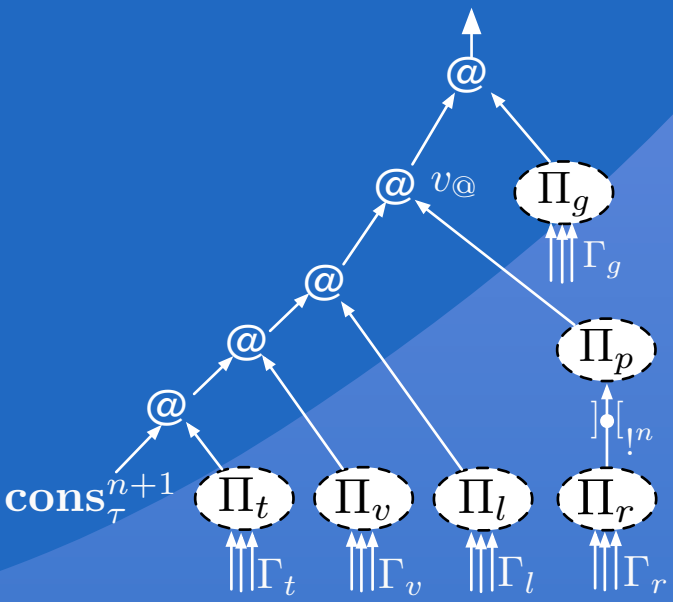
Stefan Schimanski

Polynomial Time Calculi

Dissertation an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München
vorgelegt am 16. Dezember 2008

$$\frac{\Gamma_1 \vdash l^{L^{n+1}(\tau)} \quad x^{\rho} \vdash t^{\Diamond^{n+1} \multimap \tau \multimap \sigma \multimap \sigma} \quad \Gamma_2 \vdash r^{!^n \rho}}{\Gamma_1, \Gamma_2 \vdash (l\, t[x :=]r[_{!^n}])^{\sigma \multimap \sigma}} \; (L^{n+1}(\tau)_1^-)$$



Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie. Detailierte bibliographische Daten sind im Internet über `http://dnb.ddb.de` abrufbar.

Dissertation an der Fakultät für
Mathematik, Informatik und Statistik der
Ludwig-Maximilians-Universität München

1. Berichterstatter: Prof. Dr. Helmut Schwichtenberg
2. Berichterstatter: Prof. Martin Hofmann, Ph.D.
3. Prüfer: Prof. Dr. Otto Forster
4. Prüfer: Prof. Dr. Helmut Zöschinger
Ersatzprüfer: Prof. Dr. Hans-Jürgen Schneider
Externer Gutachter: Ass. Prof. Kazushige Terui, Universität Kyoto

Vorgelegt am: 16. Dezember 2008
Tag des Rigorosums: 23. Februar 2009

ISBN 978-1-4092-7313-4
© 2009 Stefan Schimanski. Alle Rechte vorbehalten.

Abstract

This dissertation deals with type systems which guarantee polynomial time complexity of typed programs. Such algorithms are commonly regarded as being feasible for practical applications, because their runtime grows reasonably fast for bigger inputs. The implicit complexity community has proposed several type systems for polynomial time in the recent years, each with strong, but different structural restrictions on the permissible algorithms which are necessary to control complexity. Comparisons between the various approaches are hard and this has led to a landscape of islands in the literature of expressible algorithms in each calculus, without many known links between them.

This work chooses Light Affine Logic (LAL) and Hofmann's LFPL, both linearly typed, and studies the connections between them. It is shown that the *light iteration* in μLAL, the fixed point variant of LAL, is expressive enough to allow a (non-trivial) compositional embedding of LFPL. The pull-out trick of LAL is identified as a technique to type certain non-size-increasing algorithms in such a way that they can be iterated. The System T sibling LLT$_!$ of LAL is developed which seamlessly integrates this technique as a central feature of the iteration scheme and which is proved again correct and complete for polynomial time. Because LLT$_!$-iterations of the same level cannot be nested, LLT$_!$ is further generalised to LLFPL$_!$, which surprisingly can express the *impredicative iteration* of LFPL and the *light iteration* of LLT$_!$ at the same time. Therefore, it subsumes both systems in one, while still being polynomial time normalisable. Hence, this result gives the first bridge between these two islands of implicit computational complexity.

Diese Dissertation behandelt Typsysteme, welche für getypte Programme polynomielle Komplexität garantieren. Solche Algorithmen werden in der Praxis üblicherweise als effizient angesehen, weil ihre Laufzeit mit wachsender Größe der Eingaben nur mäßig ansteigt. Im Bereich der impliziten Komplexitätstheorie wurden in den letzten Jahren eine Reihe von solchen Typsystemen vorgeschlagen, wobei jedes davon sehr strikte, aber in jedem Fall unterschiedliche strukturelle Anforderungen an typisierbare Algorithmen stellt. Ein Vergleich dieser Ansätze ist schwierig, was inzwischen zu einer Insellandschaft in der Literatur geführt hat, ohne dass man viel über Verbindungen zwischen den Systemen weiß.

Diese Arbeit wählt zwei solcher Typsysteme, nämlich Light Affine Logic (LAL) und Hofmanns LFPL, beide linear getypt, und untersucht, wie diese zusammenhängen. Es wird gezeigt, dass die *leichte Iteration* in μLAL, der Fixpunkt-Variante von LAL, ausdrucksstark genug ist, um LFPL (nichttrivial) kompositional einzubetten. In LAL wird der Trick des „Herausziehens von Abstraktionen" als Technik identifiziert, mit der man in LAL bestimmte Algorithmen so typisieren kann, dass sie iterierbar sind. Mit LLT$_!$ wird eine System-T-Variante von LAL entwickelt, die diese Technik nahtlos in das Iterationsschema integriert, und welche auch wieder normalisierbar ist in polynomieller Zeit und dazu vollständig. In LLT$_!$ können Iterationen auf demselben Level, wie auch in LAL, nicht geschachtelt werden. Daher wird LLT$_!$ weiter verallgemeinert zu LLFPL$_!$, welches überraschenderweise sowohl *imprädikative Iteration* à la LFPL, als auch *leichte Iteration* à la LLT$_!$ erlaubt, und dabei auch wieder korrekt und vollständig für polynomielle Zeit ist. Auf diese Weise baut diese Arbeit eine neue, bisher unbekannte Brücke zwischen zwei Inseln der impliziten Komplexitätstheorie.

Contents

1 Introduction

This thesis is about type systems which guarantee polynomial time complexity of typed programs. A type system is a tool to give a program an additional structure. This structure makes sure that the program satisfies certain properties, e.g. it guarantees that only sensible (well-typed – whatever that may mean) programs can be written.

We propose an intuitive example: take a tool like a nut cracker. You put in a walnut and it cracks it, outputting a cracked walnut. This information can be called its *type*:

$$\text{from walnuts to cracked walnuts}$$

or in a mathematical notation:

$$\text{walnut} \to \text{cracked walnut}.$$

We say that the nut cracker is *typed* because we assigned a type to it.

Now try to plug a coconut into the same nut cracker. It will not fit and probably the tool would not be usable with the coconut. Hence, it is not *type correct* to apply a coconut to a common nut cracker.

On the other hand, take a hazelnut. Put it into the cracker and it works as well. But as we have typed the nut cracker to only take walnuts as input, this application is not type correct either. So maybe our chosen type was not the best one possible because hazelnuts work very well with our nut cracker tool.

In the real-world most programming languages provide a type system. Some of them enforce that every program is completely typed already before it is even run on the computer system. This is called *static typing*. Others type the part of the program at the very moment of its execution, i.e. dynamically when it is run. Hence, this kind is called *dynamic typing*.

From the nut cracker example we have learnt that there can be more than one type for a given program. Moreover, finding a type for a program can be complicated. Somebody who has never seen a nut cracker before most probably will not type it as we did above, at least not if we do not tell him that it is a tool in the context of nuts. Finding a type for a program can be hard or even impossible for a computer in general. There are basically two solutions to this problem:

1. We restrict the type system in such a way that it is very easy to find types. We could restrict our world to walnuts, coconuts and hazelnuts. Then it is easy to find the right type for the nut cracker, maybe even for a computer with some knowledge about nuts, how they look like and how they behave. A human could try out the tool with all nuts and infer the right type.

2. We can give a typing ourselves, or at least some of it. If we tell the computer that the nut cracker produces cracked hazelnuts, it could conclude that it needs also hazelnuts as an input.

The type system can be very simple or rather powerful. How can we express that the nut cracker outputs cracked walnuts if we input walnuts and that it outputs cracked hazelnuts if we input hazelnuts? The type

from hazelnuts or walnuts to cracked hazelnuts or cracked walnuts

certainly does not describe this precisely enough. What we want is merely something like "from hazelnuts to cracked hazelnuts or from walnuts to cracked walnuts". The type system must allow for such constructions, i.e. they must be expressible in the type language. Depending on the context, certain possible types are chosen, together with a set of typing rules. This is essentially what we call a type system: the set of types and the rules to type a program.

In computer science there are many type systems in use. They help the software developer to create usable programs. Though, it is common that programs have flaws which make them not do what they should: they can

crash once in a while or do not do what was meant by the developers. Type systems in real-world languages, like e.g. in Java or C++ certainly help to create programs. But they do not forbid writing those which cannot be considered sensible. In other words, Java's or C++'s type systems are not very strong or strict, but arguably they allow to write and type nearly every kind of algorithm[1].

On the other hand, programming languages (e.g. the Simply Typed Lambda Calculus) are developed in the academic area guaranteeing (proved by mathematical means) that every written program will terminate with a result, i.e. without crashing ever. So, why not use those in practice? Unfortunately, not every algorithm that developers would like to use can be easily written down in these. While the idea of e.g. Simply Typed Lambda Calculus is very elegant and gives the impression that it is the "right way" to type programs, it is not applicable in many cases. But luckily there are extensions of the calculus which are much more powerful, e.g. Gödel's System T, which basically adds constants to the game that allows recursion on natural numbers or lists (compare Section 2.1.1). While much more powerful, System T keeps the property that programs never crash. Moreover, it looks like a programming language which is remotely similar to those used in the industrial world. Conversely, there are features like polymorphism, which appeared in academic calculi (System F, compare Section 2.1.3) more than 30 years before they became common sense in everyday programming in Java 5 just a few years ago, in 2004.

1.1. Why (Polynomial) Complexity Matters

In this work we talk about type systems which do not only guarantee termination (i.e. without crashes and without infinite loops), but also *polynomial time complexity*. In practice, termination alone is not enough. Consider an algorithm which searches for the assignment making a propositional formula with n propositional variables $A_1, \ldots, A_n$ true (called the SAT problem), e.g. $(A_1 \vee A_3) \wedge (\neg A_2 \vee \neg A_3)$. Solving this with an algorithm on the computer might take a certain time, e.g. $1ms$. Taking one variable more into account will double the time (with all known algorithms[2]). With 10 more

[1] We write "nearly" here because this thesis is about intentional expressiveness. Of course, Java and C++ are Turing Complete, but not every algorithm is expressible in a natural formulation.

[2] In fact, the factor 2 can be improved. But in all known algorithms it is still a constant > 1. Which is essentially as bad as 2.

variables it is $2^{10} = 1024$ times as slow. Hence, an easy calculation gives the disillusioning result that with 40 more variables the runtime of the considered algorithm will easily exceed the length of a human life. It is clear that this problem, which is *exponential* according to today's knowledge, is not *feasible* in the usual time spans.

The number of 40 variables is not at all unrealistic. A lot of real-world problems can easily and naturally be broken down to SAT, but often with hundreds or thousands of variables. Even if computers get twice as fast every 18 months (Moore's Law), there is no chance that this number can be handled in the near future. Exponential algorithms are not what we want (for every input that is not tiny).

On the other side of the coin, there are a lot of algorithms which have a polynomial runtime. A look at the growth rate in Figure 1.1 shows that

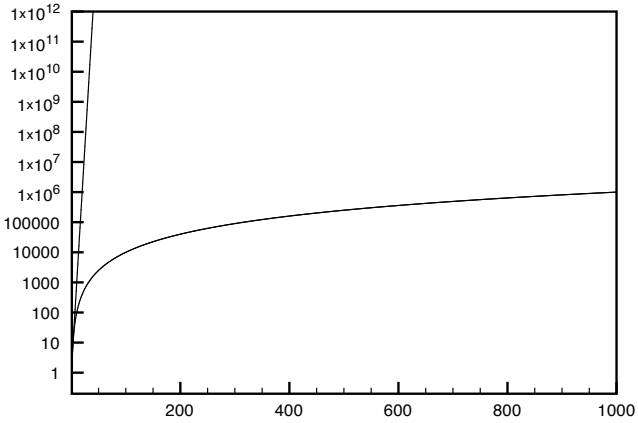


Figure 1.1.: Exponential versus polynomial growth

polynomial time is much more usable because an increase in the input size (the x-axis) results in a reasonable increase of the runtime (the y-axis). While exponential algorithms with input of size n have a runtime of e.g. 2^n processing steps, polynomial algorithms have instead only n^c steps for some constant c. In practice the constant c is often not that big either. The difference between the two worlds is fundamental.

Note that by Moore's Law the available computer power goes up linearly

on the y-axis. On the x-axis you can read then the maximal feasible input for exponential or polynomial algorithms. The exponential function is a straight line. Hence, in order to reach an input size of 100 for instance, we have to go upwards very far, up to 10^{30}.
Examples for polynomial time runtime:

- Sorting of a list of n natural numbers ($\mathcal{O}(n^2)$ "Quick Sort" or even $\mathcal{O}(n \log n)$ "Merge Sort")
- Addition and multiplication of (unary) natural numbers $\leq n$ (linear $\mathcal{O}(n)$, quadratic $\mathcal{O}(n^2)$)
- Addition and multiplication of binary numbers with $\leq n$ digits (linear $\mathcal{O}(n)$, quadratic $\mathcal{O}(n^2)$)
- Multiplication of $n \times n$-matrices ($\mathcal{O}(n^3)$)
- Shortest paths in graphs/maps with n nodes ($\mathcal{O}(n^2)$)
- Network flow optimisation for networks with n nodes ("Ford-Fulkerson", $\mathcal{O}(n^5)$).

Of course, a non-optimal implementation of those algorithms might also lead outside polynomial time. For example the sorting problem "selects" the right sorting for a given list of numbers. There are $n! = n \cdot (n-1) \cdots 2 \cdot 1$ different permutations of n numbers, i.e. even many more than exponentially many. Hence, the algorithms (e.g. Quick Sort) are much faster than a naive search for the right permutation.

What this thought tells us, is that one has to take care to really formulate the algorithm in such a way that the computation *is* polynomial. Even for polynomial problems (i.e. which can be solved in polynomial time) there are bad implementations that get exponential or worse.

The aforementioned type system guaranteed that the coconut was not applied to the nut cracker for wal- and hazelnuts. The typing "from coconuts to cracked coconuts" was not correct because the nut cracker is not able to crack a coconut.

Why should a type system not make sure that bad implementations of algorithms, i.e. those which are exponential, are not type correct?

In the same direction, termination is made sure in Typed Lambda Calculus or System T by the type system. Hence, guaranteeing polynomial time looks like a natural extension of this idea which would give *practically-terminating* (i.e. feasible) algorithms.

In the same way that, for instance, System T *does not talk about explicit measures* or well-founded relations in the typing rules as a witness of the property, type systems for polynomial time complexity should not talk about polynomials in an explicit way. Merely, the type system should force the program to have a certain structure which leads to polynomial runtime. Because no polynomials are directly involved, this area of research is called implicit complexity theory. Though, admittedly, the goal there is to characterise a complexity class, rather than to create a programming language to write algorithms in. Section 1.3 will give an overview of works in this area and the central ideas to restrict the complexity to polynomial time.

> *Remark* 1.1. Of course, for certain applications even polynomial time can be too much. In areas like real-time systems a maximal response time is needed, which might require constant time or maybe logarithmic or linear time algorithms. The same idea of using type systems for guaranteeing these complexity classes could be studied. We will not do that in this thesis, but concentrate completely on polynomial time runtime complexity.

1.2. Type Systems – A Tool to Express Program Properties

Type systems have a long tradition as a tool to guarantee properties of terms. One of the first work in this direction is due to Russel [Rus03, Rus08] from the beginning of the 20th century about methods to exclude the paradox of naive set theory which has become known as Russell's Paradox nowadays. Later, Church [Chu40] added simple types to λ-calculus, the basis for all the calculi considered in this thesis. Tait [Tai67] showed further that those simply typed terms are strongly normalising, i.e. whatever reduction order you choose, you reach the same normal form in the end. Of course, this is essential in order to use a calculus as a predictable programming language. Moreover, every such term terminates, i.e. does not loop forever. Gödel [Göd58] added constructors of natural numbers and recursion to the Simply Typed Lambda Calculus, giving System T (compare Section 2.1.1), still with the same property that every typed term will eventually terminate.

This is all folklore and types are everywhere, especially in computer science and the multitude of programming languages in this area. Even though the properties that those languages enforce using types are far from being as rigorous and clear as in Simply Typed Lambda Calculus or System T. E.g., in Java it is easy to write a program which does not terminate[3], but moreover no type can enforce termination; or another program in C, which just crashes without a proper output. Hence, types have maybe taken a much more pragmatic role in today's programming languages.

In this thesis, we will take a more theoretic position, and study variants of System T and the higher-order λ-calculus System F. This will allow us to prove mathematical theorems about the calculi because their terms, types, typing rules and reductions are formally defined. Still, we will always keep the idea at the back of our mind that we are talking about those calculi from a programmer's perspective.

Our main goal is to study how well one can type algorithms (in their natural formulation, not by some awkward embedding) and which features can be added to the calculi to make the formulation easier, keeping the meta complexity properties of the system.

We are not so much interested in the *traditional idea* of implicit complexity, which is that a correct and complete calculus gives just another definition of a known complexity class. While the presented systems can serve this need as well, this motivation would be in conflict with our desire to extend the flexibility to express more algorithms. Instead, the implicit complexity theorist might aim at the opposite: make the system as small and simple as possible while still being complete for $PTime$. Expressing algorithms other than those needed for a Turing Machine simulation would be secondary.

1.3. Related work – Approaches to Capture Complexity Classes

The idea of implicit complexity has a long tradition. The polynomial time complexity class in particular, has been explored many times and quite a

[3]Non-termination is not a bad property of course because every Turing complete calculus must allow partial functions.

number of systems were proposed. In the following, we give a small survey on the developments.

In parallel to term systems for polynomial time (in short PTime), there is another branch in the literature about *arithmetics* which have exactly the PTime provably recursive functions. The ideas behind those are often very similar to those for the term systems. Moreover, for some there is a direct correspondence (e.g. LHA [Sch06] for LT [SB01] similar to Heyting Arithmetic for System T) because the provably recursive functions in the arithmetic are exactly the terms in the term system. In this work we concentrate only on the latter.

The works reported in the literature on term systems which capture polynomial time can be roughly split into two groups:

1. the systems which implement some kind of predicativity by restricting recursion, iteration or induction (in the case of arithmetic),

2. the systems which use linear/affine typing and some kind of control of duplication.

There are plenty of papers, which are somehow related to these two central ideas, such that the list of systems that we present here will certainly not be exhaustive. We focus on those which are considered most interesting for the aim of this thesis. In Figure 1.2 on page 19 one can find a graphical overview of the landscape of polynomial time calculi with their relations and origins.

The setting In most of the cases shown in the following sections there is a type system with either unary numbers (constructors S and O), binary numbers (constructors S_0, S_1 and 0) or lists (constructors **cons** and **nil**; then binary numbers are lists of booleans). On those data types some kind of primitive recursion or iteration is available, e.g.

$$f(0, \overrightarrow{y}) := g(\overrightarrow{y})$$
$$f(Sn, \overrightarrow{y}) := h(n, \overrightarrow{y}, f(n, \overrightarrow{y}))$$

for recursion and

$$f(0, \overrightarrow{y}) := g(\overrightarrow{y})$$
$$f(Sn, \overrightarrow{y}) := h(\overrightarrow{y}, f(n, \overrightarrow{y}))$$

for iteration (i.e. without the additional argument n applied to h). Moreover, already well-typed functions can be composed in the usual mathematical sense of composition. In the case of binary numbers, sometimes the term recursion on notation is used because the structural recursion

$$f(0, \overrightarrow{y}) := g(\overrightarrow{y}) \tag{1.1}$$
$$f(S_i n, \overrightarrow{y}) := h_i(n, \overrightarrow{y}, f(n, \overrightarrow{y})) \tag{1.2}$$

recurses on the notation of the numbers, i.e. on the stack of S_i constructors.

Instead of this equational definition of terms, it is common to use System T style constants, e.g. the $\boldsymbol{\mathcal{R}}_\tau$ recursion constant with the type

$$\boldsymbol{\mathcal{R}}_\tau : N \to (N \to \tau \to \tau) \to \tau \to \tau$$

for unary numbers. $(\boldsymbol{\mathcal{R}}_\tau\, h\, g\, n)$ then gives the unary recursion from above. In analogy, only the iterator $\mathbf{It}_\tau$ might be available with

$$\mathbf{It}_\tau : N \to (\tau \to \tau) \to \tau \to \tau.$$

In System T the recursion can be simulated using iteration. In calculi which are restricted (e.g. linear) this might not be the case.

Another line of research uses type systems which are derived from *System F*, the polymorphic lambda calculus, without any constants or base types like numbers (compare Section 2.1.3). Instead, numerals use the impredicative Church encoding $N := \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$. This type is the type of an iterator and in fact a Church encoded number is its own iterator.

1.3.1. Predicativity

In the following, we present the calculi which make use of some kind of *predicativity* restriction.

Cobham "bounded recursion on notation" [Cob65] In order to capture Polynomial Time (and not to exceed it), a recursion should not return a value bigger than a (given) polynomial. Hence, Cobham's idea is to bound the result of a recursion with another function $b(x, \overrightarrow{y})$ which is already known to be polynomial. The recursion scheme is called bounded recursion on notation. A function that is defined this way can then be used to bound the next recursion, and so on. A function f, defined by *recursion on*

notation, is bounded by $b(x, \overrightarrow{y})$ (where b is defined in the system already) if $f(x, \overrightarrow{y}) \leq b(x, \overrightarrow{y})$ holds for all input.

To start somewhere, at least one polynomial function is given, e.g. the smash function

$$x\#y := \underbrace{S_0 \cdots S_0 0}_{|x|\cdot|y|},$$

which computes a number whose structural size $|x\#y|$ (which is the number of constructors) is the product of the structural sizes of x and y.

The term *predicativity* comes from the fact that a function f, which is defined by bounded recursion on notation, cannot grow faster than those functions already defined (e.g. by composition of the smash function #).

Is this system implicit in the sense of implicit complexity? There are no explicit polynomials involved. Though, one can argue that the smash function is – by composition with itself – nothing else than the building block of polynomials. Moreover, the bounding function either just cuts off the recursion value, or one has to show outside the system that the $b(x, \overrightarrow{y})$ is really an upper bound. Either case is not satisfactory, because the system essentially does not give purely syntactical rules which lead to polynomial time.

Bellantoni and Cook's BC "safe/normal recursion" [Bel92, BC92] The idea of Cobham is improved by Bellantoni and Cook's scheme safe recursion. It is based on the insight that it is enough to restrict the recursion on notation scheme in such a way, that the step function h (in Equation 1.2) is not allowed to recurse over the last argument $f(n, \overrightarrow{y})$, i.e. the recursion result of the previous step.

The philosophical idea is that it is impredicative to define f by recursion while using f already to start another nested recursion inside the definition. Predicativity means in this context that one can only recurse over completely available values, i.e. those which do not depend on intermediate results of a running recursion.

While Cobham explicitly required to bound a new function by an already defined one, Bellantoni and Cook improve this idea to only allow recursions on values which use already defined (therefore called normal) functions and values.

Technically, in Bellantoni and Cook's BC system two contexts are used, one for the safe variables (which do not allow recursions) and one for the

normal variables (which are freely usable to recurse on). The two kinds of variables in the definition of terms are separated by a semicolon:

$$f(0, \overrightarrow{x}; \overrightarrow{y}) := g(\overrightarrow{x}; \overrightarrow{y})$$
$$f(S_i n, \overrightarrow{x}; \overrightarrow{y}) := h(n, \overrightarrow{x}; \overrightarrow{y}, f(n, \overrightarrow{x}; \overrightarrow{y}))$$

with the normal variables on the left and the safe ones on the right. Moreover, there is a safe composition of functions which is restricted in such a way, that on the left of the semicolon only terms may be put which do not depend on safe variables. Hence, one can use normal variables for the safe parts, but not the other way around.

Bellantoni and Cook [BC92] prove that BC is correct and complete for PTime, i.e. every algorithm is polynomial time computable and every computable time function can somehow be implemented as an algorithm in BC.

While being a very simple and elegant restriction, BC is hardly usable for practical programming as the recursion scheme is very restrictive. The prime example for an algorithm not expressible in a natural way is Insertion Sort which is polynomial of course.

Instead of safe/normal, several other similar terms were used by others to describe the predicative and impredicative variables:

- incomplete and complete by Schwichtenberg and Bellantoni in the system LT [SB01]
- input and output by Ostrin and Wainer in the system $EA(I;O)$ [OW05a].

Schwichtenberg and Bellantoni's LT "complete/incomplete" [SB01]; arithmetic LHA [Sch06] The original BC system does not allow higher types. Especially the recursion scheme can only give ground type values, i.e. numbers. The system LT is the natural extension of BC to full System T, i.e. recursion in higher types. The main idea is to reflect the two contexts of Bellantoni and Cook's BC by having two arrows in the type system of LT:

- $\to$ for the complete/normal/input variables/abstractions (which in addition are denoted with a bar as in $\bar{x}$),
- $\multimap$ for the incomplete/safe/output variables/abstractions.

A term is called complete if it has no free variables which are incomplete[4]. A term of the $\rightarrow$-function type can only be applied to a complete term.

In the tradition of BC, the recursion constant in LT gets the following type:

$$\boldsymbol{\mathcal{R}}_{\rho,\tau} : L(\rho) \rightarrow (\rho \rightarrow L(\rho) \rightarrow \tau \multimap \tau) \rightarrow \tau \multimap \tau.$$

By far the most important arrows in this type are the first and the fourth one:

- The first makes sure that a *complete* term is required to do a recursion on.
- The fourth makes sure that the safe/incomplete recursion value cannot be used to do another nested recursion (compare with the restrictions in BC above).

There are further, smaller restrictions necessary, like linearity of higher type variables. Compare [SB01] and Section 3.2 for a more detailed account on the role of the arrow choices, and how they are reflected in the PTime correctness proof of [SB01].

As System T belongs to Heyting Arithmetic as a term system, in [Sch06] a linear Heyting Arithmetic called LHA is introduced for LT. The provably recursive functions of LHA are those of LT, and therefore they are polynomial time computable. Essentially, the separation of the complete and the incomplete contexts in LT must be extended to the quantifiers of the arithmetic LHA.

Leivant and Marion "tiering"/"ramified recurrence" [LM93, Lei94a, Lei95b, LM95] Bellantoni and Cook in BC and later Bellantoni together with Schwichtenberg in LT have split the context in two, i.e. into the normal and the safe part. Leivant and Marion [LM93] instead consider a whole hierarchy called tiers, using another data type $\mathbb{N}_i$ for each tier i. Then the recursion scheme is restricted in such a way that the result type has a smaller tier i than the numeral that is recursed over of tier $\mathbb{N}_j$, i.e. $j > i$ must be fulfilled. This gives a system which is very similar to BC. The latter can then be viewed as a tiered system with only two tiers.

[4] Note here the difference between complete/incomplete terms and complete/incomplete variables. The later are marked in the types and the terms. The former are inferred, depending on the free variables.

Leivant "data-predicative" [Lei95a, Lei01] The idea of normal/complete/input variables can be transferred into the context of arithmetic. Daniel Leivant takes his Intrinsic Theories $\mathbf{IT}(\mathbb{N})$ and identifies *data-positive* derivations which roughly correspond to terms with only ground-type recursions, resulting in elementary time provable functions. Then he restricts these further to *data-predicative* derivations, which forbid that the major premise $\mathbf{N}(t)$ (the argument to do the induction over) is bound data-positively in the derivation. This essentially means that the major premise of induction is complete, and therefore that induction is not possible over $\mathbf{N}(t)$ coming from the induction hypothesis. This is in analogy to safe recursion in BC, where this would mean to do a nested recursion over the safe/incomplete result of the previous step in the recursion.

Marion "input driven","actual arithmetic" [Mar01] In a similar setting as Leivant's Intrinsic Theories $\mathbf{IT}(\mathbb{N})$ Marion proposes an "actual" arithmetic $\mathbf{AT}(W)$, a variant of the $\to \forall\wedge$-fragment of Heyting Arithmetic, with an equational calculus to define the clauses of the algorithms, and an induction scheme which is relativised to $W(x)$ (meaning x is a (binary) computable value).

$$\text{Induction}\ \frac{\forall y.A[y], \mathbf{W}(y) \to A[\mathbf{s_0}(y)] \quad \forall y.A[y], \mathbf{W}(y) \to A[\mathbf{s_1}(y)] \quad A[\epsilon]}{\forall x.\mathbf{W}(x) \to A[x]}\ \text{Ind}(\mathbf{W})$$

A term f is provably total if $\forall \overrightarrow{x}.W(x_1), \ldots, W(x_n) \to W(t(\overrightarrow{x}))$ can be proved.

The central idea, though, is to restrict the $\forall$-elimination rule to actual terms which are those built up only from constructors and variables.

For instance, this rules out inductions over $(\text{add}\, x\, y)$, because this is not an actual term. When trying to prove $W(\exp(x))$ (defined as usual with $\exp(S_i(x)) = \text{add}(\exp(x), \exp(x))$) by induction on x, it is not possible in the step case to apply the totality proof of add to $\exp(x)$ because this is not a total or actual value. Hence, it is not definable, in analogy to the safe recursion of BC.

Again, the idea is to enforce predicativity. Yet, in this system it is not even possible to do recursion/induction over $(\text{mult}\, x\, y)$, e.g. to compute $(\text{mult}\, x\, y) + (\text{mult}\, x\, y)$, because there is no proper composition of totality proofs. On the other hand, assumptions can be freely duplicated, hence it

is possible to compute x^e for any $e \in \mathbb{N}$ by $(\text{mult}\, x\ (\text{mult}\, x\ (\text{mult}\, x\ \ldots)))$, giving polynomial time completeness by a Turing Machine simulation.

Ostrin and Wainer's $EA(I;O)$ "input/output elementary arithmetic" [OW05a] Last, but not least, Ostrin and Wainer designed the system $EA(I;O)$ as an arithmetic with input and output variables, motivated by the predicative recursion with safe/normal variables of Bellantoni and Cook [BC92] and Daniel Leivant's Intrinsic Theories [Lei95a]. They show that the provably recursive functions are exactly the elementary ones, and by switching to binary numbers and restricting the induction to $\sum_1$-formulas, polynomial time can be characterised. While these theoretical results are in line with the topic of this thesis, the flavor of $EA(I;O)$ is purely proof theoretic and not meant to be a system for writing down algorithms.

1.3.2. Linearity – Controlling Duplication

Predicative recursion is one way to outlaw algorithms which are not polynomial time computable. Another branch of research with similar results studies linearity in type systems or logics. Exponential time algorithms always duplicate data in one way or another. Hence, *general duplication* of data (or assumptions in logics) is forbidden, and only allowed then in a very restricted and controlled way.

The central idea, why this works, is that linear or affine λ-terms, i.e. where every bound variable appears once or at most once in the body, can be normalised in linear many steps. This is because in every reduction the term gets smaller if terms cannot be duplicated.

Hence, the general approach is to add further features like restricted duplication or iteration to such a linear or affine λ-calculus, but only as far as the normalisation is still possible in polynomial time. Hence, compared to predicativity based calculi from Section 1.3.1, the theme here is more in the opposite direction, i.e. adding features which do not lead outside polynomial time, instead of removing certain bad recursion schemes like e.g. in LT or BC.

Hofmann's LFPL "non-size-increasing" [Hof99a, AS00, AS02, ABHS04] Inspired by Caseiro's criteria for algorithms which are polynomial [Cas96a, Cas96b, Cas97], Hofmann came up with the Linear Functional Programming Language (LFPL). He starts with an affine λ-calculus with the base

type B and lists $L(\tau)$, case distinction and iteration. To control the complexity of the iteration, he adds another type $\diamondsuit$ which is playing the role of money during reduction. It can only come into play in a term by a free variable. In order to extend a list using the $\mathbf{cons} : \diamondsuit \multimap \tau \multimap L(\tau) \multimap L(\tau)$ constructor, one "coin" of type $\diamondsuit$ is needed. When a list is destructed in an iteration, this $\diamondsuit$-money is given back in every step. This mechanism together with linearity makes sure that the maximal list length during normalisation is bounded (essentially by the number of free variables at the beginning of the reduction sequence).

The LFPL system is very easy to grasp and as its name suggests, the initial intention is clearly to be a simple programming language. Complexity-wise LFPL can only type polynomial time algorithms which do not increase the size of the data (because the $\diamondsuit$-money cannot be created in a closed term), i.e. which are *non-size-increasing* (compare Section 2.2 for a formal definition of LFPL).

Girard, Asperti, Roversi's LLL/LAL [Gir98, AR00, AR02] Girard introduced Light Linear Logic (LLL) as a variant of Linear Logic, the discipline of (usually) higher order logics which are linear, with the exceptions of terms of type $!\tau$ (*bang*). The *bang* is a modality in the type system which marks terms that can be duplicated, i.e. where the strict linearity does not apply. The duplication itself is made explicit in the term system (or more precisely in the proof nets which are used for normalisation; compare Definition 2.30) as so called multiplexers. The introduction rule of ! is usually denoted by a graphical box in the proof nets, with the subterms inside the box.

Light Linear Logic has the §-modality in addition to ! in order to mark also the former existence of the ! in the type. This means that there is no general term $t : !\tau \multimap \tau$, but only $t : !\tau \multimap \S\tau$. In other words, if one wants to eliminate the ! in the type, one gets the § in exchange. This property is called stratification and is the central idea of LLL, which will allow the normalisation in polynomial time. Moreover, only terms/proof nets with at most one free variable/open assumption can be marked with the !, while the free variable/open assumption is marked at the same time.

The duplication (contraction) rule in LLL is only binary, i.e. from one term of type $!\tau$ two copies of type $!\tau$ are created.

Asperti and Roversi took Girard's Light Linear Logic and simplified the type system a lot to allow affine typing, resulting in *Light Affine Logic* (LAL). LAL is shown to be complete and correct for polynomial time.

Lafont's SLL [Laf04]; Soft Lambda Calculus [BM04] In Soft Linear Logic (SLL) the typing rules which relate to the !-modality of Linear Logic are replaced by

- an n-ary multiplexer for arbitrary rank n, i.e. from $!\tau$ to $\tau, \ldots, \tau$,
- and a soft promotion rule, which can mark any term/proof net with the bang !, while marking also the variables/open assumptions with the !-modality.

Soft Linear Logic is also complete and correct for polynomial time. But in contrast to LAL for example, the completeness proof (which simulates a Turing Machine) is far from trivial and makes use of a lot of coding. Another non-trivial example is the predecessor function, which requires a lot of effort to be implemented in SLL. It is questionable how usable Soft Linear Logic is from a programming point of view.

Baillot and Terui's DLAL [BT04] Proof nets are the usual representation of proofs in Linear Logic. The reason is that proof nets abstract from the uninteresting details of proofs, like certain orders of rules. Traditional proof theory adds permutative conversions for the same purpose, but these often complicate the reasoning about proofs or terms (in the case of λ-calculi).

With Dual Light Affine Lambda Calculus (DLAL) Baillot and Terui introduce a λ-calculus variant of Light Affine Logic. They remove the necessity of permutative conversions by a careful restriction of the LAL type system. The central consideration is that the !-modality is usually only needed on the left side of the arrow $\multimap$. Hence, the general bang modality ! is replaced in this system by a special arrow $\sigma \to \tau$, which morally stands for $!\sigma \multimap \tau$. Surprisingly, by this one can get rid of many difficulties when trying to find a λ-calculus representation of LAL proof nets. DLAL can be normalised in PTime without going to a proof net representation.

Roversi's WALT [Rov08a, Rov08b] It was for long an open problem [MO00, MO04, NM03] (in the time spans of implicit complexity of course) whether Bellantoni and Cook's BC system could be embedded into Light Affine Logic. A restriction of BC called BC^- can be embedded into LAL [MO00]. Mairson and Neergaard give a sketchy argument [NM03] as to why a complete embedding of BC is probably not possible.

Recently, Roversi came up with essentially an extension of LAL called Weak Affine Light Typing (WALT), which gives better control over the normalisation order.

Terui [Ter01] had shown in his thesis long before that in LAL no special normalisation order is necessary to stay in polynomial time. This property and the example by Beckmann and Weiermann (compare Section 3.1.3), which needs a special normalisation order in BC to be polynomial time normalisable, were the basis of Mairson and Neergaard's argument. An embedding from non-strongly normalising BC into the strongly normalising LAL would need to make the normalisation order explicit somehow. Roversi managed to embed full BC into WALT by exploiting the better control of when boxes are duplicated.

1.3.3. Extensional versus Intensional Point of View

A central distinction in implicit complexity is made between *extensional* and *intensional* completeness. An algorithm is a syntactical object in some calculus which implements a method to solve a problem. A function on the other hand is a mathematical, set theoretic (therefore quite abstract) relation between inputs and outputs. For a function there might be many algorithms in different calculi that compute the function. On the other hand, an algorithm computes only one function (if the coding of the input and output is fixed).

Extensional view A calculus is usually considered to be complete for polynomial time if it admits the implementation of at least one algorithm for every polynomial time function. This is called *extensional completeness*. In order to prove it, it is clearly enough to take an arbitrary computation model for all polynomial time functions, e.g. Turing Machines or Register Machines, and then to show that every polynomial time algorithm in that model can be simulated in the considered calculus. Also, in the scientific practice this is the most applied proof method to show completeness.

