterate it, using as base $\underline{1}$ of type `Int`. So applying the iteration scheme yields

$$m : !\,\texttt{Int}, n : \texttt{Int} \vdash \mathrm{IT}_{n,M,\underline{1}} : !\,\texttt{Int}$$

which represents the exponential. By abstracting and precomposing with a coercion we get the desired term $\underline{\texttt{exp}}$ of type $\texttt{Int} \multimap \texttt{Int} \multimap !\,\texttt{Int}$ which represents m^n. □

Lemma 4.2.22 (`pred` and `sub`). `pred` *and* `sub` *are representable in* **EAL**2.

Proof. The idea is always that to use a couple in which one component is used to save the last value which has been visited so that on exit we give it as output.

The definition of $\sigma \otimes \tau$ is the same used for $\sigma \times \tau$ in system **F** (see 3.2.3), provided we substitute $\multimap$ for $\to$. In particular it means that apart from exponentiating, we can use one or both the components of the pair, but at most one time each. We define the term to be iterated of type $\tau := (\alpha \multimap \alpha) \otimes (\alpha \multimap \alpha)$. So we build a term M of type $\tau \multimap \tau$:

$$M := \lambda p.p\,(\lambda x \lambda y.\langle f, x \circ y \rangle),$$

where I is the identity and $\circ$ is the composition $x \circ y := \lambda z.x\,(y\,z)$. Provided we type f, x and y with $\alpha \multimap \alpha$, and p with τ, we get

$$f : \alpha \multimap \alpha \vdash M : \tau \multimap \tau$$

flat so we can iterate it. Note no exponentials where used. If we start with base value $\vdash \langle I, I\rangle : \tau$ (empty environment) we get at the first iteration $\langle f, I \circ I\rangle$ and only from then on we begin to compose the fs in the second component. Iterating $n \geq 1$ times yields $\langle f, (f)^{n-1}\rangle$ where $(f)^{n-1} = f \circ \cdots \circ f \circ I$ with $n-1$ copies of f. Applying iteration scheme we get a term

$$f : !(\alpha \multimap \alpha), n\,\mathtt{Int} \vdash \mathrm{IT}_{n,M,\langle I,I\rangle} : !\tau.$$

We prepare apart a term ready for the one above to be plugged in by promotion. In fact we easily get a derivation ending in

$$z : \tau \vdash \lambda x.z\,\underline{\mathtt{false}}\,x : \alpha \multimap \alpha$$

which together with the other gives

$$\frac{f : !(\alpha \multimap \alpha), n\,\mathtt{Int} \vdash (!\lambda x.z\,\underline{\mathtt{false}}\,x)\{\mathrm{IT}_{n,M,\langle \pi_0, I\rangle}\,/z\} : !(\alpha \multimap \alpha)}{n\,\mathtt{Int} \vdash \lambda f.(!\lambda x.z\,\underline{\mathtt{false}}\,x)\{\mathrm{IT}_{n,M,\langle \pi_0, I\rangle}\,/z\} : \mathtt{Int}_\alpha}\ (\text{abs})$$

which then is generalized to $\mathtt{Int}$, and after abstraction becomes $\underline{\mathtt{pred}}$. Having the result of the iteration (if we plug $\underline{n}$ in place of n) $\langle I, (f)^{n-1}\rangle$ we extract the left component and have (after reverting to pure terms) $\lambda f \lambda x.((f)^{n-1}\,x)$ which is easily seen to reduce to $\underline{n-1}$. If $n = 0$ we extract the identity which yields the right result.

So $\underline{\mathtt{pred}}$ is flat: we can iterate it to get an oblique representation of $\mathtt{sub}$. □

Lemma 4.2.23 (composition, bounded sum, bounded product). *The composition, bounded sum and bounded product schemes are representable in* **EAL**2.

Proof. Composition is easy, we must just use coercion to have all modalities right. So if the f_is are represented by terms $\underline{f_i}$ of obliqueness p_i and g is represented by a term $\underline{g}$ of type $\mathtt{Int}^k \multimap !^q\,\mathtt{Int}$ we first lift $\underline{f_i}$ to the same obliqueness $p = \max(p_i)$. Then we promote $p-1$ times the term $\underline{g}\,\vec{m}$ getting

$$\vec{m} : !^{p-1}\,\mathtt{Int} \vdash \underline{(!^{p-1} g\,\vec{m})} : !^{q+p-1}.$$

In the last promotion we plug the terms $\underline{f_i}\,\vec{n}$, and finally get

$$\vdash \overrightarrow{\lambda n}.(!^p g\,\vec{m})\{\overrightarrow{(\underline{f}\vec{n})/m}\} : \mathtt{Int}^k \multimap !^{q+p}\,\mathtt{Int}\,.$$

For sums and products we have to find a way to use iteration, given that sum and product are flat. So take any g with a flat representation, and any f. We want to represent

$$h(0,\vec{n}) := f(0,\vec{n}),$$
$$h(m+1,\vec{n}) := g(f(m+1,\vec{n}),h(m,\vec{n})).$$

Say that f is represented by $\underline{f}$ of obliqueness q. First we promote the flat representation $\underline{g}$ to a term $\underline{g}'$ of type $(!^p\,\mathtt{Int})^2 \multimap !^p$. Note that $(\underline{g})^+ = (\underline{g})^+$. Then consider the type $\tau := \mathtt{Int} \otimes !^q\,\mathtt{Int}$. By assuming p_1, p_2 and p_3 of type τ, we easily derive

$$\vec{n}' : \mathtt{Int}, p_1, p_2, p_3 : \tau \vdash \langle \underline{\mathtt{succ}}\,(p_1\,\underline{\mathtt{true}}), g'\,(\underline{f}\,(p_2\,\underline{\mathtt{true}})\,\vec{n}')\,(p_3\,\underline{\mathtt{false}})\rangle : \tau.$$

We promote, contract and abstract to get

$$\vec{n}' : !\,\mathtt{Int} \vdash \lambda p.\{(!\langle \underline{\mathtt{succ}}\,(p_1\,\underline{\mathtt{true}}), g'\,(\underline{f}\,(p_2\,\underline{\mathtt{true}})\,\vec{n}')\,(p_3\,\underline{\mathtt{false}})\rangle)\}_{p\to p_1,p_2,p_3} : !\tau \multimap !\tau.$$

Let's call the above term M. Note that if $\underline{g}$ was not flat we would not obtain the same type as output. The effect of M on a pair $\langle \underline{x}, \underline{y}\rangle$ with $\underline{x}$ of type $\mathtt{Int}$ and $\underline{y}$ of type $!^q\,\mathtt{Int}$ integers is that

$$M\,\langle \underline{x}, \underline{y}\rangle \overset{\beta}{\twoheadrightarrow} \langle \underline{x+1}, \underline{g(x,y)}\rangle,$$

where we are really seeing what's happening in pure λ-calculus.

So if we apply it m times to

$$\vec{n}'' : !\,\mathtt{Int} \vdash (!\langle \underline{1}, \underline{f}\,\underline{0}, \vec{n}''\rangle) : !\tau$$

we get the pair $\langle \underline{m+1}, \underline{h(m,\vec{n})}$, if we manage to contract the n's with n''s. Let's denote by N the banged pair above. The iteration scheme yields:

$$n' : !^2\,\mathtt{Int}, n'' : !^2\,\mathtt{Int}, m : \mathtt{Int} \vdash \mathrm{IT}_{n,M,N} : !^2\tau.$$

Let's call K the term obtained from above contracting $\vec{n}'$ and $\vec{n}''$ to $\overrightarrow{\mathtt{coerc}_2\,n}$:

$$\vec{n} : \mathtt{Int}, m : \mathtt{Int} \vdash K : !^2\tau.$$

We prepare apart a term to extract what we need from K.

$$\frac{z : \tau \vdash z\,\underline{\mathtt{true}} : !^q\,\mathtt{Int}}{z : !\tau \vdash (!z\,\underline{\mathtt{true}}) : !^{q+1}\,\mathtt{Int}}\ (\mathrm{prom})$$

With another promotion we plug our iteration in z:

$$\vec{n} : \mathtt{Int}, m : \mathtt{Int} \vdash (!^2 z\,\underline{\mathtt{true}})\{K/z\} : !^{q+2}\,\mathtt{Int}\,.$$

So by abstraction we get the desired representation, with obliqueness $q+2$. □

4.2.3 LAL

Definition 4.2.24 (polytime functions). *Polytime recursive functions* are recursive functions for which there exists a Turing machine that computes them in a number of steps bounded definitely by a polynomial in the arguments of the function.

In 1992 an axiomatic definition of this class of functions was given by Bellantoni and Cook in [BC92].

Definition 4.2.25 (*BC* algebra). We define functions on integers by marking certain arguments as *safe* and the other ones as *normal*, denoting it by $f(\vec{x};\vec{y})$, where the variables to the left of the semicolon are the normal ones and the others are safe. We will now use binary notation for integers. *BC*, *Bellantoni-Cook algebra*, is the least class of function thus marked that contains:

- the constant $\mathbf{0}(;)$ that gives zero;
- the projections $\pi_i^{m,n}$, where $1 \leq i \leq m+n$, with $\pi_i^{m,n}(x_1,\ldots,x_m;x_{m+1},\ldots,x_{m+n}) := x_i$;
- the successors $\mathtt{succ}_i$, where $i = 0, 1$, with $\mathtt{succ}_i(;a) = 2 \cdot a + i = ai$;
- the predecessor $\mathtt{pred}$, with $\mathtt{pred}(;a) = \lfloor a/2 \rfloor$, so that $\mathtt{pred}(0) = 0$ and $\mathtt{pred}(ai) = a$;
- the conditional $\mathtt{if}$, with
$$\mathtt{if}(;a,b,c) = \begin{cases} b & \text{if } a \text{ is even,} \\ c & \text{otherwise.} \end{cases}$$

and closed with respect to the following schemes:

- *safe composition*, that given k functions $f_i(\vec{x};)$, ℓ functions $h_j(\vec{x};\vec{y})$ and a function $g(\vec{s}^k;\vec{t}^\ell)$ gives a function
$$\text{S-COMP}(g,\vec{f},\vec{h})(\vec{x};\vec{y}) := g(\vec{f}(\vec{x};);\vec{h}(\vec{x};\vec{y})).$$
- *safe recursion*, that given a function $g(\vec{y};\vec{z})$, and two functions $h_i(x,\vec{y};\vec{z},w)$ gives a function $f = \text{S-REC}(g,h_0,h_2)$ defined recursively:
$$f(0,\vec{y};\vec{z}) := g(\vec{y};\vec{z}),$$
$$f(xi,\vec{y};\vec{z}) := h_i(x,\vec{y};\vec{z},f(x,\vec{y};\vec{z})).$$

Theorem 4.2.26. *A function $f(\vec{n})$ is polytime if and only if $f(\vec{n};)$ is in BC.*

Proof. See [BC92]. The breakthrough is given by the fact that the place occupied by the recursive call in recursion is safe, so that it cannot be itself a recursion argument: $h_i(x,\vec{y};\vec{z},w)$ cannot be defined by recursion on w. □

Light linear logic is yet another means of capturing this complexity class.

In **LAL** we restrict further the creation of modalities. The idea is that we want to restrict duplication to the minimum necessary, and we want to forbid completely the duplication of duplication. So we may say that we mark in a separate way the resources that where obtained by duplicating something in such a way to be not duplicated again without newly marking it. To do so we need another mark, a new modality called *neutral* or *paragraph* and denoted by §. Basically once we are allowing contraction we mark the resulting type so that it may not be contracted again.

Definition 4.2.27 (types of LAL). Types are built from $\mathbb{V}$ with the following grammar:

$$\mathbb{T}_{\mathbf{LAL}} ::= \mathbb{V} \mid \mathbb{T}_{\mathbf{LAL}} \multimap Ty_{\mathbf{LAL}} \mid !Ty_{\mathbf{LAL}} \mid \S Ty_{\mathbf{LAL}}.$$

Both $\S\tau$ and $!\tau$ are both called *exponential* or *modal*.

Definition 4.2.28 (rules of LAL).

$$\frac{}{x : \tau \vdash x : \tau} \text{ (var)}$$

$$\frac{A, x : \sigma \vdash M : \tau}{A \vdash (\lambda x.M) : \sigma \multimap \tau} \text{ (abs)} \qquad \frac{A \vdash M : \sigma \multimap \tau \quad B \vdash N : \sigma}{A, B \vdash (M\ N) : \tau} \text{ (app)}$$

$$\frac{A \vdash N : !\sigma \quad \overrightarrow{x : !\sigma}, B \vdash M : \tau}{A, B \vdash M[N/\vec{x}] : \tau} \text{ (con)} \qquad \frac{A \vdash M : \tau}{A, B \vdash M : \tau} \text{ (weak)}$$

$$\frac{\overrightarrow{B \vdash N : a\sigma^n} \quad \overrightarrow{x : \sigma} \vdash M : \tau}{\vec{B}, A \vdash M[\overrightarrow{N/x}] : b\tau} \text{ (prom)}$$

where in the promotion rule $\vec{a}$ and b are exponentials, and if $b = !$ then $n \leq 1$ and $a_1 = !$ (if $n = 1$). Otherwise if $b = \S$ there is no restriction in what modalities and how many of them are used. If $b = !$ we call it a *!-promotion*, in the other case it will be called a *§-promotion*.

Again adding polymorphism is direct.

Definition 4.2.29 (types and rules of LAL2). We extend $\Lambda_{\mathbf{LAL}}$ to $\Lambda_{\mathbf{LAL2}}$ adding the quantifier:

$$\mathbb{T}_{\mathbf{LAL2}} ::= \mathbb{V} \mid \mathbb{T}_{\mathbf{LAL2}} \multimap \mathbb{T}_{\mathbf{LAL2}} \mid !\mathbb{T}_{\mathbf{LAL2}} \mid \S\mathbb{T}_{\mathbf{LAL2}} \mid \forall\mathbb{V}.\mathbb{T}_{\mathbf{LAL2}}.$$

Rules are extended with the usual (gen) and (ins).

Example 4.2.30. Church integers are typable with a type we will again call `Int` defined by

$$\texttt{Int} := \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha).$$

In fact we can easily adapt the derivation seen for **AL** in example 4.1.5, by replacing the dereliction with a §-promotion. However we will mainly use the binary representation of integers, given by type

$$\texttt{BInt} := \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha).$$

Given a sequence $w \in \{0,1\}^*$ of length k we have a representation

$$\underline{w} := \lambda f_0 \lambda f_1 \lambda x. f_{w_0}(f_{w_1} \dots (f_{w_k}\, x) \dots).$$

In particular if we adopt it to represent integers we take $\underline{n} := \underline{w}$ where w is the binary expansion of n read from the least to the most significant digit, with the difference that we represent 0 with the empty string $\lambda f_0 \lambda f_1 \lambda x.x$. If we see it as a free structure (see 3.2.6), f_i represents the constructor that appends i to the string. We will have to deal with the trailing zeros.

With the new modality we are not in linear logic anymore, strictly speaking. However we can revert back to **EAL** (and thus to AL) with a simple function.

Definition 4.2.31 (injection into EAL). The injection of **LAL** types, sequents and derivations is done by turning § into a ! and all §-promotions into normal ones.

Though again the logic system behind this typing system enjoys the intended bound on cut-elimination, in fact the same does not hold for terms typable with **LAL**. Not only strong polynomiality is absent for these terms, but even weak polynomial normalization does not hold without reverting to sharing.

Take the term $\underline{n}\,(\lambda y.y\,x\,x)\,z$, with $\underline{n}$ a Church integer. We can see it is typable with $\S!\alpha$ within the environment $y : !((!\alpha)^2 \multimap !\alpha), z : !!\alpha$. In fact we type

$$y : (!\alpha)^2 \multimap !\alpha \vdash \lambda x.y\,x\,x : !\alpha \multimap !\alpha,$$

which promoted and combined with $\underline{n} : \texttt{Int}_\alpha$ gives

$$y : !((!\alpha)^2 \multimap !\alpha) \vdash \underline{n}(\lambda x.y\,x\,x) : \S(\alpha \multimap \alpha).$$

We plug it into w in the §-promotion of

$$w : !\alpha \multimap !\alpha, z : !\alpha \vdash w\,z : !\alpha$$

and we get

$$y : !((!\alpha)^2 \multimap !\alpha), z : !!\alpha \vdash \underline{n}(\lambda x.y\,x\,x)\,z : \S!\alpha.$$

The size of this term is proportional to n. However its normal form M_n satisfies $M_{n+1} = y\,M_n\,M_n$, so that the size of the normal form is proportional to 2^n. So using a Turing machine takes up exponential space, and thus exponential time.

However if we instead see the derivation of the type for the normal forms M_n we have a representation linear in size with respect to n: we can derive n copies of

$$z : !\alpha, y : (!\alpha)^2 \multimap !\alpha \vdash y\, z\, z : !\alpha.$$

Then we §-promote one without plugging anything, and then we §-promote them one at a time plugging in z the derivation obtained at the previous step. With a final contraction we identify all the different ys. The size is clearly linear in n.

However not everything is lost. In fact if we restrict our attention to `BInt` and functions (closed terms) on `BInt`, we will always get an at least weak polynomial reduction. And the derivation becomes even strongly polynomial if we use the cut-elimination of proof nets associated to the derivations or equivalently, as we did for **EAL**, terms in which sharing and promotion are explicitly written.

Definition 4.2.32 (Λ^{EA}). The set of *light affine terms with sharing* extends the elementary ones. They are built from $\mathcal{V}$ with the following grammar

$$\Lambda^{\mathrm{EA}} ::= \mathcal{V} \mid (\Lambda^{\mathrm{EA}}\ \Lambda^{\mathrm{EA}}) \mid (\lambda\mathcal{V}.\Lambda^{\mathrm{EA}}) \mid \{\Lambda^{\mathrm{EA}}\}_{\Lambda^{\mathrm{EA}} \to \vec{\mathcal{V}}} \mid (a\Lambda^{\mathrm{EA}})\{\overrightarrow{\Lambda^{\mathrm{EA}}/\mathcal{V}}\},$$

where a is a modality. An additional condition on the construction of these terms, apart from the ones given for Λ^{EA}, is that a term boxed with ! cannot have more than one free variable.

Again there is a direct translation of the rules for standard **LAL** and **LAL**2 into a system based on Λ^{EA}: we will call these systems $\mathbf{LAL}_s$ and $\mathbf{LAL}2_s$ respectively. The rules are exactly the same of $\mathbf{EAL}_s$ and $\mathbf{EAL}2_s$, apart from the differentiation between ! and § promotions, in the same way as it is outlined for regular **LAL** rules.

Definition 4.2.33 (reduction on Λ^{EA}). We define the same one step reduction we have seen for **EAL** in 4.2.12, with the following differences and additions:

- $\to_{!-!}$ gets split in $\to_{!-!}$, $\to_{\S-!}$ and $\to_{\S-\S}$, by changing the modality of the boxed terms. In $\to_{\S-!}$ we mean that the boxed term being plugged is the one with ! modality. Note that with the condition on free variables in banged terms, $\to_{!-!}$ may be rewritten as

$$(!M)\big\{(!N)\{P/y\}/x\big\} \to_{!-!} (!M[N/x])\{P/y\},$$

  where eventually $\{P/y\}$ is empty.
- There is no $\to_{!-c}$. This comes from the fact that we cannot freely move around a contraction in a derivation, because it could have been done to make a !-promotion possible by cutting down to one the number of free variables. $\to_{\S-c}$ with § subsituting ! is permitted, as §-promotion does not have any condition on the number of type assumptions.

Note that the size-increasing power of $\rightarrow_{dup}$ is drastically lessened, as it becomes

$$\{M\}_{(!N)\{P/x\}\rightarrow\vec{y}} \rightarrow_{dup} \{M[(!N)\{x_1/x\}/y_1,\ldots,(!N)\{x_k/x\}/y_k]\}_{P\rightarrow\vec{x}}.$$

Another simpler approach is due to Baillot and Terui [BT04], by introducing a new type assignment system called **DLAL**: *dual light affine logic*. The main advantages of such an approach are that it guarantees strong polynomiality on pure λ-terms, and though it is a proper subsystem of **LAL**, it retains the property of being polytime complete. In addition it enjoys subject reduction property. This is achieved by letting the bang appear only in the form $!\sigma \multimap \tau$. In fact we erase any reference to !, reintroduce the intuitionistic arrow and keep track of what type assumptions should have been marked by ! in the environment.

Definition 4.2.34 ($\mathbb{T}_{\mathbf{DLAL}}$ and $\mathbb{T}_{\mathbf{DLAL2}}$). Types are built from $\mathbb{V}$ with the following grammar.

$$\mathbb{T}_{\mathbf{DLAL}} ::= \mathbb{V} \mid \mathbb{T}_{\mathbf{DLAL}} \multimap \mathbb{T}_{\mathbf{DLAL}} \mid \mathbb{T}_{\mathbf{DLAL}} \rightarrow \mathbb{T}_{\mathbf{DLAL}} \mid \S\mathbb{T}_{\mathbf{DLAL}}.$$

$\sigma \rightarrow \tau$ is called *intuitionistic implication*, while the other arrow is name *linear implication*. Arrows of any kind associates to the right as usual. $\sigma^k \rightarrow \tau$ is short for $\sigma \rightarrow \ldots \rightarrow \sigma \rightarrow \tau$ with k copies of σ, and similarly $\sigma^k \multimap \tau$ wit all linear arrows. Second order types are gained adding to the above rules $\forall\mathbb{V}.\mathbb{T}_{\mathbf{DLAL2}}$.

Sequents in **DLAL** are different from the usual ones: we distinguish between *intuitionistic* type assumptions and *linear* ones. Such separation is denoted by means of a semicolon: sequents have the form $A; B \vdash M : \tau$. A will be called the *intuitionistic side* of the sequent, while B is the *linear side*.

Definition 4.2.35 (rules of DLAL and DLAL2). **DLAL** is given by the following rules, where each variable introduced by (var), (weak) or (con) is fresh:

$$\frac{}{;x:\tau\vdash x:\tau}\ (\text{var})$$

$$\frac{A; x:\sigma, B\vdash \lambda x.M:\tau}{A; B\vdash \lambda x.M:\sigma\multimap\tau}\ (\text{l-abs}) \qquad \frac{A_1; B_1\vdash M:\sigma\multimap\tau \quad A_2; B_2\vdash N:\sigma}{A_1,A_2; B_1,B_2\vdash M\,N:\tau}\ (\text{l-app})$$

$$\frac{x:\sigma, A; B\vdash \lambda x.M:\tau}{A; B\vdash \lambda x.M:\sigma\rightarrow\tau}\ (\text{i-abs}) \qquad \frac{A_1; B_1\vdash M:\sigma\rightarrow\tau \quad ; z:\rho\vdash N:\sigma}{A_1, z:\rho; B_1\vdash M\,N:\tau}\ (\text{i-app})$$

$$\frac{A_1; B_1\vdash M:\tau}{A_1,A_2; B_1,B_2\vdash M:\tau}\ (\text{weak}) \qquad \frac{\vec{x}:\sigma, A; B\vdash M:\tau}{y:\sigma, A; B\vdash M[y/\vec{x}]:\tau}\ (\text{con})$$

$$\frac{\overrightarrow{A; B\vdash N:\S\sigma} \quad ;\overrightarrow{x:\sigma}, C\vdash M:\tau}{\vec{A};\vec{B},\S C\vdash M[\overrightarrow{N/x}]:\S\tau}\ (\text{prom})$$

The variants of abstraction and application are called *linear* and *intuitionistic*. In the intuitionistic application $z : \rho$ can be absent. We call a promotion *basic* if it has no left premise, i.e. if there is no plugging.