Obviously, it is highly questionable how relevant such a result is for using a calculus as a programming language, i.e. to express practical algorithms. From a theoretical point of view intensional completeness is exactly what one wants to characterise a complexity class though.

Intensional view From the intensional point of view one looks at algorithms, and not only at the functions they compute. Clearly, this view is much finer than "talking" about set theoretic functions (i.e. collections of (input, output) pairs). Intensional expressivity of a system "talks" about the algorithms which can be typed in a given calculus.

Classical complexity theory is *not* intensional and differences of computation models do not matter. But, if one is instead interested in polynomial time calculi from the view point of programming algorithms, looking at intensional properties is essential.

Intensional expressivity In the extensional setting expressivity is formulated in the language of sets, sets of functions over the natural numbers. The interesting complexity theoretic questions "talk" about complexity hierarchies like the polynomial hierarchy or sub-recursive hierarchies like NC_k. While some complexity classes are in fact defined by some computational model, the comparison is usually based only on the set-theoretic subset relation.

In the intensional setting, when talking about algorithms, it is far from being clear what one means. Intensional statements clearly depend on the calculus one is considering. Extensionally, calculi like BC and LAL are equally strong (both capture PTime). Intensionally, it is much harder to compare them. One approach of course is based on embeddings from one into the other. But by some detour embedding into a Turing Machine for instance, and then its simulation in the other calculus, one can always get an embedding "somehow". Which kind of embedding is permissible? Some kind of modularity and compositionality might be sensible requirements.

Encodings From the complexity theory point of view a characterisation via unary or binary numbers is of no qualitative difference. For practical considerations it is fundamental whether the calculus supports binary numbers (in Leivant's words "word algebras") or whether it only allows unary encodings or monadic algebras. For instance, for the latter multiplication is a quadratic operation, while in the former, it is only linear in the length of the inputs (i.e. number of constructors).

But, much more striking for practical algorithms is that unary encodings need exponentially more constructors for a numeral than a binary encoding. For example multiplying 1000 by 1000 needs two times 10 constructors to represent the input numbers in binary (and hence 20 for the result), but

1000 for each of the inputs in the unary setting and even $1,000,000$ for the output. Hence, while linear versus quadratic does not seem to be dramatic, the representation of the numbers gives the exponential blow up at the end.

1.4. Contributions

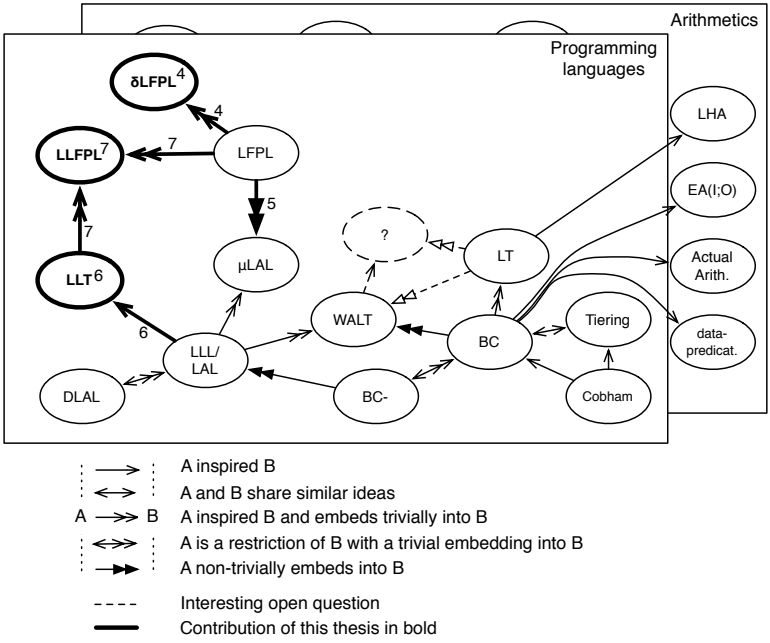


Figure 1.2.: The landscape of polynomial time calculi and arithmetics and the contribution of this thesis (the numbers are the corresponding chapters)

Figure 1.2 shows the landscape of polynomial time calculi which is very fragmented and diverse, with plenty of term systems, higher order logics and arithmetics, which all characterise polynomial time functions in one way or the other. This thesis studies this scene from an intensional point of view,

especially how different systems relate to each other. This is a broad field with a lot of unclear connections and open questions, e.g. whether sensible embeddings exist. We concentrate on Bellantoni and Schwichtenberg's LT, Hofmann's LFPL and Girard, Asperti and Roversi's Light Affine Logic LAL.

1. In Chapter 4 we develop an extension of LFPL called δLFPL that allows typing terms which are not completely linear. Subterms with a passive type like e.g. B or $B \otimes B$ can share free variables with the other part of the term. This allows δLFPL to type "if-then-else" in the usual, but non-linear way, where the predicate shares the free variables with the two branches. The contribution is the new system δLFPL itself, but more importantly, the extension of the syntactical complexity proof for normalisation of Schwichtenberg and Aehlig [AS00, AS02] to this setting with relaxed linearity. It introduces the concept of *contexts* for subterms. Essentially, the instances of the same free variable may be shared if they do not share a common context. While being a new technique covering the non-linear case, this also improves the result of [AS00, AS02] by allowing more liberal normalisation orders.

2. In order to understand the relation of LFPL and LAL, in Chapter 5 we study how the former can be embedded into the latter. Hofmann's LFPL has an impredicative iteration scheme (in the sense of Section 1.3.1), which allows the quite liberal formulation of non-size-increasing polynomial time algorithms. This scheme does not fit into the stratified iteration of Light Affine Logic, and a direct embedding (like e.g. of System T into F) does not seem to be possible. We identify the terms called "iterated iterators" which resemble exactly LFPL's iterations. They allow it to separate the impredicativity of LFPL's iteration from the remaining algorithm logic. This finally allows the embedding of LFPL into LAL in a mostly compositional way.

3. With the experience of Chapter 4 and 5, the similarities of LFPL and LAL, showing up in the pull-out trick (compare Example 3.15), ask for a setting which allows easier side-by-side comparison. LAL, as a variant of System F with all the consequences, like e.g. different possible encodings of numerals, is suited for this. In Chapter 6 we introduce a variant of System T which transfers the ideas of Light

Affine Logic into the world of System T. We prove that the calculus Light Linear T with ! ($\mathrm{LLT}_!$) is complete and correct for polynomial time.
Compared to LAL, it especially features the lack of the §-modality of LAL which is replaced by a system of levels (similar to levels for Linear Logic studied – completely independently from this thesis – by Mazza [Maz08]), that makes $\mathrm{LLT}_!$ stratified as its relative LAL.
In contrast to LAL, the calculus $\mathrm{LLT}_!$ is much easier to program with. With the introduction of System-T-like constants for iteration, case distinction, etc. $\mathrm{LLT}_!$ is not bound to the possible codings and reductions which are imposed by the System-F-like LAL. Instead, the constants can be freely adapted to the needs at hand, similarly to what Hofmann does in LFPL in order to allow only non-size-increasing algorithm in a variant of System T.
One central design choice of $\mathrm{LLT}_!$ is the seamless integration of the pull-out trick (compare Example 6.21), that allows the iteration of algorithms which would get an asymmetric type in the naive formulation in LAL.

4. Starting with $\mathrm{LLT}_!$, variations are developed which bring $\mathrm{LLT}_!$ – step by step via $\mathrm{LLT}_!$, $\mathrm{LLT}_{!\Diamond}$, $\mathrm{LLT}'_{!\Diamond}$ – nearer to LFPL, while keeping $\mathrm{LLT}_!$'s expressivity, to finally end up in $\mathrm{LLFPL}_!$. This system, the Light Linear Functional Programming Language with !, allows the direct formulation of $\mathrm{LLT}_!$ algorithms, and, *at the same time*, it features impredicative iteration of LFPL. Hence, LFPL algorithms can be expressed directly in $\mathrm{LLFPL}_!$ as well. The LFPL part is identified as being the first level of the stratified $\mathrm{LLFPL}_!$ calculus.
In contrast to LAL, each level can have an impredicative iteration of arbitrary nesting. Therefore, the polynomial, that describes the complexity to normalise each level alone, is not quadratic (or of fixed degree) like in LAL or $\mathrm{LLT}_!$, but depends on the nesting of impredicative iterations.
The system $\mathrm{LLFPL}_!$ shows that impredicative iteration and light iteration are orthogonal and can be combined into one system without loosing the polynomial time property.

The contributions from above are typeset in bold in Figure 1.2.

1.5. Structure

The structure of this thesis is the following:

- Chapter 1 introduces the ideas, the motivation, view of the world and related works of the thesis, without loosing itself in too many technicalities.
- Chapter 2 introduces the considered calculi formally.
- Chapter 3 gives the intuition behind the polynomial time calculi. A collection of example programs are implemented in these systems. For those which are exponential it is shown how the type systems outlaw them and what would happen in the complexity proofs if those examples were allowed.
- Chapter 4 develops the quasi-linear extension δLFPL of LFPL and shows that normalisation is possible in polynomial time.
- Chapter 5 embeds LFPL into μLAL using iterated iterators.
- Chapter 6 defines the System T sibling $\mathrm{LLT}_!$ of Light Affine Logic and shows its completeness and correctness for polynomial time.
- Chapter 7 exploits the higher flexibility of $\mathrm{LLT}_!$ (compared to LAL) by tweaking the type system and the constants to define $\mathrm{LLFPL}_!$, which subsumes LFPL and $\mathrm{LLT}_!$.
- Chapter 8 gives a conclusion on the results of this text. Moreover, open questions and further directions of research are presented, together with possible intuitions and ideas.

2

Polynomial Time Type Systems

This chapter introduces the different type systems that will be used throughout the thesis. They serve as a starting point for further improvements, but much more important as a very wide source of ideas and intuition of what makes up a system which captures polynomial time. As described in the introduction, there are a whole lot of calculi in the literature and we will concentrate here only on a selection of them that are of interest for this thesis.

For a deeper understanding, for the completeness and correctness proofs of the presented systems we strongly recommend the respective original papers. We will not go into the details in this chapter, because it would be outside the scope of the thesis. Merely, we aim on fixing the notations to write terms and proof nets. The Chapter 3 will then go through many examples to show the differences between the different approaches.

Structure of this chapter In Section 2.1 we start with the presentation of System T and System F as the basis of all later calculi. In Section 2.2 we introduce Hofmann's LFPL as a variant of Linear System T. In Section 2.3 Light Linear Logic and Light Affine Logic are presented, first as a term calculus with a type system, and then using proof nets. Finally, subsection 2.3.6 presents Light Affine Logic with the fixed point extension.

2.1. Preliminaries

Before introducing type systems for polynomial time though, we will start with the well known Gödel's System T [Göd58] and Girard's System F [Gir72]. All calculi introduced later are based on the *model of computation* of one or the other.

2.1.1. System T

System T (also called *Gödel's T* due to the first introduction in his Dialectica paper [Göd58]) is a non-polymorphic typed lambda calculus with a set of (inductive) types, constructor constants and primitive recursion in all finite types.

In fact, Gödel's version was restricted to the unary natural numbers. But as the basic ideas easily extend to the usual inductive types we will liberally use such extensions here, still keeping them under the name of Gödel's T (or synonymously System T).

Definition 2.1 (Types)**.** The set Ty_T of types is defined inductively as

$$\sigma, \tau ::= B \mid \sigma \to \tau \mid \sigma \otimes \tau \mid L(\sigma).$$

We write $\rho \to \sigma \to \tau$ for $\rho \to (\sigma \to \tau)$ and $\rho \otimes \sigma \to \tau$ for $(\rho \otimes \sigma) \to \tau$. Terms of type $L(\sigma)$ are called lists of type σ or lists with σ-values. Terms of type $\sigma \otimes \tau$ are called pairs of σ and τ. Terms of type B are called boolean values.

Definition 2.2 (Constants)**.** The constants Cnst_T and their types are:

$$\begin{aligned}
\mathbf{tt} &: B \\
\mathbf{ff} &: B \\
\mathbf{cons}_\tau &: \tau \to L(\tau) \to L(\tau) \\
\mathbf{nil}_\tau &: L(\tau) \\
\otimes_{\tau,\rho} &: \tau \to \rho \to \tau \otimes \rho \\
\mathbf{Case}_\tau &: B \to \tau \to \tau \to \tau \\
\boldsymbol{\mathcal{R}}_{\sigma,\tau} &: L(\sigma) \to (\sigma \to L(\sigma) \to \tau \to \tau) \to \tau \to \tau
\end{aligned}$$

$$\pi_{\rho,\sigma,\tau} : \rho \otimes \sigma \to (\rho \to \sigma \to \tau) \to \tau.$$

Definition 2.3 (Terms)**.** For a countably infinite set of variable names V the set Tm_T of (untyped) terms is inductively given by:

$$r, s, t ::= x^\tau \mid c \mid \lambda x^\tau.t \mid (t\, s)$$

for variables $x \in V$, types $\tau \in \mathrm{Ty}_T$ and constants $c \in \mathrm{Cnst}_T$.

Variables Variables carry their type in the style of Church. Variables are identified with their name. Different types for the same variable will be excluded by the typing rules, which will be introduced later on.

Subterms Let $\lhd_T$ be the subterm relation defined by the transitive closure of $t \lhd_T (t\, s)$, $s \lhd_T (t\, s)$ and $t \lhd_T \lambda x.t$ for $s, t \in \mathrm{Tm}_T$. Let $\trianglelefteq_T$ be the reflexive closure of $\lhd_T$.

Free variables A variable x is called bound in $\lambda x.t$. The set of free variables $\mathrm{FV}(t)$ is defined by

$$\begin{aligned} \mathrm{FV}((t\, s)) &:= \mathrm{FV}(t) \cup \mathrm{FV}(s) \\ \mathrm{FV}(\lambda x.t) &:= \mathrm{FV}(t) \setminus \{x\} \\ \mathrm{FV}(x) &:= \{x\} \\ \mathrm{FV}(c) &:= \emptyset. \end{aligned}$$

Terms which are equal up to the naming of bound variables are identified.

We sometimes leave out the type of a variable if it can be easily inferred. We liberally leave out parenthesis if they are clear from the context: e.g. $(f\, x\, y)$ for $((f\, x)\, y)$. We write $\lambda x, y.t$ instead of $\lambda x.\lambda y.t$ or leave out the dot for intermediate λ-abstractions as in $\lambda x \lambda y \lambda z.t$. For a product term $(\otimes_{\tau,\rho}\, t\, r)$ we also use $t \otimes r$ or $t \otimes_{\tau,\rho} r$. Note that $\lambda x.s \otimes t$ and $\lambda x.(s \otimes t)$ are different, and $(\lambda x.t\, s)$ is different from $\lambda x.\, (t\, s)$. In other words, λ binds strongest.

In order to type terms in the following definition, we first fix the meaning of a context as a finite map. In other words, the order of assignments of names to types does not matter in our setting.

Definition 2.4 (Typing)**.** A context is a finite map from the names V to types Ty_T. Untyped terms are assigned types using the ternary relation $\vdash$ between a context Γ, an untyped term $t \in \mathrm{Tm}_T$ and a type $\tau \in \mathrm{Ty}_T$, denoted $\Gamma \vdash t^\tau$ via the following rules:

$$\frac{}{\Gamma, x^\tau \vdash x^\tau}\ (\mathrm{Var}) \qquad \frac{c \text{ constant of type } \tau}{\Gamma \vdash c^\tau}\ (\mathrm{Const})$$

$$\frac{\Gamma, x^\sigma \vdash t^\tau}{\Gamma \vdash (\lambda x.t)^{\sigma\to\tau}}\ (\to^+) \qquad \frac{\Gamma_1 \vdash t^{\sigma\to\tau} \quad \Gamma_2 \vdash s^\sigma}{\Gamma_1, \Gamma_2 \vdash (t\,s)^\tau}\ (\to^-)$$

with Γ_1, Γ_2 meaning the set union, such that Γ_1 and Γ_2 agree on the assigned types on the intersection of their domains.

In the style of the (Var)- and (Const)-rules we use implicit weakening throughout this thesis.

Often we will talk about "a typed term t" (or just "a term t^τ"). This will mean that there is, implicitly, a type derivation for $\Gamma \vdash t^\tau$ for some context Γ and type τ.

Definition 2.5 (Reductions)**.** Terms are reduced via the usual β-conversion plus conversion rules for the constants:

$$\begin{aligned}
(\lambda x.t\,s) &\longmapsto t[x := s] \\
(\mathbf{Case}_\tau\ \mathbf{tt}) &\longmapsto \lambda x^\tau \lambda y^\tau.x \\
(\mathbf{Case}_\tau\ \mathbf{ff}) &\longmapsto \lambda x^\tau \lambda y^\tau.y \\
(\pi_{\rho,\sigma,\tau}\ (\otimes_{\rho,\sigma}\ r\ s)) &\longmapsto \lambda f^{\rho\to\sigma\to\tau}.\,(f\ r\ s) \\
(\boldsymbol{\mathcal{R}}_{\sigma,\tau}\ \mathbf{nil}_\sigma\ f\ g) &\longmapsto g \\
(\boldsymbol{\mathcal{R}}_{\sigma,\tau}\ (\mathbf{cons}_\sigma\ v\ l)\ f\ g) &\longmapsto (f\ v\ l\ (\boldsymbol{\mathcal{R}}_{\sigma,\tau}\ l\ f\ g))\,.
\end{aligned}$$

We write $t \longrightarrow t'$ if a subterm $s \trianglelefteq t$ reduces via $\longmapsto$ as defined above. With $t[x := s]$ we denote the capture-free substitution of $x \in \mathrm{FV}(t)$ by s and $t[x := s] = t$ if $x \notin \mathrm{FV}(t)$.

2.1.1.1. Intended Meaning

The intended meaning of the types and constants is the following:

- **tt** and **ff** of type B are boolean values and $\left(\mathbf{Case}_\tau\, b^B\right)$ is our representation of an if-then-else case distinction which depends on the "predicate" b.

- $(\otimes_{\rho,\sigma}\, r\, s)$ is a (cartesian) product of the "left side" r and the "right side" s. The constant $\pi_{\rho,\sigma,\tau}$ destructs a product by giving access to the left and right values again.

- $L(\tau)$ denotes the type of lists with values of type τ. The usual "numerals" are the chains of $\mathbf{cons}_\tau$ and $\mathbf{nil}_\tau$ like in many programming languages (such as LISP or Scheme).

- $(\boldsymbol{\mathcal{R}}_{\sigma,\tau}\, l\, f\, g)$ defines a (primitive) recursion over the list l, using the step term f and the base case g.

2.1.1.2. Located Subterms

Terms are equal if they are constructed the same way via the term Definition 2.3. In other words, terms are considered to be equal if they generate the same syntax tree (up to renaming of bound variables).

This equality definition works fine for many cases, but is not suitable for others: consider the term $w := (t\ (t\, s))$. How many subterms t does w have? Of course, the intuitive answer is 2, i.e. there are two subterms which have the same syntax tree as t. Moreover, these two occurrences are clearly equal syntactically, but they differ with respect to their location in the term w. To talk about these two instances independently, we introduce the following concept:

Definition 2.6 (Located subterm)**.** A located subterm of the term t is the pair $s \trianglelefteq_T t$ together with a unique description of its location of s in the syntax tree of t.

Of course, this definition could be made more formal by fixing a coding of syntax trees and another coding of locations therein. We will not fix these codings here though in order to keep things as simple as possible.

The concept of located subterms has not been defined inductively above. But depending on the chosen coding, it is not hard to make it into an inductive one, following the clauses of the subterm relation. There is, however, an even more direct solution to this: every proof that s is a subterm of t, i.e.

that $s \trianglelefteq_T t$ holds, is in fact just a proof term M, that is essentially built up from constants corresponding to the inductive clauses of $\trianglelefteq_T$ (which is $\triangleleft_T$ plus symmetry). Different located subterms give different proofs. Hence, we can just consider these proofs as the located subterms. Then the coding is simply that of proof terms which are inductively defined.

Throughout this thesis, the concept of located subterms will be used for different term systems, though we will not introduce it again and again. Instead, we implicitly assume that for every (non-strict) subterm relation $\trianglelefteq$ there is a definition of located subterms in the same way as in the Definition 2.6 above for System T.

2.1.1.3. System T with Applicative Notation

Logically the application $(t\,s)$ is the elimination of the arrow type. It can be convenient to use the same applicative notation to eliminate the other types as well. This way we remove the need for destruction constants $\mathbf{Case}_\tau, \pi_{\rho\otimes\sigma,\tau}, \boldsymbol{\mathcal{R}}_{\sigma,\tau}$ by adding another typing rule for each of them, in the following way:

$$\frac{\Gamma \vdash b^B \quad \Gamma \vdash f^\sigma \quad \Gamma \vdash g^\sigma}{\Gamma \vdash (b\,f\,g)^\sigma}\ (B^-) \qquad \frac{\Gamma \vdash p^{\sigma\otimes\tau} \quad \Gamma \vdash f^{\sigma\to\tau\to\rho}}{\Gamma \vdash (p\,f)^\rho}\ (\otimes^-)$$

$$\frac{\Gamma \vdash l^{L(\sigma)} \quad \Gamma \vdash f^{\sigma\to L(\sigma)\to\tau\to\tau} \quad \Gamma \vdash g^\tau}{\Gamma \vdash (l\,f\,g)^\tau}\ (L(\sigma)^-)$$

The reduction rules are replaced by the following:

$$\begin{aligned}
(\mathbf{tt}\,f\,g) &\longmapsto f\\
(\mathbf{ff}\,f\,g) &\longmapsto g\\
((\otimes_{\tau,\rho}\,r\,s)\ f) &\longmapsto (f\,r\,s)\\
(\mathbf{nil}_\sigma\,f\,g) &\longmapsto g\\
((\mathbf{cons}_\sigma\,x\,l)\ f\,g) &\longmapsto (f\,v\,l\ (l\,f\,g))\,.
\end{aligned}$$

Clearly, both notations for System T are equivalent in the sense that each term in one system can be expressed in a very similar way in the other (mainly by doing some η-expansion/reductions and then by doing simple reformulations with the other syntax). The applicative notation though can lead to shorter proofs since the rules look more uniform.

But more importantly, it is possible in the applicative System T to add

restrictions to the each of these typing rules independently (see LFPL in Section 2.2 for an example). I.e., we gain flexibility to adapt the calculus to certain needs.

Conversely, the non-applicative System T has only one elimination rule. Of course, one could restrict this instead, depending on the constant on the left of the application. But then, we have the problem that it will be necessary to normalise a term first before it is visible which constant is in the left premise of the elimination rule (and therefore a possible redex). Hence, adding restrictions is not that obvious in this non-applicative setting.

2.1.2. Linear System T

For the complexity of normalisation it can be crucial how often a variable or a term is used. Consequently, it is common to introduce this more prominently into the type system by restricting the use of bound variables to be linear (or affine).

> *Remark* 2.7. We will often use the term "linear" when in fact the technical precise term would be "affine" (affine = one or no occurrence of a variable; linear = exactly one occurrence of a variable). Since we use implicit weakening in all presented type systems (see the (Var) rule above), we do not make this distinction.

Linear System T replaces the $\to$-arrow with $\multimap$ and the recursion constant $\boldsymbol{\mathcal{R}}_{\sigma,\tau}$ with the iteration constant $\mathbf{It}_{\sigma,\tau}$. Hence, we get the following:

Definition 2.8 (Types)**.** The set Ty_{LinT} of linear types is defined inductively as

$$\sigma, \tau ::= B \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid L(\sigma).$$

Note that we use the shorthand LinT because LT is used for Linear T in the sense of Bellantoni and Schwichtenberg [SB01] (compare Section 3.1.1.2).

Definition 2.9 (Constants)**.** The constants Cnst_{LinT} and their types consist of:

$$\mathbf{tt} : B$$

$$
\begin{aligned}
\mathbf{ff} &: B \\
\mathbf{cons}_\tau &: \tau \multimap L(\tau) \multimap L(\tau) \\
\mathbf{nil}_\tau &: L(\tau) \\
\otimes_{\tau,\rho} &: \tau \multimap \rho \multimap \tau \otimes \rho \\
\mathbf{Case}_\tau &: B \multimap \tau \multimap \tau \multimap \tau \\
\mathbf{It}_{\sigma,\tau} &: L(\sigma) \multimap (\sigma \multimap \tau \multimap \tau) \multimap \tau \multimap \tau \\
\pi_{\rho,\sigma,\tau} &: \rho \otimes \sigma \multimap (\rho \multimap \sigma \multimap \tau) \multimap \tau.
\end{aligned}
$$

Definition 2.10 (Terms)**.** For a countably infinite set of variable names V the set Tm_{LinT} of (untyped) terms is inductively given by:

$$r, s, t ::= x^\tau \mid c \mid \lambda x^\tau.t \mid (t\, s)$$

for variable $x \in V$, types $\tau \in \mathrm{Ty}_{LinT}$ and constants $c \in \mathrm{Cnst}_{LinT}$. Terms which are equal up to the naming of bound variables are identified.

Subterms are defined in the same way as for Tm_T, i.e. $\lhd_{LinT} := \lhd_T$.

Free variables The variable x is called bound in $\lambda x.t$. The *multiset* of free variables $\mathrm{FV}(t)$ is defined by

$$
\begin{aligned}
\mathrm{FV}((t\, s)) &:= \mathrm{FV}(t) \cup \mathrm{FV}(s) \\
\mathrm{FV}(\lambda x.t) &:= \mathrm{FV}(t) \setminus \{x\} \\
\mathrm{FV}(x) &:= \{x\} \\
\mathrm{FV}(c) &:= \emptyset.
\end{aligned}
$$

Note the difference to the System T setting that free variables are collected in a multiset in the linear setup. The typing rules will use a multiset context in order to make it possible to express the fact that two instances of a variable are in the context in a subterm.

Definition 2.11 (Typing)**.** A context is a (unordered) finite *multiset* of type assignments from the variable names V to types Ty_{LinT} with the

context condition that no variable is assigned different types at the same time.

Untyped terms are assigned types using the ternary relation $\vdash$ between a context Γ, a untyped term $t \in \mathrm{Tm}_{LinT}$ and a type $\tau \in \mathrm{Ty}_{LinT}$, denoted $\Gamma \vdash t^\tau$ via the following rules:

$$\frac{}{\Gamma, x^\tau \vdash x^\tau}\ (\text{Var}) \qquad \frac{c \text{ constant of type } \tau}{\Gamma \vdash c^\tau}\ (\text{Const})$$

$$\frac{\Gamma, x^\sigma \vdash t^\tau}{\Gamma \vdash (\lambda x.t)^{\sigma \multimap \tau}}\ (\multimap^+) \qquad \frac{\Gamma_1 \vdash t^{\sigma \multimap \tau} \quad \Gamma_2 \vdash s^\sigma}{\Gamma_1, \Gamma_2 \vdash (t\, s)^\tau}\ (\multimap^-)$$

where Γ_1, Γ_2 denotes the multiset union which maintains the context condition. In $(\multimap^+)$ the context Γ *must not* have a type assignment for x.

Because we use multisets, the union $\Gamma_1 \cup \Gamma_2$ adds up the number of occurrences of each variable in Γ_1 and Γ_2. The $(\multimap^+)$-rule implies that x appears at most once in Γ, x^σ and therefore only linearly in t.

It is easy to see that every term of Linear System T can be translated into one of System T without linearity. The converse is not true as – by design of the type system – no duplication and therefore no growth of data can take place.

Hofmann's Linear Function Programming Language (LFPL) in Section 2.2 essentially uses this property to create a flexible type system for non-size-increasing algorithms.

Definition 2.12 (Reductions)**.** The reduction rules are the same as for non-linear System T with the exception of the recursion rules which are replaced by iteration rules:

$$(\mathbf{It}_{\sigma,\tau}\ \mathbf{nil}_\sigma\ f\ g) \longmapsto g$$
$$(\mathbf{It}_{\sigma,\tau}\ (\mathbf{cons}_\sigma\ v\ l)\ f\ g) \longmapsto (f\ v\ (\mathbf{It}_{\sigma,\tau}\ l\ f\ g))\,.$$

2.1.3. System F

In the following we present a formulation of System F, a polymorphic propositional logic, introduced by Girard [Gir72] and independently by Reynolds [Rey74]. In comparison with System T, there are no constants and no base

types in F. Instead, type variables and unrestricted quantification on the type level are available:

Definition 2.13 (Types)**.** For a countably infinite set T of type variable names the set of types Ty_F is inductively defined by:

$$\sigma, \tau ::= \alpha \mid \sigma \to \tau \mid \forall \alpha.\sigma$$

with $\alpha \in T$. The type variable α is called bound in $\forall \alpha.\sigma$. The set of free type variables in a type τ is denoted by $\mathrm{FTV}(\tau)$:

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

Types which are equal up to renaming of bound variables are identified.

Definition 2.14 (Terms)**.** For a countably infinite set V of variable names the set Tm_F of (untyped) terms is inductively defined by:

$$r, s, t ::= x^\tau \mid \lambda x^\tau.t \mid (t\, s) \mid (t\, \tau) \mid \Lambda \alpha.t$$

with types $\tau \in \mathrm{Ty}_F$, type variables $\alpha \in T$ and variable names $x \in V$. Terms are identified up to renaming of bound term variables.

Variables The variable x is called bound in $\lambda x^\tau.t$. The set $\mathrm{FV}(t)$ of free variables is defined by

$$\begin{aligned} \mathrm{FV}((t\, s)) &:= \mathrm{FV}(t) \cup \mathrm{FV}(s) \\ \mathrm{FV}(\lambda x^\tau.t) &:= \mathrm{FV}(t) \setminus \{x\} \\ \mathrm{FV}(x^\tau) &:= \{x\} \\ \mathrm{FV}((t\, \tau)) &:= \mathrm{FV}(t) \\ \mathrm{FV}(\Lambda \alpha.t) &:= \mathrm{FV}(t). \end{aligned}$$

A type variable α is called bound in $\Lambda \alpha.t$. The set $\mathrm{FTV}(t)$ of free type variables of a term $t \in \mathrm{Tm}_F$ is inductively given by

$$\begin{aligned}
\mathrm{FTV}((t\,s)) &:= \mathrm{FTV}(t) \cup \mathrm{FTV}(s) \\
\mathrm{FTV}(\lambda x^{\tau}.t) &:= \mathrm{FTV}(t) \cup \mathrm{FTV}(\tau) \\
\mathrm{FTV}(x^{\tau}) &:= \mathrm{FTV}(\tau) \\
\mathrm{FTV}((t\,\tau)) &:= \mathrm{FTV}(t) \cup \mathrm{FTV}(\tau) \\
\mathrm{FTV}(\Lambda\alpha.t) &:= \mathrm{FTV}(t) \setminus \{\alpha\}.
\end{aligned}$$

Subterms The subterm relation is defined by the same clauses as $\lhd_T$ plus $t \lhd_F (t\,\tau)$ and $t \lhd_F \Lambda\alpha.t$.

Because System F has no constants, data types are coded in an impredicative polymorphic way. The standard codings of booleans, natural number and pairs are the following:

$$\begin{aligned}
B &:= \forall\alpha.\alpha \to \alpha \to \alpha \\
\sigma \otimes \tau &:= \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha \\
N &:= \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha,
\end{aligned}$$

with the coding of N called Church numbers.

Definition 2.15 (Typing)**.** A context is a finite map from the names V to types Ty_F. Untyped terms are assigned types using the ternary relation $\vdash$ between a context Γ, a untyped term $t \in \mathrm{Tm}_F$ and a type $\tau \in \mathrm{Ty}_F$, denoted by $\Gamma \vdash t^{\tau}$, via the following rules:

$$\frac{}{\Gamma, x^{\tau} \vdash x^{\tau}}\ (\mathrm{Var})$$

$$\frac{\Gamma, x^{\sigma} \vdash t^{\tau}}{\Gamma \vdash (\lambda x.t)^{\sigma\to\tau}}\ (\to^{+}) \qquad \frac{\Gamma_1 \vdash t^{\sigma\to\tau} \quad \Gamma_2 \vdash s^{\sigma}}{\Gamma_1, \Gamma_2 \vdash (t\,s)^{\tau}}\ (\to^{-})$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash \Lambda\alpha.t}\ (\forall^{+}) \qquad \frac{\Gamma \vdash t^{\forall\alpha.t} \quad \tau \in \mathrm{Ty}_F}{\Gamma \vdash (t\,\tau)}\ (\forall^{-})$$

where Γ_1, Γ_2 means the set union, such that Γ_1 and Γ_2 agree on the intersection of their domains. In $(\to^{+})$ the context Γ must not have a type

assignment for x. In $(\forall^+)$ the type variable α must not be free in any type of Γ.

We will not give a formal definition of the normalisation of terms in System F. It should be clear how it works though – basically like System T without constants, but with substitution for the type application $(\forall^-)$.

2.1.4. Linear System F

In analogy to Linear System T, we can also restrict System F to the linear fragment by taking multisets for the contexts. Otherwise, the typing rules for the arrow (which we will denote by $\multimap$ again, as for Linear System T) look formally the same:

$$\frac{\Gamma, x^\sigma \vdash t^\tau}{\Gamma \vdash (\lambda x.t)^{\sigma \multimap \tau}} \ (\multimap^+) \qquad \frac{\Gamma_1 \vdash t^{\sigma \multimap \tau} \quad \Gamma_2 \vdash s^\sigma}{\Gamma_1, \Gamma_2 \vdash (t\, s)^\tau} \ (\multimap^-)$$

again with the restriction that Γ_1 and Γ_2 are compatible and Γ does not type x. Since now we use multisets, the union $\Gamma_1 \cup \Gamma_2$ adds up the number or occurrences of each variable in Γ_1 and Γ_2. Moreover, the $(\multimap^+)$-rule implies that x appears at most once in Γ, x^σ and therefore in t, which is exactly the linearity condition.

This Linear F is in fact the starting point of Linear Logic which enriches the types with modalities to allow restricted forms of duplication. The Light Linear Logic system in Section 2.3 will use this approach.

The quantification $\Lambda\alpha.\ldots$ will not be required to be linear (whatever this should mean if we required that). Linear type abstraction would restrict possible polymorphic algorithms considerably, e.g. not even the Church encoding of natural numbers are type-linear in this sense.

2.2. LFPL

The Linear Functional Programming Language (LFPL) was introduced by Hofmann [Hof99a] as an attempt to capture the intuition of nested iterations of non-size-increasing algorithms. This came about originally from studying syntactical restrictions of function definitions in Caseiro's thesis [Cas97]. Intuitively, iterating algorithms which do not increase the size of the data and which are polynomial time computable themselves cannot exceed polynomial time, as each iteration again is restricted by the original input size.

While Hofmann's proofs for showing that "every algorithm is polynomial time computable" use semantic arguments, another syntactical analysis was given by Aehlig and Schwichtenberg [AS00, AS02]. We will follow the latter in this thesis, from the point of view of the presentation of the type system and with respect to the proofs in Chapter 4 which will explore extensions of LFPL.

The system LFPL is a variant of Linear System T as presented in Section 2.1.2, with the following additions:

- another product type constructor $\times$ forming the cartesian product (while $\otimes$ will be called the tensor product),
- a base type $\diamondsuit$, the diamond. $\diamondsuit$ plays the central role in the system, as a kind of money which cannot be created out of nothing. Instead, the terms of type $\diamondsuit$ always have free variables where $\diamondsuit$ occurs positively in their type. Hence, the only way to get a term of type $\diamondsuit$ in a closed term is by getting it "from outside" via a lambda abstraction, i.e. as input.
- an iteration which is restricted to *closed step terms*.

There are several ways to enforce the closeness restriction of step terms via a type system. A usual iteration constant could be applied to any (also non-closed) step terms. One option would be to introduce another arrow type which requires additional rules. Instead, we follow Aehlig and Schwichtenberg with an applicative notation, as described in Section 2.1.1.3. In contrast to Aehlig and Schwichtenberg, we do not use the curly brackets as in the iteration $(l^{L(\sigma)}\,\{f\})$ with the closed step term f, which is just a cosmetic detail though. Moreover, we require that all destructor arguments are given, i.e. $(l^{L(\sigma)}\,f\,g)$ is a valid term and $(l^{L(\sigma)}\,f)$ is not.

Definition 2.16 (Types)**.** The set $\mathrm{Ty}_{\mathrm{LFPL}}$ of linear types is inductively defined by:

$$\sigma, \tau ::= \diamondsuit \mid B \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid \sigma \times \tau \mid L(\sigma).$$

Definition 2.17 (Terms and Constants)**.** The set $\mathrm{Tm}_{\mathrm{LFPL}}$ of terms is inductively defined by:

$$r, s, t ::= x^\tau \mid c \mid \lambda x^\tau.t \mid \langle t, s\rangle \mid (t\, s)\,.$$

Terms which are equal up to the naming of bound variables are identified. Free variables are defined as in Definition 2.10, with the additional clause $\mathrm{FV}(\langle t, s\rangle) := \mathrm{FV}(t) \cup \mathrm{FV}(s)$.

Constants The constructor symbols $\mathrm{Cnst}_{\mathrm{LFPL}}$ and their types are:

$$\begin{aligned}
\mathbf{tt} &: B\\
\mathbf{ff} &: B\\
\mathbf{cons}_\tau &: \Diamond \multimap \tau \multimap L(\tau) \multimap L(\tau)\\
\mathbf{nil}_\tau &: L(\tau)\\
\otimes_{\tau,\rho} &: \tau \multimap \rho \multimap \tau \otimes \rho.
\end{aligned}$$

Subterms The subterm relation $\lhd_{\mathrm{LFPL}}$ is defined the same way as $\lhd_{LinT}$, with the additional clauses $s \lhd_{\mathrm{LFPL}} \langle s, t\rangle$ and $t \lhd_{\mathrm{LFPL}} \langle s, t\rangle$.

Definition 2.18 (Typing)**.** A context Γ is defined as in Definition 2.11. The relation $\Gamma \vdash t^\tau$ is inductively defined as follows:

$$\frac{}{\Gamma, x^\tau \vdash x^\tau}\ (Var)$$

$$\frac{c \text{ constant of type } \tau}{\Gamma \vdash c^\tau}\ (Const)$$

$$\frac{\Gamma, x^\sigma \vdash t^\tau}{\Gamma \vdash (\lambda x.t)^{\sigma\multimap\tau}}\ (\multimap^+) \qquad \frac{\Gamma_1 \vdash t^{\sigma\multimap\tau} \quad \Gamma_2 \vdash s^\sigma}{\Gamma_1, \Gamma_2 \vdash (t\, s)^\tau}\ (\multimap^-)$$

$$\frac{\Gamma \vdash s^\sigma \quad \Gamma \vdash t^\tau}{\Gamma \vdash \langle s, t\rangle^{\sigma\times\tau}}\ (\times^+) \qquad \frac{\Gamma \vdash p^{\rho\times\sigma}}{\Gamma \vdash (p\,\mathbf{tt})^\rho}\ (\times^-_{\mathbf{tt}}) \frac{\Gamma \vdash p^{\rho\times\sigma}}{\Gamma \vdash (p\,\mathbf{ff})^\sigma}\ (\times^-_{\mathbf{ff}})$$

$$\frac{\Gamma_1 \vdash b^B \quad \Gamma_2 \vdash p^{\tau\times\tau}}{\Gamma_1,\Gamma_2 \vdash (b\,p)^\tau}\ (B^-)$$

$$\frac{\Gamma_1 \vdash l^{L(\tau)} \quad \emptyset \vdash h^{\diamond\multimap\tau\multimap\sigma\multimap\sigma} \quad \Gamma_2 \vdash g^\sigma}{\Gamma_1,\Gamma_2 \vdash (l\,h\,g)^\sigma}\ (\mathrm{It})$$

$$\frac{\Gamma_1 \vdash p^{\rho\otimes\sigma} \quad \Gamma_2 \vdash f^{\rho\to\sigma\to\tau}}{\Gamma_1,\Gamma_2 \vdash (p\,f)^\tau}\ (\otimes^-)$$

where Γ_1,Γ_2 denotes the multiset union which maintains the context condition. In $(\multimap^+)$ the context Γ must not have a type assignment for x.

Definition 2.19 (Conversion and Reduction)**.**

Conversion The conversion relation $\longmapsto$ is defined by:

$$\begin{aligned}
(\lambda x^\tau\, t\, s) &\longmapsto t[x := s]\\
(\langle s,t\rangle\, \mathbf{tt}) &\longmapsto s\\
(\langle s,t\rangle\, \mathbf{ff}) &\longmapsto t\\
(\mathbf{tt}\, \langle s,t\rangle) &\longmapsto s\\
(\mathbf{ff}\, \langle s,t\rangle) &\longmapsto t\\
((\otimes_{\sigma,\tau}\, s\, t)\ r) &\longmapsto (r\, s\, t)\\
(\mathbf{nil}_\tau\, h\, g) &\longmapsto g\\
((\mathbf{cons}_\tau\, d\, v\, l)\ h\, g) &\longmapsto (h\, d\, v\ (l\, h\, g))\,.
\end{aligned}$$

Reduction The reduction relation $t \longrightarrow t'$ is inductively defined by:

$$\frac{t \longmapsto t'}{t \longrightarrow t'}\ (conv)$$

$$\frac{t \longrightarrow t'}{(t\,s) \longrightarrow (t'\,s)}\ (l) \qquad \frac{s \longrightarrow s'}{(t\,s) \longrightarrow (t\,s')}\ (r)$$

In other words, the reduction is done by applying conversions inside a term, but not below a λ-node and not inside cartesian product terms $\langle s,t\rangle$.

Remark 2.20. Because variables can be shared among the branches of a cartesian product $\langle s, t\rangle$, special care must be taken that the substitution in the beta-reduction does not increase the term considerably (i.e. by duplication of the substituted term into s and t). As normalisation inside cartesian products is not allowed, it is easy to apply *sharing* of terms which appear multiple times in a such a pair.

Remark 2.21. The conversion of the list step does not depend on a completely normalised list, i.e. a list term that is a chain of $\mathbf{cons}_\tau$-constructors and the final $\mathbf{nil}_\tau$. This is in contrast to the conversions in [AS00] where the list length was used in an essential way to define the polynomial measure. In Chapter 4 we will give a proof for an extension of the LFPL system shown above, which allows certain non-linearities in terms. The proof can also be applied to the LFPL calculus without the extension. In comparison with the proof of [AS00], our proof in Chapter 4 makes much better use of the inherent size of the data in a term by taking the number of free variables with $\diamondsuit$ in positive position of their types.

2.3. Light Linear Logic (LLL) and Light Affine Logic (LAL)

In analogy to Linear System T of Section 2.1.2, we sketched Linear System F in Section 2.1.4 by considering multiset contexts and by changing the typing rules in order to enforce linear usage of variables. This has led to a linear (or, as before, more precisely "affine") Higher Order Propositional Logic. As Linear System F is not very expressive anymore (similar to Linear System T), the next step is to add a type modality to the system: the *bang*, denoted by $!\tau$. This leads to what is usually known as Linear Logic.

There has been and still is a lot of research in the field of Linear Logic. It is not possible and also not desired to give a complete account of this subject in this work. For an introduction and overview, the works of Girard [Gir95] and Danos [DC92] are suggested.

Instead, we want to concentrate on one prominent system: Light Linear Logic (LLL), originally introduced by Girard [Gir98] and later simplified by Asperti and Roversi [AR00, AR02] to Light Affine Logic (LAL). Again, we will not use the linear variant, but allow arbitrary weakening. While in this context a distinction is usually made between the linear and the affine system, we will use *linear* and *affine* synonymously in the following. If the

"purely" linear system (i.e. LLL without arbitrary weakening) is meant, this will be explicitly said.

The key property of Light Linear/Affine Logic, compared to more general Linear Logics, is the introduction of a second modality, the paragraph §.

Definition 2.22 (Types)**.** For a countably infinite set T of type variable names, the set of light linear types Ty_{LAL} is defined inductively by:

$$A, B ::= \alpha \mid A \multimap B \mid !A \mid \S A \mid \forall \alpha.A$$

with $\alpha \in T$. Bound and free variables are defined as for System F in Definition 2.13 with the additional clauses $\mathrm{FTV}(!A) := \mathrm{FTV}(\S A) := \mathrm{FTV}(A)$. Again, types equal up to renaming of bound type variables are identified.

Intuitively, the bang type $!\tau$ marks terms of type τ which can be used arbitrarily often.

The paragraph marks terms, which have subterms that were originally with a bang $!\tau$, but that are now freed from it and therefore usable as terms of type τ. Hence, the paragraph is the tool used to *stratify* the type system: if you want to use a banged term $t^{!\tau}$ as it was of type τ, it must be placed as a subterm into a $\S\sigma$ term. This allows tracing of the fact that t originally was of modal type. I.e., every term which makes use of the $!\tau$ term by "removing" the ! has to be of type $\S\sigma$, for some σ.

2.3.1. Proof Nets

In order to present LAL, we introduce another representation of typed programs: proof nets. Formally, a proof net is a directed finite graph with nodes (corresponding to the typing rules of terms), which have one typed output link (with the exception of the λ-node which has another *binding link*) and many or no input links, each labelled with a type.

Note that we only consider intuitionistic proof nets here, hence only one output. The output link corresponds to the term itself with its type, and the input links are connected to the output links of its subterms. Moreover, box nodes of a proof net can have nested proof nets. Figure 2.1 shows a typical proof net for the term

$$\S\Big[\, x = \Big] s^{\S\sigma} \Big[_{\S}\,, y = \Big]\, !^{n} \boxed{z = \Big] f^{!(\sigma \multimap \tau)} \Big[_{!}\ \text{in}\ \lambda s'.\,(z\, s')} \Big[_{!}\ \text{in}\ \ (y\, x)\,\Big].$$

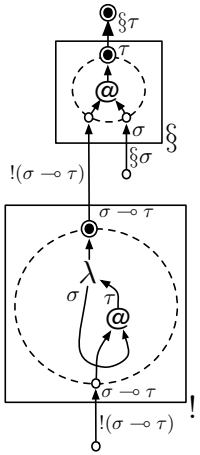


Figure 2.1.: A proof net example for LAL.

Of course, there are certain restrictions, i.e. not every oriented graph constructed with the available nodes is a proof net. Because our proof nets are quite simple, we can use a term system with usual typing rules and a translation of terms into these graphs. In other words, a given oriented finite graph is a proof net if it is the image of a term under this translation. The translation will be very similar to that which maps terms to their parse DAGs (directed acyclic graphs).

In fact, for proof nets of (Linear) System T terms, these parse DAG would be sufficient. For term systems which have multiplexers it is a bit more complicated, because the translation will not be injective anymore, but maps several terms to the same graph. In this context, an alternative view of proof nets is that of equivalence classes, that consist of those sets of terms which lead to the same graph, e.g. all those which are equal modulo certain permutative conversions.

2.3.1.1. In the Context of the Literature

Before starting the formal definition, let us put our proof nets into the context of the large amount of material available in the area of Linear Logic, and also in some sense of optimal reductions of lambda terms. Our proof nets are in fact a very simple variation of the idea to use labelled, directed graphs to represent proofs.

The common motivation to do this at all is the desire to "talk" about a proof, but in a way that structural details, like the order of the application of certain rules, do not matter, i.e. are not visible in the structure. E.g., when applying two weakenings (the introduction of new assumptions in sequent calculus), the proofs are essentially the same, independent of the order in which these two rules are applied. A proof net neglects this by the so called desequentialisation. Both proofs would lead to the same proof net.

The drawback of the use of proof nets is that their definition is not inductive anymore. Using inductive clauses for that would assign an ordering again, whereas avoiding this was the whole point of proof nets in the beginning. For the theory of proof nets it is an important and far from trivial question which kind of graphs (or proof structures) are really proof nets, i.e. to which proof nets can one assign an (inductively defined) proof. A good survey on this topic is [Gue04].

In this thesis we will not use proof nets for deep structural proof theory. Instead, we will adopt a very easy variant of proof nets, in order to normalise programs with better control over shared subterms than one would have in a traditional lambda calculus. For instance, we only consider intuitionistic proof nets, and therefore we do not need advanced tools like *polarisation* of links or multiple output links. Instead, one can see our proof nets as parse trees (i.e. directed acyclic graphs) in the style of [SB01], but with explicit duplication via multiplexer nodes and with the extension of boxes and the nesting of proof nets via these boxes. In the terminology of [Gue04] those graphs are called λ-nets with explicit *sharing* via multiplexers.

On the other hand, we do not go that far to talk about *optimal reduction*s here, i.e. the optimal avoidance of redex duplication, as it is done in [Gue99]. For that one needs two kinds of multiplexers with both polarities/orientations. Here, we will consider only multiplexers with one input and two output links. This greatly reduces the number of possible redexes for the multiplexers themselves, i.e. only the $(\lambda - up)$-rule of [Gue99] is

needed in our context. We call this redex the *polynomial redex*.

2.3.2. Term System

In the following, we will introduce the term system of Light Affine Logic. The image of the terms under a mapping into *proof net structures* will later define the proof nets. Moreover, the terms will serve as a notation system for proofs in the language of proof nets. The latter usually get big easily and hence writing down programs is not very comfortable. The term system presented here describes proof nets in the sense that the terms are one sequentialisation of a proof net. Therefore, direct normalisation of these terms would introduce the problems of *permutative conversions* again. But, we will not use the terms for that purpose:

Our model of computation is the proof net, not the term system.

Definition 2.23 (Terms)**.** For a countably infinite set V of variable names, the set of light linear terms $\mathrm{Tm}_{\mathrm{LAL}}$ is defined inductively by:

$$r, s, t ::= x \mid \lambda x.t \mid !\boxed{t} \mid !\boxed{x =]s[_! \text{ in } t} \mid \S\boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_\S}, \overrightarrow{]s[_!} \text{ in } t}$$
$$\mid \Lambda\alpha.t \mid (s\,t) \mid (s\,\tau) \mid \left(s \lhd^{x_1}_{x_2} t\right).$$

with types $\tau \in \mathrm{Ty}_{LAL}$, type variables $\alpha \in T$ and $x, y, x_1, x_2 \in V$. Terms which are equal up to the naming of bound variables and bound type variables are identified.

Variables Free and bound variables are defined as for System F in Definition 2.14 plus the cases

$$\mathrm{FV}(!\boxed{t}) := \mathrm{FV}(t)$$
$$\mathrm{FV}(!\boxed{x =]s[_! \text{ in } t}) := \mathrm{FV}(s) \cup (\mathrm{FV}(t) \setminus \{x\})$$
$$\mathrm{FV}(\S\boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_\S}, \overrightarrow{]s[_!} \text{ in } t}) := \mathrm{FV}(\overrightarrow{r}) \cup \mathrm{FV}(\overrightarrow{s}) \cup (\mathrm{FV}(t) \setminus \{\overrightarrow{x}, \overrightarrow{y}\})$$
$$\mathrm{FV}(\left(s \lhd^{x_1}_{x_2} t\right)) := \mathrm{FV}(s) \cup (\mathrm{FV}(t) \setminus \{x_1, x_2\})$$

and mutatis mutandis for FTV.

Subterms Subterms are defined as for System F in Definition 2.14 with the following additional clauses:

$$
\begin{aligned}
t &\lhd_{\mathrm{LAL}} !\boxed{t} \\
s, t &\lhd_{\mathrm{LAL}} !\boxed{x = s \text{ in } t} \\
\overrightarrow{r}, \overrightarrow{s}, t &\lhd_{\mathrm{LAL}} \S\boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_{\S}}, \overrightarrow{]s[_{!}} \text{ in } t} \\
s, t &\lhd_{\mathrm{LAL}} \left(s \lhd_{x_2}^{x_1} t\right).
\end{aligned}
$$

Note that s is not a subterm of t in $!\boxed{x = s \text{ in } t}$. The same applies to $\overrightarrow{r}$ and $\overrightarrow{s}$ in $\S\boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_{\S}}, \overrightarrow{]s[_{!}} \text{ in } t}$ for t. This is why we do not use the notation $!\boxed{t[x :=]s[_{!}]}$ officially. Though, we will write e.g. $!\boxed{\lambda x, y. \ldots]s[_{!} \ldots}$ for $!\boxed{z =]s[_{!} \text{ in } \lambda x, y. \ldots z \ldots}$ as a shorthand. The vector notation $\overrightarrow{x}, \overrightarrow{]r[_{\S}}, \overrightarrow{]s[_{!}}$ means, that an arbitrary (also zero) finite number of those assignments can be used, e.g. $\S\boxed{x, x', y =]r[_{\S},]r'[_{\S},]s[_{!} \text{ in } t}$.

Definition 2.24 (Typing)**.** A context is a multiset of type assignments from the variable names V to types, with the context condition that every variable is not assigned different types. The relation $\Gamma \vdash t^{\tau}$ is inductively defined as follows:

$$\frac{}{\Gamma, x^{\sigma} \vdash x^{\sigma}} \ (\mathrm{Var})$$

$$\frac{\Gamma, x^{\sigma} \vdash t^{\tau}}{\Gamma \vdash (\lambda x.t)^{\sigma \multimap \tau}} \ (\multimap^{+}) \qquad \frac{\Gamma_1 \vdash t^{\sigma \multimap \tau} \quad \Gamma_2 \vdash s^{\sigma}}{\Gamma_1, \Gamma_2 \vdash (t\, s)^{\tau}} \ (\multimap^{-})$$

$$\frac{\emptyset \vdash t^{\tau}}{\emptyset \vdash !\boxed{t}^{!\tau}} \ (!_0)$$

$$\frac{x^{\sigma} \vdash t^{\tau} \quad \Gamma \vdash s^{!\sigma}}{\Gamma \vdash !\boxed{x =]s[_{!} \text{ in } t}^{!\tau}} \ (!_1)$$

$$\frac{\overrightarrow{x^{\rho}}, \overrightarrow{y^{\sigma}} \vdash t^{\tau} \quad \overrightarrow{\Gamma \vdash r^{\S\rho}} \quad \overrightarrow{\Sigma \vdash s^{!\sigma}}}{\overrightarrow{\Gamma}, \overrightarrow{\Sigma} \vdash \S \boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_{\S}}, \overrightarrow{]s[_{!}} \text{ in } t}^{\S\tau}} \ (\S)$$

$$\frac{\Gamma \vdash t^{\tau}}{\Gamma \vdash \Lambda\alpha.t^{\forall\alpha.\tau}} \ (\forall^{+}) \qquad \frac{\Gamma \vdash t^{\forall\alpha.\tau}}{\Gamma \vdash (t\,\sigma)^{\tau[\alpha:=\sigma]}} \ (\forall^{-})$$

$$\frac{\Gamma_1 \vdash s^{!\sigma} \quad \Gamma_2, x_1^{!\sigma}, x_2^{!\sigma} \vdash t^{\tau}}{\Gamma_1, \Gamma_2 \vdash \left(s \lhd^{x_1}_{x_2} t\right)^{\tau}} \ (C)$$

with types $\sigma, \tau, \rho \in \mathrm{Ty}_{\mathrm{LAL}}$, contexts $\Gamma, \Gamma_1, \Gamma_2$, terms $t, s \in \mathrm{Tm}_{\mathrm{LAL}}$, $\alpha \in T$ and $x, y, x_1, x_2 \in V$. The notation Γ, Λ is the multiset union maintaining the context condition.

- In the (§)-rule either vector can be empty, and the types σ and ρ can be different for every vector component.
- In the rules $(!_0)$,$(!_1)$,(§) and (C) the variables $x, y, x_1, x_2, \overrightarrow{x}, \overrightarrow{y}$ must be fresh for the other contexts involved.
- In the $(\forall^{+})$-rule the type variable α *must not* be free in the types of Γ.
- In the $(\multimap^{+})$-rule the variable x *must not* be in Γ.

Definition 2.25 (Level of a subterm)**.** For a term w with $\Gamma \vdash w : W$ every subterm v of w is assigned a level in the following way:

- The level of w is 0 in w.
- If $!\boxed{t}$ is on level n in w, then t is on level $n+1$ in w.
- If $!\boxed{x =]s[_{!} \text{ in } t}$ is on level n in w, then t is on level $n+1$ in w and s of level n in w.
- If $\S\boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_{\S}}, \overrightarrow{]s[_{!}} \text{ in } t}$ is on level n in w, then t is on level $n+1$ in w and each $\overrightarrow{r}, \overrightarrow{s}$ of level n in w.

- In all other inductive clauses of $\lhd$ the level of the subterm $s \lhd t$ in w is the level of t in w.

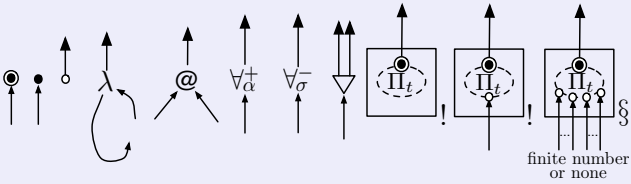
The maximal level L_t of a well-typed term t is the maximal level in t of its subterms.

Remark 2.26. The level represents the number of boxes around a term, while the terms r, s in $]s[_!\,,]r[_§$ in the $(!_1)$- and (§)-rules are seen as being outside the box, hence the inverse brackets should look like holes in the boxes. Therefore, we call $]s[_!$ the !-hole and $]r[_§$ the §-hole of a box $§\boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_§}, \overrightarrow{]s[_!} \text{ in } t}$ or $!\boxed{x =]s[_! \text{ in } t}$.

2.3.3. Proof Nets Formally

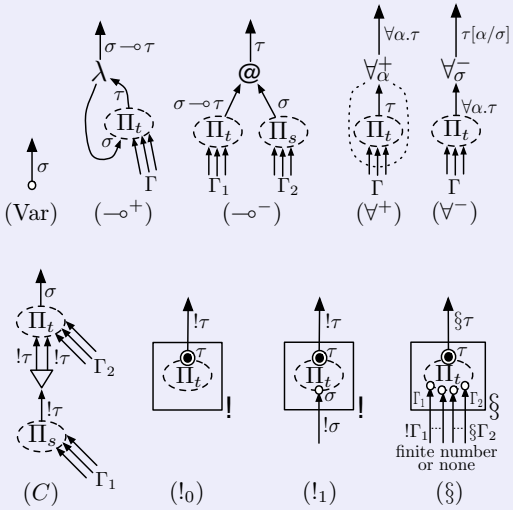
For each typing rule of Definition 2.24 we will now give a translation into a subgraph, such that the image of a typing derivation gives a proof net graph, in the following sense:

Definition 2.27 (Proof net structure)**.** A proof net structure is a labelled finite directed graph built from links (edges) and the nodes



with their given in- and output degrees and the following properties:

1. Each link is labelled with a type in $\mathrm{Ty}_{\mathrm{LAL}}$, satisfying the following restrictions given by the type annotations of the nodes' in- and output links with $\sigma, \tau \in \mathrm{Ty}_{\mathrm{LAL}}$, $\alpha \in T$:



2. There is exactly one ◉ node.
3. There is a closed $\cdots$ path around the input link of the $\forall^+_\alpha$ node, such that no σ-link with $\alpha \in \mathrm{FTV}(\sigma)$ crosses it and no input port lays within (and no node other than $\forall^+_\alpha$ lays on it).
4. The Π_t in every !- or §-box is a proof net structure with the principal port ◉ and all the ∘-, ◉-nodes laid out as displayed.

We call

- the ∘-nodes input ports or free variables,
- the •-nodes output ports or weakening ports,
- the ◉-node the principal output port of the proof net structure,
- the output link of type σ of a λ-node, with another output link $\sigma \multimap \tau$, binding link or binding port,
- and the $\bigtriangledown$-nodes multiplexers.

The type of a (non-output) node is the type of its (non-binding) output link.
The type of an output node is the type of its input link.
For ($!_0$), ($!_1$) and (§) we say that Π_t is the the proof net structure of the !- or §-box.
A proof net structure is called closed if it has no input ports.
Nodes without a path to the principal node are called garbage. A proof net structure Π' is called cleanup of Π if Π' is created from Π by removing all garbage nodes and closing potential binding links or multiplexer output links with a weakening port.

Remark 2.28. The binding link for a λ-node is well defined by the type annotations.

Definition 2.29 (Subproof net structure)**.** A subproof net structure Π' of a proof net structure Π is either

- derived from a subgraph of Π with edges, that lay only "half" inside, connected to additional input and (principal) output ports or
- a *subproof net structure* of a proof net structure Π_t of a box in Π.

This definition extends the concept of a subgraph to the nested boxes, i.e. a subproof net structure can also be inside a box or at an even deeper nested position in a proof net structure.

The typing rules in Definition 2.24 correspond *one-to-one* to the rules given in Definition 2.27. Hence, we can map each typing derivation to a proof net using the canonical mapping:

Definition 2.30 (Proof net)**.** A proof net structure Π_t is called a proof net for t if it is the image of the typing derivation $\Gamma \vdash t^\tau$ under the mapping which translates each term typing rule of Definition 2.24 to the corresponding proof net structure rule in Definition 2.27.

The output link coming from the conclusion of the typing rule at the root of $\Gamma \vdash t^\tau$ is marked as principal node ◉. Free variables are turned into input port, not appearing bound variables into non-principal output ports.

We call a proof net Π a proof net of type τ (denoted with Π^τ) if τ is the type of the principal output node.

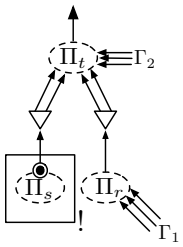
A subproof net of a proof net for the term t^τ is a proof net of a subterm $s \trianglelefteq t$.

Remark 2.31. The translation of the arrow introduction $(\multimap^+)$ that types $\Gamma \vdash \lambda x^\sigma.t^{\sigma \multimap \tau}$ with x *not free* in t uses a weakening port for the loose end of the binding link. In the type system of the terms we do not use explicit weakening, though, for sake of simplicity.

Because the content t of a box $!\boxed{x =]s[_! \text{ in } t}$ is a subterm with $x^\sigma \vdash t^\tau$ as the premise in the typing derivation, t will also map to a proof net of a box. Hence, a proof net structure of a box is also a proof net.

Remark 2.32. The central difference between the terms and the proof nets is the *desequentialisation* of the multiplexer rules. Compare the translation of the multiplexer in the term language to the multiplexer of the proof net in Figure 2.2 on the next page. While in a term the position of the contraction rule in the typing derivation is explicit, in the proof net it is not, because the proof net is just the graph. The (permutative) order of the contraction (C) and other rules for instance is lost there.

The proof net structure rules, though, induce an inductive structure over those graphs, like type derivations do for terms. But due to the lost order, there can be multiple terms for one proof net, i.e. also different inductive structures over a proof net. E.g., in the term system $\left(r^{!\rho} \lhd^{x_1}_{x_2} \left(!\boxed{\mathbf{s}} \lhd^{\mathbf{y_1}}_{\mathbf{y_2}} \mathbf{t}\right)\right)$ and $\left(\left(r^{!\rho} \lhd^{x_1}_{x_2} !\boxed{\mathbf{s}}\right) \lhd^{y_1}_{y_2} \mathbf{t}\right)$ are different, whereas as proof nets they are the same:



In this sense, proof nets are desequentialisations of type derivations. In Remark 6.28 we will go into more detail on this in the context of $\mathrm{LLT}_!$.

Because the translation from terms to proof nets is mostly (up to the lost order of the contraction rules) just that of creating a parse tree, we get the following connection between subproof net structures and subterms:

Fact 2.33. *Let Π_s be the subproof net of a proof net Π_t, Π_t the proof net for the term t^τ and $s \trianglelefteq t$ the subterm for Π_s. Then Π_s is a subproof net structure of Π_t.*

A proof net, as an image of a typing derivation, has many nice properties:

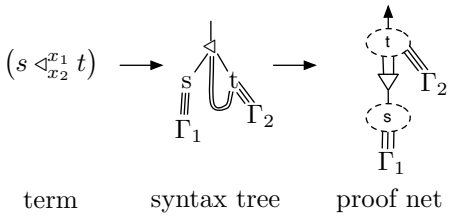


Figure 2.2.: Translation of multiplexers from LAL terms to proof nets

Definition 2.34 (Paths and proper paths)**.** A sequence of proof net links which can also go through boxes and the nested proof nets is called a proof net path (short just path). A path *which does not pass through a λ-node via the binding link* is called proper.

Fact 2.35.

1. *Each maximal proper path in a proof net ends either in a weakening port or in the principal port,*
2. *Every path in a proof net starting with the binding link of a λ-node either ends in a weakening port or passes through the input link of the starting λ-node,*
3. *There is no proper cycle in a proof net.*

Remark 2.36. The second property of the previous definition is part of what Schwichtenberg and Bellantoni [SB01] call *conformal graph*s. This essentially makes sure that the λ only binds variables in its body.

Remark 2.37. A proof net can have loops. These loops always go through a λ-node. If the binding links are "cut" and replaced with named variables (as done e.g. in [SB01]), the proof nets become trees with sharing that is made explicit via the multiplexers. Schwichtenberg and Bellantoni [SB01] call these *parse DAGs* (directed acyclic graphs) with *sharing*. We do not want to talk about variable names here. Therefore, we do not "cut" the binding links.

Notation 2.38. We use the term nodes of a proof net structure Π for the graph-theoretic nodes of Π.

We use the term nested nodes of a proof net structure Π

- for nodes of Π
- and for *nested nodes of a proof net structure of a box of* Π.

The nesting box structure of a proof net suggests a classification of nodes and redexes by the number of boxes around them. Therefore, we introduce the concept of a level of nodes in a proof net.

Definition 2.39 (Level and node of level in a proof net)**.**

1. If a box node b with Π_t as the proof net of b is a node in the proof net Π, then every node of level n in Π_t is of level $n+1$ in Π.
2. If n is a node of the proof net Π, then n is of level 0 in Π.

The maximal level n with nodes of level n in Π is denoted by L_Π.
The level of a subproof net of Π is the level of its principal node in Π.

Note that every node x of some level n in Π is also a *nested node* of Π. In fact, even the converse is true. The nested nodes of Π are exactly all nodes of arbitrary level in Π.

Compare this definition with the *level of a subterm* in Definition 2.25. Both concepts count the number of boxes around a subterm or subproof net and are equal in the sense that a subterm $s \trianglelefteq t$ is of level n iff the corresponding subnet Π_s in Π_t is also of this level. In the context of proof nets, the connection to the box nesting depth is more obvious, because a subterm s in the "hole" $]s[_!$ of a box lays clearly outside the box in the proof net representation of the same term.

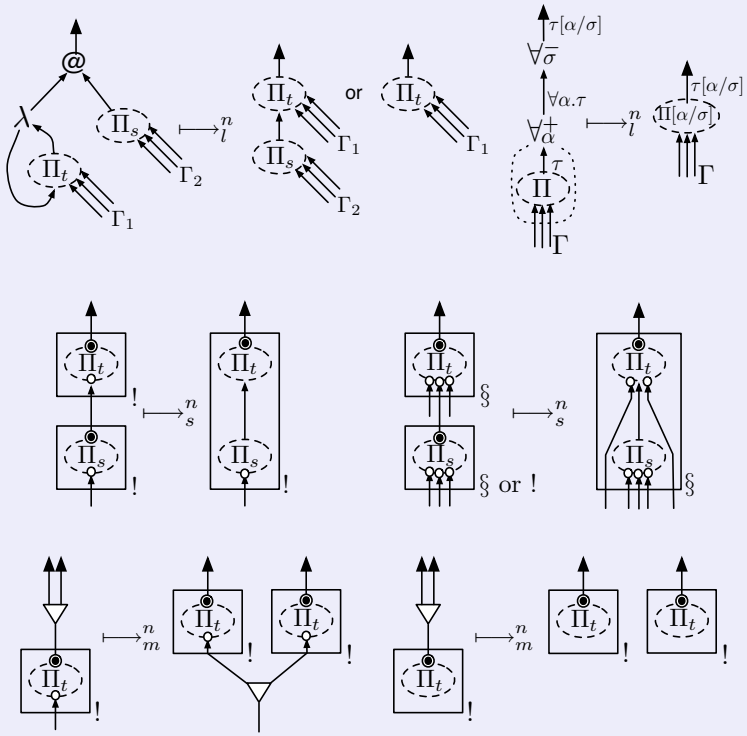
2.3.4. Normalisation

Normalisation is done in the proof nets and not in the terms. It is defined by local graph rewrite rules, to be applied to the proof net graph and the proof net graphs of the boxes and so on. Clearly, a formal and complete study of proof nets would need to show that these reductions really lead to proof nets again. We will not do that here though and refer to [AR00, AR02].

Definition 2.40 (Rewriting rules, redexes)**.** The graph rewriting relation between proof nets is split into the following redexes:

- the linear part $\longmapsto_l$ (usual beta reduction and type beta reduction) which decreases the proof net size. If the binding link ends in a weakening port, Π_s disappears,
- the shifting part $\longmapsto_s$ (box merging), which also reduces the proof net size,

- and the polynomial part $\longmapsto_m$ (duplication), which makes copies of boxes and hence increases the proof net size.



We call $\longmapsto_p^n := \longmapsto_s^n \cup \longmapsto_m^n$ polynomial redexes and $\longmapsto := \longmapsto_l \cup \longmapsto_s \cup \longmapsto_m$ arbitrary redexes.
We write $\Pi_t \longrightarrow \Pi_{t'}$ if

- $\Pi_s \longmapsto \Pi_{s'}$ for some subproof net Π_s of Π_t,
- and $\Pi_{t'}$ is Π_t with Π_s replaced by $\Pi_{s'}$ (and mutatis mutandis $\longrightarrow_\star$ for $\longmapsto_\star$-redexes) *after cleaning up garbage nodes.*

A proof net Π_t is called $\longmapsto_\star$-normal or $\longrightarrow_\star$-normal if there is no $\Pi_{t'}$ with $\Pi_t \longrightarrow_\star \Pi_{t'}$. The notation $\xrightarrow{\text{nf}}_\star$ denotes the relation which puts a term in relation with its normal form(s), i.e.

$$\Pi \overset{\text{nf}}{\sim} \Pi' :\Longleftrightarrow \Pi \sim^* \Pi' \wedge \Pi' \text{ is } \sim\text{-normal}.$$

The level of a redex $\Pi_s \longmapsto_\star \Pi_{s'}$ is the level of Π_s (which is the same as the level of $\Pi_{s'}$). We write $\longmapsto_\star^n$ for $\longmapsto_\star$ (and $\longrightarrow^n$ for $\longrightarrow$) restricted to level n redexes.

It is clear from the definition of the graph rewriting rules that the nesting of the proof nets with boxes does not change during normalisation because the rewrite rules only apply to one level at a time (with the exception of shifting a redex $\longmapsto_s$ which glues two boxes together; this though does not make any trouble either). Hence:

Fact 2.41. *During normalisation via $\longrightarrow$ the level of the nodes in a proof net Π is constant.*

This gives rise to the *normalisation by levels*:

Definition 2.42 (Normalisation by levels)**.** For a proof net Π we define the normalisation by levels

$$\longrightarrow_{NbL} := \prod_{n:=0}^{L_\Pi} (\xrightarrow{\text{nf}}_l{}^n \ \xrightarrow{\text{nf}}_p{}^n).$$

In other words, *normalisation by levels* first normalises outside boxes and then continues with redexes inside. For this reason, it is also called *"from outside to inside" normalisation*.

Normalisation by levels is a good strategy in fact, i.e. it is not any worse qualitatively than normalising in another, not-level-driven strategy (in the sense that it can lead to a normal form if there is one). The reason is that the rewrite rules are formulated in such a way that redexes do not make any assumptions about nodes on different levels, i.e. inside boxes or outside the box a redex is in. Of course, normalisation by levels duplicates boxes as early and as often as possible. Hence, complexity-wise it is not very good because a lot of redexes are duplicated.

For normalisation by levels it is essential that no new outer redexes can be created during reduction in boxes:

Fact 2.43. *Let* Π *be normal for the levels* $\leq n$ *and* $\Pi \longrightarrow^{n+1} \Pi'$. *Then also* Π' *is normal for levels* $\leq n$.

In other words, firing a redex on level $n+1$ does not create new redexes on lower levels (i.e. further outside boxes).

Corollary 2.44. *If there is a normal form of a proof net* Π *according to* $\longrightarrow$, *then this normal form can also reached by a strategy in* $\longrightarrow_{NbL}$.

To sum up, the considered normalisation for proof nets in LAL is the following:

- We start with the linear redexes on the "outside",
- Next, we fire the shifting and polynomial redexes on the "outside".
- Then, after the outer level 0 is normal, we continue with level 1,
  - again, first the linear redexes on level 1,
  - then the shifting and polynomial redexes on level 1.
  - Then, after level 1 is normal, we continued with level 2.
    - And so on...

It has been shown by Asperti and Roversi [AR00, AR02] that normalisation by levels for Light Affine Logic is done in polynomial many steps and that LAL is in fact complete for PTime:

Definition 2.45 (Proof net size)**.** The size of a proof net $|\Pi|$ is the number of nodes in Π plus the sum of $|\Pi_i|$ of the proof nets Π_i in the box nodes v_i of Π.

Theorem 2.46 (Completeness for Ptime, Asperti-Roversi)**.** *Every Turing Machine* M *which terminates after polynomially many steps in the size of the input can be simulated by a LAL proof net* Π_M *of type* $\sigma \multimap \tau$ *with*

- *the input of the Turing Machine (i.e. the input tape) coded with proof nets* Π_{in} *of type* σ *with* $L_{\Pi_{in}} \leq L_{\Pi_M}$
- *and the output of* M *coded as proof nets of type* τ.

Theorem 2.47 (Correctness for PTime, Asperti-Roversi)**.** *Every proof net* Π *of LAL can be normalised in* $\mathcal{O}(|\Pi|^{6^{L_\Pi}})$.

We will use essentially the proof idea of [AR00, AR02] as a starting point to prove the complexity bound for Light Linear T in Chapter 6, and also for LLFPL$_!$ in Chapter 7.

Furthermore, it was shown by [Ter01] that LAL is even strongly normalising in polynomial time, i.e. by any normalisation order, not only by levels. But we will not need this result in this work.

Remark 2.48. The restriction to the maximal level $L_{\Pi_{\text{in}}} \leq L_{\Pi_M}$ is essential in Theorem 2.46. The normalisation complexity is polynomial, but the order of the polynomial depends on the maximal level of the proof net which is normalised, i.e. Π_M applied to the input Π_{in}. Hence, it is important to bound the levels which appear in the input Π_{in}. It is easily possible otherwise to define inputs which, although they are all of the same type, have increasingly high levels and therefore can normalise in exponential time in their size. This way even exponential time Turing Machines could be simulated.

2.3.5. Encodings and Polynomial Time

In [Lag03] and [LB06] different kinds of encodings for natural numbers are studied. The question is which subsystems of full second order Light Linear Logic with products, fixed point types, product and co-product are sound and/or complete for PTime.

The result of the former reference is mainly that one has to restrict the use of the $(\forall^-)$- and $(\forall^+)$-rule to types whose nesting of modalities is bound by the nesting in the conclusion and premises. In other words, the inputs as cut-free proofs must not have nodes of higher level than the nesting depths in the input type. To show this necessity, a uniform input type for natural numbers is given having cut-free proofs without this property.

In the second reference, the follow-up paper, this idea is extended by not restricting the kind of cut-free input proofs, but by restricting the logic. Namely, the ∀-rules are restricted to linear types (i.e. those without ! or § in them). It is further shown that such a subsystem of Light Linear Logic can still be complete for PTime, while also being sound for any type for input and output.