Second order is achieved by adding the usual (gen) and (ins).

Remark 4.2.36. The condition of not allowing §-promotion with a non empty intuitionistic side, if seen in **LAL**, leads to not allowing appending additional modalities once ! has been put. The absence of the ! box is recuperated by the special intuitionistic application: note that it can be applied only to derivations with at most one type assumption, and this gets moved in the intuitionistic side.

Example 4.2.37. Church integers are represented by the type, which we will again call the same, $\mathtt{Int} := \forall\alpha.(\alpha \multimap \alpha) \to \S(\alpha \multimap \alpha)$. In fact we have a valid derivation for $\vdash \underline{n} : \mathtt{Int}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{;f_n : \alpha \multimap \alpha \vdash f_n : \alpha \multimap \alpha}\,(\text{var}) \quad \dfrac{}{;x : \alpha \vdash x : \alpha}\,(\text{var})}{;f_n : \alpha \multimap \alpha, x : \alpha \vdash (f_n\, x) : \alpha}\,(\text{l-app})}{\vdots}}{\dfrac{}{;f_1 : \alpha \multimap \alpha \vdash f_1 : \alpha \multimap \alpha}\,(\text{var}) \quad ;\vec{f} : \alpha \multimap \alpha, x : \alpha \vdash (f_2\,(\dots(f_n\, x)\dots) : \alpha}\,(\text{l-app})}{\vec{f} : \alpha \multimap \alpha, x : \alpha \vdash (f_1(\dots(f_n\, x)\dots) : \alpha}\,(\text{l-abs})}{;\vec{f} : \alpha \multimap \alpha \vdash \lambda x.(f_1(\dots(f_n\, x)\dots) : \alpha \multimap \alpha}\,(\text{prom})}{\vec{f} : \alpha \multimap \alpha; \vdash \lambda x.(f_1(\dots(f_n\, x)\dots) : !\alpha}\,(\text{con})}{f : !\alpha \multimap \alpha \vdash \lambda x.(f^n\, x) : !(\alpha \multimap \alpha)}\,(\text{i-abs})}{\vdash \lambda f\lambda x.(f^n\, x) : (\alpha \multimap \alpha) \to \S\alpha \multimap \alpha}\,(\text{gen})}{\vdash \lambda f\lambda x.(f^n\, x) : \mathtt{Int}}$$

As always for $\underline{1}$ no contraction is needed and for $\underline{0}$ we have to apply weakening.

We may also redefine `BInt` in **DLAL**2 as

$$\mathtt{BInt} := \forall\alpha.(\alpha \multimap \alpha) \to (\alpha \multimap \alpha) \to \S(\alpha \multimap \alpha),$$

and the derivation $\underline{w} : \mathtt{BInt}$ can be carried out like the one for `Int`, by replacing the last contraction with two distinct ones.

Let's see how this relates to standard **LAL**.

Definition 4.2.38 (injection of DLAL into LAL). The injection works in way similar to the embedding of intuitionistic into linear logic, so we will denote it in the same manner. It is defined

by:

$$\hat{\alpha} := \alpha,$$
$$\widehat{\sigma \multimap \tau} := \hat{\sigma} \multimap \hat{\tau},$$
$$\widehat{\sigma \to \tau} := !\hat{\sigma} \multimap \hat{\tau},$$
$$\widehat{\S\tau} := \S\hat{\tau},$$
$$\widehat{\forall\alpha.\tau} := \forall\alpha.\hat{\tau}.$$

DLAL2 sequents are translated prepending ! only to the intuitionistic side:

$$\Gamma = (A; B \vdash M : \tau) \mapsto \hat{\Gamma} := (!\hat{A}, \hat{B} \vdash M : \hat{\tau}),$$

where here $\hat{A}$ is defined by $\hat{A}(x) := \widehat{A(x)}$.

Proposition 4.2.39. *Every derivation $\mathcal{D} \rightsquigarrow \Gamma$ in* **DLAL**2 *can be translated into $\hat{\mathcal{D}} \rightsquigarrow \hat{\Gamma}$ in* **LAL**2.

Proof. Reasoning by induction we just need to translate each rule. Of these (var), (weak), (con), (l-abs), (l-app), (gen) and (ins) pose no problems. Also (i-abs) is easy.

(i-app): By induction hypothesis we have derivations

$$!A_1, B_1 \vdash_{\mathbf{LAL2}} M : !\hat{\sigma} \multimap \hat{\tau} \quad \text{and} \quad z : \hat{\rho} \vdash M : \hat{\sigma}.$$

We !-promote the second one and apply (app) getting the desired result.

(prom): This one is translated into a §-promotion. The passage from linear to intuitionistic side is rendered by choosing ! as modalities for those type assumptions.

□

Now let us see some properties of **DLAL**2.

Proposition 4.2.40 (subterm typing). DLAL2 *enjoys subterm typing.*

Proof. the proof is almost trivial. We reason by induction on the derivation. All rules but (con) and (prom) are easy.

For (con) we have to apply induction hypothesis making the necessary changes to the subterm for which we are searching the typing, i.e. if the term is $M = M'[x/\vec{y}^n]$, then the subterm is of the form $N = N'[x/\vec{y}^k]$ with $k \leq n$, possibly $k = 0$. So (if N is a proper subterm of M) we apply induction hypothesis to search a typing for N', and then eventually apply back contraction.

For (prom) we have that $M = M'[\overrightarrow{P/x}^n]$. If N is a subterm of one of the P_is or else if it does not contain any of the variables in $\vec{x}$ we can neatly apply induction hypothesis to one of the

premises. If on the other hand we have $N = N'[\overrightarrow{P/x^k}]$, we apply induction hypothesis to the right premise to find a typing for N' and then do a promotion plugging in the necessary P_is.

Note that because of the way a promotion works a typing for a subterm *is not* a subderivation of the typing for the main term. □

Proposition 4.2.41 (substitution). DLAL2 *enjoys the following properties:*

- *If* $\mathcal{D} \rightsquigarrow A; B \vdash_{\mathbf{DLAL2}} M : \tau$ *then*

$$A[\overrightarrow{\sigma/\alpha}]; B[\overrightarrow{\sigma/\alpha}] \vdash M : \tau[\overrightarrow{\sigma/\alpha}]$$

is derivable with a derivation with the same structure of $\mathcal{D}$.

- *If* $A; B \vdash_{\mathbf{DLAL2}} N : \sigma$ *and* $C; x : \sigma, D \vdash_{\mathbf{DLAL2}} M : \tau$ *then*

$$A, C; B, D \vdash M[N/x] : \tau$$

is derivable with a derivation which has size less than the sum of the sizes of the two derivations.

- *If* $; z : \rho \vdash_{\mathbf{DLAL2}} N : \sigma$ *(eventually with empty environment) and* $A, \vec{x} : \sigma; B \vdash_{\mathbf{DLAL2}} M : \tau$ *then*

$$A, z : \rho; B \vdash M[N/\vec{x}]$$

is derivable.

Proof. We show the properties one at a time:

- The proof carries out like in the proof of the analogous proposition 3.1.7. If we α-convert all the type variables so that there is no variable clash, we see that the substitution of type variables does not change the behaviour of any of the rules.
- By induction on the size of the derivation of $C; x : \sigma, D \vdash M : \tau$. If it is made only of the (var) rule then it must be on x, so we can directly replace it with the typing for N. All the other cases but (prom) are straightforward. Recall that as x is in the linear side it is not repeated.
  For (prom) we have to possibilities for $x : \sigma$. One is that σ is of the form $\S\sigma'$, and $x : \sigma'$ is present in the right premise of the rule: in this case we may plug the derivation for N to get the desired result, using directly the formulation of the (prom)-rule rather than the induction hypothesis. The other possibility is that $x : \sigma$ is in the environment of one of the left premises: in this case we use induction hypothesis and get what we need.
- By induction on the size of the derivation of $A, x : \sigma; B \vdash M : \tau$. The last rule just cannot be (var). All the other rules but (i-app), (prom) and (con) are straightforward, as they preserve the intuitionistic side.

If all $\vec{x} : \sigma$ are present in the left premise of (i-app), then application of the induction hypothesis is straightforward. If instead one of them (and at most one can) is the only assumption in the linear side of the right premise then we have the following situation:

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \vdots & \vdots \\ A, \vec{x}^{n-1} : \sigma; B \vdash M_1 : \tau' \to \tau & ; x_n : \sigma \vdash M_2 : \tau'\end{array}}{A, \vec{x}^{n} : \sigma; B \vdash (M_1\, M_2) : \tau}\ \text{(i-app)}$$

Then we apply induction hypothesis on $\mathcal{D}_1$ and the preceding point to $\mathcal{D}_2$, and we get

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D}'_1 & \mathcal{D}'_2 \\ \vdots & \vdots \\ A, z : \rho; B \vdash M_1[N/\vec{x}^{n-1}] : \tau' \to \tau & ; z' : \rho \vdash M_2\big[N[z'/z]/x_n\big] : \tau'\end{array}}{A, z : \rho, z' : \rho; B \vdash (M_1[N/\vec{x}^{n-1}]\, M_2\big[N[z'/z]/x_n\big]) : \tau}\ \text{(i-app)}}{A, z : \rho; B \vdash (M_1\, M_2)[N/\vec{x}]}\ \text{(con)}$$

For (prom), we have that $\vec{x} : \sigma$ is partitioned between the left premises and the linear side of the right premise. For each of the premises we apply the induction hypothesis, using for each premise a different variable instead of z (using a renaming as in the preceding case). As for the right premise, we repeatedly apply the preceding point, using different variables for z, which all get in the linear side. Applying back (prom) and shifting all the versions of z to the intuitionistic side in order then to contract all to $z : \rho$ gives the desired result. Note that if there is no $z : \rho$ then no renaming is needed.

In (con), if none of $\vec{x} : \sigma$ is the contracted assumption then the application of the induction hypothesis is direct. If instead say $x_n : \sigma$ is the assumption contracted from assumptions $\vec{y} : \sigma$, we apply induction hypothesis directly to he subderivation leading to $A, \vec{x}^{n-1}, \vec{y} : \sigma; B \vdash M' : \tau$, where $M'[x_n/\vec{y}] = M$.

□

Definition 4.2.42. We will say that a **DLAL**2 derivation is in *canonical* form if

- it is in (ins) before (gen) form (see 3.1.6);
- no conclusion of a (prom) rule is a left premise of another (prom) rule;
- every (weak) rule is either the last rule of the derivation or else it introduces only one assumption which gets abstracted away in the following rule.
- every (con) rule is either followed only by a chain of contractions followed in turn by at most a single weakening that concludes the derivation (as by the preceding property), or else it contracts an assumption which gets abstracted away by an (i-abs) immediately afterwards.

Proposition 4.2.43. *For every $\mathcal{D} \rightsquigarrow A; B \vdash M : \tau$ there is a derivation $\mathcal{D}'$ leading to the same sequent which is in canonical form.*

Proof. To obtain the first property we reason as we did for system **F** (see proposition 3.1.8). We can do it because of the first point of the previous proposition.

To obtain the second property we reason by induction on the size of the derivation. Suppose we have a (prom) rule which is a left premise of another (prom) rule. We have then the following situation:

$$\dfrac{\overrightarrow{A; B \vdash N : \S\sigma} \qquad \dfrac{\overrightarrow{C; D \vdash Q : \S\eta} \quad ; E_1, E_2, \overrightarrow{z : \eta} \vdash P : \rho}{\vec{C}, E_1; \vec{D}, \S E_2 \vdash P[\overrightarrow{Q/z}] : \S\rho}\ (\text{prom}) \qquad ; F_1, F_2, \overrightarrow{x : \sigma}, y : \rho \vdash M : \tau}{\vec{A}, \vec{C}, E_1, F_1; \vec{B}, \vec{D}, \S E_2, \S F_2 \vdash M\big[\overrightarrow{N/x}, P[\overrightarrow{Q/z}]/y\big] : \S\tau}\ (\text{prom})$$

Then we substitute $; E_1, E_2, \overrightarrow{z : \eta} \vdash P : \rho$ into y of the right premise of the second (prom), and we obtain a derivation of

$$; F_1, F_2, E_1, E_2, \overrightarrow{x_\sigma}, \overrightarrow{z : \eta} \vdash M[P/y] : \tau,$$

with size no bigger than that of the two derivations. Then we apply back (prom), putting together the left premises of the two rules. Because there is no variable collision we get a derivation for the same final sequent. Having deleted one rule means we also have a derivation with strictly less size, so we can apply induction hypothesis and get a derivation that satisfies the second property. Also the first property is trivially preserved.

For the third property we may reason by induction on the sum of the depths of all the (weak) rules contradicting the property (depth in the derivation seen as that of a tree), which should be regarded as 0 if there is none. Suppose now we have such a (weak) rule.

If the following rule is an abstraction on one of the assumptions introduced by (weak), then we can leave a (weak) rule only on that assumption and add the other ones by weakening after the abstraction, eventually merging the weakening with one following the abstraction. This lowers the depth of the eventual offending weakening.

If the following rule is a (con) on some of the assumptions introduced by weakening, we may completely absorb the contraction in the weakening if it contracts only assumptions given by the (weak) rule, or else erase both the introduced assumptions and the contraction if it contracts all weakened assumptions but one, or else just erase the weakened assumptions which should be contracted and postpone the weakening of the rest of the assumptions after the contraction. Again if there is another weakening after the contraction we merge the two.

For all the other rules, and the other cases of the two rules already treated, it is straightforward that we can lower the application of the (weak) rule. Doing this does not strip the derivation of the two preceding properties.

The fourth property is gained in a similar way: we just keep lowering the contraction until we get to where it is really needed, or eventually arrive to the final chain of contractions and weakening. The only rules with which it cannot commute are (i-abs) on the assumption being contracted[2]. Note that if the contraction is followed by another contraction that contracts the same assumption, the two merge rather than commute with one another. Once again the other properties of canonical derivations are preserved. □

Now we get to one of the main differences with standard **LAL**: subject reduction.

Proposition 4.2.44. DLAL2 *enjoys subject reduction.*

We still need two lemmas.

Lemma 4.2.45 (abstraction property). *Suppose $\mathcal{D}$ is a canonical derivation leading to $A; B \vdash \lambda x.M : \tau$ whose last rule is neither (weak) nor (con). Then the last rule is the one introducing the outermost connective of τ.*

Proof. By exclusion of the other cases done by induction.

Suppose the last rule is (ins): because the derivation is canonical the rule immediately preceding it cannot be (weak) or (con), and the term is still an abstraction. Therefore we can apply induction hypothesis to the premise, which has as outermost connective $\forall$. However this would contradict the (ins) before (gen) property.

Both the application rules and the (var) rule are excluded as they cannot yield an abstraction. So the only rules remaining are those that introduce a connective on the type. □

Corollary 4.2.46. *No left premise of a (prom) rule in a canonical derivation can be the typing of an abstraction.*

Proof. As the derivation is canonical the last rule of one of the left premises is neither (weak) nor (con). So if it were a typing of an abstraction by the above lemma its last rule would be the one corresponding to the outermost connective, which is §. But this would be a contradiction, as in a canonical derivation no (prom) rule can be a left premise of another (prom) rule. □

Proof of subject reduction. Take $A; B \vdash_{\mathbf{DLAL2}} M : \tau$ and $M \xrightarrow{\beta} N$. We have to show that $A; B \vdash_{\mathbf{DLAL2}} N : \tau$. By proposition 4.2.43 we can take a canonical derivation for M. Now we reason by induction on this derivation.

(var): No redex to contract.

[2] here a difference from standard **LAL** should be noted: in **LAL** another rule with which contraction cannot commute is the !-promotion. Here not only there is no !-promotion, but also the rule that simulates it, (i-app), does not permit contraction on the right premise,as no linear side is accepted.

(weak) and (con): Trivial. Note that if reducing a redex makes disappear all, or all but one, the variables being contracted there is no need to apply back contraction.

(l-abs) and (i-abs): Application of the inductive hypothesis is trivial as all the redexes are in the term being abstracted.

(l-app): We have $M = M_1\, M_2$, and the last part of the derivation is

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdots & \vdots\\ A_1; B_1 \vdash M_1 : \tau' \multimap \tau & A_2; B_2 \vdash M_2 : \tau'\end{array}}{A_1, A_2; B_1, B_2 \vdash M_1\, M_2 : \tau}\ \text{(l-app)}$$

with $\mathcal{D}_1$ and $\mathcal{D}_2$ still canonical. If the redex being contracted is in M_1 (resp. M_2) application of the inductive hypothesis to $\mathcal{D}_1$ (resp. $\mathcal{D}_2$) gives the desired result after we apply back (l-app). If instead the redex is $M_1\, M_2$ itself, then M_1 is an abstraction. Moreover $\mathcal{D}_1$ cannot end with a (weak) or a (con) as the derivation is canonical. Applying the abstraction property we get that in fact the last rule of $\mathcal{D}_1$ must be (l-abs), so that $\mathcal{D}_1$ is

$$\dfrac{\begin{array}{c}\mathcal{D}'_1\\ \vdots\\ A_1; B_1, x : \tau' \vdash M'_1 : \tau\end{array}}{A_1; B_1 \vdash \lambda x.M'_1 : \tau' \multimap \tau}\ \text{(l-abs)}$$

So we can apply a substitution replacing x in M'_1 with M_2, so that we get a derivation of

$$A_1, A_2; B_1, B_2 \vdash M'_1[M_2/x] : \tau$$

an in fact $N = M'_1[M_2/x]$.

(i-app): As for (l-app), with the only difference being the different version of substitution lemma being employed.

(prom): If the redex being reduced is in the term being typed by one of the premises we can just apply induction hypothesis and then back the (prom) rule. Another case would be that a redex is created by the substitution done by the promotion, but this would imply that one of the left premises is an abstraction which is impossible for a canonical derivation as seen in the corollary above.

(gen) and (ins): Here the application of the induction hypothesis is straightforward.

□

As last property we will see directly on the **DLAL**2 terms the polynomiality of a reduction. In fact we have a stronger property.

Definition 4.2.47 (exponential depth of a derivtion). The exponential depth $\mathrm{d}(\mathcal{D})$ of a **DLAL**2 derivation is the maximal number of right premises of (prom) and (i-app) rules in a branch.

Theorem 4.2.48 (strong polytime normalization). *Given a term M typable in* **DLAL**2 *with a derivation of exponential depth d, it reduces to its normal form in at most $|M|^{2^d}$ β-reduction steps, and in time proportional to $|M|^{2^{d+2}}$ in a Turing machine. This result is independent of the reduction chain.*

This result however is shown by injecting **DLAL**2 terms into an alternative definition of Λ^{EA} introduced by Terui in [Ter02] and there shown to be strongly polynomial. We refer to [BT04] and [Ter02] to get a grasp of the proof.

We show a sketch of proof of the weaker form of the above theorem.

Theorem 4.2.49 (weak polytime normalization). *There is a reduction strategy that yields the result of the above theorem.*

Proof. (sketch) The idea is to look up in the typable terms a notion of box nesting present in proof nets. In order to do this we decorate the abstractions adding a number indicating the depth, specifying also whether the abstraction is linear or intuitionistic.

So let us temporarily work with the *stratified terms* defined by

$$\Lambda_s := \mathcal{V} \mid (\lambda^{\mathbb{N}}\mathcal{V}.\Lambda_s) \mid (\lambda^{\mathbb{N}!}\mathcal{V}.\Lambda_s) \mid (\Lambda_s\,\Lambda_s),$$

and let's denote by $\lambda^{d\dagger}.M$ a term that is either $\lambda^{d}.M$ or $\lambda^{d!}.M$. d in λ^d is called the exponential depth of the abstraction, or simply depth if no misunderstanding is possible. We denote by $M[+1]$ the effect of adding one to all the depths of the abstractions. Now we adapt the derivation rules to this new notion, in way such that every **DLAL**2 derivation on pure terms induces a unique derivation of a term with exponential depths. The rules being changed are:

$$\frac{A; x:\sigma, B \vdash M:\tau}{A; B \vdash \lambda^{0}x.M : \sigma \multimap \tau}\ (\text{l-abs}) \qquad \frac{x:\sigma, A; B \vdash M:\tau}{A; B \vdash \lambda^{0!}x.M : \sigma \to \tau}\ (\text{i-abs})$$

$$\frac{A; B \vdash M_1 : \sigma \to \tau \quad ; z:\rho \vdash M_2}{A, z:\rho; B \vdash (M_1\, M_2[+1]) : \tau}\ (\text{i-app}) \qquad \frac{\overrightarrow{A; B \vdash N : \S\sigma} \quad ; \overrightarrow{x:\sigma}, C \vdash M:\tau}{\vec{A}, \vec{B}, \S C \vdash M[+1]\big[\overrightarrow{N/x}\big] : \S\tau}\ (\text{prom})$$

Note that raising the exponential depth in promotion is done before subsituting the terms, which are already ad the right depth.

It is trivial to see that with the modifications to the rules depicted above we can decorate any **DLAL**2 term with the depth, and that such decoration is unique (with respect to **DLAL**2

derivations). Also all the result seen for **DLAL**2 such as substitution, existence of canonical derivations, abstraction property and subject reduction still hold for **DLAL**2 as a type assignment for stratified terms.

We say a redex is of depth d if its main abstraction is of depth d. Now the main property of stratified terms is that

Lemma 4.2.50. *Reducing a redex of depth d does not create a redex of depth less than d.*

Proof. The only way for a redex to be created by a reduction is if the term being substituted is an abstraction or the term in which we substitute is an abstraction: so it is sufficient to see that it is not possible for a **DLAL**2 typable stratified term to have subterms of the form

1. $(\lambda^{d\dagger}x.M)\,(\lambda^{e\dagger}y.N)$ or
2. $\lambda^{d\dagger}x\lambda^{e\dagger}y.M$

with $e < d$. We reason by induction on a canonical derivation.

If the last rule is an abstraction of any type the introduced abstraction is of depth 0, therefore no term of the second type can be added. Induction hypothesis guarantees us the result.

If it is an application of any kind yielding $(\lambda^{d\dagger}x.M)\,(\lambda^{e\dagger}y).N$, then by abstraction property the last rule of the left premise must be a corresponding abstraction, so that in fact d must be 0 and so $e < d$ just can't be. As before induction hypothesis does the rest.