2.3.6. Light Affine Logic with Fixed Points (μLAL)

As Dal Lago and Baillot [LB06] point out, the normalisation process of LAL does not make essential use of the types in a proof net other than the structure of the boxes which can be read off from the modalities. Hence, it can be asked which interesting extensions of the type system are possible without breaking the normalisation proof. Dal Lago and Baillot [LB06] present an extension with *fixed point types* ($\mu\alpha.\tau$), the corresponding term constructs to fold and unfold terms and the fold- and unfold-typing rules:

Definition 2.49 (Types)**.** For a countably infinite set T of type variable names the set of light linear types with fixed points $\mathrm{Ty}_{\mu\mathrm{LAL}}$ is defined inductively by:

$$A, B ::= \alpha \mid A \multimap B \mid !A \mid \S A \mid \forall\alpha.A \mid \mu\alpha.A.$$

Definition 2.50 (Terms)**.** For a countably infinite set V of variable names the set of light linear terms $\mathrm{Tm}_{\mu\mathrm{LAL}}$ is defined inductively by:

$$r, s, t ::= x \mid \lambda x.t \mid ![\,t\,] \mid ![\,x =]s[_{!}\ \text{in}\ t\,] \mid \S[\,\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_{\S}}, \overrightarrow{]s[_{\S}}\ \text{in}\ t\,]$$
$$\mid \Lambda\alpha.t \mid (s\,t) \mid (s\,\tau) \mid \left(s \lhd^{x_1}_{x_2} t\right) \mid \{t\} \mid \}t\{$$

with types $\tau \in \mathrm{Ty}_{\mu LAL}$, type variables $\alpha \in T$ and $x, y, x_1, x_2 \in V$. Terms which are equal up to the naming of bound variables and bound type variables are identified.

Variables Free and bound variables are defined as for LAL in Definition 2.23 plus the cases $\mathrm{FV}(\{t\}) := \mathrm{FV}(\}t\{) := \mathrm{FV}(t)$ and mutatis mutandis for FTV.

Subterms Subterms are defined as for LAL in Definition 2.23, with the additional clauses $t \lhd_{\mu\mathrm{LAL}} \{t\}$ and $t \lhd_{\mu\mathrm{LAL}} \}t\{$.

Definition 2.51 (Typing)**.** The typing rules are as in Definition 2.24 for LAL with the following additional rules:

$$\frac{\Gamma \vdash t^{\tau[\alpha:=\mu\alpha.\tau]}}{\Gamma \vdash \{t\}^{\mu\alpha.\tau}}\ (\text{Fold}) \qquad \frac{\Gamma \vdash t^{\mu\alpha.\tau}}{\Gamma \vdash \}t\{^{\tau[\alpha:=\mu\alpha.\tau]}}\ (\text{Unfold})$$

The corresponding "left" rules for fixed points, which operate on the context, are derivable as usual. We refer to [LB06] for more details.

3

Building an Intuition by Examples

In this chapter, we take a collection of algorithms, which are either exponential per se or at least exponential with a badly chosen normalisation strategy. We explore how the different polynomial time type systems of Chapter 2 handle these, i.e. how the algorithms are outlawed or how a good normalisation strategy is enforced to avoid the exponential normalisation time.

It is not our goal to give a full study or formal comparison of the used calculi. Instead, the examples, seen side by side in the different settings, should help the reader build an intuition on the ideas involved to restrict the definable algorithms in a type system to polynomial time.

A real deep study of these ideas, their formalisation and the formal complexity proofs, is much more involved and mostly a very technical matter. These technicalities usually make it very hard to understand the presented system up to every detail. Here however, we want to stress the underlying ideas, how the the ideas are expressed in the calculi and how the systems outlaw certain bad algorithms.

Structure of this chapter We divide the considered examples in this chapter into three classes:

- Section 3.1 introduces the representation of the basic data types of booleans and products in the different systems and studies examples

where the normalisation complexity mainly depends of the properties of these data types.

- Section 3.2 analysis different recursion schemes which allow the definition of exponential functions. These algorithms are exponential "per se" and they must be outlawed independently from any normalisation strategy.

- Section 3.3 implements real-world polynomial time computable functions in order to compare the intentional expressivity of the considered calculi.

In Section 3.4 this chapter ends with a conclusion and an outlook on the following chapters of this work, by a collection of questions arising from the example and references to the respective chapter which will give answers to them.

3.1. Booleans and Products

We will start looking at very basic data types: the booleans and the product. The System T based calculi usually have them "hard-coded" as base types, and the typing rules and reduction rules enforce certain properties. System F based system often use the implicit Church encoding:

Definition 3.1. *The Church encoding in System F* of booleans and products (compare [GLT88]):
Boolean: $B := \forall\alpha.\alpha \to \alpha \to \alpha$
Product: $\sigma \otimes \tau := \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha$

In Light Linear Logic things are not much different. The only difference is that the linear arrows $\multimap$ are used instead of $\to$. The properties of the System F product and the Light Linear Logic product will be different due to the linearity constraints:

Definition 3.2. *Church encoding in Light Affine Logic* of booleans and products (compare [AR00]):
Boolean: $B := \forall\alpha.\alpha \multimap \alpha \multimap \alpha$
Product: $\sigma \otimes \tau := \forall\alpha.(\sigma \multimap \tau \multimap \alpha) \multimap \alpha$

Example 3.3. With the linearity requirement we have to do some extra work during destruction of a boolean (i.e. the usual if-then-else construct in a programming language) if the two alternatives $s[x]$ and $t[x]$ share free variables, here the x. This situation is not allowed in Light Linear Logic, but of course it is not hard to "abstract them out":

$$\text{if}\, b^B \,\text{then}\, s[x] \,\text{else}\, t[x] := (((b\, \sigma \multimap \tau)\ \lambda x.s[x]\, \lambda x.t[x])\ x)\,.$$

with $B := \forall\alpha.\alpha \multimap \alpha \multimap \alpha$. For each free variable one adds another arrow to the type $\sigma \multimap \tau$ and another λ-abstraction in the then- and else-cases. While not very complicated, this should illustrate the work one often has to do in a linear type system to make an algorithm typable.

3.1.1. Data Types in the Different Calculi

In the following subsections we will give an overview of the base types and products, starting with the System T based ones, which have them build into the term language and the types:

3.1.1.1. LFPL

We repeat some parts of the definition of LFPL in Section 2.2:
Types:

- boolean: B
- tensor product: $\sigma \otimes \tau$
- cartesian product: $\sigma \times \tau$
- lists: $L(\tau)$
- diamond: $\diamondsuit$

Constructor:

- $\mathbf{tt}, \mathbf{ff}$ (as constants)
- $\otimes_{\tau,\rho} : \tau \multimap \rho \multimap \tau \otimes \rho$ (as constant)
- $\langle t, s\rangle^{\tau\times\sigma}$ (as term construct)
- $\mathbf{nil}_\tau : L(\tau)$ and $\mathbf{cons}_\tau : \diamondsuit \multimap \tau \multimap L(\tau) \multimap L(\tau)$.

Destructors:

- $\left(b^B\, p^{\tau\times\tau}\right)^\tau$ (as term construct)
- $\left(p^{\rho\otimes\sigma}\, f^{\rho\to\sigma\to\tau}\right)^\tau$ (as term construct)
- $\left(\langle s,t\rangle\,\mathbf{tt}\right), \left(\langle s,t\rangle\,\mathbf{ff}\right)$ (as term construct)
- iteration: $\left(l^{L(\tau)}\, f^{\Diamond\multimap\tau\multimap\sigma\multimap\sigma}\right)^{\sigma\multimap\sigma}$ with closed f.

Typing rules:

$$\frac{\Gamma\vdash s^\sigma \quad \Gamma\vdash t^\tau}{\Gamma\vdash\langle s,t\rangle^{\sigma\times\tau}}\ (\times^+) \qquad \frac{\Gamma\vdash p^{\rho\times\sigma}}{\Gamma\vdash(p\,\mathbf{tt})^\rho}\ (\times^-_{\mathbf{tt}})$$

$$\frac{\Gamma\vdash p^{\rho\times\sigma}}{\Gamma\vdash(p\,\mathbf{ff})^\sigma}\ (\times^-_{\mathbf{ff}})$$

$$\frac{\Gamma_1\vdash b^B \quad \Gamma_2\vdash p^{\tau\times\tau}}{\Gamma_1,\Gamma_2\vdash(b\,p)^\tau}\ (B^-)$$

$$\frac{\Gamma_1\vdash p^{\rho\otimes\sigma} \quad \Gamma_2\vdash f^{\rho\to\sigma\to\tau}}{\Gamma_1,\Gamma_2\vdash(p\,f)^\tau}\ (\otimes^-)$$

$$\frac{\Gamma_1\vdash l^{L(\tau)} \quad \emptyset\vdash h^{\Diamond\multimap\tau\multimap\sigma\multimap\sigma} \quad \Gamma_2\vdash g^\sigma}{\Gamma_1,\Gamma_2\vdash(l\,h\,g)^\sigma}\ (\mathrm{It})$$

Restrictions:

Normalisation under λ and inside a term $\langle t,s\rangle$ is not allowed. The linearity restriction does not apply to $\langle t,s\rangle$, i.e. both branches can share free variables.

3.1.1.2. LT

The calculus LT is a higher-type variant of BC, i.e. Bellantoni and Cook's safe recursion. Compare [SB01] and [Sch06] for a complete definition. We will not formally introduce it here because it will only play a role in the examples of this chapter, for comparison to the other type system. Instead, we list only the main features:

Types:

- boolean: B

- tensor product: $\sigma \otimes \tau$
- cartesian product: $\sigma \times \tau$
- lists: $L(\tau)$

Constructor:

- $\mathbf{tt}, \mathbf{ff}$ (as constants)
- $\otimes^{+}_{\sigma,\tau} : \sigma \multimap \tau \multimap \sigma \otimes \tau$ (as constant)
- $\times^{+}_{\sigma,\tau,\rho} : (\rho \multimap \sigma) \multimap (\rho \multimap \tau) \multimap \rho \multimap \sigma \times \tau$ (as constant)
- $\mathbf{nil}_{\tau} : L(\tau)$ and $\mathbf{cons}_{\tau} : \tau \multimap L(\tau) \multimap L(\tau)$.

Destructors:

- $\mathbf{Case}_{\rho,\tau} : L(\rho) \multimap \tau \times (\rho \multimap L(\rho) \multimap \tau) \multimap \tau$ with linear τ (as constant)
- $\otimes^{-}_{\sigma,\tau,\rho} : \sigma \otimes \tau \multimap (\sigma \multimap \tau \multimap \rho) \multimap \rho$ (as constant)
- projections: $\mathbf{fst}_{\sigma,\tau} : \sigma \times \tau \multimap \sigma$, $\mathbf{snd}_{\sigma,\tau} : \sigma \times \tau \multimap \tau$ (as constants)
- recursion: constant $\boldsymbol{\mathcal{R}}_{\sigma,\tau} : L(\sigma) \to (\sigma \to L(\sigma) \to \tau \multimap \tau) \to \tau \multimap \tau$ with linear τ, i.e. no arrow $\to$ in τ.

Restrictions:
There is no explicit restriction of the normalisation strategy in the definition of the term system and its reductions. Though, the complexity theorem in [SB01] chooses one in the correctness proof. This strategy then is polynomial in the input size.

3.1.1.3. LAL

Light Affine Logic is formally introduced in Section 2.3. For the examples below we use the following additional definitions:
Types:

- boolean: $B := \forall\alpha.\alpha \multimap \alpha \multimap \alpha$
- tensor product: $\sigma \otimes \tau := \forall\alpha.(\sigma \multimap \tau \multimap \alpha) \multimap \alpha$
- cartesian product $\sigma \times \tau$: see Section 3.1.2 below

- lists: $L(\tau) := \forall\alpha.!(\tau \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$
- numerals: $N := \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$.

Constructor:

- $\mathbf{tt} := \Lambda\alpha.\lambda x, y.x$, $\mathbf{ff} := \Lambda\alpha.\lambda x, y.y$
- $\otimes^+_{\sigma,\tau} := \lambda s^\sigma, t^\tau.\Lambda\alpha\lambda f^{\sigma\multimap\tau\multimap\alpha}.\,(f\,s\,t)$
- $\times^+_{\sigma,\tau} := \lambda s, t.\langle s, t\rangle$: see Section 3.1.2 below.

Destructors:

- $\mathbf{Case}_\rho := \lambda b\lambda\mathrm{then}^\rho, \mathrm{else}^\rho.\,(b\,\rho\,\mathrm{then}\,\mathrm{else})$
- $\otimes^-_{\sigma,\tau,\rho} := \lambda p^{\sigma\otimes\tau}, f^{\sigma\multimap\tau\multimap\rho}.\,(p\,\rho\,f)$
- projections $\pi_{\sigma,\tau,0}$, $\pi_{\sigma,\tau,1}$: see Section 3.1.2 below.

Restrictions:

Normalisation is done by levels, i.e. from outside to inside boxes as presented in Definition 2.42. It is shown by Terui [Ter01], that not even this restriction is important and that LAL even normalises in polynomial time with an arbitrary normalisation strategy.

3.1.2. Cartesian Product in System F and LAL

Above, we have not given a definition of a cartesian product for Light Affine Logic. This has a reason: it is not obvious how to define it, due to the linearity constraints of the type system. Therefore, we study the possibilities to introduce such a cartesian product in this subsection. Our goal is that the cartesian product

- allows shared variables in both components,
- and has a uniform type, i.e. that is independent of the component terms.

Cartesian Product in System F As an instance of impredicatively defined inductive types, System F has a natural definition for a product:

$$\sigma \otimes \tau = \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha.$$

The projection functions are canonical:

$$\pi_{\sigma,\tau,0} := \lambda p^{\sigma\otimes\tau}.\,(p\,\sigma\,\lambda x^{\sigma}, y^{\tau}.x)$$
$$\pi_{\sigma,\tau,1} := \lambda p^{\sigma\otimes\tau}.\,(p\,\tau\,\lambda x^{\sigma}, y^{\tau}.y)\,.$$

The constructor is straightforward as well:

$$\otimes^{+}_{\sigma,\tau} \quad := \quad \lambda s,t.\Lambda\alpha.\lambda f^{(\gamma\to\sigma)\to(\gamma\to\tau)\to\alpha}.\,(f\,s\,t)$$

such that

$$\left(\pi_{\sigma,\tau,0}\ \left(\otimes^{+}_{\sigma,\tau}\,s\,t\right)\right) \longrightarrow^{*} s$$
$$\left(\pi_{\sigma,\tau,1}\ \left(\otimes^{+}_{\sigma,\tau}\,s\,t\right)\right) \longrightarrow^{*} t.$$

Cartesian Product in LAL - Naively How does the System F cartesian products translate to linear logic? Replacing $\to$ with the linear arrow $\multimap$ gives the type

$$\sigma \otimes \tau = \forall\alpha.(\sigma \multimap \tau \multimap \alpha) \multimap \alpha$$

with the same projections as in the non-linear case.

Take two terms of type σ and τ, e.g. $s := (f\,x^{\gamma})^{\sigma}$ and $t := (g\,x^{\gamma})^{\tau}$. We want to have a pair of s and t. Naively, one gets $\langle s,t\rangle = \Lambda\alpha.\lambda h.\,(h\,s\,t)$. Both s and t have the free variable x^{γ}. Hence, the term $\langle s,t\rangle$ is not linear and therefore not well-typed. In other words, *the cartesian product of System F does not give a cartesian product in Light Linear Logic.*

With some more effort it is possible to define a cartesian product term in LAL of type

$$\sigma \times \tau := ((\gamma \multimap \sigma) \otimes (\gamma \multimap \tau)) \otimes \gamma.$$

But this type is not uniform in the terms, i.e. depending on the (non-linear) free variables of s and t, the type γ will change. The constructor and the projections are as expected:

$$\langle s,t\rangle := ((\lambda c^{\gamma}.s \otimes \lambda c^{\gamma}.t) \otimes c^{\gamma})^{\sigma\times\tau}$$

$$\pi_{\sigma,\tau,0} := \lambda p^{((\gamma \multimap \sigma) \otimes (\gamma \multimap \tau)) \otimes \gamma}.\, (p\, \lambda x, c.\, (x\, \lambda s, t.\, (s\, c)))^{\sigma}$$
$$\pi_{\sigma,\tau,1} := \lambda p^{((\gamma \multimap \sigma) \otimes (\gamma \multimap \tau)) \otimes \gamma}.\, (p\, \lambda x, c.\, (x\, \lambda s, t.\, (t\, c)))^{\tau}\,.$$

Applying this to our example $\langle s, t\rangle$ from above, we get the term

$$((\lambda x^{\gamma}.\, (f\, x^{\gamma}) \otimes \lambda x^{\gamma}.\, (g\, x^{\gamma})) \otimes x^{\gamma}).$$

The projections take this term as the argument p, split it into the left component $x : (\gamma \multimap \sigma) \otimes (\gamma \multimap \tau)$ and the term of the shared variables $c : \gamma$. Then the branch of x (which is linearly typed now) is chosen and c is applied to it.

Because the type of the product changes depending on the free variables in a pair, this naive approach does to not give us the product that we are after.

Cartesian Product in LAL - Mendler style Inspired by the Mendler style recursion [Men88, Mat99], one can also make use of the parametricity of System F to define a product type in LAL. Parametricity (compare [Wad89]) means that a term cannot assume anything about a type variable, i.e. a term is defined uniformly in the possible substitutions of its free type variables. E.g., there is no case distinction possible on the type level, which can make a term behave differently depending on the type one substitutes into a type variable.

We use this property as follows, in order to define a uniform cartesian product in Light Affine Logic. We assume to have the usual impredicative definition of the sum type

$$\rho_1 + \rho_2 := \forall\alpha.(\rho_1 \multimap \alpha) \multimap (\rho_2 \multimap \alpha) \multimap \alpha$$

with the canonical injections $\mathbf{in}_L$ and $\mathbf{in}_R$.

First note that this sum type does not have the issues with linearity that we saw before with the product because the two arguments of type $\rho_1 \multimap \alpha$ and $\rho_2 \multimap \alpha$ are given by the term that uses the sum. Hence, the sum term itself does not have to take care of linearity.

With this in mind, we can define a Mendler style product type:

$$\sigma \times \tau \quad := \quad \forall\alpha, \beta.(\alpha + \beta) \multimap ((\sigma \otimes \alpha) + (\tau \otimes \beta)). \tag{3.1}$$

The α and β in the positive positions guarantee that a term of type $\sigma \times \tau$

always "chooses" the correct branch depending on the input $\alpha+\beta$. Although the result type is a sum type as well, the parametricity of the system *forces* the term to take the same branch as the input $\alpha + \beta$. Because there is no canonical inhabitant of α or β, there is no other (closed) choice.

On the other hand, due to parametricity and because the result is a sum type, the code that uses such a cartesian product and projects it to one component has to handle both branches of the result, i.e. both $\sigma \otimes \alpha$ and $\tau \otimes \beta$. The parametricity makes sure that it is known on the meta level that no cartesian product term with this type can choose the "wrong" branch. Hence, we get:

Theorem 3.4 (Correctness)**.** *Given a term $p \in Tm_{LAL}$ with*

$$\Gamma \vdash p : \forall\alpha,\beta.(\alpha + \beta) \multimap ((\sigma \otimes \alpha) + (\tau \otimes \beta)),$$

a term $\big(p\,\alpha\,\beta\,\big(\boldsymbol{in}_R\, b^{\beta}\big)\big)$ with $b \in V$ and $\alpha,\beta \in T$ cannot reduce to a left injection $(\boldsymbol{in}_L\, t)$ for any $t \in Tm_{LAL}$ (and conversely for the right injection).

Proof sketch. By parametricity, p is uniform in α and β. Hence, we can substitute α and β by any type without changing the behaviour.

First choose $\alpha = \bot = \forall\delta.\delta$, i.e. the type which has no inhabitants. Then a term $\big(p \bot \beta\ \big(\mathbf{in}_R\, b^{\beta}\big)\big)$ is of type $(\sigma\otimes\bot)+(\tau\otimes\beta)$ and hence the left injection is not inhabited either. If $\big(p\,\alpha\,\beta\,\big(\mathbf{in}_R\, b^{\beta}\big)\big)$ reduces to a left injection, then $\big(p \bot \beta\ \big(\mathbf{in}_R\, b^{\beta}\big)\big)$ will do it too (by parametricity). Then replace σ with some inhabited type such that t is inhabitant of $\sigma \otimes \bot$. Contradiction.

Note that we implicitly make use here of the fact that there cannot be free variables of type α or β in p because the $(\forall^+)$-rules forbids this. □

On the term level it is "not known" which branch of the sum will be chosen when using the projection on such a product. But with a simple trick this is not a problem: just choose $\alpha = U$, $\beta = \sigma \multimap \tau$ in the right projection, and $\alpha = \tau \multimap \sigma$,$\beta = U$ in the left projection (with the unit type $U := \forall\alpha.\alpha \multimap \alpha$ and $1 := \Lambda\alpha.\lambda x.x$ as its inhabitant). Then the projections are well typed, and uniform in σ and τ:

$$\pi_{\sigma,\tau,0} := \lambda p^{\sigma\times\tau}.\Big((p\,U\,\tau{\multimap}\,\sigma\,(\mathbf{in}_L\,1))\ \lambda x^{\sigma\otimes U}.(x\ \lambda s,a.s)\ \lambda x^{\tau\otimes(\tau\multimap\sigma)}.(x\ \lambda t,b.(b\,t))\Big) \tag{3.2}$$

$$\pi_{\sigma,\tau,1} := \lambda p^{\sigma\times\tau}.\Big((p\,\sigma\multimap\tau\,U\,(\mathbf{in}_R\,1))\,\lambda x^{\sigma\otimes(\sigma\multimap\tau)}.(x\,\lambda s,a.(a\,s))\,\lambda x^{\tau\otimes U}.(x\,\lambda t,b.t)\Big) \tag{3.3}$$

It remains to be shown that the type

$$\sigma\times\tau = \forall\alpha,\beta.(\alpha+\beta)\multimap((\sigma\otimes\alpha)+(\tau\otimes\beta))$$

has indeed the properties of a cartesian product:

Theorem 3.5 (Existence)**.** *In LAL for all types γ, σ, τ and terms $\Gamma_1,\Gamma\vdash s^\sigma$, $\Gamma_2,\Gamma\vdash t^\tau$ with Γ_1 and Γ_2 disjoint, there is an element $\langle s,t\rangle:\sigma\times\tau$ with $(\pi_{\sigma,\tau,0}\,\langle s,t\rangle)\longrightarrow_{NbL} s$ and $(\pi_{\sigma,\tau,1}\,\langle s,t\rangle)\longrightarrow_{NbL} t$.*

Proof. Let $\overrightarrow{x}$ be the list of variables in Γ, and $\overrightarrow{\gamma}$ the iterated tensor product of their types. Define a cartesian pair $\langle s,t\rangle^{\sigma\times\tau}$ as

$$\langle s,t\rangle := \Lambda\alpha,\beta.\lambda c^{\alpha+\beta}.\Big((c\,\delta\,\lambda a^{\alpha},\overrightarrow{x}^{\overrightarrow{\gamma}}.\overbrace{(\mathbf{in}_L\,(s\otimes a))}^{\sigma\otimes\alpha+\tau\otimes\beta}\,\lambda b^{\beta},\overrightarrow{x}^{\overrightarrow{\gamma}}.\overbrace{(\mathbf{in}_R\,(t\otimes b))}^{\sigma\otimes\alpha+\tau\otimes\beta})\,\overrightarrow{x}\Big) \tag{3.4}$$

$$\delta := \overrightarrow{\gamma}\multimap(\sigma\otimes\alpha+\tau\otimes\beta). \tag{3.5}$$

Then it is clear that the term $\langle s,t\rangle$ is well typed with $\Gamma_1,\Gamma_2,\Gamma\vdash\langle s,t\rangle^{\sigma\times\tau}$. Moreover, $(\pi_{\sigma,\tau,0}\,\langle s,t\rangle)\longrightarrow_{NbL} s$ and $(\pi_{\sigma,\tau,1}\,\langle s,t\rangle)\longrightarrow_{NbL} t$ because already

$$(\pi_{\sigma,\tau,0}\,\langle s,t\rangle)\longrightarrow_l^* s$$

and

$$(\pi_{\sigma,\tau,1}\,\langle s,t\rangle)\longrightarrow_l^* t$$

holds, i.e. the Linear System F reductions of LAL are enough (normalisation inside boxes is not necessary). □

Selection by a boolean Given a cartesian pair $\langle s,t\rangle$ of type $\tau\times\tau$, it is possible to select the side by using a boolean, i.e. to create a function of type

$$\pi:\forall\tau.B\multimap\tau\times\tau\multimap\tau$$

given by

$$\pi \quad := \quad \Lambda\tau.\lambda b^B.\,(b\,(\tau\times\tau)\multimap\tau\,\pi_{\tau,\tau,0}\,\pi_{\tau,\tau,1})\,.$$

3.1.3. Beckmann/Weiermann Example

After the discussion of the products in the different calculi, we look at the first example now that is connected with the possible normalisation orders related to products. Originally, it was given by Beckmann and Weiermann [BW96] to show the necessity of restricting the permissible normalisation strategies in Bellantoni and Cook's safe recursion [BC92]. Later, Neergaard and Mairson [NM03] used this as a hint that a complete embedding of full BC into LAL is not possible.

Our interest is to extend the analysis of the example to all the systems we consider here: LT, BC, LAL and LFPL.

> **Example 3.6** (Beckmann/Weiermann)**.** Written in (applicative) System T with lists, cartesian product and iteration:
>
> $$t := \lambda l^{L(B)}.\left(\mathbf{It}_{B,B}\, l\, \lambda b^{B} \lambda p^{B}.\, (b\, \langle p, p\rangle)\ \mathbf{tt}\right).$$

The question, that we want to ask now, is how the different calculi avoid that during the unfolding of the iteration the term $(b\,\langle p,p\rangle)$ grows as in the following illustration:

$$(t\ (\mathbf{cons}_B\,\mathbf{tt}\ (\mathbf{cons}_B\,\mathbf{tt}\ (\mathbf{cons}_B\,\mathbf{tt}\,\mathbf{nil}_B)))) \rightsquigarrow$$

$$\overbrace{(\mathbf{tt}\,\langle \quad\cdot\quad , \quad\cdot\quad \rangle)}^{}\overbrace{(\mathbf{tt}\,\langle \quad\cdot\quad , \quad\cdot\quad \rangle)}^{}.$$

$$\overbrace{(\mathbf{tt}\,\langle \quad\cdot\quad , \quad\cdot\quad \rangle)}^{}\overbrace{(\mathbf{tt}\,\langle \quad\cdot\quad , \quad\cdot\quad \rangle)}^{}\qquad\overbrace{(\mathbf{tt}\,\langle \quad\cdot\quad , \quad\cdot\quad \rangle)}^{}\overbrace{(\mathbf{tt}\,\langle \quad\cdot\quad , \quad\cdot\quad \rangle)}^{}$$

$$\overbrace{(\mathbf{tt}\,\langle \cdot\,,\,\cdot \rangle)}^{}\overbrace{(\mathbf{tt}\,\langle \cdot\,,\,\cdot \rangle)}^{}\qquad\overbrace{(\mathbf{tt}\,\langle \cdot\,,\,\cdot \rangle)}^{}\overbrace{(\mathbf{tt}\,\langle \cdot\,,\,\cdot \rangle)}^{}\qquad\overbrace{(\mathbf{tt}\,\langle \cdot\,,\,\cdot \rangle)}^{}\overbrace{(\mathbf{tt}\,\langle \cdot\,,\,\cdot \rangle)}^{}\qquad\overbrace{(\mathbf{tt}\,\langle \cdot\,,\,\cdot \rangle)}^{}\overbrace{(\mathbf{tt}\,\langle \cdot\,,\,\cdot \rangle)}^{}$$

$$\overbrace{\mathbf{tt}\ \mathbf{tt}}^{}\quad\overbrace{\mathbf{tt}\ \mathbf{tt}}^{}\quad\overbrace{\mathbf{tt}\ \mathbf{tt}}^{}\quad\overbrace{\mathbf{tt}\ \mathbf{tt}}^{}\quad\overbrace{\mathbf{tt}\ \mathbf{tt}}^{}\quad\overbrace{\mathbf{tt}\ \mathbf{tt}}^{}\quad\overbrace{\mathbf{tt}\ \mathbf{tt}}^{}\quad\overbrace{\mathbf{tt}\ \mathbf{tt}}^{}$$

It is clear that this tree like structure grows exponentially because with every $\mathbf{cons}_B$ in the input list another level in this illustration is created, doubling its size.

On the other hand, the applied normalisation strategy which leads to this size explosion is a very bad one. One can easily avoid the problem altogether by normalising the cartesian selection with the $\mathbf{tt}$ first, before unfolding and substitution. Then the algorithm runs even in linear time. But for that normalisation under λ is necessary because the selection is under the λp abstraction.

We will go through the different calculi now to see how such an optimisation is enforced by the type system or by some restriction of the normalisation strategy.

3.1.3.1. LT

$$t := \lambda \bar{l}.\, \left(\boldsymbol{\mathcal{R}}_{B,B}\, \bar{l}\, \lambda \bar{x}, \bar{l'}, p.\, (\bar{x}\, \langle p, p \rangle)\ \mathbf{tt}\right).$$

The reduction strategy, which is proposed in the complexity theorem of [SB01], will reduce this in the following way:

1. The list $\bar{l}$ is normalised first to know its length n (this is possible because $\bar{l}$ is complete) and the possibly not yet normal first arguments $b_i$of the $\mathbf{cons}_B$.

2. The iteration is unfolded n times to get $(s_n\ \ldots (s_1\ \mathbf{tt}))$ with step terms $s_i = \lambda p.\, (b_i\, \langle p, p \rangle)$, obviously of linear size in the length of the list n. Each of these s_i can be normalised independently to s_i' because of completeness of s_i.

3. Then $(s_n'\ \ldots (s_1'\ \mathbf{tt}))$ is normalised further using the strategy given by the "Sharing Normalisation" lemma in [SB01].

In the last step the "Sharing Normalisation" makes sure that the copies of p in the cartesian product subterms are properly *shared* in each step of the reduction, i.e. during beta-reduction the argument of the application is not duplicated, but only gets two pointers pointing to it. Hence, the DAG (directed acyclic graph, i.e. the model of computation of LT) stays linear in n during normalisation.

In the second step the s_i are complete. Hence, if they used another nested recursion, it could be normalised in each s_i independently. This is the central idea in LT, but not necessary for this example. In this example the sharing of duplicated terms of ground type is what avoids exponential growth of the term.

3.1.3.2. BC - Beckmann/Weiermann's Solution

When using a simple term rewriting system (and not DAGs) as the model of computation, the upper solution with sharing is not available. But even in this setting one can normalise Bellantoni-Cook-style terms in polynomial time.

Beckmann and Weiermann consider a term rewriting variant of Bellantoni and Cook's (non-higher-type) safe recursion calculus BC. I.e., in contrast to Bellantoni and Schwichtenberg's LT [SB01] they only take ground type terms into account and write program as equational definitions. The main restriction of their setting though is that they have to describe the normalisation purely as a *term rewriting system* with term rewriting rules.

Basically, they show that the term rewriting system *must not* have a general standard rewrite rule for composition and primitive recursion. Instead, the rules are restricted to have only numerals in the safe argument positions. Hence, the rewrite rule for the safe composition becomes (in their denotation)

$$\begin{aligned}&\mathcal{SUB}^{k,l}_{k',l'}[f, g_1, ..., g_{k'}, h_1, ..., h_{l'}](x_1, ..., x_k; \underline{n_1}, ..., \underline{n_l}) \longrightarrow \\ &f(g_1(x_1, .., x_k;), .., g_{k'}(x_1, .., x_k;); \\ &\qquad h_1(x_1, ...x_k; \underline{n_1}, .., \underline{n_l}), .., h_{1'}(x_1, .., x_k; \underline{n_1}, .., \underline{n_l}))\end{aligned}$$

and for recursion

$$\begin{aligned}&\mathcal{PREC}^{k+1,l}[g, h_1, h_2](S_i(;x), x_1, ..., x_k; \underline{n_1}, ..., \underline{n_l}) \longrightarrow \\ &h_i(x, x_1, ..., x_k; \underline{n_1}, ..., \underline{n_l}, \mathcal{PREC}^{k+1,l}[g, h_1, h_2](x, x_1, ..., x_k; \underline{n_1}, ..., \underline{n_l})).\end{aligned}$$

With $\underline{n}$ we denote numerals here. This means that these rules are in fact two rewrite schemes for all possible numerals.

The case which is interesting here is the one of case distinction in a safe position. When writing down the Beckmann/Weiermann example in this setting, the (safe) recursion result $(b\,\langle p, p\rangle)$ is not a numeral, and therefore the upper two rewrite rules cannot be applied then. Hence, before the recursion can be unfolded again via the $\mathcal{PREC}$ rewrite rule, the term must be normalised further to be not a case distinction anymore, but a numeral. This obviously leads to a "good" evaluation strategy, which is not exponential, in this example.

3.1.3.3. Light Linear Logic

In a linear calculus like LAL the first question is how to represent the term $\langle p, p\rangle$ which is obviously not linear.

Naive cartesian product We will first follow the idea of the naive approach in section 3.1.2 to "abstract out" the p. This gives the Beckmann/Weiermann variant for LAL:

$$t := \S\left(\right]\left(l\, B\,!\, \boxed{\lambda b^B \lambda p^B.\Big(\big((b\, B \multimap B)\, \lambda p.p\, \lambda p.p\big)\, p\Big)}\right)\Big[_{\S}\ \mathbf{tt}\Big).$$

During normalisation this term does not explode in LAL because no duplication is possible at all: from the start the shared p of the two branches is moved outside the branching which depends on b. Hence, the boolean b (which decides which branch to take) is applied to the (now closed) two branches *before* plugging in the shared variable p. So the application of p cannot give a redex before the redex of the boolean b is fired. In other words, the boolean selection blocks the duplication.

Mendler style cartesian product If we use the more general Mendler style cartesian product of Section 3.1.2, we get exactly the same effect. The canonical Mendler style cartesian pair constructor, as given in Lemma 3.5, uses the λa^α (in the left branch) and λb^β (in the right branch) to block the application of the shared variables $\overrightarrow{x}$. Hence, also in this case, duplication is avoided completely by the same mechanism as with the naive cartesian product.

3.1.3.4. LFPL

The example written in LFPL is the same as in T after replacing the iteration constant with the applicative syntax:

$$t := \lambda l^{L(B)}.\left(l\, \lambda d^{\Diamond}, b^B, p^B.\, (b\, \langle p, p\rangle)\ \mathbf{tt}\right).$$

The term is non-linear of course, but this special non-linearity in the cartesian product is allowed in LFPL. The calculus morally assumes that only one of the branches is ever chosen during normalisation. Reduction is not possible inside such a cartesian pair. As described in Remark 2.20 this makes it easy to apply sharing for shared terms inside pairs like $\langle p, p\rangle$. Like in the case of LT, this makes the term of the Beckmann/Weiermann example linear again in the input size.

The further normalisation of the term will choose the correct branch (here the left one) in each step, from outside to inside. But as no duplication can

happen due to sharing, this normalisation order is not bad anymore.

3.1.4. Necessity of the Cartesian Product in LFPL

The question we want to answer in this section is whether one really needs the cartesian product in LFPL, or if it is enough to have the usual (B^-) rule of the applicative Linear System T with booleans instead, i.e.

$$\frac{\Gamma_1 \vdash b^B \quad \Gamma_2 \vdash f^\sigma \quad \Gamma_3 \vdash g^\sigma}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash (b\, f\, g)^\sigma}\ (B^-_{aLinT})$$

with the same type for the two alternatives. In the Beckmann/Weiermann example in Section 3.1.3 we have seen that the restriction to forbid reduction inside $\langle s, t\rangle$ can be important to avoid exponential behaviour. Hence, the question about the necessity of the cartesian product in LFPL is important.

First recall that there are two kinds of elimination rules for the cartesian product in LFPL:

$$\frac{\Gamma \vdash p^{\rho\times\sigma}}{\Gamma \vdash (p\,\mathbf{tt})^\rho}\ (\times^-_{\mathbf{tt}}) \quad \frac{\Gamma \vdash p^{\rho\times\sigma}}{\Gamma \vdash (p\,\mathbf{ff})^\sigma}\ (\times^-_{\mathbf{ff}})$$

and

$$\frac{\Gamma_1 \vdash b^B \quad \Gamma_2 \vdash p^{\tau\times\tau}}{\Gamma_1, \Gamma_2 \vdash (b\,p)^\tau}\ (B^-)$$

The former two allow distinct types ρ and σ for the left and right component, but can only be applied to constants $\mathbf{tt}, \mathbf{ff}$. The latter requires symmetric types $\tau \times \tau$, but allows any term b^B to choose the projection.

The system does not have type quantification, but is simply typed. Hence, types are fixed and we can make a distinction between those terms of symmetric and those of asymmetric cartesian product types. Clearly, every application of $(\times^-_{\mathbf{tt}/\mathbf{ff}})$ to a symmetric one can be replaced by the (B^-) rule, hence we have to think about how to type the symmetric case for (B^-) and the asymmetric case for $(\times^-_{\mathbf{tt}})$, $(\times^-_{\mathbf{ff}})$ with the (B^-_{aLinT}) rule.

Symmetric case Normalisation inside cartesian products $\langle t, s\rangle$ is prohibited in LFPL. Hence, in the symmetric case we can "factor out" all free variables with the following replacement without blocking any redexes which were not blocked before already:

$$\langle s[x], t[x]\rangle^{\tau\times\tau} \quad \rightsquigarrow \quad (\lambda b.\,(b\,\langle \lambda x.s[x], \lambda x.t[x]\rangle)\ x)^{B\multimap\tau}\,.$$

By doing this again and again until the product components have distinct free variables or are even closed, we can replace $\langle \lambda\overrightarrow{x}.s[\overrightarrow{x}], \lambda\overrightarrow{x}.t[\overrightarrow{x}]\rangle$ with

$$p := \lambda b.\,((b\,\lambda\overrightarrow{x}.s[\overrightarrow{x}]\,\lambda\overrightarrow{x}.t[\overrightarrow{x}])\ \overrightarrow{x})$$

using the (B^-_{aLinT}) rule. Finally, in order to make the (B^-) elimination $\left(b\,p^{B\multimap\tau}\right)$ type correct again, we replace it with $(\multimap^-)$, i.e. $\left(p^{B\multimap\tau}\,b\right)$. This gives a derivation of a functionally equivalent term without symmetric cartesian product eliminations via (B^-).

Note that the vector $\overrightarrow{x}$ can only be substituted into $s[\overrightarrow{x}]$ or $t[\overrightarrow{x}]$ *after* the boolean b is available and reduced. But this would only be an issue if normalisation inside $\langle\cdot,\cdot\rangle$ would be possible in LFPL.

Asymmetric case – factoring out free variables What is left are asymmetric cartesian products $p := \langle t, s\rangle^{\tau\times\sigma}$. A first idea – again – is to factor out common free variables, e.g. in the following way by turning the cartesian pair into a tensor triple:

$$p' := (\lambda\overrightarrow{x}.t[\overrightarrow{x}] \otimes \lambda\overrightarrow{x}.s[\overrightarrow{x}]) \otimes \overrightarrow{x}$$

of type $((\overrightarrow{\gamma} \multimap \tau) \otimes (\overrightarrow{\gamma} \multimap \sigma)) \otimes \overrightarrow{\gamma}$ and by replacing $\left(q^{\tau\times\sigma}\,\mathbf{tt}\right)$ (q is not necessarily a $\langle\cdot,\cdot\rangle$ term) with

$$\left(q'\ \lambda p\lambda\overrightarrow{x}.\,(p\,\lambda l\lambda r.\,(l\ \overrightarrow{x}))\right).$$

The problem with this encoding though is that the type of q' depends on the free variables of p. In the term $(q'\ \lambda p\lambda\overrightarrow{x}.\,(p\,\lambda l\lambda r.\,(l\ \overrightarrow{x})))$ the subterm q' could be a bound variable. During reduction p' could be substituted into the position of q'. Because types are static in LFPL, the type of q' is the same as the one of p'. Hence, the type of q' will depend on the free variables of p which can be somewhere far away in the term. Therefore, this transformation is not what we were aiming for.

Asymmetric case – canonical inhabitants Another trick is to code asymmetric products $\sigma\times\tau$ as symmetric products $(\sigma\otimes\tau)\otimes(\sigma\otimes\tau)$. If we assume that every type ρ is inhabited by a closed term i_ρ, we can code $\langle s, t\rangle^{s\times t}$ as $\langle s\otimes i_\tau, i_\sigma\otimes t\rangle^{(\sigma\otimes\tau)\times(\sigma\otimes\tau)}$, with the obvious projections. Then we use the

encoding for the symmetric case to get rid of the cartesian product. This does not work with non-inhabited types like $\diamondsuit$ though.

Asymmetric case – sum type Assume we had a sum type $\tau + \sigma$ in the system LFPL with

$$\frac{\Gamma_1 \vdash p^{\tau+\sigma} \quad \Gamma_2 \vdash f^{\tau \multimap \rho} \quad \Gamma_3 \vdash g^{\sigma \multimap \rho}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash (p\, f\, g)^{\rho}} \; (+^-)$$

Then we could use an alternative rule like

$$\frac{\Gamma_1 \vdash b^{B} \quad \Gamma_2 \vdash p^{\tau \times \sigma}}{\Gamma_1, \Gamma_2 \vdash (b\, p)^{\tau+\sigma}} \; (B^-_{\text{sum}})$$

which looks like a good replacement for the LFPL rules $(\times^-_{\mathbf{tt}})$ and $(\times^-_{\mathbf{ff}})$. But in fact it is not: take the LFPL-term $t := \lambda p^{\diamondsuit \times B}.\, (p\, \mathbf{tt})^{\diamondsuit}$. With the (B^-_{sum}) rule this t would turn into something like

$$\lambda p.^{\diamondsuit \times B} \left((\mathbf{tt}\, p)^{\diamondsuit + B} \; \lambda x^{\diamondsuit}.x\, g^{B \multimap \diamondsuit} \right)^{\diamondsuit}.$$

There is no closed term g of type $B \multimap \diamondsuit$. In other words, the sum in (B^-_{sum}) gives us another branch in the sum elimination that we cannot cope with in this example.

> *Remark* 3.7. Compare the setting here with that of Section 3.1.2. There, a cartesian product is defined for Light Affine Logic which allows shared variables and asymmetric types. The difference is that LAL has polymorphic types, i.e. the type of a product is not as static as in LFPL. Here, we cannot define a product by a function that takes a sum or a boolean as selector for the branch because then the type will be different depending on the choice. This seems to be the very reason why LFPL has these extra typing rules $(\times^-_{\mathbf{tt}})$, $(\times^-_{\mathbf{ff}})$ for the two projections.

We will not continue investigating the issue here and leave it as an (vague) open question:

Problem 3.8. Can one transform every LFPL algorithm with occurrences of (B^-), $(\times^-_{\mathbf{tt}})$ or $(\times^-_{\mathbf{ff}})$ into an equivalent one with (B^-_{aLinT})?

By the explanation above the only remaining case is the occurrence of $(\times^{-}_{\mathbf{tt}})$ or $(\times^{-}_{\mathbf{ff}})$ with asymmetric types $\sigma \times \tau$ indeed.

3.2. Recursion Schemes

The example by Beckmann and Weiermann in 3.1.3 was very much based on the way the calculi handle cartesian products and how those can be used in a recursion. That example though, in fact, is not exponential per se, i.e. there is a normalisation strategy which computes the result in polynomial (even linear) time.

In the following, we analyse several recursive algorithms, which have an exponential growth, i.e. the result is a list whose size is exponential in the input length. It is clear that, in order to produce such a result, it needs exponentially many computation steps (at least with the usual canonical encoding and representation of numerals or lists).

Most of the examples in this section are taken from [SB01] where they were used to motivate the choice of arrows in the types of the constants and other restrictions in the type system of LT. But they are also well suited to study the restrictions of other calculi.

3.2.1. Two Recursions

The first example in this row is the possibly most direct way to calculate an exponential function, i.e. by iterating a doubling function $d(n)$:

$$d(0) := 0 \tag{3.6}$$
$$d(S(x)) := S(S(d(x))) \tag{3.7}$$

$$e(0) := S(0) \tag{3.8}$$
$$e(S(x)) := d(e(x)). \tag{3.9}$$

3.2.1.1. LT

The calculus LT has two kinds of arrows: the normal/complete one $\to$ and the safe/incomplete arrow $\multimap$. The abstracted variables of the former are overlined as in $\overline{x}$. The recursion constant has the type

$$\boldsymbol{\mathcal{R}}_{\sigma,\tau} : L(\sigma) \to (\sigma \to L(\sigma) \to \tau \multimap \tau) \to \tau \multimap \tau,$$

with the restriction that the type τ is linear, i.e. the arrow $\to$ is not allowed in τ.

To order to express the doubling function d (with $N := L(U)$ for numerals, U a unit type with constant $1 : U$), one recursion is necessary which forces us to use the complete arrow $\to$ for the input list (compare the first arrow in the type of $\boldsymbol{\mathcal{R}}_{\sigma,\tau}$):

$$d := \lambda\bar{l}.\,\left(\boldsymbol{\mathcal{R}}_{U,L(U)}\,\bar{l}\,\lambda\overline{x}\lambda\overline{l'}\lambda p.\,(\mathbf{cons}_U\,\mathbf{1}\;(\mathbf{cons}_U\,\mathbf{1}\,p))\;\mathbf{nil}_U\right)$$
$$d : L(U) \to L(U).$$

This is a valid LT term.

The iteration of e in another recursion gives:

$$e := \lambda\bar{l}.\,\left(\boldsymbol{\mathcal{R}}_{U,L(U)}\,\bar{l}\,\lambda\overline{x}\lambda\overline{l'}\lambda p.\,(d\,p)\;(\mathbf{cons}_U\,1\,\mathbf{nil}_U)\right)$$
$$e : L(U) \to L(U)$$

which is not correctly typed because d needs a complete (i.e. overlined) argument. But p as the recursion result (compare the forth arrow in the type of $\boldsymbol{\mathcal{R}}_{\sigma,\tau}$) is incomplete.

Analysis of the complexity proof A deeper look into the proof of the "$\boldsymbol{\mathcal{R}}$-elimination" in [SB01] tells us that it would break for the following reason if this term was allowed: during the elimination of the $\boldsymbol{\mathcal{R}}_{U,L(U)}$ in e the $\boldsymbol{\mathcal{R}}$-constants are removed in the step terms $s_i := \left(\lambda\overline{x}\lambda\overline{l'}\lambda p.\,(d\,p)\;r_i\,l_i\right)$ for some complete $\boldsymbol{\mathcal{R}}$-free terms r_i and l_i (i.e. by applying the induction hypothesis of the theorem to every half instantiated step). Here, it is crucial that the sizes of r_{i+1} and l_{i+1} do not depend on the previous step s_i or $(s_i\,p_i)$. If the recursion argument p (or here p_{i+1}) was complete, the size of the s_i could not be polynomially bounded anymore.

On the other hand, having an incomplete arrow for the p does not allow the recursion in d. If it was allowed, the induction hypothesis in the theorem could not be applied, as the $\boldsymbol{\mathcal{R}}_{U,L(U)}$ of d cannot be eliminated without giving also a substitution for (the now incomplete) p in s_i.

3.2.1.2. Light Affine Logic

In order to iterate the doubling function later for the definition of e, the term of d must have a symmetric type like $N \multimap N$. But in Light Affine

Logic this is not possible: the reason is that the Church encoding of natural numbers in LAL,

$$N \quad := \quad \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha),$$

requires that the (abstracted) step term of the result, which is of type $!(\alpha \multimap \alpha)$, must be used twice in the step term as in $!\boxed{\lambda z.\,(]f[_!\ (]f[_!\ z))}$ (compare Equation 3.7) which is applied to the input numbers:

$$d := \lambda x^N.\Lambda\alpha.\lambda f^{!(\alpha\multimap\alpha)}.\S\boxed{\lambda y.\,\left(\right]\left(x\,\alpha\,!\boxed{\lambda z.\,(]f[_!\ (]f[_!\ z))}\right)\left[_\S\ y\right)}.$$

This is not type-correct for LAL because the subterm $!\boxed{\lambda z.\,(]f[_!\ (]f[_!\ z))}$ cannot have two "holes" $]f[_!$ due to the $(!_1)$-rule:

$$\frac{x : A \vdash t : B \quad \Gamma \vdash s :!A}{\Gamma \vdash !\boxed{x =]s[_!\ \text{in}\ t}\,:!B}\ (!_1)$$

Of course, it is not too hard to rewrite the doubling algorithm by using the usual concatenation, i.e. addition without any iteration. Then though the input number must be duplicated (i.e. appears twice) and therefore the type of the d function must be decorated with the right modalities in order to allow this:

$$d := \lambda x^{!N}.\S\boxed{\Lambda\alpha.\lambda f^{!(\alpha\multimap\alpha)}.\S\boxed{\lambda y.(]\,(]x[_!\ \alpha\,f)[_\S\ (]\,(]x[_!\ \alpha\,f)[_\S\ y))}} \tag{3.10}$$

$$d :!N \multimap \S N.$$

We cannot replace the outer §-box with the !-rule to make the type symmetric because x is non-linear which would conflict with the $(!_1)$-rule again (see above). The intuition is that the §-modality is used to mark exactly those non-linearities. More precisely, it is a marking in the type that a banged variable (the x), which was duplicated (as two occurrences of x), is "de-banged" (as $]x[_!$), i.e. can be used inside a §-box multiple times.

In order to implement the algorithm e, one would have to iterate d. But the type of d is not symmetric anymore. So the exponential algorithm e is not definable in LAL this way. Note that we cannot say much about the

$e(x)$ function in contrast to the algorithm with this kind of argument, but the general PTime normalisation theorem by [AR02] can of course.

> *Remark* 3.9. This example gives a hint about the role of the two modalities which will be the central insight in Chapter 7: to create size-increasing algorithms (like d) in LAL an asymmetric type is needed. Symmetric algorithms are very similar to non-size-increasing algorithms of LFPL.

3.2.1.3. LFPL

In Hofmann's non-size-increasing system LFPL everything is affine (with the exception of $\langle s, t\rangle$-subterms). Moreover, the $\diamondsuit$-terms cannot be duplicated. Hence, the definition of d is not possible at all, as it increases the size of the data. Therefore, the example does not apply.

3.2.2. Higher Type Result

A variation of the previous example is the following which uses a higher type recursion to iterate the double function d:

$$\begin{aligned} e(0, y) &:= y \\ e(S(x), y) &:= e(x, d(y)), \end{aligned}$$

or written with higher types:

$$\begin{aligned} e(0) &:= \mathrm{id} \\ e(S(x)) &:= e(x) \circ d \end{aligned}$$

with the usual concatenation $\circ$ of functions and "id" as the identify. Clearly, this gives an exponential function by

$$e'(x) \quad := \quad e(x, x).$$

3.2.2.1. LT

The argument to d must be complete/normal because of the recursion in the definition of d (see 3.2.1). Hence, the return type of the recursion in e must use the complete arrow, i.e. $N \to N$. In LT only linear types τ are allowed as result types τ of recursions $\boldsymbol{\mathcal{R}}_{\sigma,\tau}$ though. Therefore, the definition of e is outlawed by a type restriction on $\boldsymbol{\mathcal{R}}_{\sigma,\tau}$.

Analysis of the complexity proof Let us check where the complexity proof in [SB01] would break if non-linear return types of recursions were allowed. In fact, like in Section 3.2.1.1 before, the $\mathcal{R}$-elimination lemma would be the one which breaks, more precisely in the case of the step term:

- Assume that $s : \sigma \to L(\sigma) \to \tau \multimap \tau$ is the recursion step for a recursion with complete arrow type, e.g. $\tau := N \to N$, i.e. in fact

$$s : \sigma \to L(\sigma) \to (N \to N) \multimap N \to N.$$

- Let s include a $\mathcal{R}$-constant which depends on the forth argument, which is of type N here. Because τ has a complete arrow, this argument is complete which is crucial to allow such a recursion.

- The normalisation of the partial steps $s_i := (s\, r_i\, l_i)$ will not remove this $\mathcal{R}$ because no numeral substitution of its principal argument is available.

- Hence, the induction step in the $\mathcal{R}$-elimination theorem fails and the polynomial normalisation bound breaks down.

3.2.2.2. Light Affine Logic

We take the modified d from Equation 3.10 in Section 3.2.1.2. Then we write down the example e without thinking about the types yet:

$$e := \lambda x^N.\Lambda\alpha.\Big(x\,\beta\,\overbrace{\underbrace{\lambda y}_{e(p)}.\underbrace{\lambda z.\,(y\ (d\,z))}_{e(p)\circ d}}^{\text{step}}\overbrace{\lambda y.y}^{\text{id}}\Big).$$

As described in Section 3.2.1.2, the type of d is not symmetric, but $!N \multimap \S N$. Therefore, y is applied to an §-argument $(d\,z)$ such that the type of y must be $\beta := \S N \multimap \gamma$ for some type γ.

On the other hand, d must be applied to a !-argument such that z is of type $!N$ and therefore $\lambda z.\,(y\ (d\,z)) : \beta = !N \multimap \gamma'$. This this would mean $\S N \multimap \gamma = \beta = !N \multimap \gamma'$, contradiction.

Hence, once again the stratification of the LAL type system makes the iteration of d (with the chosen implementation) impossible.

3.2.2.3. LFPL

Again, d cannot be defined in LFPL as it is not non-size-increasing such that the example does not apply.

3.2.3. Non-Linear Recursion Argument

The previous example in Section 3.2.2 was the motivation in LT to forbid the complete arrow $\to$ in the recursion type. The question remains whether higher type recursion that only uses the $\multimap$-arrow can be allowed.

For this consider another way to create an exponential function that uses a higher type recursion, but now with the $\multimap$ arrow. It uses the recursive value twice:

$$e(0, y) := S(y)$$
$$e(S(x), y) := e(x, e(x, y))$$

or written directly with higher types:

$$e(0) := S$$
$$e(S(x)) := e(x) \circ e(x).$$

Again clearly this growths exponentially.

3.2.3.1. LT

In the LT system this example is forbidden in an ad hoc way by enforcing linearity for incomplete higher-type arguments. This restriction of course rules out the definition of e above: in the clause for $e(S(x))$ the recursion argument $e(x)$ is used twice, i.e. non-linearly. Because the recursion type is $N \multimap N$, the use of $e(x)$ must be linear by the restriction, and hence this example is not type correct.

Analysis of the complexity proof Again, we ask where the complexity proof breaks in [SB01] if non-linear higher type variables were allowed. This time the $\mathcal{R}$-elimination lemma would not break directly. Instead, the sharing normalisation lemma does not hold anymore: in the case $(\lambda x.r\, s)$ for higher type x, it is essential that x occurs at most once (i.e. the parse dag is h-

affine). Otherwise, the s node would have to be duplicated as soon as a redex, that involves s, is fired. Naive sharing cannot cope with this case[1].

> *Remark* 3.10. In fact, LT does not enforce linearity of complete higher type variables directly. But those terms which are shown to admit a polynomial normalisation are those which are *simple*, i.e. those without any abstraction of complete higher type variables. For simple terms (which are h-affine) no sharing is ever needed for function type terms.

3.2.3.2. Light Affine Logic

The recursion result is not linear in the definition of $e(S(x))$. Hence, contraction is needed which would require a banged iteration type, e.g. $!(N \multimap N)$. Then, in order to get rid of the ! for the application of $e(x)$, the §-rule has to be used, probably similarly to

$$\S\boxed{\ldots]e(x)[_!\ldots]e(x)[_!\ldots}.$$

This though would imply a recursion type with a §. Hence, we get an asymmetric type for the step term of the iteration which is not possible. Hence, again the stratification property of LAL outlaws the definition of e.

3.2.3.3. LFPL

As before, d cannot be defined in LFPL as it is not non-size-increasing. Therefore, this example does not apply.

3.2.4. Iteration Functional

A generalisation of the example about higher type recursions in Section 3.2.1 is the iteration functional. It takes a number, a function and a base value and iterates the function:

$$I(0, f, y) := y$$
$$I(S(x), f, y) := f(I(x, f, y))$$

or written in higher types:

$$I(0, f) := \mathrm{id}$$

[1] It could be worthwhile to look into works about optimal reduction by Mairson et. al.

$$I(S(x), f) := f \circ I(x, f).$$

The example in Section 3.2.2 can be written as $I(x, d)$.

3.2.4.1. LT

$$I \quad := \quad \lambda \overline{x}^{L(U)}, \bar{f}.\, (\boldsymbol{\mathcal{R}}_{U, L(U) \multimap L(U)}\, \overline{x}\, \lambda \bar{z}, \bar{l}, p. \lambda y.\, (\bar{f}\ (p\, y))\ \lambda y. y)$$

This is a perfectly valid term if $\bar{f}$ is required to be of linear type. In this implementation though, the necessity for $\bar{f}$ to have linear type does not follow from the linear result type restriction of the recursion, but from the incompleteness of the recursion argument p. Because p appears free in the argument term of $\bar{f}$, it would not be type correct if $\bar{f}$ expected a complete argument, i.e. if $\bar{f}$ was of non-linear type.

The completeness of $\bar{f}$ comes from the fact that the step term of a recursion must be complete, such that no free incomplete variables are allowed in it (they would be duplicated during recursion otherwise).

Conversely, another equivalent implementation of the iteration functional would be the following:

$$\lambda \overline{x}^{L(U)}, \bar{f}.\, (\boldsymbol{\mathcal{R}}_{U, L(U) \multimap L(U)}\, \overline{x}\, \lambda \bar{z}, \bar{l}, p. \lambda y.\, (p\ (\bar{f}\, y))\ \lambda y. y)\,.$$

But now the type of p is required to be linear (as the result type of the recursion). Hence, y must be incomplete. Therefore, also $\bar{f}$ must have an incomplete first argument.

> *Remark* 3.11. By this example it is not clear why the whole type of $\bar{f}$ has to be linear. In fact, it is to be expected that the concept of "linear" types can be specialised to "p-linear", i.e. a type is p-linear if it has no $\rightarrow$ in positive positions.

> *Remark* 3.12. The term I is not simple because it has complete variables of higher type ($\bar{f}$). Hence, neither the $\boldsymbol{\mathcal{R}}$-elimination nor the Sharing Normalisation lemma can be applied to it directly. The solution to this in [SB01] is the following: a ground type term with only ground type free variables can be normalised using the Sharing Normalisation such that all complete higher type variables (which bound of course due to the restriction of the whole term) disappear. Hence, before a (closed)

> algorithm $t : \overrightarrow{N} \to N$ is applied to its inputs $\overrightarrow{n}$, $(t\ \overrightarrow{x})$ is normalised in this way. The complexity of this step might be big, but it is only a constant in the size of $\overrightarrow{n}$. The resulting normal form $\mathrm{nf}((t\ \overrightarrow{x}))$ is simple and the $\boldsymbol{\mathcal{R}}$-elimination and Sharing Normalisation lemmas are applicable, with the substitution $\overrightarrow{x} := \overrightarrow{n}$.

3.2.4.2. LFPL

The iteration functional is a built-in constant of LFPL. Though, by the construction of the type system, only non-size-increasing functions can be defined, and hence iterated. Therefore, the example is less interesting for LFPL, essentially because iteration of a polynomial non-size-increasing function gives a polynomial non-size-increasing function.

3.2.4.3. Light Affine Logic

The iteration function is trivially given in Linear Logic as Church numerals are used to represent natural numbers. So, any natural number is its own iteration functional. Similarly to LFPL though, also in LAL the example is less interesting because only functions can be iterated which have a symmetric type. As described in Remark 3.9 already, symmetric functions in LAL are related to non-size-increasing functions in LFPL (compare Chapter 7).

3.2.5. Iterating the Recursion Argument

The example of Section 3.2.3 can be rewritten with the iteration functional I:

$$\begin{aligned} e(0) &:= S \\ e(S(x)) &:= I(S(S(0)), e(x)). \end{aligned}$$

3.2.5.1. LT

Starting with the iteration functional from 3.2.4.1, it is obvious that the application of the incomplete $p = e(x)$ (i.e. the incomplete recursion argument) to the iterator $(I\ (S\ (S\,0)))$ is not allowed:

$$\begin{aligned} I &:= \lambda\bar{x}, \bar{f}.\, \left(\boldsymbol{\mathcal{R}}_{U,L(U)}\, \lambda\bar{z}, \bar{l}, p.\lambda y.\, \left(\bar{f}\ (p\, y)\right)\ \lambda y.y\right) \\ e &:= \lambda\bar{x}.\, \left(\boldsymbol{\mathcal{R}}_{U,L(U)\multimap L(U)}\, \bar{x}\, \lambda\bar{r}, \bar{l}, p.\, (I\ (S\ (S\,0))\ p)\ S\right). \end{aligned}$$

For illustration we want to see what happens during normalisation of $(e\,\underline{n})$ with some numeral $\underline{n}$ if we assume that this term was allowed:

- During the $\boldsymbol{\mathcal{R}}$-elimination of the outer recursion in e, the partial step terms $s_i := \left(\lambda\bar{r}, \bar{l}, p.\,(I\ (S\ (S\,0))\ p)\ r_i\,l_i\right)$ with complete and normal terms r_i and l_i are built.
- The recursions inside s_i are eliminated, which means in our example that s_i is reduced to $\lambda p, y.\,(p\ (p\,y))$.
- Hence, we get the same step term as in 3.2.3.1, which had led to the conclusion that higher type incomplete terms cannot be duplicated by using it in a non-linear fashion.

Remark 3.13. In fact, this normalisation is not exactly the one given in [SB01], but the inner iteration $(I\ (S\ (S\,0))\ p)$ would be reduced already before the $\boldsymbol{\mathcal{R}}$-elimination is started (compare Remark 3.12), giving the term $\lambda p, y.\,(p\ (p\,y))$ which is not h-affine. But, we could avoid this pre-normalisation of the inner iteration by considering

$$e' \quad := \quad \lambda\bar{x}.\,\left(\boldsymbol{\mathcal{R}}_{U,L(U)\multimap L(U)}\,\bar{x}\,\lambda\bar{r}, \bar{l}, p.\,\left(I\,\bar{l}\,p\right)\ S\right)$$

instead. Clearly, it is exponential as well and the pre-normalisation cannot remove the inner recursion constant, but has to wait until the partial step terms of the outer recursion are normalised and therefore freed from the $\boldsymbol{\mathcal{R}}$ constants.

3.2.5.2. LFPL

In LFPL the step term of the iteration must be closed. Here, in fact, the step term seems to be closed. But this is not true because the two S constants (which build up the numeral $\underline{2}$) require two terms of type $\diamondsuit$. Inside an LFPL iteration though, only one $\diamondsuit$ is available. The second $\diamondsuit$-term can only be provided by a free variable. Hence, the example is not typable in LFPL.

3.2.5.3. Light Affine Logic

In Light Affine Logic the numeral is the iterator itself, i.e. of type $\forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. Hence, we get something like

$$e := \lambda x^N.\,\left(x\,\gamma\,\lambda p^\gamma.\,\left(\underline{2}\,\gamma'\,p\right)\ \lambda y.y\right)$$

plus some modalities in the types and the corresponding boxes.

In the application to the numerals $\underline{2}$ the argument p must be banged, which means $\gamma = !(\gamma' \multimap \gamma')$ for some γ'. But the result type of the iterations (the inner one of the $\underline{2}$ numeral and the outer one of x) will get the paragraph at the outside their types. Hence, the type γ must be $!(\gamma' \multimap \gamma')$ and $\S(\gamma' \multimap \gamma')$ at the same time. In other words, the stratification of LAL makes this term impossible, once again.

3.3. Non-Artificial Algorithms

In the previous sections we studied recursion schemes which lead to exponential time and hence had to be outlawed by the type system. In a sense, those are easy tasks for the calculi we are looking at. The real challenge, where the "competing" systems differ, are polynomial time algorithms which should be expressible in a natural way.

For practical application it is not enough to know that there is some algorithm which computes a given function. E.g., completeness proofs usually show that some kind of model of computation for PTime can be simulated in the system. Of course, this gives a theoretical implementation for every PTime algorithm, but it may be far from the intended one because the simulation usually destroys the original recursive structure.

For instance a PTime Turing Machine simulation mainly consists of an outer loop, executing polynomially many steps of the machine, and another inner loop to update the tape. This structure is clearly different from the recursive structure of many algorithms, but still, every algorithm is mapped to this simple scheme.

3.3.1. Insertion Sort

A common example for PTime type systems is the Insertion Sort algorithm. It is probably so common because it is one of the few which is primitive recursive in its natural formulation. Therefore, it looks like fitting directly into the available recursion schemes. Though, this hope is far too optimistic as we will see.

Here is the algorithm implemented in System T, assuming that the (computable) predicate $\leq\colon N \to N \to B$ is given:

$$\text{insert} := \lambda x^N, l^{L(N)}.\, \big((\boldsymbol{\mathcal{R}}_{N,N\to L(N)}\, l\, \text{step}\, \text{base})\ x\big)$$

$$
\begin{aligned}
\text{step} &:= \lambda y^{N}, l'^{L(N)}, p^{N \to L(N)}, z^{N}.\, \left(\mathbf{Case}_{L(N)}\ (\leq\ y\, z)\ \text{then else}\right)\\
\text{then} &:= (\mathbf{cons}_{N}\ y\ (p\, z))\\
\text{else} &:= (\mathbf{cons}_{N}\ z\ (p\, y)) \qquad \text{or alternatively } \left(\mathbf{cons}_{N}\, z\, l'\right)\\
\text{base} &:= \lambda y.\, (\mathbf{cons}_{N}\, y\, \mathbf{nil}_{N})\\
\text{sort} &:= \lambda l^{L(N)}.\, \left(\boldsymbol{\mathcal{R}}_{N,L(N)}\, l\, \lambda x^{N}, l'^{L(N)}, p^{L(N)}.\, (\text{insert}\, x\, p)\ \mathbf{nil}_{N}\right).
\end{aligned}
$$

The crucial property of this algorithm is that there are two recursions[2] involved: the outer one in the "sort"-term and another one in the "insert"-term. The inner one does recursion over the list which is the result of the outer recursion, i.e. the recursive argument p in the "sort" line.

Clearly, the complexity is quadratic and polynomial therefore, without taking the comparison $\leq$ into account which might differ depending on unary or binary numbers.

Another important observation about Insertion Sort is that it is non-size-increasing. Obviously, the "insert"-term just inserts the number x into the list l and therefore does not increase the size of the data if one compares the input (the list of length n plus another number) and the output (a list of length $n+1$). The "sort"-term just iterates the "insert"-term such that it does not increase the size either.

3.3.1.1. System F

The translation of the above algorithm into System F is straight forward (again we assume the existence of $\leq\colon N \to N \to B$):

$$
\begin{aligned}
L(\tau) &:= \forall\alpha.(\tau \to \alpha \to \alpha) \to \alpha \to \alpha\\
N &:= \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha\\
B &:= \forall\alpha.\alpha \to \alpha \to \alpha\\
\text{insert} &:= \lambda x^{N}, l^{L(N)}.\, \overbrace{\Lambda\alpha.\lambda f^{N \to \alpha \to \alpha}, a^{\alpha}.\, ((l\, N \multimap \alpha\, \text{step}\, \text{base})\ x)}^{L(N)}\\
\text{step} &:= \lambda y^{N}, p^{N \to L(N)}, z^{N}.\, \left((\leq\ y\, z)\ \alpha\, \text{then else}\right)\\
\text{then} &:= (f\ y\ (p\, z))\\
\text{else} &:= (f\ z\ (p\, y))
\end{aligned}
$$

[2] In fact, the recursions do not use the second argument and are therefore only iterations.

$$\begin{aligned}
\text{base} &:= \lambda x.\,(f\,x\,a)\\
\text{sort} &:= \lambda l^{L(N)}.\Big(l\,L(N)\,\lambda x^{N}, p^{L(N)}.\,(\text{insert}\,x\,p)\;\underbrace{\Lambda\alpha.\lambda f, a.a}_{\textbf{nil}}\Big).
\end{aligned}$$

The Church encoding forces us to use iteration instead of recursion. Hence, the alternative implementation of the "else"-case from above is not possible although it might speed up the algorithm.

3.3.1.2. Linear System F

On the way to the version for Light Affine Logic, we first try to reformulate the algorithm in Linear System F. This seems to be possible without major changes because the algorithm is non-size-increasing and nearly linear already, with two exceptions:

- The $\leq\colon N \to N \to B$ "consumes" the two numbers. After the comparison they are gone and cannot be used for the further computation (here for the return value). Without giving a detailed implementation here, it is possible to formulate a linearly typed comparison predicate $\leq'\colon N \multimap N \multimap (N \otimes N \otimes B)$ which compares the two numbers and then returns the comparison result and the two numbers. This is essential of course in order to give a linear implementation of Insertion Sort.

- The second exception are the Church encodings of lists and natural numbers. While the latter is only of interest in the implementation of $\leq'$, the former will become clear in the following attempt to implement Insertion Sort in Linear System F:

$$\begin{aligned}
L(\tau) &:= \forall\alpha.(\tau \multimap \alpha \multimap \alpha) \multimap \alpha \multimap \alpha\\
N &:= \forall\alpha.(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha\\
B &:= \forall\alpha.\alpha \multimap \alpha \multimap \alpha\\
\text{insert} &:= \lambda x^{N}, l^{L(N)}.\overbrace{\Lambda\alpha.\lambda f^{N\multimap\alpha\multimap\alpha}, a^{\alpha}.\,((l\,N \multimap \alpha\,\text{step}\,\text{base})\;x)}^{L(N)}\\
\text{step} &:= \lambda y^{N}, p^{N\multimap L(N)}, z^{N}.\Big((\leq'\;y\,z)\;\alpha\,\lambda y, z, b.\text{ifthenelse}\Big)\\
\text{ifthenelse} &:= (\!(b\,(N\multimap\alpha)\multimap(N\multimap\alpha\multimap\alpha)\multimap N\multimap N\multimap\alpha\,\text{then}\,\text{else})\,p\,f\,y\,z)\\
\text{then} &:= \lambda p, f, y, z.\,(f\;y\;\;(p\,z))
\end{aligned}$$

$$\begin{aligned}
\text{else} &:= \lambda p, f, y, z.\,(f\ z\ (p\,y)) \\
\text{base} &:= \lambda x.\,(f\,x\,a) \\
\text{sort} &:= \lambda l^{L(N)}.\Big(l\,L(N)\,\lambda x^{N}, p^{L(N)}.\,(\text{insert}\,x\,p)\ \underbrace{\Lambda\alpha.\lambda f, a.a}_{\textbf{nil}}\Big).
\end{aligned}$$

Clearly, with the – a bit complicated – "ifthenelse"-term we manage to make the original algorithm linear which looks completely fine. But the problem lays somewhere else: which lists can be built in Linear System F using the Church encoding? Already the list of length 2 has to apply the step term twice which contradicts the linearity and is not typable therefore. In other words, while the upper algorithm is type correct, the involved (Church like) data types are not what we had in mind for lists.

Remark 3.14. In the implementation above, in System F and also in Linear System F, we have moved the $\Lambda\alpha.\lambda f \lambda a \ldots$ outside the step term of the iteration (take a look at the "insert"-line above). In this context. this is a completely arbitrary choice. It would be equally possible to do this abstraction (and therefore the elimination as well) in every step. In fact, this would fit much better to the System T formulation of Insertion Sort. In the following section though, the modalities (i.e. the boxes) will enforce the formulation from above where the abstractions are "pulled out" from the inner iteration.

3.3.1.3. Light Affine Logic

The idea of Linear Logic is to add modalities to the types of Linear System F to make (restricted) duplication possible in a controlled way. E.g., in Light Affine Logic we have the stratification property to keep complexity under control. It gives a kind of symmetry of the number of modalities in the types, e.g. that there is no general function $!\tau \multimap \tau$, but only $!\tau \multimap \S\tau$. I.e., if we eliminate the !-modality, we have to add the paragraph § to the result type.

Again, we assume having an implementation of $\leq'$: $N \multimap N \multimap N \otimes N \otimes B$ with

$$\begin{aligned}
N &:= \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha) \\
L(\tau) &:= \forall\alpha.!(\tau \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha) \\
B &:= \forall\alpha.\alpha \multimap \alpha \multimap \alpha.
\end{aligned}$$

Before giving the term of the "insert"-term, we look at the simpler, but in fact very similar algorithm of iterating the identity:

Example 3.15 (Identity iteration, pull-out trick)**.** In System T the iteration of the identity step term would look like this:

$$\mathrm{IdIt} \quad := \quad \lambda l^{L(N)}.\, (\boldsymbol{\mathcal{R}}_{N,L(N)}\, l\, \lambda x, l', p.\, (\mathbf{cons}_N\, x\, p)\ \mathbf{nil}_N)\,.$$

In System F we would get something similar, but we now have two choices to place the Church abstraction $\Lambda\alpha\lambda f\lambda y$:

$$\begin{aligned}
\mathrm{IdIt} &:= \lambda l^{L(N)}.\Big(l\, L(N)\ \overbrace{\lambda x\lambda p.\Lambda\alpha\lambda f\lambda y.\, (f\, x\ (p\, \alpha\, f\, y))}^{\text{step}}\ \overbrace{\Lambda\alpha\lambda f\lambda x.x}^{\text{base}}\Big)\\
\mathrm{IdIt}' &:= \lambda l^{L(N)}.\Lambda\alpha\lambda f\lambda y.\big(l\, L(N)\ \underbrace{\lambda x\lambda p.\, (f\, x\, p)}_{\text{step}}\ \underbrace{y}_{\text{base}}\big).
\end{aligned}$$

The first version is the direct translation of the System T term into F. The second though needs some deeper insight into the structure of the algorithm to see that $\Lambda\alpha\lambda f\lambda y$ can be placed in front of the iteration.

If we now try to add the boxes of LAL to both terms, we get the following:

$$\mathrm{IdIt} := \lambda l^{L(N)}.\Big(l\, L(N)\, !\boxed{\lambda x\lambda p.\Lambda\alpha\lambda f\,\S\boxed{\lambda y.\big(f\, x\big](p\, \alpha\, f\, \S\boxed{y}\,)\big]_{\S}}}\ \S\boxed{\Lambda\alpha\lambda f.\S\boxed{\lambda x.x}}\Big)$$

$$\mathrm{IdIt}' := \lambda l^{L(N)}.\Lambda\alpha\lambda f.\S\boxed{\lambda y.\Big(\big](l\, L(N)\, !\boxed{\lambda x\lambda p.(]f[_!\ x\, p)}\,)\big]_{\S}\ y\Big)}\,.$$

The crucial difference of these two variations is the type:

$$\begin{aligned}
\mathrm{IdIt} &: L(N) \multimap \S L(N)\\
\mathrm{IdIt}' &: L(N) \multimap L(N).
\end{aligned}$$

Of course, the second type is much better in the sense that the first can be defined by the second, but not the other way round. The used method, i.e. to pull out the abstractions in front of the iteration and the §-box, we will call the pull-out trick in the following.

With the pull-out trick in mind, we can now formulate the "insert"-term. In order to iterate this later, we have to find a formulation which has a

symmetric type, i.e. like

$$\text{insert} : \S N \multimap L(N) \multimap L(N).$$

Note that the first parameter has to be on the same level (in the sense of Definition 2.39) as the numbers inside the list. This is necessary because the comparison $\leq'$ takes two numbers on the same level as arguments. Hence, the first parameter of "insert" has to be of type $\S N$ and *not* just N.