If the last rule is (prom) then none of the left premises types an abstraction, so that no new subterms of one of the two types are created, which together with induction hypothesis gives the desired result.

The other rules do not introduce new subterms of one of the two types, so that induction hypothesis suffices. □

Note also that redexes are created using abstraction that were already present: so the total depth of the derivation is preserved, we cannot create an abstraction of higher depth out of nowhere.

The intended strategy is one that starting from depth 0 reduces all the redexes at a fixed depth before going on to the next depth. The above lemma guarantees that this in fact leads to the normal form, as eventual new redexes are created at the same or higher depth, and so there is no way of missing some redexes.

Let us denote by $\rightarrow_d$ for some d the one step β-reduction restricted to contracting redexes of depth d, and as usual by $\twoheadrightarrow_d$ its reflexive and transitive closure. We need he following lemma

Lemma 4.2.51. *If $M \twoheadrightarrow_d N$ then the number of steps in the reduction is bounded by $|M|$.*

Proof. Cosder the two cases of redexes being reduced: in case of $(\lambda^d x.M)\, N$ then as x was abstraced from the intuitionistic side the number of its occurrences in M is at most one, so that eventual abstractions of depth d in N are not duplicated. If the situation is $(\lambda^{d!} x.M)\, N$ instead, we can prove by induction on a canonical derivation that N has no abstractions of depth d. Note how the induction hypothesis goes down to considering $d-1$ when passing the right premise of (i-app) or (prom). In fact the key case is that in which the last rule is (i-app) and the redex being considered is the term itself: by the abstraction property the last rule on the left is (i-abs) and so $d = 0$, while all the depths in N are raised by one and thus cannot be 0. So also in this case the number of abstractions of depth d decreases by one. So the length of the reduction is bounded by the number of abstractions at depth d which in turn is bounded by the size of the term. □

Then we go on proving

Lemma 4.2.52. *If* $|M| \geq 2$ *and* $M \twoheadrightarrow_d N$ *then* $|N| \leq |M|\,(|M| - 1)$.

Proof. We reason by looking up what happens to a given symbol of the construction tree of M (an abstraction, an application or a variable as a leave). If it gets involved in the reduction of a linear redex then no duplication happens: a single occurrence of variable bounded at depth d disappears and no other one gets duplicated. If instead the symbol is in N in a reduced redex of form $(\lambda^{d!} x.M)\, N$, then it gets repeated say n times, and in the meanwhile n variables bound at depth d (the n occurrences of x) disappear, and we have already seen in the previous lemma that no other ones can be duplicated in this way. The symbol gets in a subterm to be duplicated again only if to this subterm an intuitionistic abstraction has been applied. But the condition on (i-app) means that such subterm can have only one free variable, so that there is only one place the symbol can find himself in, and this in turn implies that when the subterm is duplicated, then again to every variable bounded at depth d that disappears in the process it corresponds a single new occurrence of the symbol.

So, for any given symbol of the construction tree of M the number of its copies at the end of the reduction sequence is either 1 or in any case bounded by the number of variable occurrences bounded at depth d in M, which is trivially bounded in turn by $|M| - 1$. So the number of symbols in N, i.e. its size, is bounded by the number of symbols in M times $|M| - 1$. □

Now we can get to the final result. Take a chain of reductions

$$M = M_0 \twoheadrightarrow_0 M_1 \twoheadrightarrow_1 \ldots M_d \twoheadrightarrow_d M^*.$$

If some of the M_i has size less than 2 then it has no redexes and the chain of reductions ends there, so we can suppose $|M_i| \geq 2$. The whole length of the reduction is bounded by $\sum_{i=0}^{d} |M_i|$ by

lemma 4.2.51. Then by induction on d we get that $\sum_{i=0}^{d} |M_i| \leq |M|^{2^d}$, using as base $|M_0| \leq |M|^{2^0}$ and as step:

$$\sum_{i=0}^{d} |M_i| = \sum_{i=0}^{d-1} |M_i| + |M_d| \leq \sum_{i=0}^{d-1} |M_i| + |M_{d-1}|\,(|M_{d-1}| - 1) \leq$$
$$\leq \sum_{i=0}^{d-1} |M_i| + \sum_{i=0}^{d-1} |M_i| \left(\sum_{i=0}^{d-1} |M_i| - 1 \right) = \left(\sum_{i=0}^{d-1} |M_i| \right)^2 \leq \left(|M|^{2^{d-1}} \right)^2 = |M|^{2^d}.$$

Clearly $|M|^{2^d}$ bounds every term appearing in the reduction, so that a Turing machine can simulate each step in this reduction in a number of steps proportional to $\left(|M|^{2^d}\right)^2$, so that the final bound is proportional to $|M|^{2^d} \cdot |M|^{2^{d+1}} \leq |M|^{2^{d+2}}$. □

4.2.4 Representation theorem for LAL

We will prove the representation theorem for **DLAL**2 using the approach developed by Murawski and Ong in [MO04] for **LAL**2. Representation for **DLAL**2 then implies the same result for **LAL**2, however this is an occasion to see the easier functioning of **DLAL**2 at work.

Definition 4.2.53 (representation of functions). We say $f : \mathbb{N}^k \to \mathbb{N}$ is represented by a term M typable in **DLAL**2 with $(\mathtt{BInt})^k \multimap \S^q\,\mathtt{BInt}$ if for any $\vec{n} \in \mathbb{N}^k$ we have $M\,\underline{\vec{n}} \overset{\beta}{\twoheadrightarrow} \underline{f(\vec{n})}$. We say the representation has *obliqueness* q.

The aim of this section is to prove the following theorem.

Theorem 4.2.54. *Any polytime function is representable by a pure term typable in* **DLAL**2*. On the converse every closed term typable in* **DLAL**2 *with* $\mathtt{BInt}^k \multimap \S^q\,\mathtt{BInt}$ *represents a polytime function.*

Remark 4.2.55. There is a little imprecision about the second claim. As for `Int` it is easy to prove that every normal term of type `BInt` (and we can work in system **F**) is in one of the three forms $\lambda f_0 \lambda f_1 . f_0$, $\lambda f_0 \lambda f_1 . f_1$ and $\underline{w}$. The first is equivalent to $\underline{0}$ (seen as a sequence with only 0), the second to $\underline{1}$. In any case another ambiguity is represented by the eventual presence of trailing zeros: so in order for the second claim to have sense we must identify sequences that differ in this manner. However we will take the strict meaning when proving the other direction: we will be able to have terms that represent functions that output binary integers in the correct form.

By the property of polynomiality of the reduction of a term typable in **DLAL**2 (we need just the weak one), we know that $M\,\underline{\vec{n}}$ reduces in polynomial time to a normal term typable with $\S^q\,\mathtt{BInt}$. By the above remark we know it must be (eventually equivalently) the representation of a binary expansion.

For the other direction we will adapt to **DLAL** an approach used by Murawski and Ong in [MO04] to partly represent safe recursion in **LAL**. We may say is is a semi-axiomatic approach: we start from a proper subclass BC^- of BC (see definition 4.2.25), and give a compositional translation in **DLAL** as we did with elementary functions and **EAL**. So we will bring **DLAL** closer to the polytime class, however the last steps will be done by completing BC^- using low-level functions that allow to complete the encoding of polytime Turing machines and that are representable in **DLAL**. So in fact We will be halfway axiomatic and halfway oriented to Turing machine encoding.

Definition 4.2.56 (BC^-). BC^- is the least class of functions containing the base functions of BC but closed to these restricted forms of safe composition and recursion:

- *restricted safe composition*, which given functions $f_i(\vec{x};)$ and $h_i(\vec{x};\vec{y}_i)$, and a function $g(\vec{s};\vec{t})$ gives a function
$$\text{RS-COMP}(\vec{f},\vec{g})(\vec{x};\vec{y}_1,\dots,\vec{y}_\ell) := g(\vec{f}(\vec{x};);h_1(\vec{x};\vec{y}_1),\dots,h_\ell(\vec{x};\vec{y}_\ell));$$
- *restricted safe recursion*, which given functions $g(\vec{y};\vec{z})$ and two functions $h_i(x,\vec{y};w)$ gives a function $f = \text{RS-REC}(g,h_0,h_1)$ defined by
$$f(0,\vec{y};\vec{z}) := g(\vec{y};\vec{z}),$$
$$f(xi,\vec{y};\vec{z}) := h_i(x,\vec{y};f(x,\vec{y};\vec{z})).$$
To differentiate these functions from the most general class BC we substitute the semicolon with a colon, writing $f(\vec{x}:\vec{y})$.

Remark 4.2.57. What we have done is that we have *linearized* the safe variables, in the sense that they cannot be used more than once, i.e. they are not contractible. in fact in composition we do not allow safe variables to be the argument of more than one function, and in recursion the step functions have only the place for recursive call as safe variable, so that the safe variables $\vec{y}$ are used only in the base function.

Now we will see how to translate safeness.

Definition 4.2.58 (safeness in DLAL2). We will say that $f(\vec{x}:\vec{y})$ is represented by the term M with free variables $\vec{x},\vec{y}$ (we use the same names on purpose) if the following sequent is derivable
$$;\vec{x}:\mathtt{BInt},\vec{y}:\S^q\,\mathtt{BInt} \vdash M:\S^q\,\mathtt{BInt}$$
and
$$M[\overrightarrow{\underline{n}/x},\overrightarrow{\underline{m}/y}] \overset{\beta}{\twoheadrightarrow} \underline{f(\vec{n}:\vec{m})}.$$

We will denote the property by writing $M = M[\vec{x} : \vec{y}|q]$, and we will write $M[\vec{P} : \vec{Q}]$ for the substitution $M[\overrightarrow{P/x}, \overrightarrow{Q/y}]$. q is called the *obliqueness* of M; as with elementary representations (though here we are avoiding the final abstraction in the programming of a function) we will say that the term is *flat* if $q = 0$. Note that if M is flat all variables can be regarded as safe.

Before enquiring into the meaning of the definition let's introduce some tools, and begin to bring down to representation the base functions. As usual $\mathbf{0}(:)$ is easy, and so are projections, for which we have representations $\underline{\pi_i^{m,n}}[\vec{x} : \vec{y}|0]$ using the usual projections. Note that because the term is flat in fact we can regard any variable as safe or normal at will.

Lemma 4.2.59 (iteration). *If* $; z_i : \sigma_i \vdash M_i : \tau \multimap \tau$ *with* $i = 0, 1$ *(eventually with empty environment) and* $; A_1, A_2 \vdash N : \tau$ *are derivable we have a typing that leads to*

$$A_1, z_0 : \sigma_0, z_1 : \sigma_1; \S A_2, n : \mathtt{BInt} \vdash n\, M_0\, M_1\, N : \S\tau.$$

This term, if we plug $\underline{n}$ *into* n*, represents applying repeatedly* M_0 *and* M_1 *to* N *in the order determined by the binary digits of* n*.*

Proof. First we derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{; n : \mathtt{BInt} \vdash n : \mathtt{BInt}}\,(\text{var})}{; n : \mathtt{BInt} \vdash n : \mathtt{BInt}_\tau}\,(\text{gen}) \qquad ; z_0 : \sigma_0 \vdash M_0 : \tau \multimap \tau}{z_0 : \sigma_0; n : \mathtt{BInt} \vdash n\, M_0 : (\tau \multimap \tau) \to \S(\tau \multimap \tau)}\,(\text{i-app}) \qquad ; z_1 : \sigma_1 \vdash M_1 : \tau \multimap \tau}{z_0 : \sigma_0, z_1 : \sigma_1; n : \mathtt{BInt} \vdash n\, M_0\, M_1 : \S(\tau \multimap \tau)}\,(\text{i-app})$$

Then we plug it into a promotion

$$\dfrac{\overrightarrow{z : \sigma}; n : \mathtt{BInt} \vdash n\, M_0\, M_1 : \S(\tau \multimap \tau) \qquad \dfrac{\dfrac{}{; y : \tau \multimap \tau \vdash y : \tau \multimap \tau}\,(\text{var}) \qquad ; A_1, A_2 \vdash N : \tau}{; A_1, A_2, y : \tau \multimap \tau \vdash y\, N : \tau}\,(\text{l-app})}{A_1, z_0 : \sigma_0, z_1 : \sigma_1; \S A_2, n : \mathtt{BInt} \vdash n\, M_0\, M_1\, N : \S\tau}\,(\text{prom})$$

□

Lemma 4.2.60 (strip). *There is a context* $\mathtt{strip}[\ ]$ *such that* $n : \mathtt{BInt} \vdash_{\mathbf{DLAL}^2} \mathtt{strip}[n] : \mathtt{BInt}$ *and*

$$\mathtt{strip}[\lambda f_0 \lambda f_1 \lambda x. f_{b_0}(\dots f_{b_{k-1}}(f_1(f_0(\dots(f_0\, x)\dots)))\dots)] \overset{\beta}{\twoheadrightarrow} \lambda f_0 \lambda f_1 \lambda x. f_{b_0}(\dots f_{b_{k-1}}(f_1\, x)\dots),$$

and

$$\mathtt{strip}[\lambda f_0 \lambda f_1 \lambda x. (f_0^k\, x)] \overset{\beta}{\twoheadrightarrow} \lambda f_0 \lambda f_1 \lambda x. x.$$

Practically $\mathtt{strip}[\ ]$ *erases trailing zeros.*

Proof. Consider the type $\tau = \mathtt{Bool} \otimes (\alpha \multimap \alpha)$, where $\mathtt{Bool} = \forall\beta.\beta \multimap \beta \multimap \beta$ and as usual

$$\sigma_1 \otimes \sigma_2 := \forall\gamma.(\sigma_1 \multimap \sigma_2 \multimap \gamma) \multimap \gamma.$$

We will use the first component to mark as whether we are still erasing the trailing zeros or not, distinguishing the two cases with $\underline{\mathtt{true}}$ and $\underline{\mathtt{false}}$. Until we don't encounter f_1 we do not save the symbol we are passing, while from then on we have to. Consider the terms (recall $x \circ y := \lambda z.x\,(y\,z)$):

$$M_1 := \lambda p.\langle \underline{\mathtt{false}}, f_1 \circ (p\,\underline{\mathtt{false}})\rangle,$$
$$M_0 := \lambda p.p\,(\lambda x\lambda y.\langle\langle \underline{\mathtt{true}}, I\rangle, \langle \underline{\mathtt{false}}, f_0 \circ y\rangle\rangle\, x).$$

They both have the derivation $; f_i : \alpha \multimap \alpha \vdash M_i : \tau \multimap \tau$. The first one is straightforward In the second one we have to assume $x : \mathtt{Bool}$ and then instantiate it with type $\tau \multimap \tau \multimap \tau$. The pair of pairs is built with type $\tau \otimes \tau = \forall\alpha.(\tau \multimap \tau \multimap \alpha) \multimap \tau$ which can be instantiated to $(\tau \multimap \tau \multimap \tau) \multimap \tau$ and thus the pair can be applied to x yielding type τ. What do those terms do?

$$M_0\,\langle \underline{b}, y\rangle \overset{\beta}{\twoheadrightarrow} \begin{cases} \langle \underline{\mathtt{true}}, I\rangle & \text{if } b = \mathtt{true}, \\ \langle \underline{\mathtt{false}}, f_0 \circ y\rangle & \text{otherwise.} \end{cases}$$
$$M_1\,\langle \underline{b}, y\rangle \overset{\beta}{\twoheadrightarrow} \langle \underline{\mathtt{true}}, f_1 \circ y\rangle.$$

So applying the iteration scheme with base $N := \langle \underline{\mathtt{true}}, I\rangle$ typable with τ to a representation with at least 1 we get a term $\langle \mathtt{false}, F\rangle$ where F is the chained composition of f_0 and f_1 in the same order they appear in the iterator, apart from the initial f_0 which get ignored. If there are no f_is, or there are only f_0s, F will be the identity. The typing proceeds by

$$f_0 : \alpha \multimap \alpha, f_1 : \alpha \multimap \alpha; n : \mathtt{BInt} \vdash n\,M_0\,M_1\,N : \S\tau.$$

Then we can plug it into z in the promotion of

$$z : \tau \vdash \lambda x.z\,\underline{\mathtt{false}}\,x : \alpha \multimap \alpha$$

which gives

$$f_0 : \alpha \multimap \alpha, f_1 : \alpha \multimap \alpha; n : \mathtt{BInt} \vdash \lambda x.n\,M_0\,M_1\,N\,\underline{\mathtt{true}}\,x : \S(\alpha \multimap \alpha)$$

which by two abstractions and a generalization becomes what we wanted

$$; n : \mathtt{BInt} \vdash \mathtt{strip}[n] : \mathtt{BInt},$$

with $\mathtt{strip}[\ ] := \lambda f_0\lambda f_1\lambda x.\square\,M_0\,M_1\,N\,\underline{\mathtt{true}}\,x$. In fact

$$\mathtt{strip}[\underline{b_0 \dots b_{k-1}10\dots0}] \overset{\beta}{\twoheadrightarrow} \lambda f_0\lambda f_1\lambda x.F\,x$$

where F is the term described above, so that $F\,x \overset{\beta}{\twoheadrightarrow} f_{b_0}(\dots(f_{b_{k-1}}(f_1\,x))\dots)$. □

Now we are ready to handle the more delicate case of the successor $\mathtt{succ}_0$.

Lemma 4.2.61 (successors). *The successors* $\mathtt{succ}_i(:n)$ *are representable in* **DLAL**2.

Proof. $\mathtt{succ}_1$ is easy. We derive with the help of three axioms and two intuitionistic applications

$$f_0 : \alpha \multimap \alpha, f_1 : \alpha \multimap \alpha; n : \mathtt{BInt} \vdash n\, f_0\, f_1.$$

Then we prepare a term where to plug the above derivation by promotion:

$$; f_1' : \alpha \multimap \alpha, z : \alpha \multimap \alpha \vdash \lambda x.f_1\,(z\,x) : \alpha \multimap \alpha.$$

During promotion we shift f_1' to the intuitionistic side, and we get

$$\frac{f_0 : \alpha \multimap \alpha, f_1' : \alpha \multimap \alpha, f_1 : \alpha \multimap \alpha; n : \mathtt{BInt} \vdash \lambda x.f_1'\,(n\, f_0\, f_1\, x) : \S(\alpha \multimap \alpha)}{f_0 : \alpha \multimap \alpha, f_1 : \alpha \multimap \alpha; n : \mathtt{BInt} \vdash \lambda x.f_1\,(n\, f_0\, f_1\, x) : \S(\alpha \multimap \alpha)}\ (\mathrm{con})$$

after which two abstractions and a generalization give

$$; n : \mathtt{BInt} \vdash \underline{\mathtt{succ}_1} : \mathtt{BInt}\,.$$

with $\underline{\mathtt{succ}_1} := \lambda f_0 \lambda f_1 \lambda x.f_1\,(n\, f_0\, f_1\, x)$. So n is safe with respect to our definition, and it is trivial to see $\underline{\mathtt{succ}_1}$ does what it's designed for.

For $\underline{\mathtt{succ}_0}$ we reason in the same way, but then we compose with $\mathtt{strip}[\ ]$:

$$\underline{\mathtt{succ}_0} := \mathtt{strip}[\lambda f_0 \lambda f_1 \lambda x.f_0\,(n\, f_0\, f_1\, x)],$$

so that in effect $\underline{\mathtt{succ}_0}[\underline{0}/n] \overset{\beta}{\twoheadrightarrow} \underline{0}$ as should be. Thus we have the two successors $\underline{\mathtt{succ}_i}[:n|0]$. □

Remark 4.2.62. The successors given above have the desired effect on integers to yield $n \mapsto 2n+i$. However we will need another kind of successors, such that they append at the most significant digits instead of the least significant ones. We call them the same way because of syntactical similarity, and the parallelism with the case for Church integers where the two versions of the successor did the same thing because there was no order to be careful of.

These two successors are

$$\underline{\mathtt{succ}_i'} := \lambda f_0 \lambda f_1 \lambda x.n\, f_0\, f_1\, (f_i\, x),$$

It is easy to see they still get the derivation $; n : \mathtt{BInt} \vdash \underline{\mathtt{succ}_i'} : \mathtt{BInt}$. We do not use $\mathtt{strip}$ because we need them mostly for technical reasons, not to compute integer values. We need them to always put the digit they have to put.

With them we can program two kinds of coercions.

Lemma 4.2.63 (coercions). *There is a context* $\mathtt{coerc}[\ ]$ *such that*

$$;n : \mathtt{BInt} \vdash \mathtt{coerc}[n] : \S\,\mathtt{BInt}$$

and such that $\mathtt{coerc}[\underline{n}] \overset{\beta}{\twoheadrightarrow} \underline{n}$. *We can chain this context to obtain in general*

$$;n : \S^p\,\mathtt{BInt} \vdash \mathtt{coerc}^q[n] : \S^{p+q}\,\mathtt{BInt}\,.$$

There is a context with two holes $\mathtt{coerc}'_x[\Box_1, \Box_2]$ *such that if* $n : \mathtt{BInt}; A \vdash M : \tau$ *then*

$$;m : \mathtt{BInt}, \S A \vdash \mathtt{coerc}'_n[m, M] : \S\tau$$

and $\mathtt{coerc}'_n[\underline{m}, M] \overset{\beta}{\twoheadrightarrow} M[\underline{m}/n]$.

Proof. We do the iteration

$$;n : \mathtt{BInt} \vdash n\,(\lambda x.\underline{\mathtt{succ}_0}[:x])\,(\lambda x\underline{\mathtt{succ}_1}[:x])\,\underline{0}.$$

The claim about chaining comes from use of multiple promotions.