The result type is a list. Hence, the question is where to put the §-box (which comes from the § in $L(N)$). Here, we follow the Example 3.15 from above and apply the pull-out trick to the "insert" formulation: the abstractions of α and of the step function of the result are placed outside the box. Like in Example 3.15 we could not get the desired symmetric type otherwise (but something like $\S N \multimap L(N) \multimap \S L(N)$; compare Remark 3.14). These considerations finally lead to the following implementation of "insert":

$$\begin{aligned}
\text{insert} &:= \lambda x^{\S N}, l^{L(N)}.\overbrace{\Lambda\alpha.\lambda f^{!(N\multimap\alpha\multimap\alpha)}.\S\boxed{\lambda a^{\alpha}.\big((]\,(l\ N{\multimap}\alpha\ \text{step})[_{\S}\ \text{base})\ ]x[_{\S}\big)}}^{L(N)} \\
\text{step} &:=!\boxed{\lambda y^{N}, p^{N\multimap\alpha}, z^{N}.\Big((\leq'\ y\ z)\,\alpha\,\lambda y, z, b.\text{ifthenelse}\Big)} \\
\text{ifthenelse} &:= \big((b\,(N{\multimap}\alpha){\multimap}(N{\multimap}\alpha{\multimap}\alpha){\multimap}N{\multimap}N{\multimap}\alpha\,\text{then else})\,p\ ]f[_{!}\ y\,z\big) \\
\text{then} &:= \lambda p, f', y, z.\ \big(f'\ y\ \ (p\,z)\big) \\
\text{else} &:= \lambda p, f', y, z.\ \big(f'\ z\ \ (p\,y)\big) \\
\text{base} &:= \lambda x.\ \big(]f[_{!}\ x\,a\big)\,.
\end{aligned}$$

First, we try to understand how the instances of the step function f of the result are used: it appears in the step term of the iteration exactly once (of course in the "de-banged" version $]f[_{!}$) as this is required by the $(!_1)$-rule. Another copy is used in the base case. Here though, the (§)-rule is used to type the term such that we could even use more than one copy there.

More interesting is the first occurrence that should be analysed a bit further:

- The step f of the result is used once in the iteration step. There is no other way to use it twice or more times. The $(!_{0/1})$-rules do not allow this. But as the step's type in $L(N)$ is $!(N \multimap \alpha \multimap \alpha)$ there is

no other choice than using the $(!_{0/1})$-rules to type it because we have to get rid of the ! to apply the step f in the step of the iteration.

- If f had to be used more than once (e.g. like in a doubling function, compare Section 3.2.1.2), we could not use the pull-out trick to move the abstractions outside the box. If we kept the abstractions inside the box though, we could use arbitrary many instances of f because we could use the (§)-rule to "de-bang" f. Of course, we would get an asymmetric type at the end (compare with Example 3.15).
- The f corresponds to the use of the $\mathbf{cons}_N$-constant in the System T formulation. Hence, being forced to use at most one instance of f is equivalent to use only one $\mathbf{cons}_N$ in the step term. This is the central idea for the $\mathrm{LLT}_!$ calculus in Chapter 6. Moreover, this reminds a lot of the restriction in LFPL that we only have one instance of the $\diamondsuit$-term in the step of an LFPL-iteration. This similarity will be the topic of Chapter 7: symmetric LAL-function are non-size-increasing (also compare Remark 3.9).

Because we are able to formulate "insert" with the symmetric type, it is straightforward to iterate it in order to get the full Insertion Sort algorithm (with a little exception as we will see below):

$$\mathrm{sort}' \quad := \quad \lambda l^{L(N)}.\left(l\, L(N)\, !\, \boxed{\lambda x^N \lambda p^{L(N)}\, (\mathrm{insert}\, x\, p)}\, \Lambda\alpha\lambda f.\S \boxed{\lambda x.x}\right).$$

This term is not correctly typed yet because the x is of the wrong level. In order to apply it to "insert" term, we have to lift it, i.e. apply a function $\uparrow_N^{\S N}: N \multimap \S N$:

$$\mathrm{sort} : L(N) \multimap \S L(N)$$

$$\mathrm{sort} := \lambda l^{L(N)}.\Big(l\, L(N)\, !\, \boxed{\lambda x^N \lambda p^{L(N)}\, \Big(\mathrm{insert}\, (\uparrow_N^{\S N}\ x)\, p\Big)}\, \Lambda\alpha\lambda f.\S \boxed{\lambda x.x}\Big)$$

$$\uparrow_N^{\S N} := \lambda x^N.\S\boxed{\left(\Big(x\, N\, !\, \boxed{\lambda p^N.\Lambda\alpha.\lambda f.\S\boxed{\lambda a.(]f[_!\ (]\,(p\,\alpha\, f)[_\S\ a))}}\Big)\right[_\S\ \Lambda\alpha\lambda f.\S\boxed{\lambda x.x}\Big)}.$$

The type of "sort" is not symmetric anymore such that it is not possible to iterate "sort". So, why is that? Obviously, "sort" is non-size-increasing. Can we apply the pull-out trick here again to get symmetric version of the Insertion Sort algorithm?

- A first theoretical consideration about the complexity: assume sort_0:=sort was of the symmetric type $L(N) \multimap L(N)$ by applying the pull-out trick. Then, we could iterate sort_0 to get sort_1. Its complexity would be at least $\mathcal{O}(n^3)$ (n times Insertion Sort, which is quadratic). If we could apply the pull-out trick again to get a symmetric sort_1, we could iterate it again to get sort_2 with complexity of at least $\mathcal{O}(n^4)$, and so on. Hence, one could create a polynomial complexity hierarchy with this construction. But it is known that the degree of the complexity polynomial for fixed depth LAL proof nets is constant. Here, the proof net depth would be fixed indeed. In other words, this whole construction cannot work as described.

- On the syntactical level: to apply the pull-out trick we first have to identify the instances of the step f of the result $\Lambda\alpha\lambda f.\S\boxed{\lambda a...}$. Here, the result of the "insert"-iteration is also the result of the "sort"-iteration. In other words, the f of the "insert"-term must be "pulled out" somehow to apply the trick. But in the "insert"-term it must be "available" as a banged variable because it is used more than once (in each step of the iteration). By "pulling it out" in front of the "sort"-iteration though, the f would be singly banged as in $!(\alpha \multimap \alpha)$, which would allow only exactly one copy $]f[_!$ in the "sort"-iteration step which has no ! anymore in its type. In fact, in order to get the f often enough, in the inner iteration step a type like $!!(\alpha \multimap \alpha)$ would be needed: one ! for enough copies of $]f[_!$ in the "sort"-step, and one ! for enough copies of $]]f[_!\,[_!$ in the "insert"-step. For this though one would need a different type for the result of the whole sorting algorithm, something like $L(N)' := \forall\alpha.!!(N \multimap \alpha \multimap \alpha) \multimap \S\S(\alpha \multimap \alpha)$, a completely different data type of course.

Hence, it seems that we do not have a chance to get the symmetric variant of the Insertion Sort. This again is due to the stratification property of LAL. We will see further below the same effect in the definition of polynomials.

3.3.1.4. LT

In the Bellantoni-Cook-style [Bel92] predicative recursion of the system LT the recursion argument is incomplete. Hence, another (nested) recursion over the recursion argument is not possible.

The Insertion Sort algorithm needs exactly this though, i.e. in the step of the "sort"-recursion an inner recursion is executed to insert the element

at the right position. The result is given to the next step of the "sort"-recursion. But there it is not complete to drive the insertion.

In fact, this observation was, originally, one of the motivations to invent LFPL [Hof99b].

3.3.1.5. LFPL

Hofmann's LFPL is exactly built after the recursion scheme of Insertion Sort. Hence, it is not surprising that we can just take the System T algorithm and translate into the LFPL syntax (with $N := L(U)$):

$$\begin{aligned}
\text{insert} &:= \lambda x^{N}, e^{\Diamond}, l^{L(N)}.\,((l\,\text{step}\,\text{base})\ x)\\
\text{step} &:= \lambda y^{N}, d^{\Diamond}, p^{N\to L(N)}, z^{N}.\,\Big((\leq\ y\,z)\ \lambda b^{B}, y^{N}, z^{N}.\,(b\,\langle\text{then},\text{else}\rangle)\Big)\\
\text{then} &:= (\mathbf{cons}_{N}\,y\,d\ (p\,z))\\
\text{else} &:= (\mathbf{cons}_{N}\,z\,d\ (p\,y))\\
\text{base} &:= \lambda y.\,(\mathbf{cons}_{N}\,y\,e\,\mathbf{nil}_{N})\\
\text{sort} &:= \lambda l^{L(N)}.\,(l\,\text{insert}\,\mathbf{nil}_{N})\,.
\end{aligned}$$

Of course, we also need the special comparison predicate $\leq\colon N \multimap N \multimap (B \otimes N \otimes N)$ in this linear setting. It gives back the original values of the two inputs in order to reuse them to compute the selected branch.

> *Remark* 3.16. In contrast to the considerations for LAL about the construction of the polynomial hierarchy by iterating "sort" we will not get a contradiction here. The "sort"-term here is symmetric (there are no modalities in LFPL at all) and iterating it is possible without restriction. The complexity proof by [AS00] even gives an explicit way to calculate the degree of the complexity polynomial. Essentially, the nesting of the closed step terms of the LFPL iteration determines the degree. For Insertion Sort we have two (or three with the iteration inside $\leq$) nested iterations. Hence, the complexity is $\mathcal{O}(n^2)$ (or $\mathcal{O}(n^3)$ with $\leq$).

3.3.2. Polynomials

Most completeness proofs for polynomial time systems are based on a simulation of polynomial time Turing Machines, i.e. the completeness proofs give a mapping of a Turing Machine program to a simulation of it computed

in polynomial time on the considered computation model. One basic ingredient for these completeness results is a way to make the step function of a Turing Machine being executed polynomially often. For that polynomials on natural numbers, lists or whatever data structure is involved, are to be defined using

- addition (i.e. an algorithm which creates a data structure of the size equal to the sum of the inputs)
- and multiplication (i.e. an algorithm which creates a data structure of the size equal to the product of the inputs)

both with unary numbers $N := L(U)$[3].

For System T this gives:

$$+ := \lambda l_1, l_2.\, \left(\boldsymbol{\mathcal{R}}_{U,L(U)}\, l_1\, \lambda x, r, p.\, (\mathbf{cons}_U\, x\, p)\; l_2\right)$$
$$* := \lambda l_1, l_2.\, \left(\boldsymbol{\mathcal{R}}_{U,L(U)}\, l_1\, \lambda x, r, p.\, (+\, l_2\, p)\; \mathbf{nil}_U\right).$$

3.3.2.1. LT

The System T terms from above translate directly into LT:

$$+ := \lambda \bar{l_1}, l_2.\, \left(\boldsymbol{\mathcal{R}}_{U,L(U)}\, \bar{l_1}\, \lambda \bar{x}, \bar{r}, p.\, (\mathbf{cons}_U\, \bar{x}\, p)\; l_2\right)$$
$$* := \lambda \bar{l_1}, \bar{l_2}.\, \left(\boldsymbol{\mathcal{R}}_{U,L(U)}\, \bar{l_1}\, \lambda \bar{x}, \bar{r}, p.\, \left(+\, \bar{l_2}\, p\right)\; \mathbf{nil}_U\right).$$

Addition makes use of (or in other words: consumes) one complete argument. For multiplication a second complete one is needed to "drive" the outer recursion.

Similarly, in order to increase the degree of the calculated polynomial by one, another complete argument is needed. Because in LT there is no restriction on the contraction of ground types, all those complete inputs of polynomial can be contracted freely to give enough copies of the input to drive all recursions. I.e., a value like n^k for fixed k can be computed by taking k copies of the input n and applying $*$.

[3] The data type does not really matter here. The size of the data structures as the number of constructors is important at this point. For simplicity usually unary numbers are used.

3.3.2.2. LFPL

Addition is non-size-increasing and its implementation in LFPL is straightforward:

$$+ := \lambda l_1^{L(U)}, l_2^{L(U)}.\, (l_1\ \mathbf{cons}_U\ l_2)\,.$$

Multiplication, as a size-increasing algorithm, is not definable in LFPL. If one took the usual System T multiplication to implement it in LFPL, it cannot be typed because the step term is not closed (compare above in the LT case).

3.3.2.3. Light Affine Logic

Addition is simple concatenation in LAL which is possible even without iteration, and with a symmetric type $N \multimap N \multimap N$:

$$+ := \lambda l_1, l_2.\Lambda\alpha.\lambda f.\S\boxed{\lambda x.\left(](l_1\ \alpha\ f)[_\S\ \left(](l_2\ \alpha\ f)[_\S\ x\right)\right)}^{N\multimap N\multimap N}\,.$$

The impredicative encoding "abstracts out" the base case. Hence, it is enough to plug the second number into the base case of the first iteration functional (i.e. the first Church number). In other words, concatenation of data structures is a constant time operation in this system.

In contrast, the multiplication requires iteration. In every step addition is applied with l_2 as the argument. In order to make this possible, l_2 must have a banged type $!N$. This, though, also forces us to lift the result type to the next level via $\S N$:

$$* := \lambda l_1, l_2.\left(l_1\ N\ !\boxed{(+\ ]l_2[_!)}\ \S\boxed{0}\right)^{N\multimap !N\multimap \S N}\,.$$

Can we do any better? Is there an implementation of type $N \multimap N \multimap N$ for multiplication as well? This can be answered via some complexity considerations: If we had $* : N \multimap N \multimap N$, we could formulate:

$$t := \lambda l^{!N}.\S\boxed{(*\ ]l[_!\ (*\ ]l[_!\ \ldots(*\ ]l[_!\ ]l[_!)))}$$

with k copies of $]l[_!$. Obviously, $\left(t\,!\boxed{l}\right)$ will compute $\left|l\right|^k$, and the maximal level of $\left(t\,!\boxed{l}\right)$ is independent of k. Hence, we can increase k freely and compute any polynomial $\left|l\right|^k$. By the complexity theorem of e.g. Asperti

and Roversi[AR00], it is known that the maximal degree of the complexity polynomial only depends on the maximal level appearing in the term. This contradicts this situation were we can set k freely.

3.4. Conclusion and Outlook

In this chapter we have discussed a collection of algorithms:

- some which require a certain normalisation strategy to be efficient (Section 3.1),
- some which are exponential (or worse) per se (Section 3.2),
- some which stress the intentional expressivity of the type system (Section 3.3).

In the first case, we have analysed how the considered calculi enforce this strategy while reducing a term.

In the second case, the implementation cannot be type correct, and hence, the type systems must outlaw the algorithm by some structural restriction on the allowed recursion schema. Furthermore, we have traced back the exponential behaviour of the normalisation if such an implementation was typable in a system or we pointed out where the complexity proofs of the normalisation would break in this case.

In the third case, we have identified several difficulties to express algorithms in their natural formulation and structure:

- Strict linearity in a type system makes the usual formulation of "if-then-else" constructions impossible if the predicate and the branches share free variables (compare Insertion Sort in Section 3.3.1.3 and 3.3.1.5). This means that predicates have to be reformulated in a way that they "give back" the original values in order to pass these to the selected branch. In Chapter 4 the extension δLFPL of LFPL is introduced and shown to be sound for polynomial time, which solves this very problem by allowing certain non-linearities.
- An essential technique in Light Affine Logic is the pull-out trick. It is often necessary to type an algorithm symmetrically, i.e. in such a way that it can be iterated (compare Example 3.15). Chapter 6 will introduce a complete and correct System T variant LLT$_!$ which

integrates this trick seamlessly, i.e. without the need to explicitly pull-out the abstractions.

- The pull-out trick is not always applicable (compare Insertion Sort in LAL), even though the algorithm is non-size-increasing. Chapter 7 will give a solution to this problem by introducing a calculus LLFPL$_!$ which has light iteration and impredicative iteration (like in LFPL) in the same system.

- Hofmann's LFPL does – by design – only support non-size-increasing algorithms. Those can be expressed very elegantly in many cases. But even simple multiplication increases the size and is therefore outside the expressivity of LFPL. In this sense LLFPL$_!$ in Chapter 7 can also be considered as an extension of LFPL to support size-increases in a controlled way (using levels and stratification).

4

Relaxing Linearity

A central concept of this thesis is linearity, i.e. the occurrence of a bound variable in a term at most once. The reason that this can help to control complexity is that the substituent does not have to be duplicated during a beta reduction. In Hofmann's LFPL system this leads to the invariant of the system that the number of diamonds does not increase during normalisation.

Linearity though is a very restrictive property of a program. Many algorithms, although they make essentially linear usage of every bound variable if you consider the recursion schemes, use some variables in a non-linear way which does not harm the complexity. This chapter studies an extension of Hofmann's LFPL that allows certain non-linearities in terms. The normalisation proof of [AS00] is extended in order to also cover this extension. Moreover, the technical tools needed for this allow us to improve the original complexity theorem in such a way that the normalisation order is less restrictive. This leads to a better understanding of LFPL's normalisation in general.

Structure of this chapter After the motivation of extensions to Hofmann's LFPL in Section 4.1, we introduce the extended calculus δLFPL formally by giving the types, the terms and the typing rules. In Section 4.3 the normalisation of δLFPL is defined. Section 4.4 is the core of this chapter. It consists of the introduction of the *variable interaction* concept in Subsection

4.4.1 and the definition of the *list length measure* in Subsection 4.4.2. The latter is the basis of the complexity analysis of the normalisation in Section 4.5. The chapter ends in Section 4.6 with a conclusion by discussing the results and their limitations, and it gives an outlook of possible further work.

4.1. Motivation

Consider the following algorithm, written in System T with the predefined function $\leq\colon L(B) \to L(B) \to B$ to compare two numbers:

$$\begin{aligned}
&\text{insert} : L(B) \to L(L(B)) \to L(L(B))\\
&\text{insert} := \lambda x, l.\big((\boldsymbol{\mathcal{R}}_{L(B),L(B)\to L(L(B))}\, l\, \text{step}\, \lambda z.z)\, x\big)\\
&\quad \text{step} := \lambda y, l, p.\lambda z.\big(\mathbf{Case}_{L(L(B))}\, (\leq\, z\, y)\, (\mathbf{cons}_{L(B)}\, z\, (p\, y))\, (\mathbf{cons}_{L(B)}\, y\, (p\, z))\big).
\end{aligned}$$

The algorithms inserts the number x into the list l. This is done by iterating through the list in order to find the first position where the element y in the list is bigger than z. We see that the linearity restriction for x forces us to change the result type of the recursion, e.g. into a function $L(B) \to L(L(B))$. Otherwise, x would have to be free in every step term.

Of course, even this formulation of the algorithm is not linear yet. First, the two branches of the **Case** share the variables z, p, y. Though, the calculus LFPL has a construct $\langle s, t\rangle$ for that. Hence, this can easily be made linear in LFPL.

The bigger problem is the occurrence of the z and the y in the boolean argument to **Case** and in the branches of **Case**. LFPL does not provide any way to type that. The usual solution in [Hof99a, AS00] is to postulate that $\leq$ is of type $L(B) \to L(B) \to L(B) \otimes L(B) \otimes B$, i.e. it "gives back" the original numbers (compare Section 3.3.1.2). This way one can use the left and middle component of the result of $(\leq\, z\, y)$ in the branches of the **Case**.

This kind of program transformations complicates the program. Changing the definition of $\leq$ into the second variant is not trivial either, and one might wonder whether this is needed at all.

The "insert" algorithm, that uses z and y twice, looks completely harmless because the result of the comparison $(\leq\, z\, y)$ is just a boolean, i.e. the two numbers are thrown away during the computation. Hence, from the point of

view of non-size-increasing computation, this non-linearity *does not harm*. The questions we will answer in the following are:

- How can we extend LFPL with harmless non-linearities?
- How restricted does the normalisation order have to be to stay in polynomial time?

This chapter answers these questions by introducing a variant of LFPL with a duplication operator, which is then studied in detail. Under certain restrictions of this new operator (compare Assumptions 4.22 and 4.49), the calculus will be shown to be sound for polynomial time.

4.2. The Extended Calculus δLFPL

Definition 4.1 (Types)**.** The set $\mathrm{Ty}_{\delta\mathrm{LFPL}}$ of linear types is defined inductively by:

$$\sigma, \tau ::= \diamondsuit \mid B \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid \sigma \times \tau \mid L(\sigma).$$

Definition 4.2 (Terms and Constants)**.** Let V be a countably infinite set of variable names. The set $\mathrm{Tm}_{\delta\mathrm{LFPL}}$ of terms is inductively defined by

$$r, s, t ::= x^{\tau} \mid c \mid \lambda x^{\tau}.t \mid \langle t, s\rangle \mid (t\, s) \mid \delta t$$

with $c \in \mathrm{Cnst}_{\delta\mathrm{LFPL}}$, $x \in V$ and $\tau \in \mathrm{Ty}_{\delta\mathrm{LFPL}}$. Terms which are equal up to the naming of bound variables are identified. Free and bound variables are defined as in Definition 2.17 for LFPL, with the additional clause $\mathrm{FV}(\delta t) := \mathrm{FV}(t)$, but by collecting the free variables as located subterms (see Definition 2.6).

The set of variable subterms $x \trianglelefteq t$ with $x \notin \mathrm{FV}(t)$ (i.e. bound variables) is denoted by $\mathrm{BV}(t)$.

Constants are as in Definition 2.17 for LFPL, i.e. $\mathrm{Cnst}_{\delta\mathrm{LFPL}} = \mathrm{Cnst}_{\mathrm{LFPL}}$. The subterm relation $\lhd_{\delta\mathrm{LFPL}}$ is defined as $\lhd_{\mathrm{LFPL}}$ with the additional clause $t \lhd_{\delta\mathrm{LFPL}} \delta t$.

Definition 4.3 (Typing)**.** Contexts are defined as in Definition 2.4 as finite mappings from variable names V to types $\mathrm{Ty}_{\delta\mathrm{LFPL}}$.

The relation between contexts Γ and Λ, a untyped term $t \in \mathrm{Tm}_{\delta\mathrm{LFPL}}$ and a type $\tau \in \mathrm{Ty}_{\delta\mathrm{LFPL}}$, denoted $\Gamma;\Lambda \vdash t^\tau$, is inductively defined as follows:

$$\frac{}{\Gamma, x^\tau;\Lambda \vdash x^\tau}\ (\mathrm{Var})$$

$$\frac{}{\Gamma;\Lambda, x^\tau \vdash x^\tau}\ (\mathrm{Var}_l)$$

$$\frac{c \text{ constand of type } \tau}{\Gamma,\Lambda \vdash c^\tau}\ (\mathrm{Const})$$

$$\frac{\Gamma;\Lambda, x^\sigma \vdash t^\tau}{\Gamma;\Lambda \vdash (\lambda x^\sigma.t)^{\sigma\multimap\tau}}\ (\multimap^+) \qquad \frac{\Gamma;\Lambda_1 \vdash t^{\sigma\multimap\tau} \quad \Gamma;\Lambda_2 \vdash s^\sigma}{\Gamma;\Lambda_1,\Lambda_2 \vdash (t\,s)^\tau}\ (\multimap^-)$$

$$\frac{\Gamma;\Lambda \vdash s^\sigma \quad \Gamma;\Lambda \vdash t^\tau}{\Gamma;\Lambda \vdash \langle s,t\rangle^{\sigma\times\tau}}\ (\times^+) \qquad \frac{\Gamma;\Lambda \vdash t^{\sigma\times\tau}}{\Gamma;\Lambda \vdash (t\,\mathbf{tt})^\sigma}\ (\times_0^-) \frac{\Gamma;\Lambda \vdash t^{\sigma\times\tau}}{\Gamma;\Lambda \vdash (t\,\mathbf{ff})^\tau}\ (\times_1^-)$$

$$\frac{\Gamma;\Lambda_1 \vdash t^B \quad \Gamma;\Lambda_2 \vdash \langle s,r\rangle^{\tau\times\tau}}{\Gamma;\Lambda_1,\Lambda_2 \vdash (t\,\langle s,r\rangle)^\tau}\ (B^-)$$

$$\frac{\Gamma;\Lambda \vdash t^{L(\tau)} \quad \emptyset;\emptyset \vdash h^{\diamond\multimap\tau\multimap\sigma\multimap\sigma}}{\Gamma;\Lambda \vdash (t\,h)^{\sigma\multimap\sigma}}\ (L(\tau)^-)$$

$$\frac{\Gamma;\Lambda_1 \vdash t^{\rho\otimes\tau} \quad \Gamma;\Lambda_2 \vdash s^{\rho\multimap\tau\multimap\sigma}}{\Gamma;\Lambda_1,\Lambda_2 \vdash (t\,s)^\sigma}\ (\otimes^-)$$

$$\frac{\Gamma;\Lambda \vdash f^{\sigma\multimap\tau}}{\Gamma;\Lambda \vdash \delta f^{\sigma\multimap\tau\otimes\sigma}}\ (\mathrm{dup}^-)$$

with Γ_1,Γ_2 meaning the set union, such that Γ_1 and Γ_2 agree on the assigned types on the intersection of their domains. For $\Gamma;\Lambda$ the contexts Γ and Λ must have disjoint domains.

The left context is the *non-linear* one, i.e. variables in $(t\,s)$ on both sides can overlap if they are typed via the left context. The right context is

the *linear* part which corresponds to the contexts in the original LFPL in Definition 2.18.

The non-linear variables cannot be bound by a lambda abstraction. Hence, those variables are always free. In other words, a closed term (with minimal contexts) has no variables in the left context.

> *Remark* 4.4. Compare the context definition with that of System T (Definition 2.4) and LFPL (Definition 2.18). In the setting of this chapter, we split the environment into to two contexts. Both are represented as finite maps, *not* multisets as for LFPL. Basically, multiple occurrences of variables in a multiset LFPL context can be translated by putting the variable once into the left context of δLFPL. Therefore, we do not need to use multisets here.
>
> On the other hand, we cannot even use multisets for δLFPL because this would break subject reduction in the case of the reduction rule for $\Gamma; \emptyset \vdash (\delta f\ s)$ which duplicates s. With the left context Γ being a set we do not have to take care to have enough "copies" of $x \in \mathrm{FV}(s)$ in Γ.

The typing of the δf term construction expresses what was suggested in the motivation in Section 4.1: a term of type σ should be duplicated, one copy applied to the function $f^{\sigma \multimap \tau}$ and one copy given back as the right side of the pair of type $\tau \otimes \sigma$.

The typing rule (dup^-), as given above, *is not complete* yet. We will discuss necessary side-conditions later in this chapter before proving the complexity theorem. Just to give a hint: the type τ cannot be arbitrary because e.g. by choosing $\tau = \sigma = \diamondsuit$, one could duplicate the diamond easily. This cannot be allowed of course because then non-polynomial time algorithms become definable.

By the shape of (Var) and (Var_l), it is clear that we have full weakening in the sense of the following lemma:

Lemma 4.5 (Weakening)**.** *If* $\Gamma; \Lambda \vdash t^\tau$*, then also* $\Gamma'; \Lambda' \vdash t^\tau$ *holds for finite contexts* $\Gamma' \supseteq \Gamma$ *and* $\Lambda' \supseteq \Lambda$*.*

Proof. Easy induction over the typing derivation. □

Definition 4.6 (linear, almost-closed)**.** A variable subterm $x \trianglelefteq t$ is called linear in t if x is not free in t or x is typed in $\Gamma; \Lambda \vdash t^\tau$ via Λ.
A term t is called linear if all variables $x \in \mathrm{FV}(t)$ are linear in t.
A term t is called almost-closed if all free variables in t are of type $\diamondsuit$.
A subterm $s \trianglelefteq t$ is called bound in t if s has free variables which are bound in t.

In other words, a variable x is *linear* in t if it does not occur at all in t or, if it does, there is a typing derivation, such that x is in the right context, the linear one.

A *linear* term has only linear variables, i.e. the term can be typed with an empty left context.

In order to allow case distinctions in proofs later on, we state the following easy lemma:

Lemma 4.7. *For every variable x exactly one of the following cases holds:*

1. *$x \trianglelefteq t$ and x is not free in t,*
2. *$x \trianglelefteq t$ and x is free in t,*
3. *or $x \not\trianglelefteq t$.*

Proof. By definition of free and bound variables. □

Lemma 4.8. *For every typed $\lambda x^\sigma.t$ the variable x is linear in t.*

Proof. If $\lambda x^\sigma.t$ is typed, by the rule $(\multimap^+)$ there is a context $\Gamma; \Lambda$ with $\Gamma; \Lambda, x^\sigma \vdash t^\tau$. Hence, by definition x is linear in t. □

Lemma 4.9 (Linearity)**.** *Assume $\Gamma; \Lambda \vdash (t_1\, t_2)^\tau$. Then there are types τ_1 and τ_2 and disjoint contexts Λ_1, Λ_2 with $\Lambda_1 \cup \Lambda_2 = \Lambda$, such that $\Gamma; \Lambda_1 \vdash t_1^{\tau_1}$ and $\Gamma; \Lambda_2 \vdash t_2^{\tau_2}$. If τ_1 is a list type $L(\tau_1')$ then t_2 is closed.*

Proof. By the shape of the typing rules, the last rule of the typing derivation of $\Gamma; \Lambda \vdash (t_1\, t_2)^\tau$ must be one of $(\multimap^-)$, $(\times_0^-)$, $(\times_1^-)$, (B^-), $(\otimes^-)$ or $(L(\tau)^-)$. In the last case t_2 and with the weakening Lemma 4.5 we get $\Gamma; \Lambda_2 \vdash t_2^{\tau_2}$,

and similarly for $(\times_0^-)$ and $(\times_1^-)$. For the other cases we get Λ_1 and Λ_2 directly from the premises of the typing rules. □

Having two contexts implies the question which one is stronger in the sense in which direction we can move type assignments between them. The intuition behind the left non-linear and the right linear contexts suggests that variables can be moved freely from right to left (while adapting the typing derivation of course), and formally this is also the case:

Lemma 4.10. *Let be* $\Gamma; \Lambda \vdash t^\tau$*, then also* $\Gamma, \Lambda; \emptyset \vdash t^\tau$ *holds.*

Proof. We show that $\Gamma, \Lambda_1; \Lambda_2 \vdash t^\tau$ follows from $\Gamma; \Lambda_1, \Lambda_2 \vdash t^\tau$ for all disjoint $\Gamma, \Lambda_1, \Lambda_2$ which implies the claim of the lemma.

Induction on the typing of t and in each step case distinction on the shape of t:

Case x, c: trivial by (Var), (Var_l) and $(Const)$.

Case $\langle r, s\rangle$: Follows from $(\times^+)$ and the IH on r and s.

Case $\lambda x^\sigma.r^\tau$: It follows from $(\multimap^+)$ that $\Gamma; \Lambda_1, \Lambda_2, x^\sigma \vdash r$ and by IH we get $\Gamma, \Lambda_1; \Lambda_2, x^\sigma \vdash r^\tau$ which implies $\Gamma, \Lambda_1; \Lambda_2 \vdash \lambda x^\sigma.r^\tau$ by $(\multimap^+)$.

Case $(s\, r)$:

Subcase $(r^{\sigma\multimap\tau}\, s^\sigma)^\tau$: Typed by $(\multimap^-)$. By IH we get $\Gamma, \Lambda_1; \Lambda_{12} \vdash r^{\sigma\multimap\tau}$ and $\Gamma, \Lambda_2; \Lambda_{22} \vdash s^\sigma$ and by applying $(\multimap^-)$ again the claimed $\Gamma, \Lambda_1, \Lambda_2; \Lambda_{12}, \Lambda_{22} \vdash t^\tau$.

Subcase $\left(r^{\sigma\times\tau}\, \mathbf{tt}\right)^\sigma$: Typed by $(\times_1^-)$, trivial with IH.

Subcase $\left(r^{\sigma\times\tau}\, \mathbf{ff}\right)^\tau$: Typed by $(\times_2^-)$, trivial with IH.

Subcase $(v\, \langle r, s\rangle)^\tau$: Typed by (B^-). By IH $\Gamma, \Lambda_1; \Lambda_{12} \vdash v^B$ and $\Gamma, \Lambda_2; \Lambda_{22} \vdash \langle r, s\rangle^{\tau\times\tau}$. Applying (B^-) again yields $\Gamma, \Lambda_1, \Lambda_2; \Lambda_{12}, \Lambda_{22} \vdash (v\, \langle r, s\rangle)^\tau$.

Subcase $(r\, \lambda x^\tau, y^\rho.s)^\sigma$: Typed by $(\otimes^-)$. Analogous to the first subcase.

Subcase $\left(r^{L(\tau)}\, h\right)^{\sigma\multimap\sigma}$: Typed by $(L(\tau)^-)$. Trivial using IH for r.

Case $\delta f^{\sigma\multimap\tau\otimes\sigma}$: Typed by (dup^-). Trivial using IH for f. □

4.3. Normalisation

Definition 4.11 (Conversion and Reduction)**.** The conversion relation $\longmapsto$ is defined by:

$$
\begin{aligned}
(\lambda x^{\tau}.t\, s) &\longmapsto t[x := s] \\
(\langle s, t\rangle\, \mathbf{tt}) &\longmapsto s \\
(\langle s, t\rangle\, \mathbf{ff}) &\longmapsto t \\
(\mathbf{tt}\, \langle s, t\rangle) &\longmapsto s \\
(\mathbf{ff}\, \langle s, t\rangle) &\longmapsto t \\
((\otimes_{\sigma,\tau}\, s\, t)\ r) &\longmapsto (r\, s\, t) \\
(\mathbf{nil}_{\tau}\, h\, g) &\longmapsto g \\
((\mathbf{cons}_{\tau}\, d\, v\, x)\ h\, g) &\longmapsto (h\, d\, v\ (x\, h\, g)) \\
(\delta f\, s) &\longmapsto (\otimes_{\sigma,\tau}\ (f\, s)\ s) \qquad \text{if } s \text{ is normal.}
\end{aligned}
$$

The reduction relation $t \longrightarrow t'$ is inductively defined by:

$$
\frac{t \longmapsto t'}{t \longrightarrow t'}\ (\text{conv}) \qquad \frac{t \longrightarrow t'}{(t\, s) \longrightarrow (t'\, s)}\ (l) \qquad \frac{s \longrightarrow s'}{(t\, s) \longrightarrow (t\, s')}\ (r).
$$

Note that by this definition *reduction is not possible under λ-abstraction* (the λ-restriction) *or inside $\langle s, t\rangle$*.

Remark 4.12. The normalisation is mostly the standard LFPL reduction, given e.g. by Aehlig and Schwichtenberg [AS00], but with one important difference which makes it far more general: the conversion of the list step does not depend anymore on a full normalised list on the left of the redex. In [AS00] this term had to be normalised until at least the length of the resulting list is known precisely. The reason for this was the simple measure used to bound the number of times the step term h had to be duplicated. Here, we will use another, more global measure which does not need this special normalisation strategy.

Remark 4.13. We need to use the sharing technique as described in Remark 2.20 for pure LFPL. Otherwise, the beta-reduction would potentially double the term size. But due to forbidden reduction inside carte-

sian pairs, it is easy to apply sharing of subterms which the branches of cartesian pairs have in common.

Lemma 4.14 (Substitution). *If* $\Gamma; \Lambda, x^\sigma \vdash t^\tau$ *and* $\Gamma; \emptyset \vdash s^\sigma$*, then* $\Gamma; \Lambda \vdash t[x := s]^\tau$.

Proof. Induction on t:

Case $t = x$ or $t = c$: Trivial, with weakening Lemma 4.5.

Case $t = (t_1\, t_2)$: By Lemma 4.9 either $t[x := s] = (t_1[x := s]\, t_2)$ or $t[x := s] = (t_1\, t_2[x := s])$. Apply the IH to the substitution subterm.

Case $t = \lambda y^\nu.r$: Then t is typed with $(\multimap^+)$ s.t. $\Gamma \setminus y; (\Lambda, x^\sigma) \setminus y, y^\nu \vdash r^\rho$ with $\tau = \nu \multimap \rho$.

Subcase $x = y$: This implies $\Gamma; \Lambda, y^\nu \vdash r^\rho$ due to $(\multimap^+)$, i.e. x is not free in r. Hence, the claim $\Gamma; \Lambda \vdash t[x := s]^\rho$ follows with $t[x := s] = t$.

Subcase $x \neq y$: We get $\Gamma \setminus y; \Lambda \setminus y, y^\nu \vdash r[x := s]^\rho$ by IH. By $(\multimap^+)$ this gives the claim $\Gamma; \Lambda \vdash t[x := s]^\rho$.

The other cases are easy. □

By Lemma 4.10 every typed term can be "re-typed" in such a way, that all free variables are moved into the left context. As we cannot normalise below λ-abstractions anyway, this is good enough (remember: only variables in the right context can be bound).

For this kind of typed terms we can easily prove subject reduction, i.e. that the reduct can be typed with the same type after a normalisation step:

Lemma 4.15 (Subject reduction). *If* $\Gamma; \emptyset \vdash t^\tau$ *and* $t \longrightarrow t'$*, then* $\Gamma; \emptyset \vdash t'^\tau$.

Proof. By definition of $\longrightarrow$ only conversions have to be considered:

Case $(\lambda x^\tau.t\, s) \longmapsto t[x := s]$: by Lemma 4.14 with $\Lambda = \emptyset$.

Case $(\delta f\, s) \longmapsto (f\, s) \otimes s$: $(\delta f\, s)$ can only be typed due to $(\multimap)^-$, i.e. $\Gamma, \Lambda_1 \vdash \delta f^{\sigma \multimap \tau}$. But as normalisation under lambda abstractions is not allowed and $(\multimap^+)$ is the only rule which has a smaller context in the conclusion, we get $\Gamma; \emptyset \vdash \delta f^{\sigma \multimap \tau}$ and with the same argument $\Gamma; \emptyset \vdash s^\sigma$. δf can only be typed due to $(\mathrm{dup})^-$ s.t. $\tau = \nu \otimes \sigma$ for some ν and $\Gamma; \emptyset \vdash f^{\sigma \multimap \nu}$. Conversely, with $\Gamma; \emptyset \vdash f^{\sigma \multimap \nu}$ and $\Gamma; \emptyset \vdash s^\sigma$, $(\multimap)^-$, $(\otimes^-)$ and (Const) one gets $\Gamma; \emptyset \vdash (f\, s) \otimes s^{\nu \otimes \sigma}$.

The other cases are easy. □

4.4. Data Size Analysis

Next to the fact that the presented type system is sound, the more interesting question for us is the complexity of the normalisation. The standard LFPL system is polynomial time normalisable [AS00]. I.e., there is an normalisation strategy which needs a polynomial number of steps in the size of the input term.