For the second claim we define for $i = 0, 1$:

$$H_i := \lambda g \lambda p.(g\,(\underline{\mathtt{succ}'_i}[:p])),$$

with the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{;g : \mathtt{BInt} \to \tau \vdash g : \mathtt{BInt} \to \tau}\,(\text{var}) \quad \dfrac{\vdash \underline{\mathtt{succ}_i} : \mathtt{BInt} \multimap \mathtt{BInt} \quad \dfrac{}{;p : \mathtt{BInt} \vdash p : \mathtt{BInt}}\,(\text{var})}{;p : \mathtt{BInt} \vdash \underline{\mathtt{succ}_i}\,p : \mathtt{BInt}}\,(\text{l-app})}{p : \mathtt{BInt}; g : \mathtt{BInt} \to \tau \vdash g\,(\underline{\mathtt{succ}_i}\,p) : \tau}\,(\text{i-app})}{;g : \mathtt{BInt} \to \tau \vdash \lambda p.g\,(\underline{\mathtt{succ}_i}\,p) : \mathtt{BInt} \to \tau}\,(\text{i-abs})}{;\vdash H_i : (\mathtt{BInt} \to \tau) \multimap \mathtt{BInt} \to \tau}\,(\text{l-abs})$$

Then we instantiate $m : \mathtt{BInt}$ with $\mathtt{BInt}_{\mathtt{BInt}\to\tau}$, and with two intuitionistic applications we get

$$;m : \mathtt{BInt} \vdash m\,H_0\,H_1 : \S((\mathtt{BInt} \to \tau) \multimap \mathtt{BInt} \to \tau).$$

Apart we prepare

$$\dfrac{\vdash \underline{0} : \mathtt{BInt} \qquad \dfrac{\dfrac{}{;\cdots \vdash z : (\mathtt{BInt} \to \tau) \multimap \mathtt{BInt} \to \tau}\,(\text{var}) \quad \dfrac{n : \mathtt{BInt}; A \vdash M : \tau}{;A \vdash \lambda n.M : \mathtt{BInt} \to \tau}\,(\text{i-abs})}{;A, z : (\mathtt{BInt} \to \tau) \multimap \mathtt{BInt} \to \tau \vdash z\,(\lambda n.M) : \mathtt{BInt} \to \tau}\,(\text{l-app})}{;A, z : (\mathtt{BInt} \to \tau) \multimap \mathtt{BInt} \to \tau \vdash z\,(\lambda n.M)\,\underline{0} : \tau}\,(\text{i-app})$$

We then promote and plug into z the derivation of $m\,H_0\,H_1$, obtaining at last

$$;m : \mathtt{BInt}, \S A \vdash \mathtt{coerc}'_n[m, M] : \S\tau$$

where

$$\mathtt{coerc}'_x[\Box_1, \Box_2] := (\Box_1\, H_0\, H_1\, \lambda x.\Box_2)\, \underline{0}.$$

Now suppose $m = (b_k \dots b_0)_2$:

$$\begin{aligned}
\mathtt{coerc}'_n[\underline{m}, M] &\overset{\beta}{\twoheadrightarrow} (H_{b_0}(\dots(H_{b_k}\, \lambda n.M)\dots))\, \underline{0} \overset{\beta}{\to} \\
&\overset{\beta}{\to} (H_{b_0}(\dots(H_{b_{k-1}}\, \lambda p.((\lambda n.M)\, (\underline{\mathtt{succ}'_{b_k}}[: p])))\dots))\, \underline{0} \overset{\beta}{\to} \\
&\overset{\beta}{\to} (H_{b_0}(\dots(H_{b_{k-1}}\, \lambda p.(M[\underline{\mathtt{succ}'_{b_k}}[: p]/n]))\dots))\, \underline{0} \overset{\beta}{\twoheadrightarrow} \\
&\overset{\beta}{\twoheadrightarrow} (\lambda p.M[\underline{\mathtt{succ}'_{b_k}}[: \dots \underline{\mathtt{succ}'_{b_0}}[: p]\dots]/n])\, \underline{0} \overset{\beta}{\to} \\
&\overset{\beta}{\to} M[\underline{\mathtt{succ}'_{b_k}}[: \dots \underline{\mathtt{succ}'_{b_0}}[: \underline{0}]\dots]/n] \overset{\beta}{\twoheadrightarrow} M[\underline{m}/n].
\end{aligned}$$

Note that the second coercion may be applied only if there is one variable to move from intuitionistic to linear, and then again it applies paragraphs to all the rest of the environment and to the derived type. Next we will see how to chain this kind of coercion to shift more than one variable. □

Remark 4.2.64. There is a way to chain also the second coercion, letting us move multiple `BInt` variables from intuitionistic to linear side.

Suppose we can derive $\vec{n}^k : \mathtt{BInt}; A \vdash M : \tau$. Let us denote by $\tau_h := \mathtt{BInt}^h \multimap \tau$, with $\tau_0 = \tau$. Now let us use $\mathtt{coerc}'$ for k times applying it to the following sequents for $1 \leq h \leq k$:

$$t_h : \mathtt{BInt}; z_h : \tau_{k-h+1} \vdash z_h\, t_h : \tau_{k-h}$$

getting

$$; m_h : \mathtt{BInt}, z_h : \S\tau_{k-h+1} \vdash \mathtt{coerc}_{t_h}[m_h,\, z_h\, t_h] : \S\tau_{k-h}.$$

Let us denote by N_h the term thus obtained. We can substitute each N_h into z_{h+1} in the derivation of the following N_{h+1}: each derivation inherits all the preceding y_hs and only z_1. So in N_h we obtain:

$$; \vec{m}^h : \mathtt{BInt}, z_1 : \S\tau_k \vdash P_h : \S\tau_{k-h},$$

where $P_1 = N_1$ and $P_{h+1} = N_{h+1}[P_h/z_{h+1}]$. Now take the sequent we started with, and apply k intuitionistic abstractions:

$$; A \vdash \overrightarrow{\lambda n}.M : \tau_k.$$

We promote it and substitute it into z_1 in P_h and we get:

$$; \vec{m}^h : \mathtt{BInt}, \S A, \vdash P_h[M/z_1] : \S\tau_{k-h}.$$

Note now that we cannot recuperate the rest of the intuitionistic variables without raising the level. In fact we should plug the derivation above in a promotion of a derivation for $w\, n_{h+1} \dots n_k$,

but then the n_is should be on the intuitionistic side, preventing the promotion from happening. So we have to take $P_k[M/z_1]$ above shifting all the variables to the linear side, and in case we want back in the intuitionistic side some variables we have to promote adding a paragraph to the other variables.

Let us in future write $\mathtt{coerc}'_{\vec{n}}[\vec{m}, M]$ for the construction of P_k as done above.

Remark 4.2.65. As we did for elementary functions, now we may strip of all paragraphs the environment of a represented function, i.e. we can turn a safe variable into a normal one. Given

$$;\vec{x} : \mathtt{BInt}, y : \S^q\,\mathtt{BInt}, z : \S^q\,\mathtt{BInt} \vdash M[\vec{x} : y, \vec{z}|q] : \mathtt{BInt},$$

we can substitute into it

$$;y' : \mathtt{BInt} \vdash \mathtt{coerc}^q[y'] : \S^q : \mathtt{BInt}$$

and get

$$;\vec{x} : \mathtt{BInt}, y' : \mathtt{BInt}, z : \S^q\,\mathtt{BInt} \vdash M'[\vec{x}, y' : \vec{z}|q] : \mathtt{BInt}$$

where $M' := M[\mathtt{coerc}^q[y']/y]$ and thus represents the same function.

We can also *lift* a representation: given $M[\vec{x} : \vec{y}|q]$ we can increase the obliqueness at will, just by taking $M' := \mathtt{coerc}^p[M]$, so that $M' = M'[\vec{x} : \vec{y}|p+q]$.

Proposition 4.2.66. *Normal variables are contractible, i.e. given $M[\vec{x}, \vec{y} : \vec{z}|k]$ that represents $f(\vec{x}, \vec{y} : \vec{z})$, we can give a term $M'[\vec{x}, w : \vec{z}|q+2]$ such that it represents $f(\vec{x}, w, \ldots, w : \vec{z})$.*

Proof. We do a promotion turning normal variables needed to be contracted into intuitionistic and then contract them:

$$\dfrac{\dfrac{;\vec{x} : \mathtt{BInt}, \vec{y} : \mathtt{BInt}, \vec{z} : \S^q\,\mathtt{BInt} \vdash M : \S^q\,\mathtt{BInt}}{\vec{y} : \mathtt{BInt}; \vec{x} : \S\,\mathtt{BInt}, \vec{z} : \S^{q+1}\,\mathtt{BInt} \vdash M : \S^{q+1}\,\mathtt{BInt}}\,(\text{prom})}{y : \mathtt{BInt}; \vec{x} : \S\,\mathtt{BInt}, \vec{z} : \S^{q+1}\,\mathtt{BInt} \vdash M[y/\vec{y}] : \S^{q+1}\,\mathtt{BInt}}\,(\text{con})$$

Now we use the second coercion and get

$$;w : \mathtt{BInt}, \vec{x} : \S^2\,\mathtt{BInt}, \vec{z} : \S^{q+2}\,\mathtt{BInt} \vdash \mathtt{coerc}'_y[w, M[y\vec{y}]] : \S^{q+2}\,\mathtt{BInt}\,.$$

Finally if we precompose with the other coercion we return $\vec{x}$ to normal state:

$$;w : \mathtt{BInt}, \vec{x} : \mathtt{BInt}, \vec{z} : \S^{q+2}\,\mathtt{BInt} \vdash \mathtt{coerc}'_y[w, M[y/vecy, \overrightarrow{\mathtt{coerc}^2[x]/x}]] : \S^{q+2}\,\mathtt{BInt}\,.$$

We changed the term using only the coercions, so we have not changed its algorithmic content. □

We will see that safe variables are not contractible, making clear the parallelism with safe variables in BC^-. Now we will go on completing the codification of the base functions.

Lemma 4.2.67 (predecessor). *The predecessor* $\mathtt{pred}(:x)$ *is representable in* **DLAL**2.

Proof. The idea is again that of using pairs. Let τ be $(\alpha \multimap \alpha) \otimes (\alpha \multimap \alpha)$. We will use the first component to record the symbol we are passing at the moment, and the other one to chain the previous symbols. It is not distant from the representation of the predecessor on unary representations given for **EAL**2. So let M_i be the following terms

$$M_i := \lambda p.p\,(\lambda x \lambda y.\langle f_i, x \circ y\rangle),$$

which is easily typable with $\tau \multimap \tau$. Iteration with $\underline{n} = \underline{b_k \dots b_0}$ started on base $\langle I, I\rangle$ yields in the end $\langle f_{b_0}, f_{b_1} \circ \dots \circ f_{b_k}\rangle$ of type $\S\tau$. As usual we plug

$$f_0 : \alpha \multimap \alpha, f_1 \alpha \multimap \alpha; n : \mathtt{BInt} \vdash n\, M_0\, M_1 : \S\tau$$

into z in the following derivable sequent to be promoted

$$; z : \tau \vdash \lambda x.z\ \mathtt{false}\ x : \alpha \multimap \alpha,$$

giving after abstractions and generalizations

$$; n : \mathtt{BInt} \vdash \underline{\mathtt{pred}} : \mathtt{BInt}$$

where $\underline{\mathtt{pred}}[:n|0] := \lambda f_0 \lambda f_1 \lambda x.n\, M_0\, M_1\, \underline{\mathtt{false}}\, x$. □

Lemma 4.2.68 (conditional). *The conditional clause* $\mathtt{if}(;a,b,c)$ *is representable in* **DLAL**2.

Proof. We simply have to see only the least significant digit. However the delicacy lies in wanting a representation with all safe variables. We put $M_0 = \lambda d.\underline{\mathtt{true}}$ and $M_1 = \lambda d.\underline{\mathtt{false}}$, but type them with $\mathtt{Bool}_{\{}\, \alpha \multimap \alpha\} \multimap \mathtt{Bool}_{\alpha \multimap \alpha}$. Iterating on base value $\underline{\mathtt{true}}$ gives the term

$$; a : \mathtt{BInt} \vdash a\, M_0\, M_1\, \underline{\mathtt{true}} : \S\, \mathtt{Bool}_{\alpha \multimap \alpha}\,.$$

Now we carefully prepare a term to plug it in. We use b to derive, using an instantiation of $\mathtt{BInt}$ and two intuitionistic applications:

$$f_0^b : \alpha \multimap \alpha, f_1^b : \alpha \multimap \alpha; b : \mathtt{BInt} \vdash b\, f_0^b\, f_1^b : \S(\alpha \multimap \alpha),$$

and we do the same for c. Then we derive

$$; w_1 : \alpha \multimap \alpha, w_2 : \alpha \multimap \alpha, z : \mathtt{Bool}_{\alpha \multimap \alpha} \vdash z\, w_1\, w_2 : \alpha \multimap \alpha.$$

Now we have all we need. We promote the last derivation and plug in it the other two:

$$\frac{f_0^b, f_1^b, f_0^c, f_1^c : \alpha \multimap \alpha; a : \mathtt{BInt}, b : \mathtt{BInt}, c : \mathtt{BInt} \vdash a\, M_0\, M_1\, \underline{\mathtt{true}}\, (b\, f_0^b\, f_1^b)\, (c\, f_0^c\, f_1^c) : \S(\alpha \multimap \alpha)}{f_0, f_1 : \alpha \multimap \alpha; a : \mathtt{BInt}, b : \mathtt{BInt}, c : \mathtt{BInt} \vdash a\, M_0\, M_1\, \underline{\mathtt{true}}\, (b\, f_0\, f_1)\, (c\, f_0\, f_1) : \S(\alpha \multimap \alpha)}\ (\mathrm{con})$$

and then two abstractions and a generalization give the proper type. So in the end

$$\mathtt{if}[:a,b,c|0] := \lambda f_0 \lambda f_1 . a\, M_0\, M_1\, \underline{\mathtt{true}}\, (b\, f_0\, f_1)\, (c\, f_0\, f_1),$$

and in fact

$$\mathtt{if}[:\underline{a},\underline{b},\underline{c}] \overset{\beta}{\twoheadrightarrow} \lambda f_0 \lambda f_1 . \underline{d}\, (\underline{b}\, f_0\, f_1)\, (\underline{c}\, f_0\, f_1)$$

where $d = \mathtt{true}$ if $a_0 = 0$ and $\mathtt{false}$ otherwise. □

Before going on we generalize to **DLAL**2 the usual product of types.

Definition 4.2.69 (dual product). The *dual product* is defined by

$$\bigotimes_{m,n}(\vec{\sigma}^m, \vec{\tau}^n) := \forall\alpha.(\sigma_1 \to \ldots \to \sigma_m \to \tau_1 \multimap \ldots \multimap \tau_n \multimap \alpha) \multimap \alpha.$$

$\bigotimes_{m,n}(\sigma,\tau)$ is short for $\bigotimes_{m,n}(\sigma,\ldots,\sigma,\tau,\ldots,\tau)$. It should be seen in **LAL** as the product of the types in which the first m are being banged. A *dual* m,n*-ple* is defined on λ-terms as

$$\langle \vec{x}, \vec{y} \rangle := \lambda z . z\, \vec{x}\, \vec{y}.$$

Then by assuming

$$z : \sigma_1 \to \ldots \to \sigma_m \to \tau_1 \multimap \ldots \multimap \tau_n \multimap \alpha$$

and then doing m intuitionistic applications and n linear ones we get the typing:

$$\overrightarrow{x:\sigma}; \overrightarrow{y:\tau} \vdash \langle \vec{x}, \vec{y} \rangle : \bigotimes_{m,n}(\vec{\sigma}^m, \vec{\tau}^n).$$

Using substitution we therefore can have as derived rule:

$$\frac{\overrightarrow{;z:\rho \vdash M:\sigma} \quad \overrightarrow{A;B \vdash N:\tau}}{\overrightarrow{z:\rho}, \vec{A}; \vec{B} \vdash \langle \vec{M}, \vec{N} \rangle : \bigotimes_{m,n}(\vec{\sigma}, \vec{\tau})}$$

As for extraction, we can easily extract the linear part of the m,n-uple, applying it to a projection $\underline{\pi}_i^{m+n}$, which can be typed right by weakening. If we want to extract something in the linear side however we have to put a promotion in the derivation of the projection, so that to the final type a paragraph will be appended. This reflects the fact that in **LAL**2 the product $\bigotimes_{1,1}(\sigma,\tau)$ is translated in $!\sigma \otimes \tau$. Extracting the first in **LAL**2 should give $!\sigma$. However we have made this type illegal in **DLAL**2, so we must revert to the weaker $\S\sigma$.

Lemma 4.2.70 (restricted safe composition and recursion). *Restricted safe composition and restricted safe recursion are representable in* **DLAL**2.

Proof. Suppose we have representations $F_i[\vec{x}^n : |p_i]$ and $H_j[\vec{x}^n : \vec{y}_j|q_j]$ and $G[\vec{s} : \vec{t}|r]$. First we lift all the F_is to the same obliqueness $p = \max(p_i)$, and then plug them by substitution in the right places of G promoted p times, renaming in each the free variables (or else we cannot put them together). So if F'_i are F_i lifted and with $\vec{x}$ rename to $\vec{z}_i$, we have:

$$;\vec{z}_1, \dots, \vec{z}_k : \mathtt{BInt}, \vec{t} : \S^{r+p}\,\mathtt{BInt} \vdash G[\overrightarrow{F'/s}] : \S^{p+r}\,\mathtt{BInt}\,.$$

Now we lift the term given above and all the H_js to a common obliqueness $u = \max(q_j, p + r)$: if $G'[\vec{z}_1, \dots, \vec{z}_k : \vec{t}|u]$ is the term so obtained, and H'_j are the terms obtained by lifting and renaming $\vec{x}$ to $\vec{w}_j$, we finally have

$$;\vec{z}_1, \dots, \vec{z}_k, \vec{w}_1, \dots, \vec{w}_\ell : \mathtt{BInt}, \vec{y}_1, \dots, \vec{y}_\ell : \S^u\,\mathtt{BInt} \vdash G'[\overrightarrow{H'/t}] : \S^u\,\mathtt{BInt}\,.$$

Now we apply lemma 4.2.66 n times to identify the normal variables back to $\vec{x}$, obtaining finally the term $N[\vec{x} : \vec{y}_1, \dots, \vec{y}_\ell|u + 2n]$ that is the safe composition of F_i and H_j with G.

For restricted safe recursion, we first lift all the representation we have to the same obliqueness. So suppose we have

$$;\vec{y}^m : \mathtt{BInt}, \vec{z}^n : \S^q\,\mathtt{BInt} \vdash G : \S^q\,\mathtt{BInt}, \quad ;x : \mathtt{BInt}, \vec{y}^m : \mathtt{BInt}, w : \S^q \vdash H_i : \S^q\,\mathtt{BInt}\,.$$

Now we promote the above to shift $\vec{y}$ and x to the intuitionistic side.

$$\vec{y} : \mathtt{BInt}; \vec{z} : \S^{q+1}\,\mathtt{BInt} \vdash G : \S^{q+1}\,\mathtt{BInt}, \quad x : \mathtt{BInt}, \vec{y} : \mathtt{BInt}, w : \S^{q+1} \vdash H_i : \S^{q+1}\,\mathtt{BInt}\,.$$

Now consider the types $\tau := \bigotimes_{1,1}(\mathtt{BInt}, \S^{q+1}\,\mathtt{BInt})$ and $\sigma := \mathtt{BInt}^m \to (\S^{q+1}\,\mathtt{BInt})^n \multimap \tau$. We will do an iteration on type $\sigma \multimap \sigma$, where the content of σ should denote the function $\overrightarrow{\lambda y}\overrightarrow{\lambda z}.\langle z,\, H_i[z, \vec{y} : F[z, \vec{y} : \vec{z}]]\rangle$, where F is the term we are building. Clearly we will extract all we need from the previous iteration.

For the base value N we derive

$$\dfrac{\dfrac{;\vdash \underline{0} : \mathtt{BInt} \quad \vec{y} : \mathtt{BInt}; \vec{z} : \S^{q+1}\,\mathtt{BInt} \vdash G : \S^{q+1}\,\mathtt{BInt}}{\vec{y} : \mathtt{BInt}; \vec{z} : \S^{q+1}\,\mathtt{BInt} \vdash \langle \underline{0}, G\rangle : \tau}}{;\vdash \overrightarrow{\lambda y}\overrightarrow{\lambda z}.\langle \underline{0}, G\rangle : \sigma}\ (\text{abs})$$

The step functions, are derived:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vec{y}; \vec{z}, f : \sigma \vdash f\,\vec{y}\,\vec{z} : \tau}}{\vec{y}; \vec{z}, f : \sigma \vdash f\,\vec{y}\,\vec{z} : (\mathtt{BInt} \to \S^{q+1}\,\mathtt{BInt} \multimap \tau) \multimap \tau}\ (\text{ins}) \quad \begin{matrix}\mathcal{D}\\ \vdots\end{matrix}}{\vec{y}, \vec{y'} : \mathtt{BInt}; \vec{z} : \S^{q+1}\,\mathtt{BInt}, f : \sigma \vdash f\,\vec{y}\,\vec{z}\,(\lambda x \lambda w.\langle \underline{\mathtt{succ}_i}\, x,\, H_i[\underline{\mathtt{succ}_i}\, x, \vec{y'} : w]\rangle) : \tau}\ (\text{l-app})}{\vec{y} : \mathtt{BInt}; \vec{z} : \S^{q+1}\,\mathtt{BInt}, f : \sigma \vdash f\,\vec{y}\,\vec{z}\,(\lambda x \lambda w.\langle \underline{\mathtt{succ}_i}\, x,\, H_i[\underline{\mathtt{succ}_i}\, x, \vec{y} : w]\rangle) : \tau}\ (\text{con})}{;\vdash \lambda f \overrightarrow{\lambda y}\overrightarrow{\lambda z}.f\,\vec{y}\,\vec{z}\,(\lambda x \lambda w.\langle \underline{\mathtt{succ}_i}\, x,\, H_i[\underline{\mathtt{succ}_i}\, x, \vec{y} : w]\rangle) : \sigma \multimap \sigma}\ (\text{abs})$$

where $\mathcal{D}$ is the derivation

$$\dfrac{\dfrac{\dfrac{\vdots}{;x_1:\mathtt{BInt}\vdash \underline{\mathtt{succ}_i}\,x_1:\mathtt{BInt}} \quad \dfrac{\vdots}{\vec{y'}:\mathtt{BInt},x_2:\mathtt{BInt};w:\S^{q+1}\,\mathtt{BInt}\vdash H_i[\underline{\mathtt{succ}_i}\,x_2,\vec{y'}:w]:\S^{q+1}\,\mathtt{BInt}}}{\vec{y'},x_1:\mathtt{BInt},x_2\,\mathtt{BInt}\vdash\langle\underline{\mathtt{succ}_i}\,x_1,\,H_i[\underline{\mathtt{succ}_i}\,x_2,\vec{y'}:w]\rangle:\tau}}{\vec{y'}:\mathtt{BInt},x:\mathtt{BInt};w:\S^{q+1}\vdash\langle\underline{\mathtt{succ}_i}\,x,\,H_i[\underline{\mathtt{succ}_i}\,x,\vec{y'}:w]\rangle:\tau}\ (\mathrm{con})}{\vec{y'}:\mathtt{BInt}\vdash\lambda x\lambda w.\langle\underline{\mathtt{succ}_i}\,x,\,H_i[\underline{\mathtt{succ}_i}\,x,\vec{y'}:w]\rangle:\mathtt{BInt}\to\S^{q+1}\,\mathtt{BInt}\multimap\tau}\ (\mathrm{abs})$$

The two step functions M_i take a function f of type σ, and return a function that given $\vec{y}$ and $\vec{z}$ returns $\langle\underline{\mathtt{succ}_i}\,x,\,H_i[\underline{\mathtt{succ}_i}\,x,\vec{y}:w]\rangle$, where here x and w represent the result of $f\,\vec{y}\,\vec{z}$. Now we apply the iteration, obtaining

$$;x:\mathtt{BInt}\vdash x\,M_0\,M_1\,N:\S(\tau).$$

Now let us derive apart

$$\vec{y'};\vec{z}:\S^{q+1}\,\mathtt{BInt},f:\tau\vdash f\,\vec{y'}\,\vec{z}\,\underline{\mathtt{false}}:\S^{q+1}\,\mathtt{BInt}\,.$$

To this we apply the chained second coercion (see remark 4.2.64), and we get

$$;\vec{y}:\mathtt{BInt},\vec{z}:\S^{q+2}\,\mathtt{BInt},f:\S\tau\vdash\mathtt{coerc}_{\vec{y'}}[\vec{y},\,f\,\vec{y'}\,\vec{z}\,\underline{\mathtt{false}}]:\S^{q+2}\,\mathtt{BInt}\,.$$

In f we substitute the iteration and finally get

$$;\vec{x}:\mathtt{BInt},\vec{y}:\mathtt{BInt},\vec{z}:\S^{q+2}\,\mathtt{BInt}\vdash\mathtt{coerc}_{\vec{y'}}[\vec{y},\,x\,M_0\,M_1\,N\,\vec{y'}\,\vec{z}\,\underline{\mathtt{false}}]:\S^{q+2}\,\mathtt{BInt}$$

which represents the restricted safe recursion. □

So the final result (at least for now) is ready.

Theorem 4.2.71. *BC^- is representable in* **DLAL**2.

We can also see now how the parallelism between the notion of safeness in BC^- and in **DLAL**2 works out. safe variables in BC^- are non contractible ones. The same holds for safe variables in **DLAL**2.

Proposition 4.2.72. *Safe variables in* **DLAL**2 *are not contractible, i.e. it is not true that for any representation $M[\vec{x}:\vec{y},\vec{z}]$ we can have a representation $M'[\vec{x}:w,\vec{z}]$ with the same algorithmic content of $M[\vec{x}:w,\ldots,w,\vec{z}]$.*

Proof. We will build a flat term with two variables which if contracted would bring outside polytime functions.

This term represents $\mathtt{concat}(x,y) := xy$ in binary notation, that is $\mathtt{concat}(m,n) = 2^{k_n} \cdot m + n$ where $k_n = \lfloor \log_2 n \rfloor + 1$ is the number of binary digits of n. In fact this is much like $\underline{\mathtt{add}}$ for Church integers. We plug the following two derivations

$$f_0, f_1 : \alpha \multimap \alpha; m : \mathtt{BInt} \vdash m\, f_0\, f_1 : \S(\alpha \multimap \alpha), \quad f_0', f_1' : \alpha \multimap \alpha; n : \mathtt{BInt} \vdash n\, f_0'\, f_1' : \S(\alpha \multimap \alpha)$$

into the promotion of

$$; z : \alpha \multimap \alpha, z' : \alpha \multimap \alpha \vdash \lambda x.z\,(z'\,x) : \alpha \multimap \alpha$$

getting by contraction

$$f_0, f_1 : \mathtt{BInt}; m : \mathtt{BInt}, n : \mathtt{BInt} \vdash \lambda x.(m\, f_0\, f_1\, (n\, f_0\, f_1)) : \S(\alpha \multimap \alpha).$$

With two abstractions we get $\underline{\mathtt{concat}}[: m, n|0]$ of the right type.

Now, if we could have $M[: x|q]$ with $M[: \underline{n}] \equiv_\beta \underline{\mathtt{concat}}[: \underline{n}, \underline{n}]$ we would have represented the function $f(n) = 2^{k_n} \cdot n + n$ with a safe argument. So we could apply restricted safe recursion with step functions $H_i[x : w] := M[: w]$ (possible with safe composition) and base value $N[:] := \underline{1}$ and then we would have a term $P[x :]$ which would compute $f^{k_x}(1) = 2^{2^{k_n}} - 1$ which requires exponential space just to write the result.

In fact this proves also that safeness in BC is more strict than in **DLAL**2: the same argument above applies to proving that $\mathtt{concat}(; x, y)$ with both variables safe is not in BC, as here we can contract safe variables by using the more relaxed safe composition. Moreover saying that safe variables are not contractible, means that the unrestricted safe composition is not representable in **DLAL**2 with respect to this notion of safeness.

□

Remark 4.2.73. `concat` is definable in BC^-, but with only one safe variable, because the normal one is needed for recursion. The normal variable is the one being appended at the least significant digits.

$$\mathtt{concat}(0 : x) := \pi_1^{0,1}(: x), \mathtt{concat}(yi : x) \qquad := \mathtt{succ}_i(: \pi_2^{1,1}(y : \mathtt{concat}(y : x))),$$

so that $\mathtt{concat}(y : x) = xy$ in binary notation.

Now we must fill up the distance from polytime Turing machines. This is done by augmenting BC^- with low-level functions that are interpretable in **DLAL**2 with the right safe variables (so that they also do not break polytime soundness), and such that BC^- with these new functions can simulate polytime Turing machines.

The two functions we will add o the algebra are `case` and `e-shift`.

Definition 4.2.74 (`case` and `e-shift`). The `case` function is defined with the use of $m+1$ functions $f_i(:x)$ of a fixed form and m binary strings u_i with $|u_i| < K$, by which we define

$$\mathtt{case}_K\{u_1 : f_1 \mid \cdots \mid u_m : f_m \mid \mathtt{else}\, f_{m+1}\}(:x) := \begin{cases} f_i(:x) & \text{if } x_0 \dots x_K = u_i, \\ f_{m+1}(:x) & \text{otherwise.} \end{cases}$$

So the case construct should return f_i applied on the argument if the $K+1$ least significant digits of the argument match u_i. Clearly we require the u_i to be different. We require that if $|u_i| < K$ then he most significant digit of u_i must be 1 or else $u_i = 0$, and in such cases x matches u_i only if $x = u_i$. Also We require that f_i are of the form

$$f_i(:x) := \mathtt{concat}(\mathtt{succ}_{s_j}(:\mathtt{pred}^{k_j}(:x)) : p_j)$$

where $\mathtt{succ}_s$ with s binary string is defined inductively by $\mathtt{succ}_\varepsilon(:x) := x$ and $\mathtt{succ}_{ia}(:x) := \mathtt{succ}_i(\mathtt{succ}_a(:x))$. So practically f_i can only truncate some least significant bits (as resulting from the multiple application of `pred`) and then add some bits to the head (successors) and to the tail (`concat`).

The *even shift* function `e-shift` shifts all even bits of one place towards the most significant bits putting a 0 in the place left empty:

$$\mathtt{e\text{-}shift}(:b_{2k+1}\dots b_2b_1b_0) := b_{2k}b_{2k+1}\dots b_2b_3b_0b_10,$$
$$\mathtt{e\text{-}shift}(:b_{2k}\dots b_2b_1b_0) := b_{2k}0b_{2k-2}b_{2k-1}\dots b_2b_3b_0b_10,$$

with the convention in the first case to strip the string of the eventual zero brought by b_{2k}.

Lemma 4.2.75 (`case`). *The* `case` *function is representable in* **DLAL**2.

Proof. Recall we have to give a representation such that the depth is preserved. Let k be $\max(k_i)$. We will first do an iteration on type

$$\tau := \left(\bigotimes_{0,K+1} (\forall\beta.\beta^3 \multimap \beta)\right) \otimes \left(\bigotimes_{0,k+1} (\alpha \multimap \alpha)\right).$$

The division between the two products is just for simplicity. The intended value at a given step of an iteration is that the first $K+1$ projections should keep track of the digits being visited so that at the end they should contain $\pi_i := \overrightarrow{\lambda y_0^2}.y_i$ with i being the digit or π_2 for the absence of any digit, all this regarding the least significant digits. Recall that $\vec{X}_0^n$ means by notation a sequence starting with index 0 and ending with index N. The next k spots will be occupied by an f_i corresponding

again to the digits being visited: we will need them to add them to the last variable destined to hold the value in f_is of $\mathtt{pred}^k(x)$. So as base value we will take

$$;\vdash N := \langle\langle \pi_2, \dots, \pi_2\rangle, \langle I, \dots I\rangle\rangle : \tau$$

and the step functions will just add a proper term at the beginning of the two lists, and shift all the rest, and the kth term of the second list instead of being discarded will be composed with the last one.

$$; f_i : \alpha \multimap \alpha \vdash M_i := \lambda p.p\,(\lambda\ell_1\lambda\ell_2.\langle \ell_1\,(\overrightarrow{\lambda y_0^K}.\langle \pi_i, \vec{y}_0^{K-1}\rangle), \ell_2\,(\overrightarrow{\lambda z_0^k}.\langle f_i, \vec{z}_0^{k-2}, z_{k-1} \circ z_k\rangle)\rangle : \tau \multimap \tau.$$

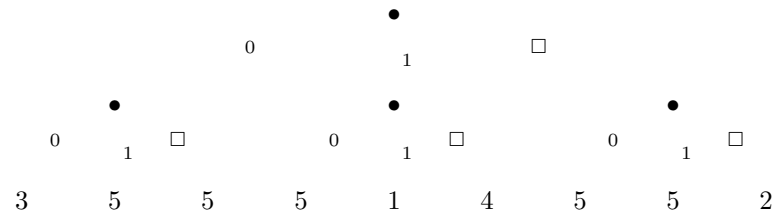
So when we iterate we get

$$f_0, f_1 : \alpha \multimap \alpha; n : \mathtt{BInt} \vdash n\, M_0\, M_1\, N : \S\tau.$$

Now we will use the information in the first K-uple to choose what function to apply, and then sewhat we gathered in the second one to apply the functions without needing the argument anymore (so we will keep it linear).

First create a ternary complete tree with depth $K+1$: it will be the decision tree leading to the right function. So label the three branches of every node with 0, 1 and □. Now starting from the root, for every i we have to follow the path corresponding to u_i read from least significant digit, where □ corresponds to empty digit (we fill up u_i with boxes up to digit number K). When we get to the end of the branch we label the leave with the corresponding i. All the leaves that do not get labeled in this way get label $m+1$.

Now inductively build the term `selection` from the tree: for every leave labeled with i the corresponding term is $\pi_i^m := \overrightarrow{\lambda y^{m+1}}.y_i$, while for every node the corresponding term is the 3-uple built with the three terms corresponding to the three subtrees originating from that node. So for example if $K = 1$ and we have the four cases 11, 0 (meaning the empty string), 00 , and 1, then the corresponding tree is



and the corresponding term is

$$\mathtt{selection} = \langle\langle \pi_3, \pi_5, \pi_5\rangle, \langle \pi_5, \pi_1, \pi_4\rangle, \langle \pi_5, \pi_5, \pi_2\rangle\rangle.$$

This term is typable with type $\rho(K)$, where $\rho(0) = \bigotimes_{0,3}(\forall\beta.\beta^{m+1} \multimap \beta)$ and $\rho(h{+}1) = \bigotimes_{0,3}(\rho(k))$. In particular we can apply it repeatedly to terms of type $\forall\beta.\beta^3 \multimap \beta$ to access its subtrees and in the end the leaves.

So given something of type $\bigotimes_{0,K+1}(\forall\beta.\beta^3 \multimap \beta)$, we may design a term that selects the right selector to pick the right function.

$$;\vdash \lambda s.s\,(\overrightarrow{\vec{y}}.\texttt{selection}\,\vec{y}) : \bigotimes_{0,K+1}(\forall\beta.\beta^3 \multimap \beta) \multimap \forall\beta.\beta^{K+1} \multimap \beta.$$

Let's call this term `select`. Note that in the derivation every y_i gets typed a different way by instantiation.

Now we will design the terms F_i that represent the actions of the functions f_i, which work on the second product. Practically we take the last component (which represents the predecessor applied k times), recuperate from the other bits what is needed (if $k_i < k$), and then manually put in front and in the end the other bits. So if we must represent $\texttt{succ}_s\,\texttt{concat}(\texttt{pred}^h(:x)) : p)$ we just use

$$;f_0, f_1 : \alpha \multimap \alpha, q : \bigotimes_{0,k+1}(\alpha \multimap \alpha) \vdash q\,(\overrightarrow{\lambda z}.\lambda w.$$
$$f_{s_0}\,(f_{s_1}\dots(f_{s_{|s|-1}}\,(z_h\,(\dots z_{k-1}\,(z_k\,(f_{p_0}\,(\dots(f_{p_{|p|-1}}\,w)\dots)))\dots)))\dots) : \alpha \multimap \alpha.$$

Let's call this term $F_{s,h,p}$, and consider in particular $F_i := \lambda q.F_{s_i,k_i,p_i}$, typable with $\bigotimes_{0,k+1}(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$. Note that while all the other symbols are meant to hold only one bit each, z_k instead holds everything that's not erased by the predecessor. Now combine all the functions to get

$$;\vec{f_0}, \vec{f_1} : \alpha \multimap \alpha \vdash \langle\vec{F}\rangle : \bigotimes_{0,m+1}(\bigotimes_{0,k+1}(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha).$$

Note the multiple f_is. Now we put together he last term and the `select` term, making it all dependant on a variable of type τ:

$$;\vec{f_0}, \vec{f_1} : \alpha \multimap \alpha, r : \tau \vdash r\,(\lambda\ell_1\lambda\ell_2.\langle\vec{F}\rangle\,(\texttt{select}\,\ell_1)\,\ell_2) : \alpha \multimap \alpha.$$

Finally we promote, plugging the iteration in r and shifting the various f_is on the linear side, and then contracting them together with the two already present for the iteration, and we get:

$$f_0, f_1 : \alpha \multimap \alpha; n : \texttt{BInt} \vdash n\,M_0\,M_1\,N\,(\lambda\ell_1\lambda\ell_2.\langle\vec{F}\rangle\,(\texttt{select}\,\ell_1)\,\ell_2) : \S(\alpha \multimap \alpha)$$

which by two abstractions and a generalization becomes the term $\underline{\texttt{case}}_K[\dots]$. □

Lemma 4.2.76 (`e-shift`). `e-shift` *is representable in* **DLAL**2.

Proof. First let us give a flat term that almost does the operation required, as it shift the even bits, but counting from the most significant one. As usual we do an iteration on a list where one of the components is $\alpha \multimap \alpha$ that contains the composition of the bits of the result.

The type on which we do the iteration is $\bigotimes_{0,3}(\rho, \alpha \multimap \alpha, \alpha \multimap \alpha)$, where $\rho := \forall\beta.(\beta^2 \multimap \beta)$. The intended meaning of a value of this type is that the first component is $\pi_i := \lambda x_0 \lambda x_1 . x_i$ where i tells us whether we are passing an even ($i = 0$) or odd bit ($i = 1$), the second is the bit we eventually put apart for use two digits later, and the third holds the result so far. The step functions will therefore be:

$$; f_i : \alpha \multimap \alpha \vdash M_i := \lambda p.p\,(\lambda x \lambda y \lambda z.x\,\langle \pi_1, f_i, y \circ z\rangle\,\langle \pi_0, y, f_i \circ z\rangle) : \tau \multimap \tau.$$

So the current scanned bit gets directly appended when passing odd digits, while if we are passing an even one we put the bit aside and use the saved one instead. The base value is

$$; f_0 \vdash N := \langle \pi_0, f_0, I\rangle : \tau,$$

as the first bit must be changed to 0. Note how we are on purpose forgetting about trailing zeros. The iteration gives (we shift the f_0 of the base value to intuitionistic side and then contract it with the one from M_0):

$$f_0, f_1 : \alpha \multimap \alpha; n : \texttt{BInt} \vdash n\,M_0\,M_1\,N : \S\tau.$$

The result of the iteration, apart from being extracted, must be then corrected to add the saved bit at the right position. If we arrive at the end when we expected an even digit we just add the saved bit, while otherwise we have to add 0 and the saved bit. This operation is carried out by the following term:

$$; p : \tau, f_0' : \alpha \multimap \alpha \vdash p\,(\lambda x \lambda y \lambda z.x\,(y \circ z)\,(y \circ (f_0' \circ z))) : \alpha \multimap \alpha.$$

Note that here x gets instantiated with type $(\alpha \multimap \alpha)^2 \multimap \alpha \multimap \alpha$. Then we promote this last derivation, plugging in p the result of the iteration, and shifting f_0' to the linear side to contract it with the f_0 already present:

$$f_0, f_1 : \alpha \multimap \alpha; n : \texttt{BInt} \vdash n\,M_0\,M_1\,N\,(\lambda x \lambda y \lambda z.x\,(y \circ z)\,(y \circ (f_0' \circ z))) : \S(\alpha \multimap \alpha).$$

Finally we abstract the two f_is and generalize to get a term we will call $\texttt{rev} - \texttt{shift}[: n|0]$ (reversed shift).

Now we will program a term that reverses the order of bits of a term of type `BInt`, as if it was just a list of bits. We do it by just composing the bits in reversed order.The type on which we iterate is $\alpha \multimap \alpha$, the step functions are

$$; f_i : \alpha \multimap \alpha \vdash \lambda x.x \circ f_i : (\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$$

and the base value is I, so that iteration, two abstractions and generalization give

$$; n : \mathtt{BInt} \vdash \mathtt{rev} : \mathtt{BInt}$$

where $\mathtt{rev}[: n|0] := \lambda f_0 \lambda f_1 . n\,(\lambda x . x \circ f_0)\,(\lambda x . x \circ f_1)\,I$.

Now we can obtain `e-shift` just by first reversing the bits, then applying $\mathtt{rev} - \mathtt{shift}$, then reversing again and finally applying `strip` to cut off the eventual 0 at the beginning:

$$; n : \mathtt{BInt} \vdash \underline{\mathtt{e\text{-}shift}} : \mathtt{BInt}$$

with $\underline{\mathtt{e\text{-}shift}}[: n|0] := \underline{\mathtt{strip}}[: \mathtt{rev}[: \mathtt{rev} - \mathtt{shift}[: \mathtt{rev}[: n]]]]$. □

Let us call $BC^{\pm}$ the class BC^{-} augmented with the `case` constructs and `e-shift`.

Corollary 4.2.77. *All functions in $BC^{\pm}$ are polytime.*

Proof. We have just seen that the `case` constructs and `e-shift` are encodable with **DLAL**2 terms. Together with the compositional representation of BC in **DLAL**2 it ensures us that we remain in the polytime functions. □

Theorem 4.2.78 (polytime completeness of $BC^{\pm}$). *Every polytime function f can be written as $f(\vec{n} :)$ in $BC^{\pm}$*

Proof. (sketch) First of all we see that the function

$$f_p(x :) := \underbrace{11 \ldots 1}_{p(|x|)}$$

is representable in BC^{-}, where p is a polynomial and $|x|$ is the length of the binary expansion of x. We define it by induction on the construction of the polynomial:

$$\begin{aligned} f_1(x :) &:= \pi_1^{1,0}(\mathtt{succ}_1(: \mathbf{0}) :), \\ f_{p_1+p_2}(x :) &:= \mathtt{concat}(f_{p_1}(x :) : f_{p_2}(x :)), \\ f_{X \cdot p}(x :) &:= g_p(\pi_1^{1,0}(x :), \pi_1^{1,0}(x :) :), \end{aligned}$$

where $g_p(x_1, x_2 :)$ is defined to give $|x_1| \cdot p(|x_2|)$ 1s,using the definition by restricted safe recursion that uses f_p:

$$g_p(0, x_2) := \mathbf{0}; g_p(xi, x_2) \qquad := \mathtt{concat}(f_p(x_2) : g_p(x, x_2 :)).$$

The function f_p is needed as a clock for the polytime computation.

Suppose now we have a polytime Turing machine that computes a function, with a polynomial $p(X)$ tat bounds he time needed for any computations. For simplicity we can suppose the function

being computed is in only one argument. We suppose that the machine has a single tape, and that at the end the output is written in binary starting where the head of the machine stands, and all the rest of the tape is empty. We encode all the states with numbers with binary expansion of length K with K even, all with least significant digit 1, and all symbols with binary strings of length L, with a 1 at the beginning an at the end (so that in fact $L \geq 3$), apart from the blank symbol that is encoded by L zeros. For commodity we require that the encoding of 1 and 0 begin with 11 and 10 respectively.

The configuration of a machine is then coded by a binary string such that the first K less significant digits are the code of the current state, and then the codes of the left and right parts of the tape are written with alternating bits starting from he position of the head, i.e. if concatenating the chosen code for symbols of the whole tape (apart from the infinite blank symbols) gives

$$\dots s_{-4}s_{-3}s_{-2}s_{-1}s_0s_1s_2s_3s_4\dots$$

and the current state is coded by $q_0 \dots q_{K-1}$ then the corresponding code for our purposes is

$$q_0 \dots q_{K-1}s_0s_{-1}s_1s_{-2}s_2s_{-3}s_3s_{-4}s_4\dots,$$

starting from the least significants bits. Note that if the left and right part are different in length as to the written part, then the distance is filled up with zeros which is the right encoding of the blank symbol. Also the fact of having required that the encodings of non-blank symbols are delimited by ones makes so that no information is lost if we regard the string as an integer and delete trailing zeros.

Now the encoding of the initial configuration is easily carried out by safe recursion and composition of concatenations. For the transition we use $\texttt{case}_{K+2L-2}$: we have to look only at the bits corresponding to the state and the symbol under the head of the machine. Note that each case given by the Turing machine must be repeated here for all the possible combinations of the bits regarding the left part of the tape. In any case the effect of `case` must be to truncate all the bits of the state, put back the ones of the new state, and eventually put an additional 0 if the head must move to the right.

The movement of the tape is simulated using `e-shift`: in fact take the encoding of the tape after a change of state and a movement to the left by one bit has been done (we will have to repeat it L times):

$$q'_0 \dots q'_{K-1}s_{-1}s_{-2}s_0s_{-3}s_1s_{-4}s_2s_{-5}s_3\dots,$$

while the result of applying `e-shift` two times is (recall K is even):

$$0q'_10q'_3q_0\dots q'_{K-1}q'_{K-4}s_{-1}q'_{K-2}s_{-2}s_0s_{-3}s_1s_{-4}s_2s_{-5}s_3\dots$$

Also when moving to the right `e-shift` works in a similar manner because we have put an additional 0 at the beginning. Moreover we can distinguish between the two cases by just watching the 5th bit: it is zero only if we have added it, otherwise it is q_0 which we have chosen to be 1. Correcting what's wrong with the result of $\texttt{e-shift}^2$ (the mixed up state) requires dealing only with the first $K+5$ bits, and that can be accomplished by a convenient `case` construct with finite though many patterns. Then we can repeat the bit shifting L times and in the end with a $\texttt{case}_1$ remove the eventual zero appended for movemet to the right. What we get is a function $\texttt{transition}(:n)$ with a safe variable, so that it can be iterated by the following function:

$$\begin{aligned}\texttt{iteration}(0:m) &:= m,\\ \texttt{iteration}(xi:m) &:= \texttt{transition}(:\texttt{iteration}(x:m)).\end{aligned}$$

The extraction of the result is then done by first deleting the bits of the state with the predecessor, then extracting the right part of the tape using the following function

$$\begin{aligned}\texttt{aux}(0:) &:= 1,\\ \texttt{aux}(xi) &:= \texttt{case}_1[\\ &\qquad 1: \quad \texttt{succ}_0(:\texttt{succ}_i(:\texttt{pred}(:u)))\ |\\ &\qquad 0: \quad \texttt{succ}_1(\texttt{pred}(:u))\\ &\quad](:\texttt{aux}(:x)),\end{aligned}$$

which practically uses the least significant digit as a flag. The extraction of the content of the left tape is then done by erasing with `pred` this flag. Similarly we can then extract the value by looking at the second bit of every codification (because alone it can already tell us if the symbol is 0 or 1) and then jumping the next L bits with the use of several flags.