Example 4.16. In Section 4.2 we added the δf construction to the system. As mentioned before, this extension in the general version will break the polynomial time property. Here is an example for this:

$$d := \lambda x^{L(\tau)}.\,(\delta\mathrm{Id}\,x)$$
$$e := \lambda x^{L(\tau)}.\left(x\,\lambda d^{\diamondsuit}, y, p^{L(\tau)}.\,((d\,p)\ \mathrm{cat})\ \left(\mathbf{cons}_\tau\ e^{\diamondsuit}\ y\ \mathbf{nil}_\tau\right)\right).$$

The function d duplicates a list. The function e uses the non-size-increasing concatenation function cat, and doubles the size of the list in every iteration step using d. This is obviously exponential, but expressible without problems with our extension so far.

In order to restrict the δf-construction, we first introduce the kind of types which are safe to duplicate:

Definition 4.17 (Passive types, passive subterms, passive free variables, active)**.** A type τ is called passive if every almost-closed normal term w^τ is closed, i.e. if every almost-closed term normalises to a closed term.

Let be $s^\tau \lhd t$ of a passive type τ and $\mathrm{FV}(s) \cap \mathrm{BV}(t) = \emptyset$, then s is called a passive subterm in t. The free variables of t with $x \trianglelefteq s \lhd t$ are called passive free variables for t. Free variables in t that are not passive for t are called active for t. The set of all active free variables (denoted as located subterms) for t is denoted by $\mathrm{FV}_a(t)$.

Example 4.18.

1. B is a passive type because every almost-closed term of type B reduces to either **tt** or **ff**. Moreover, when starting with an almost-

closed term t and $t \longrightarrow^* t'$, every subterm s'^B of t' that is not under λ-abstraction is almost-closed again.

2. $B \otimes B$ is passive.
3. $L(B)$ is not passive because $\left(\mathbf{cons}_B\, x^\Diamond\, \mathbf{tt}\, \mathbf{nil}\right)$ is almost-closed and normal, but not closed.
4. $B \multimap B$ is not passive because $\lambda x^B.\left(\langle x, y^\Diamond\rangle\, \mathbf{tt}\right)$ is normal (reduction under lambda is not allowed), almost-closed, but not closed.
5. $B \times B$ is not passive because $\langle\left(\lambda x^\Diamond.\mathbf{tt}\, y^\Diamond\right), \mathbf{ff}\rangle$ is normal (reduction inside cartesian products is not allowed), almost-closed, but not closed.

Example 4.19. A passive free variable of t is a free variable of t which is "under" a passive strict subterm of t. Take $t := \left(f^{\Diamond \multimap B}\, x^\Diamond\right) \otimes y^\Diamond$. Then x is passive in t because $(f\, x)$ is a passive subterm. But y is active for t because there is no such strict subterm s of t above y which is passive.

In other words, a passive strict subterm s of t turns all free variables "below" s into passive ones for t. Those which are not "covered" by such a passive subterm stay active for t. Hence, the intuition is that passive subterms "mask away" everything below in the syntax tree, such that it does not contribute to the data size outside anymore.

Remark 4.20. A passive subterm does not have to be almost-closed. In fact, even x^B is a passive term. The definition of passive subterm only requires that the type is passive, i.e. if the subterm was almost-closed, then it would normalise to a closed term. Due to this distinction between passive types and passive terms it makes sense to talk about passive variables in the first place.

Free variables under a passive subterm cannot "escape" it, i.e. they cannot become active during normalisation. If the passive subterm disappears the variables under it will do so as well. Otherwise, they stay passive:

Lemma 4.21. *Let be $t \longrightarrow^n t'$ and $x \in FV(t) - FV_a(t)$, i.e. x is passive in t. Then $x \notin FV_a(t')$ holds.*

Proof. Easy induction over n and case distinction according to the definition of $\longmapsto$. □

The central idea to control the normalisation complexity in the setting *with duplication*, is the following:

Duplication does not harm if only one copy of the duplicated data stays active and the other copy is protected under a passive subterm.

In order to study the normalisation with this idea, we make the following assumption:

Assumption 4.22. *Only allow passive types τ for $(\delta f\, s)^{\tau \otimes \sigma}$.*

Note here that σ (the duplicated type) is not restricted, only the type of the subterm $(f\, s)$, i.e. the left side of the reduct $(f\, s) \otimes s$. This passive type stops the left copy of s from taking any influence on the data size outside the subterm $(f\, s)$: after normalisation $(f\, s)$ will not have any free variable of type $\diamondsuit$ anymore because $(f\, s)$ is almost-closed (compare Definition 4.17).

Cartesian products The cartesian product allows common free variables in the branches because during normalisation it will be chosen which branch is used. I.e., not both can take effect on the data outside the product at the same time. Hence, we will always assume to know which branch is the right one by using the correct forecast for the choice:

Definition 4.23. [Forecast, effective]Let t be a term.
A mapping $\eta : \{\langle v, w\rangle \mid \langle v, w\rangle \trianglelefteq t\} \to \{\mathbf{tt}, \mathbf{ff}\}$ is called forecast for t.
A subterm s of t is called effective according to η if $s \not\trianglelefteq w$ in case of $\eta(\langle v, w\rangle) = \mathbf{tt}$ and $s \not\trianglelefteq v$ in case of $\eta(\langle v, w\rangle) = \mathbf{ff}$ for all $\langle v, w\rangle \trianglelefteq t$.
The set $\mathrm{FV}_\eta(t) \subseteq \mathrm{FV}(t)$ consists exactly of those free variables (as located subterms) that are effective according to η.

Lemma 4.24. *If $\Gamma; \Lambda \vdash \lambda x^{\tau}.s^{\tau \multimap \sigma}$, then for any forecast η for s the variable x occurs at most once in s as a subterm, that is effective for η.*

Proof. Induction on the term structure. □

Passive subterms (which are not variables themselves) make all of their free variables passive. They, kind of, protect the data below in the syntax tree from having an influence on list in other branches of the tree.

Every subterm is either below such a passive subterm, or there is no such subterm above. This passive subterm strictly above or the root of the syntax tree (if the passive subterm strictly above does not exist) is called the *context:*

Definition 4.25 (Context)**.** For $s \trianglelefteq t$ the term

$$c_t(s) = \min\{u \mid s \triangleleft u \trianglelefteq t \text{ and } u \text{ has passive type, or } u = t\}$$

is called the context of s in t.

4.4.1. Interacting Variables

In the following, we relate those subterms which can form a list together, i.e. where one is not protected from the other by a passive subterm in between. In order to get an idea of this subterm interaction, we start with three examples:

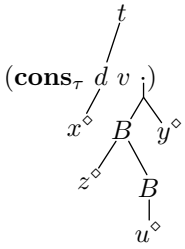
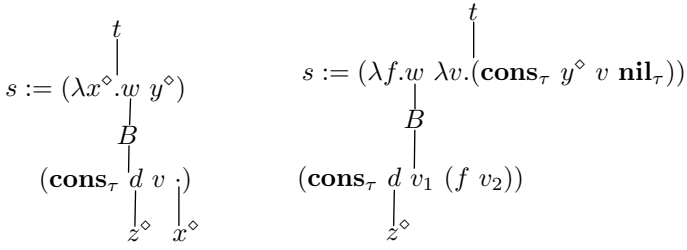


Figure 4.1.: Example syntax tree for interacting subterms

Example 4.26. In order to create a list, terms of the $\diamondsuit$-type are needed. Hence, consider the syntax tree in Figure 4.1.

Which variables of type $\diamondsuit$ can form list together of length ≥ 2?

Obviously, x and y could be part of the same list. On the other hand, z and u are both separated from x and y. Moreover, z and u cannot interact either because u has another passive subterm of type B above. Of course, z and u have a common passive subterm above, but the minimal one (i.e. the *context*) is a different one for each. x and y have the same *context*, namely $c_t((\mathbf{cons}_\tau\, d\, v\, \cdot))$.



(a) direct (b) higher type

Figure 4.2.: Interaction via abstractions

Example 4.27. Consider the syntax tree in Figure 4.2a. z and x interact in the sense of the previous example. But in this example abstractions come into play: the variable x is bound, and via substitution during a beta reduction, z can also interact with y.

Note that the context of z is the B-type term in the middle, while the context of y is the root of the tree. But even in such a case with different *contexts*, the abstraction makes the interaction possible.

Example 4.28. The interaction via a λ-abstraction in the previous example is quite direct. Without deeper understanding of the normalisation, the connection between z and y is visible. But this is not always the case: consider the third example in Figure 4.2b.

Can z and y interact in this case?

The λ-abstraction λf is of higher type. The variable f is a function and the term $\lambda f.w$ is applied to the subterm with y. During beta reduction, the right term is substituted into the position of f. Finally, it is applied to v_2, and obviously z and y form a list together.

These three example motivate the formalisation of the concept of interaction. The last example suggests that interaction, somehow, resembles the whole inner abstraction structure of a lambda term. In order to capture this, we introduce two simultaneously inductively defined relations:

Definition 4.29 (In the context, interact)**.** For every term t and a forecast η for t

- the relation $v \succ_t x$ between terms $v \trianglelefteq t$ and free variables $x \in FV(t)$
- and the relation $x \prec\!\!\diamond_t^\eta y$ between $x, y \in FV_\eta(t)$

are simultaneously defined as follows:

1. $c_t(x) \succ_t x$
2. $z \prec\!\!\diamond_s^\eta x \wedge \lambda z^\tau.s \trianglelefteq t \to c_t(\lambda z^\tau.s) \succ_t x$
3. $\exists c(c \succ_t x \wedge c \succ_t y) \to x \prec\!\!\diamond_t^\eta y.$

We say for $x \prec\!\!\diamond_t^\eta y$ that x interacts with y, and for $s \succ_t x$ that x is in the context of s.

The intuition for the three clauses is the following:

1. x is in its own context in the term t.
2. z and x can interact in a term s, and z is bound above, then x is also in the context of the abstraction. This is basically the case of Figure 4.2a.

3. If x and y have a common context c that they are in, then x can interact with y and vice versa.

The inductive closure over these clauses makes situations like in Figure 4.2b possible which go though more than one abstraction.

> *Remark* 4.30. The forecasts are part of the definition by restricting the domain of the $\multimap_t^\eta$ relation to the variables which are effective in t, i.e. $\mathrm{FV}_\eta(t)$. In other words, we do not care about ineffective variables at all.

In the following, we will show that the relation $\multimap_t^\eta$, in fact, is a *equivalence relation* over the (active) free variables of a term. But this alone would not tell us much. We also need the property that the equivalence relation is somehow compatible with the normalisation, especially that duplication via the δf construction creates copies of variables which reside in different equivalence classes. This will be the main work in Theorem 4.36.

Finally, we will show that, because of this well behaved duplication reduction, the sizes of the equivalence classes do not grow. I.e., they are non-size-increasing during normalisation.

But before we have to prove basic properties of the two relations:

Lemma 4.31.

1. *If $c \succ_t x$, then $x \trianglelefteq c$.*
2. *If $c \succ_t x$, then $c_t(x) \trianglelefteq c$.*

Proof. Claim 1: Induction on $\succ_t$:

- Base case: clear.
- Step case: If $c_t(\lambda z^\tau.s) \succ_t x$, we have $z \multimap_s^\eta x$ and by the 3rd clause the existence of an common context c of x and z with $c \trianglelefteq s$ and therefore $x \trianglelefteq s \trianglelefteq c_t(\lambda z.s)$.

Claim 2: By the shape of the clauses of $\succ_s$, the term c must be a context, i.e. a passive term or s itself. $c_t(x)$ is the smallest passive term above x. Hence, by the first claim we have $c_t(x) \trianglelefteq c$. □

Lemma 4.32.

1. *For $c \trianglelefteq s \lhd t$: If $c \succ_t x$, then $c \succ_s x$.*
2. *For $c \lhd s \lhd t$: If $c \succ_s x$, then $c \succ_t x$.*
3. *For $c \trianglelefteq s \trianglelefteq t$ with s of passive type: $c \succ_s x$ iff $c \succ_t x$.*

Proof. Claim 1 follows from the observation that $c_t(v) = c_s(v)$ holds for every $v \trianglelefteq s \trianglelefteq t$ with $c_t(v) \trianglelefteq s$. In either case, Clause 1 or Clause 2 of Definition 4.29, this gives $c \succ_s x$.

Claim 2 follows from the observation that $c_t(v) = c_s(v)$ holds for every $v \lhd s \trianglelefteq t$ with $c_s(v) \lhd s$.

For Claim 3 the case $c = s$ from left to right remains. Then $c = s = c_s(v)$ holds for some $v \lhd s$, the witness for $c \succ_s x$. Because s is passive, c is also the smallest passive term in t above v. Hence, $c_s(v) = c_t(v)$. □

Remark 4.33. For $c \trianglelefteq s \lhd t$ the Claim 2 of Lemma 4.32 would be wrong. Take the term

$$(f\,x)^{\diamondsuit} = c = s \lhd \lambda x.\left(g^{\diamondsuit \multimap B}\ (f\,x)^{\diamondsuit}\right)^{B}.$$

Then the variable x is in the context s in the term s, i.e. $c_s(x) = s \succ_s x$. But in the term t the context of x is the smallest passive term strictly above x which is $\left(g^{\diamondsuit \multimap B}\ (f\,x)^{\diamondsuit}\right)^{B}$ here.

This counter-example works because s is of type $\diamondsuit$, i.e. not a passive type. Otherwise, the Claim 3 of Lemma 4.32 applies.

Lemma 4.34.

1. *For $c_1 \succ_t x$, $c_2 \succ_t x$ either $c_1 \trianglelefteq c_2$ or $c_2 \trianglelefteq c_1$ is the case.*
2. *If $c \succ_t x$, $x \trianglelefteq s \lhd c \trianglelefteq t$ and no free variables of s are bound in t, then $c_t(s) \succ_t x$.*
3. *If $c \succ_{t[u:=s]} x$, $u \in FV(t)$ and $x \trianglelefteq s$, then $c \succ_t u$.*
4. *If $c \succ_t x$, $x \trianglelefteq s \lhd c \trianglelefteq t$ and s is not under a λ-abstraction, then $c = c_t(s)$.*

Proof. By $x \trianglelefteq c_1$, $x \trianglelefteq c_2$ due to Lemma 4.31 and the tree structure of terms, the Claim 1 is obvious.

Claim 2: First let c be $\trianglelefteq$-minimal with $c \succ_t x$ and $s \lhd c$. The case $c = c_t(s) = c_t(x)$ is trivial. Otherwise, there is a witness $\lambda z^\tau.v$ for $c \succ_t x$ (due to Clause 2 of Definition 4.29). As no free variables of s are bound in t and c is chosen to be $\trianglelefteq$-minimal, $\lambda z^\tau.v \trianglelefteq s \lhd c$ is the case and therefore $c = c_t(\lambda z^\tau.v) = c_t(s)$, because the minimal passive term strictly above s would be the context of $\lambda z^\tau.v$ as well. Hence, this proves the claim $c = c_t(s) \succ_t x$.

If c is not minimal from the beginning, we trace back the witnesses $\lambda z.v$ of $c \succ_t x$ until $\lambda z.v \trianglelefteq s$ holds (which it will at some point because $\succ$ is inductively defined). This will give a $\trianglelefteq$-minimal $c \succ_{t'} x$ with $x \trianglelefteq s \lhd c \trianglelefteq t' \trianglelefteq t$. If no variables of s are bound in t, then neither in t'. Hence, we can apply the $\trianglelefteq$-minimal case above.

Claim 3 follows from the fact that no free variables of s are bound in $t[u := s]$ by easy induction over the inductive predicate $\succ_\star$.

Claim 4 is a special case of Claim 2: By the latter we have $c_t(s) \succ_t x$. If it was $c \rhd c_t(s)$, then only due to Clause 2 of Definition 4.25 which would imply a λ-abstraction above c and hence above $c_t(s)$. Hence, $c \trianglelefteq c_t(s)$ and with minimality of $c_t(s)$ above s we have $c = c_t(s)$. □

Claim 1 of this lemma is clear.

Claim 2 tells us that a variable x can only be in the context of a c above s with no bound variables in s if it is already in the context of this very s. In other words, formulated in the negative way: a term s above x in the syntax tree with a context of x even further above *can only not interact* with x (i.e. $c_t(x) \not\succ_t x$) if this s has free variables bound by λ-abstractions in t above. Only those bound variables would make it possible to "jump over" $c_t(s)$ with the $\succ_\star$ -relation.

Claim 3 finally tells us that the $\succ_\star$ relation is somewhat stable under substitution, but in the reverse way: if x is in context of c after substitution, it was already in the context c before.

Claim 4 is a special case of Claim 2 and says that a context of x above $c_t(s)$ is not possible without λ-abstractions above s.

Lemma 4.35 (Equivalence relation)**.** *The relation $\leadsto_t^\eta$ is an equivalence relation.*

Proof. Symmetry and reflexivity are obvious by definition. The transitivity is shown by (course-of-value) induction on t. Assume $x \leadsto_t^\eta y$, $y \leadsto_t^\eta z$ with the witnesses c_1 and c_2 (for Clause 3 of Definition 4.29). Then:

- **Base cases** with x and y not under λ-abstraction: The only possible situation is $c_1 = c_t(x) = c_t(y) = c_t(z) = c_2$ and hence the transitivity of the base case.

- **Step case:** Assume transitivity of $\leadsto_s^\eta$ for all $s \lhd t$. By the previous lemma, Claim 1 w.l.o.g. assume $c_1 \lhd c_2$. Let c be minimal with $c_1 \lhd c$ and $c \succ_t y$. Because c_1 is passive, there must be a $\lambda u^\tau.s$ with $u \leadsto_s^\eta y$ and $c = c_{t'}(\lambda u^\tau.s)$ as witness for $c \succ_{t'} y$ for some $t' \trianglelefteq t$. Because c is the smallest passive term above $\lambda u^\tau.s$ and $c \neq c_1$, it follows $c_1 \lhd \lambda u^\tau.s$. Hence, by Lemma 4.32 part 1 and the symmetry of $\leadsto_s^\eta$, one gets $x \leadsto_s^\eta y$, $y \leadsto_s^\eta u$ and by induction hypothesis $x \leadsto_s^\eta u$. Therefore, $c \succ_{t'} x$ holds. By repeating this method, c_2 will be reached after a finite numbers of steps, such that $c_2 \succ_t x$ holds, which implies $x \leadsto_t^\eta z$. □

What one basically does here in order to show $x \leadsto_t^\eta z$ is to take the two witnesses c_1 and c_2 which are both contexts of y, and then – step for step – it is shown that the contexts in between up to c_2 are also contexts of x. Hence, finally c_2 is the witness as well for $x \leadsto_t^\eta z$.

The equivalence relation $\leadsto_t^\eta$ induces partitions on the free variables of t that are effective according to η. These partitions will play an important role as an upper bound of the length of a list that is recursed over. More precisely, the size of the equivalence class of the list term is watched during reduction for any forecast η. That this size does not increase and therefore is bounded by the number of free variables in the whole term t at the beginning, will be proved in the next theorem:

Theorem 4.36 (Non-size-increasing equivalence classes)**.** *For a typed term t and $t \longrightarrow t'$, a forecast η' for t' and an equivalence class C' of $\leadsto_{t'}^{\eta'}$, there is a forecast η for t and an equivalence class C of $\leadsto_t^\eta$ with $|C'| \leq |C|$.*

Proof. Assume $t \longrightarrow t'$ due to the conversion $s \longmapsto s'$ with $s \trianglelefteq t$ and $s' \trianglelefteq t'$. Let η' be a forecast for t'.

For variables x, y in t' as located subterms, for readability, we will use the same names x, y for the "corresponding" located subterms in t before normalisation. The conversions of δLFPL allow us to find such a "corresponding" original subterm in t in any case. In the case of the $(\delta f\, s)$-conversion all free variables in the two copies of s in the reduct of course share the "corresponding" subterms in t.

We show that there is a forecast η for t, such that

$$\forall x, y \trianglelefteq t'.x \mathrel{\prec\!\!\!\succ}^{\eta'}_{t'} y \to x \mathrel{\prec\!\!\!\succ}^{\eta}_{t} y$$

and that $x_1 \not\mathrel{\prec\!\!\!\succ}^{\eta'}_{t'} x_2$ if $x \in FV(t)$ is the "corresponding" subterm in t for the copies x_1 and x_2 in t', i.e. different occurrences in t' of the same free variable of t fall in different equivalence classes of $\mathrel{\prec\!\!\!\succ}^{\eta'}_{t'}$. This is done for all the conversions by case distinction, which finally implies the claim of the theorem.

So let η' be a forecast for t', distinct $x, y \in FV_{\eta'}(t')$ with $x \mathrel{\prec\!\!\!\succ}^{\eta'}_{t'} y$ with witness $c' \trianglelefteq t'$. Case distinction for $s \longmapsto s'$:

1. **Case** $(\lambda u^{\tau}.v\, w) \longmapsto v[u := w]$:
   According to Lemma 4.24 u does occur t most once effectively a in v according to η'. Hence, no $x \in FV_{\eta}(w)$ will be duplicated in $v[u := w]$, effectively according to η'.

   The forecast η for t is selected to be the canonical counterpart to η', i.e. η operates on the $\langle p, q\rangle$-subterms of t in same way η' does on t'.

2. **Subcase** $x, y \trianglelefteq w$ in t':

   a) **Subsubcase** $c' \triangleleft w$: Apply Lemma 4.32 part 1 to $c' \trianglelefteq w \triangleleft t'$ and then with part 2 to $c' \triangleleft w \triangleleft t$, giving the witness c' for $x \mathrel{\prec\!\!\!\succ}^{\eta}_{t} y$.

      i. **Subsubcase** $c' = w$: Then w must be passive. Apply Lemma 4.32 part 3, once from right to left with $c' = w \trianglelefteq t'$ and then from left to right for $c' = w \trianglelefteq t$.

      ii. **Subsubcase** $w \triangleleft c' \trianglelefteq t'$: The redex $s = (\lambda u.v\, w)$ is not under λ-abstraction (the reduction would not be allowed then), and hence, with Claim 4 of Lemma 4.34 we have $c' = c_{t'}(w)$ and by Claim 1 of Lemma 4.32 we get $w \succ_w x$ and $w \succ_w y$. By the uniqueness of a minimal passive term

above w, there is an $c \rhd w$ in t with $c \succ_t x$ and $c \succ_t y$ acting as a witness for $x \prec\!\!\prec^{\eta}_t y$.

b) **Subcase** $x, y \trianglelefteq v[u := w]$, $x, y \not\trianglelefteq w$ in t':

   i. **Subsubcase** $c' \trianglelefteq v[u := w]$: No free variable of w is bound in t', but outside w itself (again the reduction would not be allowed then). Therefore, no free variable subterm of w can act as a witness for some $c'' \succ_{v[u:=w]} x$, i.e. the substitution can be ignored, such that also $c'' \succ_v x$ holds in t. Therefore, $x \prec\!\!\prec^{\eta}_v y$.

   ii. **Subsubcase** $v[u := w] \triangleleft c'$: Then $v[u := w] \succ_{v[u:=w]} x$, $v[u := w] \succ_{v[u:=w]} y$ hold by Claim 1 of Lemma 4.32, and with the same argument about bound variables in w as in the previous subsubcase, we have $v \succ_v x$, $v \succ_v y$. Again by uniqueness of the minimal passive term c above v, this c acts as a witness of $x \prec\!\!\prec^{\eta}_t y$.

c) **Subcase** $x, y \trianglelefteq v[u := w]$, $x \not\trianglelefteq w$, $y \trianglelefteq w$ in t':
By $y \trianglelefteq w$ we can assume that $u \in \mathrm{FV}(v)$.

   i. **Subsubcase** $c' \trianglelefteq v[u := w]$: By Lemma 4.34 part 3 $c' \succ_v u$. Therefore, c' is a witness for $x \prec\!\!\prec^{\eta}_v u$. Moreover, w is not passive, because then the biggest c with $c \succ_{t'} y$ could not be above w as w has no bound variables in t' and t outside itself. Therefore, by Lemma 4.34 part 2 the smallest passive term c above w in t gives $c \succ_t y$, but $c_t(\lambda u^\tau.v) = c$, such that c is a witness for $x \prec\!\!\prec^{\eta}_t y$.

   ii. **Subsubcase** $v[u := w] \trianglelefteq c'$: With a similar argumentation as in the previous subsubcase, w is not passive and no $d \trianglelefteq w$ acts as a witness for $c' \succ_{t'} x$. Therefore, $c_t(v) = c_t(\lambda u^\tau.v) = c_t(w)$ holds and $x \prec\!\!\prec^{\eta}_t y$ with $c_t(w) = c_t(\lambda u^\tau.v)$ as the witness.

d) **Subcase** with $x, y \not\trianglelefteq v[u := w]$ in t' is easy because again $v[u := w]$ has no variables bound outside.

e) **Subcase** with $x \not\trianglelefteq v[u := w]$,$y \trianglelefteq v[u := w]$,$y \not\trianglelefteq w$ in t' (and vice-versa):
$v[u := w]$ has no free variables bound above, but c' must be above $v[u := w]$. Hence, by Lemma 4.34 Claim 4 $c' = c_{t'}(v[u :=$

$w]) \succ_{t'} x, y$. But, as there is no variable in w bound above that is below $v[u := w]$ either, we can ignore the substitution again and get $c_t(v) \succ_t y$. By $x \not\trianglelefteq v[u := w]$ and therefore also $x \not\trianglelefteq v$ in t, we have that v is not passive, such that $c_t(v) \neq v$. Hence, $c_t(v)$ is the corresponding subterm of $c_{t'}(v)$ and therefore the witness for $x \mathrel{-\!\!\diamond\!\!\!\succ}_t^{\eta} y$ we aimed at.

3. **Case** $(\delta f\, v) \longmapsto (f\, v) \otimes_{\sigma,\tau} v$:
   By Assumption 4.22 we have that $(f\, v)$ is of passive type and not under λ-abstraction. Hence, for every $x \trianglelefteq (f\, v)$ and $c \succ_{t'} x$ the term c is also subterm of $(f\, v)$. But for every $d \succ_{t'} y$, $y \not\trianglelefteq (f\, v)$ with $x \trianglelefteq d$ the term d is above or equal to $(f\, v) \otimes v$, such that no witness for $x \mathrel{-\!\!\diamond\!\!\!\succ}_{t'}^{\eta} y$ can be found for those cases. Hence, two instances of $x \in FV(v)$, once as subterm of $(f\, v)$ and once as subterm of the outer v, fall in distinct equivalence classes.

   Interaction outside $(f\, v) \otimes_{\sigma,\tau} v$**:** As $(f\, v)$ is not under λ-abstraction in t', no free variable of $(f\, v)$ acts as a witness for any $e \succ_{t'} x$ with $x \trianglelefteq t'$ and $x \not\trianglelefteq (f\, v)$, such that from $x \mathrel{-\!\!\diamond\!\!\!\succ}_{t'}^{\eta'} y$ it easily follows $x \mathrel{-\!\!\diamond\!\!\!\succ}_{t}^{\eta} y$.

4. **Case** $(\langle s, t\rangle\, \mathbf{tt}) \longmapsto s$:
   The forecast η is equal to η' on the common domain, but extended by $\langle s, t\rangle \mapsto \mathbf{tt}$ and arbitrary values for any $\langle v, w\rangle \trianglelefteq t$. Otherwise, this case is easy.

5. **Case** $(\langle s, t\rangle\, \mathbf{ff}) \longmapsto t$: Analogous.

6. **Case** $(\mathbf{tt}\, \langle s, t\rangle) \longmapsto s$: Analogous.

7. **Case** $(\mathbf{ff}\, \langle s, t\rangle) \longmapsto t$: Analogous.

8. **Case** $(s \otimes_{\sigma,\tau} t\, r) \longmapsto ((r\, s)\ t)$: clear.

9. **Case** $(\mathbf{nil}_{\tau}\, h\, g) \longmapsto g$: clear.

10. **Case** $((\mathbf{cons}_{\tau}\, d\, v\, x)\ h\, g) \longmapsto (h\, d\, v\ (x\, h\, g))$: Clear, as h is closed. □

Corollary 4.37. *The size of every equivalence class of* $\mathrel{-\!\!\diamond\!\!\!\succ}_{t'}^{\eta'}$ *is bounded by* $|FV(t)|$ *for* $t \longrightarrow^n t'$ *and every* $n \in \mathbb{N}$.

Proof. By the Theorem 4.36 every equivalence class after reduction is not bigger than one class before reduction. At the start of the reduction chain, the size of every equivalence class is obviously bounded by $|\mathrm{FV}(t)|$, i.e. by all variables, in the worst case in one equivalence class. By induction on n it follows that no equivalence class of $\multimap^{\eta'}_{t'}$ can be bigger than $|\mathrm{FV}(t)|$. □

By this corollary the equivalence classes are shown to be non-size-increasing. During normalisation the number of classes may grow, but each class stays small.

If one takes the original LFPL system and tries to identify the classes, one finds exactly the same structure. The key difference is that in LFPL there is no duplication and hence the overall number of free variables will never increase. In other words, *in LFPL the sum of the class sizes is already non-size-increasing*. Hence, there is not much sense to look at this fine grained interaction structure at all, which induces the equivalence relation $\multimap^{\eta}_{t}$ onto the term t.

In δLFPL this fine look is essential though, because the overall number of free variables increases. Every time a term is duplicated, one might duplicate the free variables and possibly also increase the number of equivalence classes. The Assumption 4.22 makes sure that the equivalence classes under the passive term cannot interact with the outside.

4.4.2. Lengths of Lists

In [AS00] the polynomial measure is (recursively) defined for each LFPL term and shown to strictly decrease in each normalisation step. For the iteration construct in the LFPL language this measure uses the number of free variables as an upper bound for the length of lists which can occur in the term.

If we transfer this idea to the δLFPL-system, it is natural to take the size of the $\multimap^{\eta}_{t}$-equivalence class of the list as this upper bound. With the same argument as in [AS00] and the fine analysis of the equivalence classes during normalisation in Theorem 4.36, it is clear that this makes sense, because a list can never cross different equivalence classes. In order to see this, we take a list term:

$$l := \left(\mathbf{cons}_\tau\, d_0^\Diamond\, v_0\, \left(\mathbf{cons}_\tau\, d_1^\Diamond\, v_1\, \left(\mathbf{cons}_\tau\, d_2^\Diamond\, x_2\, \ldots\right)\right)\right).$$

The list type $L(\tau)$ is never passive (compare Example 4.18), and the $\diamondsuit$ is not passive either. Hence, we get

$$c_t(d_0) = c_t(d_1) = \cdots = c_t(l).$$

Moreover, it is clear that a variable $x_i^{\diamondsuit}$ (or at least with $\diamondsuit$ positively in the type of x_i) is needed in every term of type $\diamondsuit$, and this x_i must be in the context of $c_t(d_i)$. Therefore, the size of the equivalence class of $c_t(l)$ bounds the length of l.

With this intuition what the bound of the list lengths could be, we need to define the length of a (non-normal) term of list type $L(\tau)$. In order to consider a length measure $l(s^{L(\tau)})$ to be reasonable, it has to satisfy certain conditions:

1. $l(s^{L(\tau)}) < l((\mathbf{cons}_\tau\, d\, x\, s))$
2. $l(s^{L(\tau)}[x^{\diamondsuit} := v]) = l(s)$
3. $l(s^{L(\tau)}) \leq M(t)$ for $t \longrightarrow^* t'$, $s \trianglelefteq t'$ and $M(t)$ is constant.

The first condition describes that applying $\mathbf{cons}_\tau$ creates a longer list.

The second property makes sure that substituting a variable of type $\diamondsuit$ with a term of the same type does not alter the list length. Note here that because of this property it is not enough to count the variables of e.g. type $\diamondsuit$ (or at least with $\diamondsuit$ positively in the type). If we did that, we could substitute a term like $v := \left(y^{\diamondsuit} \otimes_{\diamondsuit,\diamondsuit} z^{\diamondsuit}\, \lambda y, z.y\right)$ for x. This would increase the length of the list $s[x := v]$.

The last condition makes sure that every length of a list which occurs during normalisation can be bound by some constant *which only depends on the starting term* t.

We will now fix a specific length measure. We then show that it is reasonable in the sense of the upper conditions. In the next section, about the complexity of normalisation, we will exploit those properties in order to show the complexity theorem.

Definition 4.38 (List length measure)**.** For a term $s \trianglelefteq t$ of list type the length of s is defined as

$$l(s) \quad = \quad \max_{\eta} \left| \left\{ v^{\diamondsuit} \trianglelefteq s \mid v \text{ effective for } \eta \wedge c_s(v) = s \right.\right.$$

$$\left. \wedge\ v \text{ is } \trianglelefteq\text{-maximal with these properties} \right\} \Big|$$

where η ranges over the forecasts for s.

Condition 1 for a reasonable length measure is easy:

Lemma 4.39. *$l(s) < l((\boldsymbol{cons}_\tau\, d\, v\, s))$.*

Proof. The subterm d of type $\diamondsuit$ contributes to $l((\mathbf{cons}_\tau\, d\, v\, s))$ because it is maximal of $\diamondsuit$-type and $c_{(\mathbf{cons}\, d\, v\, s)}(d) = (\mathbf{cons}_\tau\, d\, v\, s)$. Conversely, all $w \lhd s$ which contribute to $l(s)$ are also maximal of $\diamondsuit$-type in $(\mathbf{cons}_\tau\, d\, v\, s)$. Moreover, s is not passive, such that by Lemma 4.32 part 3 $c_{(\mathbf{cons}\, d\, v\, s)}(w) = (\mathbf{cons}_\tau\, d\, v\, s)$ holds. Hence, w also contributes to $l((\mathbf{cons}_\tau\, d\, v\, s))$. □

Corollary 4.40. *The recursion $t = ((\boldsymbol{cons}_\tau\, d\, v\, s)\ h\, g) \trianglelefteq t$ can only be unfolded $l((\boldsymbol{cons}_\tau\, d\, v\, s))$ times.*

Proof. The length of the remaining list s is strictly decreasing when unfolding once. Because the length is non-negative by definition, the unfolding must stop after less than $l((\mathbf{cons}_\tau\, d\, v\, s))$ times. □

Condition 2 for a reasonable length measure:

Lemma 4.41. *$l(s^{L(\tau)}[x^\diamondsuit := v]) = l(s)$ holds.*

Proof. If x does not count for $l(s)$, the claim is trivial. If x itself counts for $l(s)$, then v will count for $l(s^{L(\tau)}[x^\diamondsuit := v])$ and vice-versa. □

Lemma 4.42. *For every term $v^\diamondsuit \trianglelefteq s$ there is an $x^\sigma \in FV_\eta(v)$ with $\diamondsuit$ positively in σ and $v \succ_v x$.*

Proof. We show that for every term $v^\tau \trianglelefteq s$ with $\diamondsuit$ positively in τ there is an $x^\sigma \in FV_\eta(v)$ with $\diamondsuit$ positively in σ and $v \succ_v x$ by induction on the term v. □

Corollary 4.43. *For $s^{L(\tau)}$ the following holds:*

1. *$l(s) \leq \max_\eta |\{x \mid x \in FV_\eta(s) \wedge s \succ_s x\}|$,*
2. *For a fixed η all those x from 1 are in the same equivalence class of $\multimap^\eta_s$.*

Proof. Every v contributing to $l(s)$ contains an $x^\sigma \in FV_\eta(v)$ with $\diamondsuit$ positively in σ and $v \succ_v x$ due to Lemma 4.42. Moreover, $c_s(v) = s$ holds, and as v is not passive, we get $s \succ_s x$.
The second claim is true because all the x from 1 share the same common context s. □

With Corollary 4.37, which said that the equivalence classes do not grow, we have just proved condition 3 of a reasonable length measure, but not in full generality, but only if s is not under λ-abstraction. Because in that case, from $x \multimap^\eta_s y$ we can conclude $x \multimap^\eta_t y$ with Lemma 4.32 part 3 because s is not passive. Hence, we get the constant $M(t) := \mathrm{FV}_\eta(t)$ of condition 3 for a reasonable length measure:

Lemma 4.44. *If $s' \trianglelefteq t'$ is not bound in t' and $t \longrightarrow^* t'$, then $l(s') \leq |FV(t)|$ holds.*

Proof. If s' is not bound in t', every free variable of s' is also free in t'. By Corollary 4.43 those free $x \in FV_\eta(s')$ which contribute to $l(s)$ are in the same equivalence class of $\multimap^\eta_{t'}$. The size of this class is bounded by $|\mathrm{FV}(t)|$ by Corollary 4.37. □

A special case of this lemma is the one where s' is *not* under λ-abstraction.

4.5. Complexity

With the data size analysis, i.e. the definition of the length of lists, we can proceed in applying essentially the complexity proof by Aehlig and Schwichtenberg [AS00] that the normalisation is bound by a polynomial. The basic idea is that we first define the polynomial $\vartheta_q(t) \in \mathbb{N}_0[X]$, where t is a δLFPL-term. Then, we show that there is a constant c (which is linear in $|t|$), such that $\vartheta_q(t)(c) > \vartheta_q(t')(c)$ for $t \longrightarrow t'$. This means that any reduction sequence, starting from t, is at most $\vartheta_q(t)(c)$ steps long.

And, because $c \in \mathcal{O}(|t|)$ we get the polynomial time complexity of the normalisation.

Definition 4.45 (Complexity measure)**.** For each subterm $t \trianglelefteq q$ the complexity measure $\vartheta_v(t) \in \mathbb{N}_0[X]$ is defined by recursion on t:

$$
\begin{aligned}
\vartheta_q(x) &:= 0 \\
\vartheta_q(c) &:= 0 \\
\vartheta_q((s\,t)) &:= \begin{cases}
\vartheta_q(s)+(l(s)+1)\cdot\vartheta_q(t)+l(s)+1 & \text{if } s \text{ is not bound in } q \\
& \text{and } s \text{ is of list type} \\
\vartheta_q(s)+(X+1)\cdot\vartheta_q(t)+X+1 & \text{if } s \text{ is bound in } q \\
& \text{and } s \text{ is of list type} \\
\vartheta_q(s)+\vartheta_q(t)+1 & \text{if } s \text{ is of tensor type } \sigma\otimes\tau \\
\vartheta_q(s)+\vartheta_q(t) & \text{otherwise}
\end{cases} \\
\vartheta_q(\lambda x^{\tau}.t) &:= \vartheta_q(t)+1 \\
\vartheta_q(\langle s,t\rangle) &:= \sup_{\preceq}\{\vartheta_q(s),\vartheta_q(t)\}+1 \\
\vartheta_q(\delta f) &:= \vartheta_q(f)+1
\end{aligned}
$$

with $\sup_{\preceq}$ being the supremum of two polynomials, and $\preceq$ as the lexicographic order on the coefficients.