So in the end the whole computation of the Turing machine is represented by the function

$$\texttt{extraction}(\texttt{iteration}(f_p(x:):\texttt{init}(x:)):)$$

where `extraction` is the extraction of the output, and `init` is the injection of the input. □

Corollary 4.2.79. *A function is representable in* **DLAL**2 *if and only if it is polytime.*

4.3 TC, TYP and type inference

When dealing with light logics type checking and typability assume new importance. They do not only provide a way to check the correctness as to termination, but even give complexity bounds

on the computation. Again the best thing would be to leave to a machine the part consinsting in deriving the type, while the programmer just writes the pure term.

In fact type checking and type inference have been proved decidable for both the propositional fragments of **EAL** and **LAL**, and also an adaptation of the algorithm for **EAL** has been proposed for **DLAL**.

4.3.1 Type inference for EAL

Typing for the first order *simple* fragment of **EAL** has been solved by Coppola and Martini in [CM01], and then refined in a stronger way by Coppola and Ronchi Della Rocca in [CRDR03]. By simple it is meant that a restriction on contractions is in place: contractions are made only on variables, without plugging in more complex terms[3].

Definition 4.3.1 (simple terms). Let $\#$ be the following measure on Λ^{EA} terms (it consists of a length function in which variables are considered of length 0):

$$\begin{aligned}
\#(x) &:= 0,\\
\#(M_1\, M_2) &:= 1 + \#(M_1) + \#(M_2),\\
\#(\lambda x.M) &:= 1 + \#(M),\\
\#((!M)\{\overrightarrow{P/x}) &:= \#(M) + \sum \overrightarrow{\#(P)},\\
\#(\{M\}_{N\to\vec{x}}) &:= \#(M) + \#(N).
\end{aligned}$$

Recall that $(\,.\,)^+ : \Lambda^{\mathrm{EA}} \to \Lambda$ is the function resolving the substitutions meant by the two additional constructs in Λ^{EA}. Clearly $\#(M) \leq \#((M)^+)$.

We say a Λ^{EA}-term M is *simple* if it contains no subterm $\{M\}_{N\to\vec{x}}$ such that N is not a variable, and moreover $\#(M) = \#((M)^+)$. The second condition is required so that not even deferred contractions of non-variable terms happens, in the sense of a contraction of a variable into which we plug a complex term afterwards using a box. In fact it implies that in a boxed term $(!M)\{\overrightarrow{P/x}\}$ if P_i is not a variable then x_i occurs only once in $(M)^+$, i.e. it is not contracted by a previous contraction.

We say a pure term M is *simply* typable in **EAL** if there is a typable simple Λ^{EA}-term N such that $(N)^+ = M$.

We further say that a typing of a term is *simple* if the corresponding Λ^{EA} term is simple.

The idea is first of all to adapt Λ^{EA} to a grammar that merges multiple promotions and contractions.

[3] this has a justification in the fact that **EAL** typing is related to the application of optimal reduction by the Lamping algorithm, which requires the initial translation into sharing graphs to have only variables being shared.

Definition 4.3.2 (Abs^{EA}). The set of *abstract elementary affine terms* is built from $\mathcal{V}$ with the following grammar:

$$Abs^{\mathrm{EA}} ::= \mathbb{V} \mid (Abs^{\mathrm{EA}}\ Abs^{\mathrm{EA}}) \mid \lambda\mathcal{V}.Abs^{\mathrm{EA}} \mid \{Abs^{\mathrm{EA}}\}_{\overrightarrow{Abs^{\mathrm{EA}} \to \vec{\mathbb{V}}}} \mid (\nabla Abs^{\mathrm{EA}})\{\overrightarrow{Abs^{\mathrm{EA}}/\mathcal{V}}\}.$$

The last rules introduce respectively contracted and boxed terms as in Λ^{EA} (see definition 4.2.6): they undergo the same conditions and for them we adopt the same conventions. Free variables are also defined like for Λ^{EA}:

$$\begin{aligned}
\mathrm{FV}(\{M\}_{\overrightarrow{N \to x}}) &:= \mathrm{FV}(M) \setminus \{\overrightarrow{x}\} \cup \mathrm{FV}(\vec{N}),\\
\mathrm{FV}((\nabla M)\{\overrightarrow{N/x}\} &:= \mathrm{FV}(M) \setminus \{\vec{x}\} \cup \mathrm{FV}(\vec{N}),\\
\mathrm{BV}(\{M\}_{\overrightarrow{N \to x}}) &:= \mathrm{BV}(M) \cup \mathrm{BV}(\vec{N}),\\
\mathrm{BV}((!M)\{\overrightarrow{N/x}\} &:= \mathrm{BV}(M) \cup \mathrm{BV}(\vec{N}).
\end{aligned}$$

We will call *sharing list* the list $\overrightarrow{N \to x}$ in $\{M\}_{\overrightarrow{N \to x}}$, and we will range over them with letters such as ℓ. We will allow such a list to be empty, in which case $\{M\}_{\emptyset}$ denotes just M. The set of *contracted variables* is defined on sharing lists as

$$\mathrm{CV}(\overrightarrow{N \to \vec{x}}) := \{\overrightarrow{\vec{x}}\},$$

while we extend the notion of free variables by

$$\mathrm{FV}(\overrightarrow{N \to \vec{x}}) := \mathrm{FV}(\vec{N}).$$

One should interpret abstract terms as sets of Λ^{EA}-terms, so that contracted abstract terms stand for the terms that contract the same variables in different order, and $(\nabla M)\{\overrightarrow{N/x}\}$ stands for terms of the form $(!^m M)\{\overrightarrow{N/x}\}$.

Together with the new grammar for terms comes a new type assignment system we will call *Abs*. The rules are the ones for Λ^{EA}, in which the only differences regard (prom) and (con), which take the form:

$$\frac{\overrightarrow{A \vdash N : !\sigma} \quad \overrightarrow{\vec{x} : !\sigma}, B \vdash M : \tau}{\vec{A}, B \vdash \{M\}_{\overrightarrow{N \to \vec{x}}} : \tau}\ (\mathrm{con}) \qquad \frac{\overrightarrow{B \vdash N : !^m\sigma} \quad \overrightarrow{x : \sigma} \vdash M : \tau}{\vec{A} \vdash (\nabla M)\{\overrightarrow{N/x}\} : !^m\tau}\ (\mathrm{prom})$$

Again for ease of notation we may write the promotion as

$$\frac{\overrightarrow{B \vdash N : !^m\sigma} \quad \overrightarrow{x : \sigma}, B \vdash M : \tau}{\vec{A}, !B \vdash (\nabla M)\{\overrightarrow{N/x}\} : !^m\tau}\ (\mathrm{prom})$$

There is a function that easily injects Λ^{EA} terms into Abs^{EA}-terms. We will here denote it by emb, and it is easily defined by:

$$\begin{aligned}
\mathrm{emb}(x) &:= x,\\
\mathrm{emb}(\lambda x.M) &:= \lambda x.\,\mathrm{emb}(M),\\
\mathrm{emb}(M_1\,M_2) &:= \mathrm{emb}(M_1)\;\mathrm{emb}(M_2),\\
\mathrm{emb}(\{M\}_{N\to\vec{x}} &:= \{\mathrm{emb}(M)\}_{\mathrm{emb}(N)\to\vec{x}},\\
\mathrm{emb}((!M)\{\overrightarrow{N/x}\}) &:= (\nabla\,\mathrm{emb}(M))\{\overrightarrow{\mathrm{emb}(N)/x}\}.
\end{aligned}$$

It is easy then to see that

$$A \vdash_{\mathbf{EAL}_s} M : \tau \implies A \vdash_{Abs} \mathrm{emb}(M) : \tau.$$

In the other direction we can define a function bme that brings Abs^{EA} to sets of terms in Λ^{EA}, by expanding the possibilities ∇ represents.

$$\begin{aligned}
\mathrm{bme}(x) &:= \{x\},\\
\mathrm{bme}(\lambda x.M) &:= \{\,\lambda x.M' \mid M' \in \mathrm{bme}(M)\,\},\\
\mathrm{bme}(M_1\,M_2) &:= \{\,(M_1'\,M_2') \mid M_1' \in \mathrm{bme}(M_1),\ M_2' \in \mathrm{bme}(M_2)\,\},\\
\mathrm{bme}(\{M\}_{\ell,N\to\vec{x}} &:= \{\,\{M'\}_{N'\to\vec{x}} \mid M' \in \mathrm{bme}(\{M\}_\ell),\ N' \in \mathrm{bme}(N)\,\},\\
\mathrm{bme}((\nabla M)\{\overrightarrow{N/x}\}) &:= \{\,(!^m M')\{\overrightarrow{N'/x} \mid M' \in \mathrm{bme}(M),\ N_i' \in \mathrm{bme}(N_i),\ m > 0\,\}.
\end{aligned}$$

This definition makes sense once we fix a particular way in which the sharing lists are ordered.

Now in this opposite way we can show that

$$A \vdash_{Abs} M : \tau \implies \exists M' : A \vdash_{\mathbf{EAL}_s} M' : \tau;$$

the particular M' is practically chosen by the actual number of bangs involved in the rules used to derive ∇s in M.

In a way similar to how we defined canonical derivations in **DLAL** we here define canonical terms by means of a special reduction, designed to reduce to the minimum nesting of contracted and boxed terms.

Definition 4.3.3 (canonical reduction). The *canonical one step reduction* is denoted by $\to_{\mathrm{e}}$

and is the least relation that passes to context for which

$$
\begin{aligned}
(\nabla(\nabla M))\{\overrightarrow{N/x}\} &\to_{\mathcal{C}} (\nabla M)\{\overrightarrow{N/x}\},\\
\{\{M\}_{\ell_1,y\to\vec{x}}\}_{\ell_2,N\to(\vec{z},y)} &\to_{\mathcal{C}} \{\{M\}_{\ell_1}\}_{\ell_2,N\to(\vec{z},\vec{x})},\\
\{\{M\}_{\ell_1,N\to\vec{x}}\}_{\ell_2} &\to_{\mathcal{C}} \{\{M\}_{\ell_1}\}_{\ell_2,N\to\vec{x}} \qquad \text{if } \mathrm{FV}(N)\cap\mathrm{CV}(\ell_2)=\emptyset,\\
(\nabla M)\{\overrightarrow{N/x},(\nabla P)\{\overrightarrow{Q/y}\}/z\} &\to_{\mathcal{C}} (\nabla M[P/z])\{\overrightarrow{N/x},\overrightarrow{Q/y}\},\\
(\nabla M)\{\overrightarrow{N/x},\{P\}_\ell/y\} &\to_{\mathcal{C}} \{(\nabla M)\{\overrightarrow{N/x},P/y\}\}_\ell,\\
(\nabla\{M\}_{\ell,x\to\vec{y}})\{\overrightarrow{N/z},w/x\} &\to_{\mathcal{C}} \{(\nabla\{M\}_\ell)\{\overrightarrow{N/z},\overrightarrow{t/y}\}\}_{w\to\vec{t}},\\
\{M\}_{\ell_1,\{N\}_{\ell_2}\to\vec{x}} &\to_{\mathcal{C}} \{\{M\}_{\ell_1,N\to\vec{x}}\}_{\ell_2},\\
(\{M\}_\ell\, N) &\to_{\mathcal{C}} \{(M\,N)\}_\ell,\\
(M\,\{N\}_\ell) &\to_{\mathcal{C}} \{(M\,N)\}_\ell,\\
\lambda x.\{M\}_\ell &\to_{\mathcal{C}} \{\lambda x.M\}_\ell \quad \text{if } x\notin\mathrm{FV}(\ell),\\
(\nabla M)\{\overrightarrow{N/x},\overrightarrow{P/y}\} &\to_{\mathcal{C}} (\nabla M)\{\overrightarrow{N/x}\} \quad \text{if } \vec{y}\cap\mathrm{FV}(M)=\emptyset,\\
(\nabla x)\{M/x\} &\to_{\mathcal{C}} M,\\
\{M\}_{\ell,N\to(\vec{x},\vec{y})} &\to_{\mathcal{C}} \{M\}_{\ell,N\to\vec{x}} \quad \text{if } \vec{y}\cap\mathrm{FV}(M)=\emptyset,\\
\{M\}_{\ell,N\to\vec{x}} &\to_{\mathcal{C}} \{M\}_\ell \quad \text{if } \vec{x}\cap\mathrm{FV}(M)=\emptyset,\\
\{M\}_{N\to x} &\to_{\mathcal{C}} M[N/x],
\end{aligned}
$$

Note we make extensive use of the convention that no variables are repeated twice, so that there is no clash when moving around shared lists.

As usual the reflexive and transitive closure is denoted by $\twoheadrightarrow_{\mathcal{C}}$.

Aided by the fact that the system is syntax directed apart from the term-invariant (ins) and (gen) it is easy to see that $\twoheadrightarrow_{\mathcal{C}}$ enjoys subject reduction. Moreover by inspection of all the possible cases it is also Church-Rosser. So for M a Λ^{EA} we may define $\mathcal{C}(M)$ as the unique normal form. We only prove an important property that is not stated in [CRDR03], which allow us to define $\mathcal{C}(M)$.

Proposition 4.3.4. $\twoheadrightarrow_{\mathcal{C}}$ *is strongly normalizing.*

Proof. We define some measures of terms. We first define a subterm in the usual way, using a notion that does not take into account the substitutions implied by the construction rules, i.e. for example $(x\,y)$ is considered a subterm of $\{x\,y\}_{I\to(x,y)}$.

The *relative depth* $\mathrm{d}(N,M)$ of a subterm N in a term M is defined inductively so that

$\mathrm{d}(M,M)=0$ and in the other cases (which excludes $M=x$):

$$\begin{aligned}
\mathrm{d}(N,\lambda x.M) &:= 1+\mathrm{d}(N,M),\\
\mathrm{d}(N,M_1\,M_2) &:= 1+\mathrm{d}(N,M_i) \quad \text{where } N \text{ is a subterm of } M_i,\\
\mathrm{d}(N,\{M\}_{\overrightarrow{P\to x}}) &:= \begin{cases} 1+\mathrm{d}(N,M) & \text{if } N \text{ is a subterm of } M,\\ 1+\mathrm{d}(N,P_i) & \text{if } N \text{ is a subterm of } P_i,\end{cases}\\
\mathrm{d}(N,(\nabla M)\{\overrightarrow{P/x}\}) &:= \begin{cases} 1+\mathrm{d}(N,M) & \text{if } N \text{ is a subterm of } M,\\ 1+\mathrm{d}(N,P_i) & \text{if } N \text{ is a subterm of } P_i.\end{cases}
\end{aligned}$$

We denote by $\#_1(M)$ the number of ∇ in the term plus the total number of auxiliary variables appearing in sharing lists.

Then define recursively the following measure we will here denote by $\#_2(N,M)$ where N is a subterm of M:

$$\begin{aligned}
\#_2(x,M) &:= 0,\\
\#_2(\lambda x.N,M) &:= \#_2(N,M),\\
\#_2(N_1\,N_2,M) &:= \#_2(N_1,M)+\#_2(N_2,M),\\
\#_2(\{N\}_{\overrightarrow{P\to x^n}},M) &:= n\cdot\mathrm{d}(N,M)+\#_2(N,M)+\sum\#_2(P,M),\\
\#_2((\nabla N)\{\overrightarrow{P/x^n}\},M) &:= n+\#_2(N,M)+\sum\#_2(P,M).
\end{aligned}$$

With respect to lexicographic order $\to_{\mathcal{C}}$ strictly reduces the measure $(\#_1(M),\#_2(M,M))$. This is based on the fact that most of the rules bring the contractions down while leaving the rest as it is. Of the only two cases that may increase $\#_2$ by doing actual substitutions (which may bring contractions deeper in the term), one erases a ∇ and the other erases an auxiliary variable in a sharing list, and the substitution cannot be a duplicating one, so that $\#_1$ decreases strictly. □

Lemma 4.3.5. *The canonical forms of simple typable* Λ^{EA} *terms respect the following properties:*

1. *Every contracted subterm apperas either in the form* $\lambda x.\{M\}_{x\to\vec{y}^n}$ *where* $\vec{y}\subseteq\mathrm{FV}(M)$ *and* $n\geq 2$*, or else the term itself is in the form* $\{M\}_\ell$ *with* $\mathrm{CV}(\ell)\subseteq\mathrm{FV}(M)$ *and* ℓ *contracting only variables and always to at least two variables. Basically contractions are either final or just before they are needed by an abstraction, and they are always on a variable.*
2. *There is no subterm of the form* $(\nabla(\nabla M)\{\overrightarrow{x/y}\}\{\overrightarrow{P/x}\}$*, i.e. a boxed term that plugs in only variables cannot be boxed again, so it can only terminate a chain of boxings.*
3. *No variable is ever boxed or contracted, i.e. there are no subterms of the form* $\{x\}_{N\to\vec{y}}$ *or* $(\nabla x)\{\overrightarrow{P/y}\}$.

4. *The only terms that can be plugged in a boxed term are variables and applications. The boxed and contracted terms are excluded because of the rules of canonical reduction, and moreover no abstraction can have as type one with the bang necessary to be plugged in.*
5. *No boxed term appears at the left of an application, as it would not get a suitable type.*

On the converse if a term respects these properties it is the canonical form of a simple term, though it can be untypable.

Next we will show in what sense *Abs* admits principal pairs. To do so we need types that do not only have type variables, but also variables on the number of modalities.

Definition 4.3.6 (EAL type schemes). The set of type schemes $\mathbb{T}_{Abs}$, ranged over again by Greek letters such as σ, τ, is defined from the set of type variables by the grammar

$$\mathbb{T}_{Abs} ::= \mathbb{V} \mid \mathbb{T}_{Abs} \multimap \mathbb{T}_{Abs} \mid !^{\Lambda_{\mathbb{N}}}(\mathbb{T}_{Abs}),$$

where $\Lambda_{\mathbb{N}}$ is the set of *exponent schemes* ranged over by letters such as p, q and built from a countable set of natural literals $\mathcal{V}_{\mathbb{N}}$ by the grammar

$$\Lambda_{\mathbb{N}} := \mathcal{V}_{\mathbb{N}} \mid \Lambda_{\mathbb{N}} + \Lambda_{\mathbb{N}}.$$

The set of exponent schemes is considered quotiented with respect to commutativity and associativity of $+$. We denote by $\sum \vec{n}$ the exponent scheme $n_1 + \cdots + n_k$ where n_i are variables in $\mathcal{V}_{\mathbb{N}}$ rather than actual integers. The set of type schemes is quotiented with respect to the least equivalence relation $\sim$ that passes to context such that

$$!^p(!^q(\tau)) \sim !^{p+q}(\tau).$$

We will always take as representant the one without subtypes of the form $!^p(!^q(\sigma))$.

A *scheme substitution* is a function T with $\mathrm{DOM}(T) = \mathbb{V} \cup \mathcal{V}_{\mathbb{N}}$ and finite support such that $T(\mathbb{V}) \subseteq \mathbb{T}_{\mathbf{AL}}$ and $T(\mathcal{V}_{\mathbb{N}}) \subseteq \mathbb{N} \setminus \{0\}$. T is extended to $T : \Lambda_{\mathbb{N}} \to \mathbb{N}$ linearly by $T(\sum \vec{n}) := \sum \overrightarrow{T(n)}$, and then to $T : \mathbb{T}_{Abs} \to \mathbb{T}_{\mathbf{AL}}$ by the relations:

$$T(\sigma \multimap \tau) := T(\sigma) \multimap T(\tau),$$
$$T(!^p(\sigma)) := !^{T(p)}T(\sigma).$$

Note that on the left the exponent is an actual number, which means a certain number of exponential prepended to the type.

We will use a notion of syntactical equivalence that ignores only the exponent scheme. It is the least equivalence relation that passes to context such that

$$\sigma_1 \multimap \sigma_2 =_e \tau \iff \tau = \tau_1 \multimap \tau_2, \quad \sigma_i =_e \sigma_i,$$
$$!^p\sigma' =_e \tau \iff \tau = !^q\tau', \quad \sigma' =_e \tau'.$$

The first part is to design a unification function U: its job is to unify the implicational structure of the type schemes as the unifier seen for system **S** did (see subsection 2.2.5), and in the same time design a system of linear equations on the literals involved.

Definition 4.3.7 (unifiers and most general unifiers). We say a scheme substitution T is a solution to a set C of linear constraints if for every $p = q \in C$ we have that $T(p) = T(q)$. It is trivial that asking whether there exist a solution of C is decidable.

We will call *unifier* for σ and τ a pair (C, S) where C is a set of linear constraints on literals in $\Lambda_{\mathbb{N}}$ and S a substitution based on $S : \mathbb{V} \to \mathbb{T}_{Abs}$, if $S(\sigma) =_e S(\tau)$ and for every T such that T is a solution of C we have that $T(S(\sigma)) = T(S(\tau))$[4]. We say it is *a most general one* if for every T such that $T(\sigma) = T(\tau)$ then T is a solution of C and there exist a scheme substitution T' such that $T|_{\mathbb{T}_{Abs}} = T' \circ S$.

Proposition 4.3.8. *There is an algorithm that taken in input two type schemes gives as output* `fail` *of there is no unifier, otherwise* (C, S) *where* C *is a set of linear constraints on* $\mathbb{V}_{\mathbb{N}}$ *in the form* $p = q$ *with* p *and* q *exponent schemes, and* S *is a substitution on types.*

Proof. U can be given by the following algorithm:

Require: σ and τ type schemes;

1: **if** $\sigma = \alpha$ **then**
2: **if** $\tau = \alpha$ **then return** $(\emptyset, [\ ])$;
3: **else**
4: **if** $\alpha \notin \mathrm{FTV}(\tau)$ **then return** $(\emptyset, [\tau/\alpha])$;
5: **elsereturn** `fail`;
6: **else**
7: **if** $\tau = \alpha$ **then return** $U(\alpha, \sigma)$;
8: **else**
9: **if** $\sigma = !^p\sigma'$ **then**
10: **if** $\tau = !^q\tau'$ **then**
11: $(C', S') \leftarrow U(\sigma', \tau')$;
12: **if** $(C', S') =$ `fail` **then return** `fail`;
13: **else return** $(C' \cup \{p = q\}, S')$;
14: **else**
15: we have $\sigma = \sigma_1 \multimap \sigma_2$;

[4] in fact we can require this property only for T that is the identity on type variables, as the syntactical identity is already required. The previous condition is needed so that we cannot build a unifier by just using an unsolvable linear system.

16: **if** $\tau = \tau_1 \multimap \tau_2$ **then**
17: $S_1 \leftarrow U(\sigma_1, \tau_1)$;
18: **if** $(C_1, S_1) =$ `fail` **then return** `fail`;
19: **else**
20: $(C_2, S_2) \leftarrow U(S_1(\sigma_2), S_1(\tau_2))$;
21: **if** $(C_2, S_2 =$ `fail` **then return** `fail`;
22: **else return** $(C_1 \cup C_2, S_2 \circ S_1)$;
23: **elsereturn** `fail`;

Then by induction:

$\sigma = \alpha$**:** the case in which the result is `fail` is when no possible substitution compatible with $=_e$. On the other hand a positive result is easily seen to be a unifier. Moreover if T is such that $T(\alpha) = T(\tau)$, we have that T is automatically a solution of $\emptyset$, and on the other hand we may see that $T|_{\mathbb{T}_{Abs}} = T \circ S$:

$$(T \circ [\tau/\alpha])(\beta) = \begin{cases} T(\tau) & \text{if } \beta = \alpha, \\ T(\beta) & \text{otherwise.} \end{cases}$$

and, as $[\tau/\alpha]$ does not instantiate any exponent schemes, all is set up right by T. Clearly the case $\tau = \alpha$ is identical.

$\sigma = !^p(\sigma')$**:** if τ is not a banged type clearly no unifier is possible, as $=_e$ ignores the exponents but not their presence. Then if $\tau = !^q(\tau')$ an eventual unifier unifies also σ' and τ'. On the converse for a unifier of σ' and τ' the only thing lacking from being a unifier for σ and τ is satisfying the equality between the number of bangs, i.e. it must satisfy also $p = q$. The result is then a most general one because if $T(\sigma) = T(\tau)$ then also $T(\sigma') = T(\tau')$ and necessarily $T(p) = T(q)$, so that by induction hypothesis we get what we desired.

$\sigma = \sigma_1 \multimap \sigma_2$**:** again if τ is neither an implication nor a variable no substitution can satisfy $=_e$, so `fail` is a correct answear. It is easy to see that also in the other cases `fail` is correct or else the answear is indeed a unifier. If on the other hand T is such that $T(\sigma) = T(\tau)$, then we have $T(\sigma_1) = T(\tau_1)$, so that necessarily T is a solution of C_1 and $T|_{\mathbb{T}_{Abs}} = T_1' \circ S_1$, so that $T_1'(S_1(\sigma_2)) = T_1'(S_1(\tau_2))$. Therefore by induction hypothesis T_1' (and thus T, as S_1 has no information on exponents) solves C_2, and $T_1' = T' \circ S_2$, so that $T = T' \circ (S_2 \circ S_1)$.

□

We extend the above algorithm so that it may unify more variables. In order to compute $U(\vec{\sigma})$

we define recursively

$$(C_1, S_1) := U(\sigma_1, \sigma_2),$$
$$(C_{i+1}, S_{i+1}) := U(S_1 \circ \cdots \circ S_i(\sigma_{i+1}), S_1 \circ \cdots \circ S_i(\sigma_{i+2}),$$

and then we define $U(\vec{\sigma}^n) := (C_1 \cup \cdots \cup C_n, S_1 \circ \cdots \circ S_n)$. We extend further U to multiple equations. Given sequences $\vec{\sigma}_j$ we define in a similar manner as before:

$$(C_1, S_1) := U(\vec{\sigma}_1),$$
$$(C_{i+1}, S_{i+1}) := U(\overrightarrow{S_1 \circ \cdots \circ S_i(\sigma_{i+1})}),$$

and then we denote by $U(\vec{\sigma}_1 \mid \cdots \mid \vec{\sigma}_m)$, which unifies all the sequences separately.

Now we can design a function that assigns the equivalent of principal pairs to terms in *Abs*.

Definition 4.3.9 (principal triples). Given a term M we call a *principal triple* the object (C, A, τ) where C is a set of linear constraints on literals A a *type scheme environment*, i.e. a function $A : \mathcal{V} \to \mathbb{T}_{Abs}$ with finite domain, and $\tau \in \mathbb{T}_{Abs}$, if the two following properties hold:

- for every T that solves C, we have $T(A) \vdash_{Abs} M : T(\tau)$, where $T(A)$ is defined by $T(A)(x) := T(A(x))$;
- if $B \vdash_{Abs} M : \sigma$, then there exist T scheme substitution such that T solves C, $T(A) \subseteq B$ and $\sigma = T(\tau)$.

Proposition 4.3.10. *There is an algorithm ptr that taken as input an abstract term M outputs* `fail` *if M is not typable in Abs, otherwise it gives a principal triple for M.*

Proof. The algorithm is an extension for the last one given for **S** in subsection 2.2.5, based on the fact that *Abs* is syntax directed. Let *ptr* be the following algorithm, where for the sake of brevity we adopt the convention that each time one of the functions called returns `fail` then the whole algorithm returns `fail` (as if by an exception call):

Require: a term M in Abs^{EA};

1: **if** $M = x$ **then return** $(\emptyset, \{x : \alpha\}, \alpha)$;
2: **else**
3: **if** $M = \lambda x.M'$ **then**
4: $(C', A', \tau') \leftarrow ptr(M')$;
5: **if** $x \in \mathrm{DOM}(A')$ **then return** $(C', A' \setminus \{x : A'(x)\}, A'(x) \multimap \tau')$;
6: **else**
7: choose α fresh;
8: **return** $(C', A', \alpha \multimap \tau')$;

9: **else**

10: **if** $M = M_1\, M_2$ **then**

11: $(C_1, A_1, \tau_1) \leftarrow pp'(M_1)$;

12: $(C_2, A_2, \tau_2) \leftarrow pp'(M_2)$;

13: in (C_1, A_1, τ_1) rename variables so that type variables and literals do not appear also in (C_2, A_2, τ_2);

14: choose α fresh;

15: $(C_3, S) \leftarrow U(\tau_1, \tau_2 \multimap \alpha)$; **return** $(C_1 \cup C_2 \cup C_3, S(A_1 \cup A_2), S(\alpha))$;

16: **else**

17: **if** $M = \{N\}_{\overrightarrow{P \to x^n}}$ **then**

18: **for** $1 \leq i \leq n$ **do**

19: $(C_i, A_i, \tau_i) \leftarrow ptr(P_i)$;

20: in (C_i, A_i, τ_i) rename variables so that type variables and literals do not appear also in (C_j, A_j, τ_j) for all $j < i$;

21: $(C', A', \tau') \leftarrow ptr(N)$;

22: in (C', A', τ') rename variables so that type variables and literals do not appear also in (C_i, A_i, τ_i) for all i;

23: choose $\vec{\alpha}^n$ and $\vec{m}^n$ fresh;

24: $(C'', S) \leftarrow U(\overrightarrow{A'(x_1)}, \tau_1, !^{m_1}\alpha_1 \mid \cdots \mid \overrightarrow{A'(x_n)}, \tau_n, !^{m_1}\alpha_n)$;

25: **return** $(C' \cup C'' \cup \bigcup_i C_i, S(A' \setminus \{\overrightarrow{\overrightarrow{x : A'(x)}}\} \cup \bigcup_i A_i), S(\tau'))$;

26: **else**

27: necessarily $M = (\nabla N)\{\overrightarrow{P/x^n}\}$;

28: **for** $1 \leq i \leq n$ **do**

29: $(C_i, A_i, \tau_i) \leftarrow ptr(P_i)$;

30: in (C_i, A_i, τ_i) rename variables so that type variables and literals do not appear also in (C_j, A_j, τ_j) for all $j < i$;

31: $(C', A', \tau') \leftarrow ptr(N)$;

32: in (C', A', τ') rename variables so that type variables and literals do not appear also in (C_i, A_i, τ_i) for all i;

33: choose m fresh;

34: $(C'', S) \leftarrow U(!^m A'(x_1), \tau_1 \mid \cdots \mid !^m A'(x_n), \tau_n)$;

35: **return** $(C' \cup C'' \cup \bigcup_i C_i, S(\bigcup_i A_i), !^m S(\tau'))$;

Let's see the two points of the definition of principal triple by induction. Suppose $ptr(M) = (C, A, \tau)$. In the first point we will always suppose T is such that T solves C, and we will need to show that $T(A) \vdash_{Abs} M : T(\tau)$. In the second one we will suppose $B \vdash_{Abs} M : \sigma$ and we will have

to show that there exist T solution to C and such that $T(A) \subseteq B$ and $T(\tau) = \sigma$. Also the case in which `fail` is given will be discussed.

$M = x$**:** The first point is given by a simple (var); the other point is shown by stripping the derivation from all weakenings down to a single (var) yielding $x : \sigma \vdash x : \sigma$, and then choosing $T = [\sigma/\alpha]$.

$M = \lambda x.M'$**:** Untypability of M' implies untypability of M. By induction hypothesis (as $C = C'$) $T(A') \vdash_{Abs} M' : T(\tau')$. Now if $x \in \mathrm{DOM}(A')$ we derive with an (abs) $T(A) \vdash \lambda x.M' : T(A'(x)) \multimap T(\tau')$, and in fact $T(A'(x)) \multimap T(\tau') = T(\tau)$. Otherwise we introduce $x : \alpha$ by weakening and then abstract it away to get the same result.
For the second point if we go up the derivation we necessarily have $\sigma = \sigma_1 \multimap \sigma_2$ and $x : \sigma_1, B \vdash_{Abs} M' : \sigma_2$. So by induction hypothesis there is T that solves $C' = C$ and such that $T(A') \subseteq B \cup \{x : \sigma_1\}$ and $T(\tau') = T(\sigma_2)$. If $x \in \mathrm{DOM}(A')$ then $T(\tau) = T(A'(x)) \multimap T(\tau') = \sigma_1 \multimap \sigma_2$, and clearly $T(A) \subseteq B$. Otherwise take T' such that $T'(\alpha) = \sigma_1$ and such that it is equal to T on all other variables. As $A = A'$, $x \notin \mathrm{DOM}(A)$ and $\alpha \notin \mathrm{FTV}(A)$ we automatically have $T'(A) \subseteq B$. Clearly also $T'(\tau) = \sigma$ holds.

$M = M_1\, M_2$**:** Untypability of any of the two implies untypability of the whole term. The other case in which `fail` is given is when the types τ_1 and $\tau_2 \multimap \alpha$ are not unifyable: the correctness of this comes from the second point of the induction hypothesis.
Let's get back to the first one. By induction hypothesis (as T solves both C_1 and C_2, and so also $T \circ S$ does) we have that $T(S(A_i)) \vdash_{Abs} M_1 : T(S(\tau_i))$ for $i = 1, 2$. As T solves C_3 by definition of unifier we have that

$$T(S(\tau_1)) = T(S(\tau_2 \multimap \alpha)) = T(S(\tau_2)) \multimap T(S(\alpha))$$

so that we can combine the two by (app), and obtain a derivation of $T(A) \vdash_{Abs} M : T(\tau)$.
For the other point we climb the derivation to the two subderivations giving

$$B_1 \vdash_{Abs} M_1 : \sigma' \multimap \sigma, \qquad B_2 \vdash_{Abs} M_2 : \sigma'.$$

By induction hypothesis we have T_i such that $T_i(A_i) \subseteq B_i$, with $T_1(\tau_1) = \sigma' \multimap \sigma$ and $T_2(\tau_2) = \sigma'$. Define T to be equal to T_i when dealing with variables (both type variables and literals) appearing in (C_i, A_i, τ_i), equal to σ on α and the identity on all the other ones. As the two sets of variables have been made disjoint and α is fresh such definition is possible. So T solves both C_1 and C_2, and as $T(\tau_2 \multimap \alpha) = \sigma' \multimap \sigma = T(\tau_1)$ by definition of most general unifier T solves also C_3 and there is T' such that $T|_{\mathbb{T}_{Abs}} = T' \circ S$. So

$$T'(A) = T'(S(A_1 \cup A_2)) = T(A_1 \cup A_2) = T_1(A_1) \cup T_2(A_2) \subseteq B_1 \cup B_2$$

and $T'(\tau) = T'(S(\alpha)) = T(\alpha) = \sigma$.

$M = \{N\}_{P \to \vec{x}}$**:** It is necessary that all the subterms involved are typable for the term to be in turn typable. And again applying the second point of induction hypothesis yields that it is right for the algorithm to output `fail` when the unifying function fails.
If T is as in the hypothesis then $T \circ S$ solves C' and C_i for every i, so that $T(S(A')) \vdash_{Abs} N : T(S(\tau))$ and $T(S(A_i)) \vdash_{Abs} P_i : T(S(\tau_i))$. T also solves C'': if we fix i, and say $\vec{x}_i = x_i^1, \dots, x_i^{k_i}$ by definition of unifier we have that $T(S(A'(x_i^j)))$ is constantly equal to $T(S(\sigma_i))$ which in turn is equal to $T(S(!^{m_i}(\alpha_i))) = !^{T(M_i)}(T(S(\alpha)))$. So for a fixed i there is a certain fixed number of bangs (namely $T(m_i)$) in front of the types $T(S(A'(x_i^j)))$ which are all equal, so we can contract these variables and plug into them the derivation for P_i getting what we needed.
The last rule of the derivation must be a (con), so that we have subderivations for

$$\overrightarrow{B \vdash P : !^n \rho}, \qquad \overrightarrow{\vec{x} : !^n \rho}, B' \vdash N : \sigma,$$

where we are supposing ρ is not a bang-type, and $n_i < 0$ for all i, and we have $B = B' \setminus \{\overrightarrow{\vec{x} : !^n \rho}\} \cup \bigcup_i B_i$. By induction hypothesis we have T_i such that T_i solves C_i, $T_i(A_i) \subseteq B_i$ and $T_i(\tau_i) = !^{n_i}\rho_i$, and T' such that T' solves C', $T'(A') \subseteq B'$ and $T'(\tau') = \sigma$. Now define T so that it has the values of T_i and of T' when dealing with the respective variables in the principal triple, and has values ρ_i and n_i on α_i and m_i respectively. Again it is possible because the variables are rendered disjoint. Now for each fixed i we have

$$\begin{aligned} T(A'(x_i^j)) = T'(A'(x_i^j)) &= !^{n_i}\rho \quad \text{for all } j, \\ T(\tau_i) = T_i(\tau_i) &= !^{n_i}\rho, \\ T(!^{m_i}\alpha_i) = !^{T(m_i)}T(\alpha_i) &= !^{n_i}\rho. \end{aligned}$$

So by definition of most general unifier T solves C'' and there is T'' such that $T|_{\mathbb{T}_{Abs}} = T'' \circ S$. T'' solves the same linear constraints as T, so that it solves C, and moreover

$$T''(S(A' \setminus \{\overrightarrow{\overrightarrow{x : A'(x)}}\} \cup \bigcup_i A_i)) = T'(A') \setminus \{\overrightarrow{\vec{x} : !^n \rho}\} \cup \bigcup_i T_i(A_i) \subseteq B' \setminus \{\overrightarrow{\vec{x} : !^n \rho}\} \cup \bigcup_i B_i = B,$$

and $T''(S(\tau')) = T'(\tau') = \sigma$.

$M = (\nabla N)\{\overrightarrow{P/x}\}$**:** This case is handled basically in the same way as the preceding one.

□

Now we have to relate this typing to the pure terms.

Proposition 4.3.11. *Given a pure term M let $C(M)$ be the following set:*

$$C(M) = \{\, N \in \Lambda^{Abs} \mid N = \mathcal{C}(R), \quad R \text{ simple}, \quad (R)^+ = M \,\}.$$

There is an algorithm ct (which stands for "canonical typable") that given a pure term M gives as output a set such that

- $ct(M) \subseteq C(M)$;
- *if $N \in C(M)$ and N is typable in Abs then $N \in ct(M)$.*

Proof. Let $\mathbb{L}$ be the (computable) function that linearizes the free variables of a term, in the sense that it replaces each occurrence of a variable occurring more than once with a different fresh variable. Let $\mathbb{L}_x(M)$ be the set of variables substituted for x in its linearization. So for example if $M = (x\,(y\,(x\,(y\,z))))$ then $\mathbb{L}(M) = (x_1\,(y_1\,(x_2\,(y_2\,z))))$, and $\mathbb{L}_x(M) = \{\, x_1, x_2 \,\}$. Let's denote by $x \in_{>1} \mathrm{FV}(M)$ the property of occurring free more than once in M, and if X and Y are set of terms let's denote

$$\lambda x.(X) := \{\, \lambda x.M \mid M \in X \,\}, \qquad X \mathbin{@} Y := \{\, (M\,N) \mid M \in X, \quad N \in Y \,\}.$$

Similarly we define $\{X\}_{x \to \vec{y}}$ and $(\nabla X)\{\overrightarrow{Y/x}\}$. Note that all this operations, though they can greatly increase the size of the sets, still preserve their finiteness.

First we define this supporting procedures: *bl*, which stands for "build linear", and *bb*, for "build boxes". *bb* has an auxiliary variant *bb′*, used to distinguish between the building of boxes that plug in only variables from the ones that plugs in non trivial terms. *bl* is the following algorithm:

Require: M pure linearized term;

1: **if** $M = x$ **then return** $\{x\}$;
2: **else**
3: **if** $M = \lambda x.M'$ **then**
4: **if** $x \in_{>1} \mathrm{FV}(M')$ **then return** $\lambda x.\{bl(\mathbb{L}(M')) \cup bb(\mathbb{L}(M'))\}_{x \to \mathbb{L}_x(M')}$;
5: **else return** $\lambda x.(bl(M') \cup bb(M'))$;
6: **else**
7: necessarily $M = M_1\,M_2$; **return** $bl(M_1) \mathbin{@} (bl(M_1) \cup bb(M_2))$;

bb instead is

Require: M pure linearized term;

1: **if** $M = x$ **then return** $\emptyset$;
2: **else**
3: $\vec{x}^n \leftarrow \mathrm{FV}(M)$;
4: choose $\vec{y}^n$ fresh; **return** $bb'(M) \cup \Big(\nabla\big(bl(M[\overrightarrow{y/x}]) \cup bb'(M[\overrightarrow{y/x}])\big)\Big)\,\{\overrightarrow{x/y}\}$;

By *free subterm* we intend a subterm whose free variables are not later bounded. Basically N is a free subterm of M if we can write $M = P[N/x]$ for some P and x occurring only once in P, without the need of a context. Let fs be an algorithm that accepts as input a pure term M and by each subsequent call it returns M written in the form $P[\overrightarrow{Q/y}]$, defined by all the possible non-empty sequences of free disjoint subterms $\vec{Q}$ such that every Q_i is an application, P is not a variable and y_i occurs only once in P. When all such possibilities are depleted it returns `fail`. Clearly the possibilities are finite. Finally bb', the more complex of the three auxiliary algorithms, is the following one. Note that it cannot be called on a variable.

Require: M pure linearized term;

1: $X \leftarrow \emptyset$;
2: **while** $P[\overrightarrow{Q/y}^n] \leftarrow fs(M)$ is not `fail` **do**
3: $\quad \vec{x}^m \leftarrow \mathrm{FV}(P) \setminus \vec{y}$;
4: $\quad$ choose $\vec{z}^n$ and $\vec{w}^m$ fresh;
5: $\quad X \leftarrow X \cup \Big(\nabla\big(bl(P[\overrightarrow{z/y}, \overrightarrow{w/x}]) \cup bb'(P[\overrightarrow{z/y}, \overrightarrow{w/x}])\big)\Big)\, \{\overrightarrow{bl(Q)/y}, \overrightarrow{x/w}\}$;

All these algorithms make heavy use of the restrictions we have on canonical forms of simple terms as they are described in lemma 4.3.5. The last thing to do is design all the possible contractions when first evaluating a term. In the end the main algorithm $C(M)$ is simply defined by the following pseudo-code.

Require: M pure term;

1: $\vec{x} \leftarrow \{\, x \in \mathrm{FV}(M) \mid x \in_{>1} \mathrm{FV}(M) \,\}$;
2: **if** $\vec{x} = \emptyset$ **then return** $bl(M) \cup bb(M)$;
3: **else return** $\{bl(\mathbb{L}(M)) \cup bb(\mathbb{L}(M))\}_{x_1 \to \mathbb{L}_{x_1}(M), \ldots, x_n \to \mathbb{L}_{x_n}(M)}$;

So by inspection of the algorithms, confronting with the properties listed in 4.3.5, we get the two properties. □

Finally we can say how we can have all and only the simple typings for a pure term.

Proposition 4.3.12. *Fix a pure term M.*

If $N \in ct(M)$ and $ptr(N) = (C, A, \tau)$ then for every T scheme substitution that solves C we have $T(A) \vdash_{\mathbf{EAL}} M : T(\tau)$.

If on the converse $B \vdash M : \sigma$ is derivable with a simple typing then there exists a term N in $ct(M)$ such that if we denote $(C, A, \tau) = ptr(N)$ there exists a scheme substitution T that solves C, such that $T(A) \subseteq B$ and $T(\tau) = \sigma$.

In particular M is simply typable if and only if there exists $N \in ct(M)$ for which there exists a principal triple with a solvable system.

4.3.2 Type inference for DLAL

A type inference algorithm has been developed for propositional **LAL** in [Bai04], using ideas similar to the ones described for **EAL**. However the constraints one has to solve are not linear equalities, but equations on words which are hard to solve.

This is based on the fact that ! and § can occur mixed up in the types. This problem is overcome with **DLAL**: if we read this system as a restriction of **LAL**, we see that practically ! is used exclusively for handling eventual duplications, while § is kept to ensure stratification. The two processes can be separated: the proposal of algorithm in [BT04] works by two stages. The first one scans a simple type derivation to put eventual !, i.e. to see were (i-app) and (i-abs) are needed. The second one uses the type inference algorithm designed for **EAL** to extract the typings compatible with **DLAL**, using the fact that all **DLAL** typings can be translated in **EAL** ones. Note that **DLAL** has built in the system the condition of contracting only variables. We briefly expose this algorithm.

Let's see one step at a time.

Definition 4.3.13. *Basic abstract type schemes* $\mathbb{T}_B$ will here denote are built from $\mathbb{V}$ by the grammar

$$\mathbb{T}_B ::= \mathbb{V} \mid \mathbb{T}_! \to \mathbb{T}_B,$$

where in turn $\mathbb{T}_!$ are the *bang abstract type schemes* built by the grammar

$$\mathbb{T}_! ::= \Lambda_{\mathbb{B}} \mathbb{T}_B.$$

$\Lambda_{\mathbb{B}}$ is the set of disjunctions of boolean parameters

$$\Lambda_{\mathbb{B}} := \{\, a_1 \vee \cdots \vee a_n \mid n \geq 0, \quad a_i \in \mathcal{V}_{\mathbb{B}} \,\},$$

where $\mathcal{V}_{\mathbb{B}}$ is a countable set of literals intended to represent boolean values.

We can abbreviate $a_1 \vee \cdots \vee a_n$ by $\vec{a}$. We call an *interpretation* a function $\Phi : \mathcal{V}_{\mathbb{B}} \to \mathbb{B}$ extended to encompass all $\Lambda_{\mathbb{B}}$ by $\Phi(\vec{a}) = (\exists i : \Phi(a_i))$ seen as boolean values.

Abstract environments will here mean $A : \mathcal{V} \to \mathbb{T}_!$, and abstract sequents are $A \vdash M : \tau$ where A is an abstract environment and τ belongs to the basic abstract types. aA denotes the environment defined by $(aA)(x) := a(A(x))$.

Now the basic idea is to decorate each application and abstraction with a parameter that when instantiated will tell whether it is linear or intuitionistic. So given a pure term M take the typing $\mathcal{D}$ resulting from applying *pp* described in subsection 2.2.5. We place in this derivation (weak) and (con) rules: due to our knowledge of canonical derivations in **DLAL** (see proposition 4.2.43), we can automatically place them before the abstraction that needs them or at the end of the whole

derivation. In fact no (weak) is placed at the end as *pp* does not give superfluous variables. Now from a $\mathcal{D}$ such treated we build a sort of *abstract* derivation where (app) and (abs) are decorated with a parameter in $\mathcal{V}_{\mathbb{B}}$.

Definition 4.3.14 (maximal decoration). We decorate simple types turning them into basic abstract ones by

$$\hat{\alpha} := \alpha\widehat{\sigma \to \tau} \qquad\qquad := a\hat{\sigma} \to \hat{\tau},$$

where a is each time a fresh parameter.

We design a special unifier for abstract types. As we will work on derivations already obtained in the simple case no unifying of the implicational structure of the types will be needed. So only a set C of constraints of the form $\vec{a} = \vec{b}$ with $\vec{a}, \vec{b} \in \Lambda_{\mathbb{B}}$ is given as output.

So let U be the following algorithm.

Require: σ and τ two bang abstract types with the same implicational structure;

1: **if** $\sigma = \vec{a}\alpha$ and $\tau = \vec{b}\alpha$ **then return** $\{\vec{a} = \vec{b}\}$;

2: **else**

3: necessarily $\sigma = \vec{a}(\sigma_1 \to \sigma_2)$ and $\tau = \vec{b}(\tau_1 \to \tau_2)$; **return** $\{\vec{a} = \vec{b}\} \cup U(\sigma_1, \tau_1) \cup U(\sigma_2, \tau_2)$;

We turn $\mathcal{D}$ into an abstract derivation $\hat{\mathcal{D}}$ where in the final sequent all types are obtained by some decoration of the simple ones. We proceed by induction on the last rule of $\mathcal{D}$, building in the meanwhile a set $C(\mathcal{D})$ of equations on $\Lambda_{\mathbb{B}}$.

(var): $x : \sigma \vdash x : \sigma$ becomes $x : \hat{\sigma} \vdash x : \hat{\sigma}$, and $C(\mathcal{D}) = \emptyset$.

(abs): We have $M = \lambda x.M'$ and the subderivation $\mathcal{D}'$ that types M'. By induction we get a derivation $\hat{\mathcal{D}}' \rightsquigarrow x : \sigma, A \vdash M' : \tau$ with A, τ and σ abstract, and we also have $C(\mathcal{D}')$. We choose a fresh parameter a: if $x \in {>}\,1\,\mathrm{FV}(M')$ then we define $C(\mathcal{D}) := C(\mathcal{D}') \cup \{a = \mathtt{true}\}$, otherwise we leave $C(\mathcal{D}) := C(\mathcal{D}')$. In any case $\hat{\mathcal{D}}$ ends with

$$\dfrac{\begin{array}{c}\hat{\mathcal{D}}' \\ \vdots \\ A, x : \sigma \vdash M' : \tau\end{array}}{A \vdash M' : a\sigma \to \tau}\ (a\text{-abs})$$

(app): We have $M = M_1\, M_2$ and the corresponding two subderivations $\mathcal{D}_1$ and $\mathcal{D}_2$. So by induction hypothesis we have $\hat{\mathcal{D}}_1 \rightsquigarrow A_1 \vdash M_1 : \sigma_1 \to \tau$ and $\hat{\mathcal{D}}_2 \rightsquigarrow A_2 \vdash M_2 : \sigma_2$, and the constraints $C(\mathcal{D}_1)$ and $C(\mathcal{D}_2)$. σ_1 and σ_2 have the same implicational structure but are not necessarily equal in the boolean parameters. First we rename all parameters so that

they are disjoint in the two derivations. Then we choose a fresh parameter a, and define $C(\mathcal{D}) := C(\mathcal{D}_1) \cup C(\mathcal{D}_2) \cup U(\sigma_1, a\sigma_2)$, and $\hat{\mathcal{D}}$ is

$$\dfrac{\begin{array}{cc}\hat{\mathcal{D}}_1 & \hat{\mathcal{D}}_2 \\ \vdots & \vdots \\ A_1 \vdash M_1 : \sigma_1 \to \tau & A_2 \vdash M_2 : \sigma_2\end{array}}{A_1, aA_2 \vdash M_1\, M_2 : \tau}\ (a\text{-app})$$

(con): We have $M = M'[x/\vec{y}]$, and a subderivation $\mathcal{D}'$, which by induction hypothesis yields $\hat{\mathcal{D}}' \rightsquigarrow \overrightarrow{y:\sigma}, A \vdash M' : \tau$. The σ_is have all the same structure but different parameters. We leave $C(\mathcal{D}) := C(\mathcal{D}')$ and the derivation $\hat{\mathcal{D}}$ is

$$\dfrac{\begin{array}{c}\hat{\mathcal{D}}' \\ \vdots \\ \overrightarrow{y:\sigma}, A \vdash M' : \tau\end{array}}{x : m(\vec{\sigma}), A \vdash M'[x/\vec{y}] : \tau}\ (\text{con})$$

where m ("merge") is defined by

$$\begin{aligned} m(\vec{a}\alpha, \vec{b}\alpha) &:= \vec{a}\vec{b}\alpha, \\ m(\vec{a}(\sigma_1 \to \sigma_2), \vec{b}(\tau_1 \to \tau_2)) &:= \vec{a}\vec{b}(m(\sigma_1, \tau_1) \to m(\sigma_2, \tau_2)), \\ m(\overrightarrow{\sigma}, \tau) &:= m(m(\overrightarrow{\sigma}), \tau). \end{aligned}$$

Note that the case of contraction does not pose any additional conditions. In fact this phase is interested in just distinguishing intuitionistic and linear applications and abstractions in a plausible way. All the rest is left to phase two.

(weak): The set of constraints is directly inherited and the weakening is turned into one on the maximal decoration of the type.

We then can find all the possible solutions of $C(\mathcal{D})$, i.e. interpretations Φ such that $\Phi(\vec{a}) = \Phi(\vec{b})$ for every $\vec{a} = \vec{b} \in C$: this is possible as the number of parameters is finite. At least one exists: for example $\Phi \equiv \texttt{true}$. For each of this solutions Φ (which are finite) we get a so called *!-derivation* $\mathcal{D}_\Phi$ where in $\hat{\mathcal{D}}$ we have labeled as (i-abs) and (i-app) the (a-abs) and (a-app) with $\Phi(a) = \texttt{true}$ and (l-abs) and (l-app) otherwise. We now have that each **DLAL** derivation $\mathcal{D}$ is such that applying to it the forgetful function we get a derivation $\mathcal{D}'$ such that there exists Φ that solves $C(\mathcal{D}')$ and $\mathcal{D} = \mathcal{D}'_\Phi$ up to the structure of the rules.

Another filter is now given by checking on each of these derivations if the following two conditions hold:

1. in $\mathcal{D}_\Phi$ the right premise of each (i-app) has an environment of at most one assumption;

2. the eventual only variable in the environment of a right premise appears only once in the term on the right of $\vdash$.

Clearly these conditions are necessary for the derivation to be turned into a **DLAL** one.

The next phase is to apply to the term the algorithm $ct(M)$ described in the previous section for **EAL**. With two further steps we can filter out from $ct(M)$ the canonical terms not corresponding to a possible **DLAL** derivation. We are comparing the two on the base of the translation of **DLAL** to **LAL**, which in turn can be injected in **EAL**. We work separately with each $\mathcal{D}_\Phi$ obtained in the previous stage.

Using $\mathcal{D}_\Phi$ we draw a syntactic tree with boxes drawn around each right premise of $(i-app)$. This means a !-promotion is needed there if we look at the derivation in **LAL**. Similarly we draw a syntactical tree with boxes for each term in $ct(M)$: a box is put around each of the boxed subterms. Let T be the tree obtained from $\mathcal{D}_\Phi$ and T_i the trees obtained from $ct(M)$. We confront each T_i with T and we leave it out if it does not satisfy the following conditions:

- each box of T has a corresponding one in T_i, and in particular (as boxes in T are all with at most a single variable in the environment) the corresponding box cannot have a complex term plugged in it;
- for each box B of T that has a variable x in the environment (and we know it must be the only one) there is no box in T_i that contains B and not an abstraction on x.

The second condition is due to the fact that in a **DLAL** derivation such an x shifts automatically to the intuitionistic side. So no further boxes (be they ! ones that prelude to an (i-app) or S ones given by a promotion) can be drawn until x disappears form the environment, which can happen only when it is abstracted away.

Thus we obtain $\widehat{ct}(M)$, a subset of $ct(M)$. To each Λ^{Abs}-term N in it we apply the algorithm *ptr* that finds a triple (C, A, τ) as described in the previous section. Let $B_1, \dots B_k$ be all the boxes in the syntactical tree of N that corresponds to boxes in $\mathcal{D}_\Phi$. Suppose now that the scheme substitution T is a solution of C: then there is a valid **EAL** derivation $\mathcal{D}'$ typing N. In it each of the boxes B_i corresponds to n_i nested promotions, with $n_i > 0$. These consecutive promotions (which are made on a term with just one variable occurring only once) are translated into $n_i - 1$ (possibly none) **DLAL** promotions. The last one is implicit in (i-app). All the other promotions are turned into **DLAL** ones. The fact that they can be applied is seen by shifting to the intuitionistic side the least as possible. So given n nested promotions, $n-1$ of them are done without shifting any variable. In the last one for each assumption we check down the derivation if it is needed intuitionistic in a contraction or an (i-abs) before another promotion is encountered: if it is the case we shift it to the intuitionistic side. However such a case means that the assumption will

disappear before any other promotion is needed: in the first case because a contraction is either followed immediately by an abstraction or else it is the last part of the derivation, in the second one because it is directly abstracted away. So promotions are always possible, i.e. there are never assumptions in the intuitionistic side that prevent them from happening. So each term in $\widehat{ct}(M)$ together with a solution to the corresponding set of linear constraints induces a **DLAL** typing for M.

On the converse we have already said that each **DLAL** typing $\mathcal{D}$ is of the form $\mathcal{D}'_\Phi$ for some abstract derivation $\mathcal{D}'$ and some solution Φ to the set of boolean constraints induced by $\mathcal{D}'$. On the other hand by injection in **EAL** we see that $\mathcal{D}$ is also a valid **EAL** simple typing. It is simple because **DLAL** permits contraction only on variables, and does not allow plugging into contracted assumptions. So by the results on ct there must be $N \in ct(M)$ with $(N)^+ = M$ and N typable in *Abs*. This N however comes from a **DLAL** derivation, so it must respect the properties set for $\widehat{ct}(M)$.

So the algorithm we have outlined gives a way to find all the possible typings for M in propositional **DLAL**.

4.3.3 Typing in polymorphic light logic

As we have seen in the representation theorem we have used polymorphism heavily. We cannot expect much expressiveness from a system without second order quantifying. However it poses a drawback: we have already seen as the problem of type checking and typability is undecidable in system **F**.

Regarding type checking, we can easily adapt the proof of undecidability for **F** (see 3.4.1) to **EAL**2 and **LAL**2.

Theorem 4.3.15 ($\mathrm{SUP} \leq \mathrm{TC}_{\mathbf{DLAL2}}, \mathrm{SUP} \leq \mathrm{TC}_{\mathbf{LAL2}}, \mathrm{SUP} \leq \mathrm{TC}_{\mathbf{EAL2}}, \mathrm{SUP} \leq \mathrm{TC}_{\mathbf{AL2}}$)**.** SUP *with two pairs is reducible to* TC *in all of the four systems* **AL**2*,* **EAL**2*,* **LAL**2*,* **DLAL**2*.*

Proof. Basically we follow the same thread of the proof of the analogous proposition 3.4.3. Let $\Gamma = (\sigma_1, \tau_1), (\sigma_2, \tau_2)$ be any instance of SUP with two pairs. We will build a unique instance for $\mathrm{TC}_{\mathbf{DLAL2}}$. Then we will inject it in **LAL**2, then into **EAL**2, and from there into **AL** and finally with the forgetful function we will return to the same instance we have used for system **F**. We can assume without problems that the arrows in the SUP instance are linear ones instead of intuitionistic.

We take the same term $M = b\,(\lambda x.c\,x\,x)$. Take the **DLAL**2 environment

$$A_1 := ; b : \forall\gamma.(\gamma \to \S\gamma) \multimap \beta,\ c : \S\forall.(\tau_1 \multimap \delta_1) \multimap (\delta_2 \multimap \tau_2) \multimap (\sigma_1 \multimap \sigma_2),$$

and define A_2 as the above environment injected in **LAL**2, and A_3 as the environment obtained injecting A_2 in **EAL**2 (and **AL**2), and finally by A the environment obtained with the forgetful function, which is the same environment used in the proof of proposition 3.4.3. We will show that

$$\Gamma \text{ has a solution } \implies A_1 \vdash_{\mathbf{DLAL2}} M : \beta.$$

Then we will have what is needed, as we can complete the chain: the last statement implies that $A_2 \vdash M : \beta$ is derivable in **LAL**2, and $A_3 \vdash M : \beta$ is then derivable both in **EAL**2 and **AL**2, and then $A \vdash_{\mathbf{F}} M : \beta$, which in turn as showed in the proof of proposition 3.4.3 implies that SUP has a solution. So the six statements involved are equivalent.

Suppose we have a solution S of Γ, together with S_1 and S_2 such that $S_i(S(\sigma_i)) = S(\tau_i)$. Now we will basically take the derivation already depicted for system **F** and decorate it with the necessary modalities. For brevity denote by

$$\rho := \forall.(\tau_1 \multimap \delta_1) \multimap (\delta_2 \multimap \tau_2) \multimap (\sigma_1 \multimap \sigma_2),$$

the type of c in A_1 without the leading modality. φ is the type $S(\sigma_1) \multimap S(\sigma_2)$, so that

$$S_1(\varphi) = S(\tau_1) \multimap S_1(S(\sigma_2)), \qquad S_2(\varphi) = S_2(S(\sigma_1)) \multimap S(\tau_2).$$

Then we have the following derivation:

$$\dfrac{\dfrac{\dfrac{}{;b : A(b) \multimap \beta \vdash b : \forall\gamma.(!\gamma \multimap \S\gamma) \multimap \beta}\,(\text{var})}{;b : A(b) \vdash ((\forall.\varphi) \to \S\forall.\varphi) \multimap \beta}\,(\text{ins}) \qquad \begin{array}{c}\mathcal{D}\\ \vdots \\ ;c : A(c) \vdash \lambda x.c\,x\,x : (\forall.\varphi) \to \S\forall.\varphi\end{array}}{A \vdash b\,(\lambda x.c\,x\,x) : \beta}\,(\text{l-app})$$

where $\mathcal{D}$ is the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{;c : \rho \vdash c : \rho}\,(\text{var})}{;c : \rho \vdash c : S_1(\varphi) \multimap S_2(\varphi) \multimap \varphi}\,(\text{ins}) \qquad \dfrac{\dfrac{}{;x_1 : \forall.\varphi \vdash x_1 : \forall.\varphi}\,(\text{var})}{;x_1 : \forall.\varphi \vdash x_1 : S_1(\varphi)}\,(\text{ins})}{;c : \rho, x_1 : \forall.\varphi \vdash c\,x_1 : S_2(\varphi) \multimap \varphi}\,(\text{l-app}) \qquad \dfrac{\dfrac{}{;x_2 : \forall.\varphi \vdash x_2 : \forall.\varphi}\,(\text{var})}{;x_2 : \forall.\varphi \vdash x_2 : S_2(\varphi)}\,(\text{ins})}{;c : \rho,\, x_1, x_2 : \forall.\varphi \vdash c\,x_1\,x_2 : \varphi}\,(\text{l-app})}{;c : \rho,\, x_1, x_2 : \forall.\varphi \vdash c\,x_1\,x_2 : \forall.\varphi}\,(\text{gen})}{x_1, x_2 : \forall.\varphi; c : \S\rho \vdash c\,x_1\,x_2 : \S\forall.\varphi}\,(\text{prom})}{x : \forall.\varphi; c : \S\rho \vdash c\,x\,x : \S\forall.\varphi}\,(\text{con})}{;c : \S\rho \vdash \lambda x.c\,x\,x : (\forall.\varphi) \to \S\forall.\varphi}\,(\text{i-abs})$$

So $\mathrm{TC}_{\mathbf{DLAL2}}$, and type checking for *all* type assignment systems with unrestricted instantiation which find themselves between **DLAL** and **F**, is undecidable. □

Moreover we may see that the proof is again based on the possibility of simply instantiating arrow types, and on the fact that a term like M above is typable without many problems. If we want to circumvent this problem we cannot expect to design a system where M is not typable, because it would signify a heavy loss of expressiveness. Neither suffices one in which some kind of discipline over modalities and quantifiers permitted to be instantiated is set up, because the proof uses just arrow types.

Decidability of typability in these system is up to today an open question. An adaptation of the proof given for system **F** seems out of reach.

A possible breakthrough could follow from the fact that system **F** restricted to rank 2 is decidable. We recall that rank is defined inductively: if we denote with $\mathbb{T}_k$ the set of types of rank k then in system **F** we define

$$\mathbb{T}_0 ::= \mathbb{V} \mid \mathbb{T}_0 \to \mathbb{T}_0, \qquad \text{(the open types)},$$
$$\mathbb{T}_{k+1} ::= \mathbb{T}_k \mid \forall\mathbb{V}.\mathbb{T}_{k+1} \mid \mathbb{T}_k \to \mathbb{T}_{k+1},$$

so basically the rank one less than the maximal depth permitted for quantifiers on a left-going path in the syntactical tree of a type. The types used in the proof of undecidability of TC are of rank 3: more precisely the final type assigned to c is (of the form) $(\forall.\varphi \to \forall.\varphi) \to \beta$ which puts quantifiers at left depth 2. In fact as we already mentioned Wells has proved that both typability and type checking for rank 2 are decidable, while they are both undecidable for every rank greater than 2.

Clearly the definition of rank is extendable to linear logics. The types would include `Int` and functions on integers. However we would have a restriction on the instantiable types that would lessen, but not cancel completely, the power of iteration (based on instantiation of the quantifier in `Int`). The corresponding loss of expressiveness has yet to be investigated. The advantage would be decidability of type checking and typability. Baillot and Terui have proved (though they have yet to publish the result) that the problem of decorating a system **F** derivation into a valid **LAL** one, or else give negative answear is decidable. So in an eventual rank 2 fragment of **LAL** we could search for system **F** derivations and then decorate them to eventually obtain **LAL** ones.

Bibliography

[AC98] Roberto M. Amadio and Pierre-Louis Curien. *Domains and lambda-calculi*. Cambridge University Press, New York, NY, USA, 1998.

[ACM04] Andrea Asperti, Paolo Coppola, and Simone Martini. (Optimal) duplication is not elementary recursive. *Inform. and Comput.*, 193(1):21–56, 2004.

[AM01] Andrea Asperti and Harry G. Mairson. Parallel beta reduction is not elementary recursive. *Inform. and Comput.*, 170(1):49–80, 2001.

[AR02] Andrea Asperti and Luca Roversi. Intuitionistic light affine logic (proof-nets, normalization complexity, expressive power). *ACM Trans. Comput. Log.*, 3(1):1–38 (electronic), 2002.

[Bai04] Patrick Baillot. Type inference for light affine logic via constraints on words. *Theoretical Computer Science*, 2004. 35 pages, to appear.

[Bar92] H. P. Barendregt. Lambda calculi with types. pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[BC92] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 283–293, New York, NY, USA, 1992. ACM Press.

[BT04] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. *CoRR*, cs.LO/0402059, 2004.

[Chu36] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *Amer. J. Math.*, 58:345–363, 1936.

[CLRDR04] Paolo Coppola, Ugo Dal Lago, and Simona Ronchi Della Rocca. Elementary affine logic and the call by value lambda calculus. Accepted for presentation at TLCA'05, 2004.

[CM01] Paolo Coppola and Simone Martini. Typing lambda terms in elementary logic with linear constraints. In *TLCA*, pages 76–90, 2001.

[CRDR03] Paolo Coppola and Simona Ronchi Della Rocca. Principal typing for elementary affine logic. In Hofmann M., editor, *Typed Lambda Calculi and Applications: 6th International Conference (TLCA 2003)*, volume 2701 of *LNCS*, pages 90–104. Springer-Verlag, 2003. A preliminary version has been presented at the workshop "LL—Linear Logic", affiliated to FLoC'02, Copenhagen.

[DJ03] Vincent Danos and Jean-Baptiste Joinet. Linear logic and elementary time. *Inf. Comput.*, 183(1):123–137, 2003.

[FLO83] Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30(1):151–185, 1983.

[Gan80] Robin O. Gandy. Proof of strong normalization. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

[Gir98] Jean-Yves Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.

[GTL89] Jean Y. Girard, Paul Taylor, and Yves Lafont. *Proofs and types.* CUP, Cambridge, 1989.

[Hoo66] Philip K. Hooper. The undecidability of the turing machine immortality problem. *J. Symb. Log.*, 31(2):219–234, 1966.

[Kri90] Jean-Louis Krivine. *Lambda-calcul: types et modèles.* Masson, 1990.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993.

[KW94] Assaf J. Kfoury and Joe B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 196–207, New York, NY, USA, 1994. ACM Press.

[Loa98] Ralph Loader. Notes on simply typed lambda calculus. Technical Report 381, Laboratory for Foundations of Computer Science, 1998.

[MO04] A. S. Murawski and C.-H. L. Ong. On an interpretation of safe recursion in light affine logic. *Theor. Comput. Sci.*, 318(1-2):197–223, 2004.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Sch76] Helmut Schwichtenberg. Definierbare funktionen im λ-kalkül mit typen. *Archiv für Mathematische Logik und Grundlagenforschung*, 17:113–114, 1976.

[Sco93] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121(1-2):411–440, 1993.

[Ter01] Kazushige Terui. Light affine calculus and polytime strong normalization. In *LICS*, 2001.

[Ter02] Kazushige Terui. *Light logic and polynomial time computation.* PhD thesis, Keio University, 2002.

[Wel99] Joe B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.