Note that the index q in $\vartheta_q(t)$ is needed to express whether s is bound in q in the iteration cases.

Remark 4.46. There are two cases for the iteration, i.e. for a list type term s applied to a step term t in $(s\,t)$:

- If s is *not bound*, the list will not change anymore. This property, therefore, makes sure that the length measure $l(s)$ is precise and we can use it instead of X in the polynomial $\vartheta_q(((\mathbf{cons}_B\, x\, \mathbf{tt}\, \mathbf{nil}_B)\ h))$.
- If s is *bound*, there is still a variable in s bound above, such that we do not know much about $l(s)$. Hence, we use the placeholder X for the length of s. X will be set later to a value bigger than the number of free variables in the whole term $|\mathrm{FV}(q)|$. In other

words, X will later be an upper bound for $l(s)$ as long as we do not know better. In the moment the last bound variable is substituted, the polynomial $\vartheta_q((s\,t))$ switches from the second to the first case in the definition.

- If s is *almost-closed*, by Lemma 4.41 we know that the length of s will not change anymore during beta reduction. But this is not enough in order to use $l(s)$ already, i.e. if s is still bound. We further have to require that s is not bound in q. Take the term

$$q := \langle \lambda x^{\diamond}.\,((\mathbf{cons}_B\, x\, \mathbf{tt}\, \mathbf{nil}_B)\ h)\,, \mathbf{tt}\rangle.$$

  Is the list length $l((\mathbf{cons}_B\, x\, \mathbf{tt}\, \mathbf{nil}_B))$ bounded by $|\mathrm{FV}(q)|$?
  Obviously, this is not the case because the term q is closed, and therefore $|\mathrm{FV}(q)| = 0$. But clearly, $(\mathbf{cons}_B\, x\, \mathbf{tt}\, \mathbf{nil}_B)$ is bound in q (because x is a bound variable). Hence, the size measure of this list in the complexity measure $\vartheta_q(q)$ must be X, and not $l(l((\mathbf{cons}_B\, x\, \mathbf{tt}\, \mathbf{nil}_B))) = 1$.

- Would it be enough to require "s is under λ-abstraction" instead of "s is bound" in the definition of $\vartheta_q(t)$?
  The answer can be seen in the following example:

$$q := \left(\lambda f^{\sigma\multimap\sigma}, p^{\mathbf{tt}}.f\ \left(\left(\mathbf{cons}_B\, x^{\diamond}\, \mathbf{tt}\, \mathbf{nil}_B\right)\ h\right)\right).$$

  Assume that we use "s is under λ-abstraction" instead of "s is bound". The polynomial $\vartheta_q(\left(\left(\mathbf{cons}_B\, x^{\diamond}\, \mathbf{tt}\, \mathbf{nil}_B\right)\ h\right))$ will use the precise length measure 1 for the list involved because there is no λ above. After beta reduction $q \longrightarrow q'$, the list though will suddenly be under the λp and hence the polynomial $\vartheta_{q'}(\left(\left(\mathbf{cons}_B\, x^{\diamond}\, \mathbf{tt}\, \mathbf{nil}_B\right)\ h\right))$ would switch to the X-case. This would increase the value $\vartheta_{q'}(q')(|\mathrm{FV}(q)|)$ if $|\mathrm{FV}(q)| > 1$ holds. In the definition with "s is bound" this cannot happen.

Lemma 4.47. *If x is linear and free in $(t\,s)$, then x is free in either t or s, but not in both.*

Proof. By Definition 4.6 of *linear* and Lemma 4.42. □

The most complicated case in the analysis of the complexity is the behaviour of the polynomial $\vartheta_q(t)$ during beta-reduction, i.e. when substituting a variable with another term. This is non-trivial, as usual, because substitution is a non-local operation on a term. Hence, we study this very case first in its own lemma:

Lemma 4.48 (Substitution Complexity Lemma)**.** *Let be* $\Gamma;\emptyset \vdash (\lambda x^\sigma.u\, s^\sigma)$ *with* $q := (\lambda x^\sigma.u\, s^\sigma)$ *and* $q' := u[x := s]$ *and* η *a forecast for* q *and* η' *the corresponding forecast for* q'*. Then*

$$\vartheta_q(u[x := s])(X) < \vartheta_q((\lambda x^\sigma.u\, s^\sigma))(X)$$

holds for all $X \geq |FV((\lambda x^\sigma.u\, s^\sigma))|$.

Proof. We show $\vartheta_{q'}(t[x := s])(X) \leq \vartheta_q(t)(X) + \vartheta_q(s)(X)$ for all $t \trianglelefteq u$ by induction on t. Then the claim follows from the case $t = u$ with

$$\vartheta_q(u)(X) + \vartheta_q(s)(X) < \vartheta_q(u)(X) + \vartheta_q(s)(X) + 1 = \vartheta_q((\lambda x^\sigma.u\, s^\sigma))(X).$$

We write $f \preceq g$ for $\forall x.f(x) \leq g(x)$.

The only non-trivial cases are those where t is an application, i.e. t is of the shape $(r\, r')$. For all other cases the induction step is easy.
Case $r = \delta f$: x only occurs free at most once in t by Lemma 4.47. Assume x is free in f. Then:

$$\begin{aligned}
\vartheta_{q'}((\delta f\, r')\,[x := s]) &= \vartheta_{q'}((\delta f[x := s]\, r')) \\
&= \vartheta_{q'}(f[x := s]) + \vartheta_q(r') + 1 \\
&\preceq \vartheta_q(f) + \vartheta_q(s) + \vartheta_q(r') + 1 \quad \text{by IH} \\
&= \vartheta_q((\delta f\, r')) + \vartheta_q(s).
\end{aligned}$$

The case where x is free in r' is similar.
Case r is of list type and not bound in q: Then $r' = h$, the step term. h is closed and r has no variable bound above. Hence, the x cannot be free in $(r\, r')$.
Case r is of list type and bound in q: The case with x not-free in r is trivial again. If x is free in r we get two subcases:
Subcase $r[x := s]$ is bound in q':

$$\vartheta_{q'}((r\, h)\,[x := s]) = \vartheta_{q'}((r[x := s]\, h))$$

$$
\begin{aligned}
&= \vartheta_{q'}(r[x := s]) + (X+1) \cdot \vartheta_{q'}(h) + X + 1 \\
&\preceq \vartheta_q(r) + \vartheta_q(s) + (X+1) \cdot \vartheta_q(h) + X + 1 \quad \text{by IH} \\
&= \vartheta_q((r\, h)) + \vartheta_q(s).
\end{aligned}
$$

Subcase $r[x := s]$ is not bound in q': The substitution replaced the last bound variable of r, such that we switched from the iteration case of the definition of $\vartheta_q((r\, r'))$ with length measure X to the case with length measure $l(r[x := s])$. By assumption of this lemma, we know

$$
X \quad \geq \quad |\mathrm{FV}(q)| \geq |\mathrm{FV}_\eta(q)| \geq \left|\mathrm{FV}_{\eta'}(q')\right| \geq l(r[x := s]), \tag{4.1}
$$

such that

$$
\begin{aligned}
&\vartheta_{q'}((r\, h)\,[x := s])(X) \\
&= \vartheta_{q'}((r[x := s]\, h))(X) \\
&= \vartheta_{q'}(r[x := s])(X) + (l(r[x := s]) + 1) \cdot \vartheta_{q'}(h)(X) + l(r[x := s]) + 1 \\
&\leq \vartheta_{q'}(r[x := s])(X) + (X+1) \cdot \vartheta_{q'}(h)(X) + X + 1 \quad \text{by Eq. 4.1} \\
&\leq \vartheta_q(r)(X) + \vartheta_q(s)(X) + (X+1) \cdot \vartheta_q(h)(X) + X + 1 \quad \text{by IH} \\
&= \vartheta_q((r\, h))(X) + \vartheta_q(s)(X).
\end{aligned}
$$

Case r is not of list type: Then, by Lemma 4.47 we know that x can only occur at most once in t because x is linear in t. Therefore, either

$$
\vartheta_{q'}(\left(r\, r'\right)\,[x := s]) = \vartheta_{q'}(\left(r[x := s]\, r'\right))
$$

or

$$
\vartheta_{q'}(\left(r\, r'\right)\,[x := s]) = \vartheta_{q'}(\left(r\, r'[x := s]\right)).
$$

In either case, the result follows from the definition of $\vartheta_q((r\, r'))$ and the induction hypothesis. □

Note that the second last case of this proof makes use of the special distinction of the two list cases which was already pointed out in Remark 4.46

4.5.1. Complexity Theorem

The main goal of this chapter is the proof that δLFPL-terms normalise in polynomially many steps in their size (measured as the size of the syntax

tree). Before giving a proof for that, we have to restrict the system a bit further:

Assumption 4.49. *Only allow those types σ for $(\delta f\, s^{\sigma}) \trianglelefteq q$ s.t. every normal and almost-closed s^{σ} has the property $\vartheta_q(s) = 0$.*

This restriction mainly makes sure that a normal term which is closed up to free variables of type $\diamondsuit$ does not contain blocked redexes anymore. When looking at the definition of $\vartheta_q(\cdot)$ the most common case of those almost-closed, normal terms are that of so-called numerals, i.e. applications of constructors. Constructors do not count, i.e. $\vartheta_q(c) = 0$ and the $\vartheta_q((s\,t)) = \vartheta_q(s) + \vartheta_q(t)$. Hence, we get the following:

$$\begin{aligned}
&\vartheta_q(\Big(\mathbf{cons}_B\, y^{\diamondsuit}\, \mathbf{tt}\ \Big(\mathbf{cons}_B\, y^{\diamondsuit}\, \mathbf{tt}\, \mathbf{nil}_B\Big)\Big)) \\
&:= \vartheta_q(\mathbf{cons}_B) + \vartheta_q(y) + \vartheta_q(w) + \vartheta_q(\mathbf{cons}_B) + \vartheta_q(y) + \vartheta_q(\mathbf{tt}) + \vartheta_q(\mathbf{nil}_B) \\
&= 0 + 0 + 0 + 0 + 0 + 0 + 0 = 0.
\end{aligned}$$

The restriction in Assumption 4.49 will be crucial in the last case of the following main theorem:

Theorem 4.50 (Complexity Theorem)**.** *Assume $\Gamma; \emptyset \vdash q^{\delta}$ with almost-closed q and $|FV(q)| \leq N \in \mathbb{N}$. If $q \longrightarrow q'$, then*

$$\vartheta_q(q)(N) > \vartheta_{q'}(q')(N).$$

In particular, any reduction sequence starting from q has length at most

$$\vartheta_q(q)(|FV(q)|).$$

Proof. We proof the claim by induction on the (inductive) relation $q \longrightarrow q'$:
Case $(r\,s) \longrightarrow (r'\,s)$ via $r \longrightarrow r'$: We distinguish whether r is a list, a tensor or another type.
Subcase r is a list: Because normalisation under λ-abstraction is not possible, r cannot be bound in q. Moreover, we know the length of lists cannot increase, i.e. $l(r) \geq l(r')$. Hence:

$$\begin{aligned}
&\vartheta_q((r\,s))(N) \\
&= \vartheta_q(r)(N) + (l(r) + 1) \cdot \vartheta_q(s)(N) + l(r) + 1 \\
&> \vartheta_{q'}(r')(N) + (l(r') + 1) \cdot \vartheta_{q'}(s)(N) + l(r') + 1 \quad \text{by IH}
\end{aligned}$$

$$= \vartheta_{q'}((r' \, s))(N).$$

Subcase r is a tensor product:

$$\begin{aligned} & \vartheta_q((r \, s))(N) \\ &= \vartheta_q(r)(N) + \vartheta_q(s) + 1 \\ &> \vartheta_{q'}(r')(N) + \vartheta_{q'}(s)(N) + 1 \quad \text{by IH} \\ &= \vartheta_{q'}((r' \, s))(N). \end{aligned}$$

Otherwise:

$$\begin{aligned} & \vartheta_q((r \, s))(N) \\ &= \vartheta_q(r)(N) + \vartheta_q(s)(N) \\ &> \vartheta_{q'}(r')(N) + \vartheta_{q'}(s)(N) \quad \text{by IH} \\ &= \vartheta_{q'}((r' \, s))(N). \end{aligned}$$

Case $(r \, s) \longrightarrow (r \, s')$ via $s \longrightarrow s'$:
Subcase r is a list: Because normalisation under λ-abstraction is not possible, r cannot be bound in q. Hence:

$$\begin{aligned} & \vartheta_q((r \, s))(N) \\ &= \vartheta_q(r)(N) + (l(r) + 1) \cdot \vartheta_q(s)(N) + l(r) + 1 \\ &> \vartheta_{q'}(r)(N) + (l(r) + 1) \cdot \vartheta_{q'}(s')(N) + l(r) + 1 \text{ by IH and } l(r) + 1 \geq 1 \\ &= \vartheta_{q'}((r \, s'))(N). \end{aligned}$$

Otherwise: Like for the previous case.
Case $r \longmapsto r'$:
Subcase $(\lambda x^\tau.t \, s) \longmapsto t[x := s]$: Because the right context of q^δ is empty and r is not under λ-abstraction, also r has an empty right context, and every subterm $u \trianglelefteq r$ is bound in r if and only if it is bound in q. With $N \geq |\mathrm{FV}(q)| \geq |\mathrm{FV}(r)|$ and Lemma 4.48 we get:

$$\vartheta_q((\lambda x^\tau.t \, s))(N) = \vartheta_r((\lambda x^\tau.t \, s))(N) > \vartheta_{r'}(t[x := s])(N) = \vartheta_{q'}(t[x := s])(N).$$

Subcase $(\langle s, t\rangle \, \mathbf{tt}) \longmapsto s$: easy
Subcase $(\langle s, t\rangle \, \mathbf{ff}) \longmapsto t$: easy
Subcase $(\mathbf{tt} \, \langle s, t\rangle) \longmapsto s$: easy
Subcase $(\mathbf{ff} \, \langle s, t\rangle) \longmapsto t$: easy

Subcase $((\otimes_{\sigma,\tau}\, s\, t)\ v) \longmapsto ((v\, s)\ t)$:

$$\begin{aligned}
&\vartheta_q(((\otimes_{\sigma,\tau}\, s\, t)\ v))(N)\\
&= \vartheta_q((\otimes_{\sigma,\tau}\, s\, t))(N) + \vartheta_q(v)(N) + 1\\
&= \vartheta_q(s)(N) + \vartheta_q(t)(N) + \vartheta_q(v)(N) + 1\\
&> \vartheta_{q'}(s)(N) + \vartheta_{q'}(t)(N) + \vartheta_{q'}(v)(N)\\
&= \vartheta_{q'}(((v\, s)\ t))(N).
\end{aligned}$$

Subcase $(\mathbf{nil}_\tau\, h\, g) \longmapsto g$: Because normalisation under λ-abstraction is not possible, $\mathbf{nil}_\tau$ cannot be bound in q. Hence:

$$\begin{aligned}
&\vartheta_q((\mathbf{nil}_\tau\, h\, g))(N)\\
&= \vartheta_q((\mathbf{nil}_\tau\, h))(N) + \vartheta_q(g)(N)\\
&= \vartheta_q(\mathbf{nil}_\tau)(N) + (l(\mathbf{nil}_\tau) + 1)\cdot\vartheta_q(h)(N) + l(\mathbf{nil}_\tau) + 1 + \vartheta_q(g)(N)\\
&= 1 + \vartheta_q(h)(N) + \vartheta_q(g)(N)\\
&> \vartheta_{q'}(g)(N).
\end{aligned}$$

Subcase $((\mathbf{cons}_\tau\, d\, v\, x)\ h\, g) \longmapsto (h\, d\, v\ (x\, h\, g))$: Because normalisation under λ-abstraction is not possible, $(\mathbf{cons}_\tau\, d\, v\, x)$ cannot be bound in q. Hence:

$$\begin{aligned}
&\vartheta_q(((\mathbf{cons}_\tau\, d\, v\, x)\ h\, g))(N)\\
&= \vartheta_q(((\mathbf{cons}_\tau\, d\, v\, x)\ h))(N) + \vartheta_q(g)(N)\\
&\geq \vartheta_q((\mathbf{cons}_\tau\, d\, v\, x))(N) + (l(x)+2)\cdot\vartheta_q(h)(N) + (l(x)+1) + 1 + \vartheta_q(g)(N)\\
&\geq \vartheta_q(h)(N) + \vartheta_q(d)(N) + \vartheta_q(x)(N) + (l(x)+1)\cdot\vartheta_q(h)(N) + l(x) + 1 + \vartheta_q(g)(N) + 1\\
&> \vartheta_{q'}((h\, d\ (x\, h\, g)))(N).
\end{aligned}$$

Subcase $(\delta f^{\sigma\multimap\tau}\, s) \longmapsto (\otimes_{\tau,\sigma}\ (f\, s)\ s)$: Because normalisation under λ-abstraction is not possible and q is almost-closed, also s is almost closed. With Assumption 4.49 we have $\vartheta_q(s)(N) = 0$. Then:

$$\begin{aligned}
&\vartheta_q((\delta f\, s))(N)\\
&= \vartheta_q(f)(N) + \underbrace{\vartheta_q(s)(N)}_{0} + 1\\
&> \vartheta_{q'}(f)(N) + \vartheta_{q'}(s)(N) + \vartheta_{q'}(s)(N)\\
&= \vartheta_{q'}((\otimes_{\tau,\sigma}\ (f\, s)\ s))(N).
\end{aligned}$$

4.6. Conclusion and Outlook

This chapter extends the complexity proof of Aehlig and Schwichtenberg[AS00] to a more relaxed type system, which allows certain non-linearities. Obviously, it is considerably more complicated because the original measure of the free variables in LFPL must be replaced with the equivalence classes of $\prec\!\!\diamond\!\!\succ_t^\eta$. But the proof structure is very similar in the end.

Next to the support for the δf-construct in the language of δLFPL, the complexity result here is stronger than the one of Aehlig and Schwichtenberg [AS00]: we do not require that a list is computed in order to know its exact size before an iteration can be unfolded. The size of the $\prec\!\!\diamond\!\!\succ_t^\eta$-classes as an estimate for the lengths of lists is precise enough, even without normalisation of the list argument.

An interesting question is how essential the restriction is that normalisation below λ-abstraction is forbidden. Our proof makes use of this in a number of places, mainly in order to simplify the length measure below a λ-abstraction. If there are bound variables around, it is not easy to say which length a list will possibly have during further normalisation. This of course depends on the substituted term when the abstraction is going away during beta-reduction.

An idea to improve the length measure to those cases would be to take the whole context of a list into account. But then, it is not very clear how the length decreases when an iteration is normalised. Also the result of the iteration will stay in the context and the length measure will not obviously become smaller.

5

Embedding LFPL into Light Affine Logic with Fixed Points

In this chapter we embed LFPL-terms into Light Affine Logic with fixed points. In Section 3.3.1.3 the Insertion Sort example is discussed in the context of Light Affine Logic. Insertion Sort is the prime example of a non-size-increasing algorithm which is easily typable in LFPL. In a sense, LFPL is just a variant of Gödel's System T which captures the idea of non-size-increasing functions. Such a function can only create outputs at most as big as the input. For this purpose a special diamond type $\diamondsuit$ is used which is not inhabited by a closed term. The only way to "get" a diamond is through the iteration operator, which passes one $\diamondsuit$ to each step function. Moreover, the $\mathbf{cons}_\tau$-operation to enlarge a list "consumes" one diamond. The consequence is that the number of diamonds around during normalisation is bounded by the number of diamonds in the beginning of normalisation, hence every function definable in this system is non-size-increasing.

After LFPL, the second system of interest in this chapter is Light Affine Logic as described in Section 2.3. Light Affine Logic is based on System F and therefore no base types are available. The list type $L(\tau)$ of LFPL can be easily defined in an impredicative way using type abstractions.

The meaning of the diamond type is not obvious though when one tries to reformulate LFPL algorithms in LAL. The Insertion Sort of Section 3.3.1.3,

as shown there, can be expressed in LAL, but its type is asymmetric and cannot be iterated therefore. In LFPL the iteration scheme of the Insertion Sort algorithm can be expressed directly and iterated again without restriction. Hence, the question arises how far apart LAL and LFPL really are.

The goal of this chapter is to give an embedding of LFPL programs into Light Affine Logic or more precisely into Light Affine Logic with *fixed points*.

The latter will make it easier to represent the non-recursive parts of algorithms in a flat way, because fixed point types allow us to use inductive data types without any modality (and, at the same time, without the ability to iterate over them)[1].

The embedding should be of a compositional nature, without the long way of simulating a computational model like Turing Machines, SECD machines (compare [NM03]) or any other low-level machine.

Structure of this chapter This chapter starts with the *iterated iterator* concept in Section 5.1, that is used to do impredicative iteration in LAL. After an intuitive introduction and motivation in Subsection 5.1.1, we present the implementation of the *iterated iterators* in Light Affine Logic in Subsection 5.1.2. Section 5.2 uses the definitions of the previous section in order to implement a flat embedded of the non-recursive parts of LFPL-algorithms into Light Affine Logic with fixed points. The chapter ends in Section 5.3 with a conclusion by discussing whether the presented embedding is satisfactory.

5.1. Iterating Iterators

The main tool for this chapter will be the concept of iterated iterators. Before introducing them formally we try to get an idea first why they do in LAL what the built-in iteration scheme does in LFPL:

5.1.1. Building an Intuition

The complexity proof in [AS00] constructs a polynomial measure which strictly decreases during normalisation in every step (compare Section 4.5).

[1] This is purely technical and, probably, the traditional Light Affine Logic, as described in Section 2.3, would work as well with some other, much more involved coding for data types without iteration.

one iteration in the outer recursion of sort	$(l\,\text{insert}\,\mathbf{nil})$
one iteration in each insert	$(l\,...\leq ...\,\text{base})$
one iteration in each $\leq$ in the step of insert	$\leq$
no further iteration in the the step of $\leq$	...

Figure 5.1.: Iteration nesting of Insertion Sort (compare Section 3.3.1)

The degree of the measure is uniform in the input, i.e. there is polynomial $\vartheta_q(t)(X) = a_k X^k + a_{k-1}X^{k-1} + \ldots + a_0$ for every typable term $t^{L(\tau)\multimap\sigma}$ of LFPL, with k and the coefficients a_i independent of the input $l^{L(\tau)}$, such that $(t\,l)$ normalises in at most $\vartheta_q(t)(|l|)$ many steps.

The main insight in the complexity proof of [AS00] is the way this polynomial can be constructed: Due to the requirement of the type system that step terms must be closed (see rule (It) of LFPL in Section 2.18) it is possible to derive $\vartheta_q(t)(X)$ from t mainly by recursion on the nested iterations in the term, increasing the degree of the polynomial with each further nesting. In other words, the polynomial for $(l\,h)$ is essentially $X \cdot \vartheta_q(h)(X)$.

This construction leads to the intuition that the power of LFPL lays in the *nesting of iterations* inside the step terms of iterations. This nesting dominates the normalisation time, i.e. the degree of the normalisation time polynomial $\vartheta_t(t)$ is bounded by the nesting depth of the iterations in t. Consequently, in order to get an embedding of LFPL into another calculus, the crucial point will be to model this nesting. Compare Figure 5.1 for an non-trivial nesting of iterations.

An algorithm without any nesting, e.g. $(l\,\lambda x, d, p.\,(\mathbf{cons}_\tau\,x\,d\,p)\;\mathbf{nil}_\tau)$, basically just needs one iterator which runs $|l|$ many steps. If the nesting is one level deep, e.g. in

$$t := (l\,\lambda y, e, q.\,(\mathbf{cons}_\tau\,y\,e\;\,(q\,\lambda x, d, p.\,(\mathbf{cons}_\tau\,x\,d\,p)\;\mathbf{nil}_\tau))\;\mathbf{nil}_\tau)\,, \qquad (5.1)$$

one needs another "fresh" iterator in each step of the outer iteration. The outer step is called $|l|$ times. Due to the property of the system that only non-size-increasing algorithms can be formulated, we know that also the inner iteration will run no more than $|l|$ steps. Hence, we have to find an iterator "data structure" for the embedding which provides the iterators of length $|l|$ in each outer step, i.e. $|l|$ many of them, in order to drive the inner iteration. This process can be repeated if the nesting is even deeper.

simple iteration functional	$N \cong L(U)$	⟍	linear
iterations in iterations	$L(N)$	□	quadratic
it. in it. in it.	$L(L(N))$	⧈	cubic
it. in it. in it. in it.	$L(L(L(N)))$	⧈	hyper-cubic

Figure 5.2.: Iterated iterators illustrated

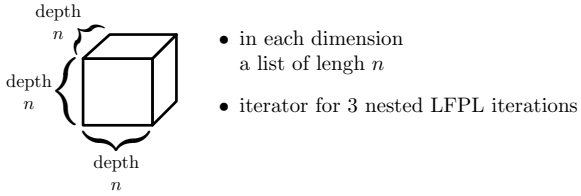


Figure 5.3.: Iterated iterator of dimension 3 and depth n

In LAL an iterator is the list itself (remember, lists are Church encodings in LAL) . Therefore, an iterator of length $|l|$ is just a list $L(\tau)$ of that size, especially $|l|$ itself of course. An iterator which provides another iterator in each step is of type $L(L(\tau))$ and so on. We will call this construction iterated iterators. Compare Figure 5.2 for a graphical intuition.

Because the outer iterations are of length at most $|l|$ and the inner iterations as well, we need an inhabitant of $L(N)$ which is of the right length, and each numeral in the list is again of length $|l|$. This idea is to be repeated for higher nestings. For nesting depth 1 we get a square shape, for nesting depth 2 we get a cube, and so on.

Before going into the technical details how these constructions are possible in LAL, we want to get a better idea how these nested iterators can be used to translate LFPL terms later on. For this purpose assume that It_n^k is such an *iterated iterator* of length n (i.e. every occurring list at every nesting level has length n) and nesting depth k (hence, geometrically $k+1$ is the dimension of its shape). In Figure 5.3 It_n^2 is shown, of dimension 3 as a cube.

Then, one can replace the term in Equation 5.1 with

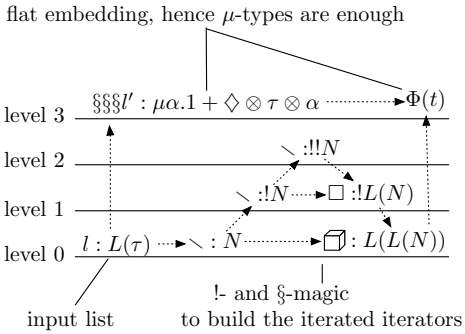


Figure 5.4.: Separation of iteration and remaining logic of an algorithm

$$t' := \left(It_n^1\, \lambda It_n^0, e, q.... \left(It_n^0\, \lambda x^\tau, d, p..... ...\right) ...\right),$$

i.e. the list It_n^1 provides the iterators It_n^0 for the inner recursion.

Of course, the question is where the "real" list l is gone. The idea is that we can implement the actual computation on l just by using the destructor $\mathbf{Case}_{\tau,\sigma}$ (or in fact the case distinction for the sum type because we will use fixed point types for the lists), without any iteration. We know that our iterators will run the step term often enough to destruct the list l (and any other occurring list) $\mathbf{cons}_\tau$ by $\mathbf{cons}_\tau$ until the empty list is reached. The whole complexity is encapsulated in the iterators. They make the iteration scheme of LFPL explicit.

So, why does this help to create an embedding into LAL? While the iterated iterators can be implemented in LAL as we will see in the next section, the main point is the following: by separating the iteration structure from the remaining logic of an algorithm, we are able to embed the latter in a completely flat way. This means that (while the iterators need a lot of modalities, i.e. magic with !- and §-boxes) the algorithm can be essentially embedded into one level of the LAL box structure. This idea is illustrated in Figure 5.4

5.1.2. Iterated Iterators in Light Affine Logic

In order to implement the idea of iterated iterators in LAL, we first fix the list type as $L(\tau) := \forall\alpha.!(\tau \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$, the numerals as $N := \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ and the unit as $U := \forall\alpha.\alpha \multimap \alpha$, with

$$\mathrm{Succ} := \lambda n.\Lambda\alpha\lambda f^{!(\alpha\multimap\alpha)}\S\boxed{\lambda g^{\alpha}\left(\rfloor\left(n\,\alpha\,!\boxed{\lambda p.\,(\rfloor f[_{!}\,p)}\right)\Big[_{\S}\,g\right)} : N \multimap N$$

$$0 := \Lambda\alpha\lambda f^{!(\alpha\multimap\alpha)}\S\boxed{\lambda a.a} : N$$

$$\mathrm{Cons}_{\tau} := \lambda l^{L(\tau)}\lambda x^{\S\tau}.\Lambda\alpha\lambda f.\S\boxed{\lambda g^{\alpha}\left(\rfloor f[_{!}\,\rfloor x[_{\S}\,\left(\rfloor\left(l\,\alpha\,!\boxed{\lambda p, x.\,(\rfloor f[_{!}\,x\,p)}\right)\Big[_{\S}\,g\right)\right)}$$

$$: L(\tau) \multimap \S\tau \multimap L(\tau)$$

$$\mathrm{Nil}_{\tau} := \Lambda\alpha\lambda f^{\tau\multimap\alpha\multimap\alpha}.\S\boxed{\lambda g^{\alpha}.g} : L(\tau)$$

$$1^{U} := \Lambda\alpha.\lambda x^{\alpha}.x.$$

We want to build an iterated iterator of arbitrary, but constant nesting depth. The size of the iterated iterators (on each nesting level) is the length of the input of type N.

Liftings As a first step, we need a number of lifting terms. Essentially, the first goal is to duplicate the input number n by lifting it to a type with the bang ! in front. Such a lifting term of type $N \multimap !N$ is too ambitious though, but for the cost of an additional paragraph § it is possible to express it in LAL, even to an arbitrary number of !-modalities:

$$\uparrow^{\S}_{N} := \lambda n.\S\boxed{\left(\rfloor\,(n\,N\,!\boxed{\mathrm{Succ}})\Big[_{\S}\,0\right)} : N \multimap \S N$$

$$\uparrow^{\S!}_{N} := \lambda n.\S\boxed{\left(\rfloor\left(n\,!N\,!\boxed{\lambda p^{!N}!\boxed{(\mathrm{Succ}\ \rfloor p[_{!})}}\right)\Big[_{\S}\,!\boxed{0}\right)} : N \multimap \S!N$$

$$\uparrow^{\S\S^{h}!^{i}\S^{j}}_{N} := \lambda n.\S\boxed{\left(\rfloor\,(n\,\S^{h}!^{i}\S^{j}N\,!\boxed{\mathrm{step}})\Big[_{\S}\,\S\boxed{\cdots!\boxed{\cdots\S\boxed{0}\cdots}\cdots}\right)} : N \multimap \S\S^{h}!^{i}\S^{j}N$$

$$\mathrm{step} := \lambda p^{!!N}\S\boxed{\cdots!\boxed{\cdots\S\boxed{(\mathrm{Succ}\ \rfloor\cdots\rfloor\cdots\rfloor p[_{\S}\cdots\Big[_{!}\cdots\Big[_{\S}\,)\cdots}\cdots}\cdots}.$$

With this we can create any number of copies of the input, with $\tau^n := \underbrace{\tau \otimes \cdots \otimes \tau}_{n \text{ times}}$ and $0^n := 0 \otimes \cdots \otimes 0$, again with arbitrary many ! in front:

$$\mathrm{Succ}_n := \lambda p.\, (p\, \lambda x_1 \ldots x_n.\, (\mathrm{Succ}\, x_1) \otimes \cdots \otimes (\mathrm{Succ}\, x_n)) : N^n \multimap N^n$$

$$p_n^{\S} := \lambda n.\S \boxed{\left(\right] (n\, N^n\, !\, \boxed{\mathrm{Succ}_n}) \left[_{\S}\ 0^n\right)} : N \multimap \S N^n$$

$$p_n^{\S\S^h!^i\S^j} := \lambda n.\S \boxed{\left(\right] (n\, \S^h!^i\S^j N^n\, !\, \boxed{\mathrm{step}}) \left[_{\S}\ \S \boxed{\cdots !\, \boxed{\cdots \S \boxed{0} \cdots} \cdots}\right)} : N \multimap \S\S^h!^i\S^j N^n$$

$$\mathrm{step} := \lambda p.\S \boxed{\cdots !\, \boxed{\cdots \S \boxed{(\mathrm{Succ}_n\ \left.\right] \cdots \left.\right] \cdots] p [_{\S} \cdots \left[_{!} \cdots \left[_{\S}\right)} \cdots} \cdots}.$$

Base cases for nesting one and two With this preparation we can easily create the smallest non-trivial iterated iterator of type $L(N)$. It consists of a list of length $|n|$ with copies of x in the list cells:

$$\circlearrowleft_0^{\S} := \lambda n, x.\S \boxed{\left(\right] \left(n\, L(N)\, !\, \boxed{(\mathrm{Cons}_N\ ] x [_{!})}\right) \left[_{\S}\ \mathrm{Nil}_N\right)}$$
$$: N \multimap \S! N \multimap \S L(N).$$

Using $\uparrow_N^{\S!}$ and then $\uparrow_N^{\S}$, we can get the arguments for $\circlearrowleft_0^{\S}$:

$$\lambda n^N.\S \boxed{\left(\lambda m^{!N}.\S \boxed{\left(\circlearrowleft_0^{\S}\ ] m [_{!}\ \left(\uparrow_N^{\S!}\ ] m [_{!}\right)\right)}\right] \left(\uparrow_N^{\S!}\ n\right) \left[_{\S}\right)} : N \multimap \S\S\S L(N).$$

In order to go one step further, i.e. the next deeper nesting level, we have to modify $\circlearrowleft_0^{\S}$ in such a way that we also get another banged copy of the input number in each list cell:

$$\circlearrowleft_0^{\S!} : \lambda n, x.\S \boxed{!\, \boxed{\left(\right] \left(\right] (n\, !(!N \otimes L(N))\, !\, \boxed{\mathrm{step}}) \left[_{\S}\ !\, \boxed{\left.\right]] x [_{\S} \left[_{!}\ \mathrm{Nil}_N\right)} \left[_{!}\ \lambda l, r.r\right)}}$$
$$: N \multimap \S!! N \multimap \S! L(N)$$

$$\mathrm{step} := \lambda p.! \boxed{\left(] p [_{!}\ \lambda l, r.\, \left(\mathrm{Cons}_N\, \S \boxed{] l [_{!}} r\right)\right)}.$$

Again, using the lifting term $\uparrow_N^{\S\S^h!^i\S^j}$, we are able to create the needed inputs for $\circlearrowleft_0^{\S!}$:

$$\lambda n^N.\S\boxed{\left(\lambda m^{!N}.\S\boxed{\left(\circlearrowleft_0^{\S!}\ ]m[_!\ \left(\uparrow_N^{\S!!}\ ]m[_!\right)\right)}\right]\left(\uparrow_N^{\S!}\ n\right)\Big[_\S\Big)} : N \multimap \S\S\S!L(N).$$

This can be used to provide $\circlearrowleft_1^{\S}$, the equivalent of $\circlearrowleft_0^{\S}$ for the next nesting level:

$$\circlearrowleft_1^{\S} := \lambda n, x.\S\boxed{\left(\right]\left(n\, L(L(N))\, !\boxed{\left(\mathrm{Cons}_{L(N)}\ ]x[_!\right)}\right)\Big[_\S\ \mathrm{Nil}_N\Big)}$$
$$: N \multimap \S!L(N) \multimap \S L(L(N)).$$

This scheme can be continued. After nesting one and two, we will now switch to the general case.

General case for arbitrary nesting For this we first need some helper terms which take a number and another argument of generic type α with enough bangs in front, and which produces a list with the length of the former and copies of the banged α argument in the cells of the list, with the cost of one !-modality. For the base case with only 2 bangs we get the following:

$$\circlearrowleft_1 := \Lambda\alpha\lambda n^N, x^{\S!!\alpha}.\S\boxed{!\boxed{\left(\right]\left(\circlearrowleft_1'\ \alpha\, n\, x\right)[_\S\Big[_!\ \lambda l, r.r\Big)}}$$
$$: \forall\alpha.N \multimap \S!!\alpha \multimap \S!(L(\alpha))$$
$$\circlearrowleft_1' := \Lambda\alpha\lambda n^N, x^{\S!!\alpha}.\S\boxed{\circlearrowleft_1''} : \forall\alpha.N \multimap \S!!\alpha \multimap \S!(!\alpha \otimes L(\alpha))$$
$$\circlearrowleft_1'' := \left(\right]\left(n\,\tau\,!\boxed{\lambda p.!\boxed{\left(]p[_!\ \lambda y^{!\alpha}, l^{L(\alpha)}.y \otimes \left(\mathrm{Cons}_\alpha\ \S\boxed{]y[_!}\ l\right)\right)}}\right)\Big[_\S\ \mathrm{base}\Big)$$
$$\tau := !(!\alpha \otimes L(\alpha))$$
$$\mathrm{base} := !\boxed{\left.\right]x[_\S\Big[_!\ \otimes \mathrm{Nil}_\alpha}.$$

For an arbitrary number of bang modalities we get the following with the same idea, but of course many more boxes depending on the number of bangs:

$$\circlearrowright_i := \Lambda\alpha\lambda n^N, x^{\S!!\cdots!\alpha}.\S\boxed{!\cdots!\boxed{\left(\right]\cdots\left.\right]\right](\circlearrowright_i'\ \alpha\, n\, x)\left[_\S\left[_!\cdots\left[_!\ \lambda l, r.r\right)\cdots}$$

$$: \forall\alpha.N \multimap \S\ \underbrace{!!\cdots!}_{i+1\text{ many}}\ \alpha \multimap \S\ \underbrace{!\cdots!}_{i\text{ many}}(L(\alpha))$$

$$\circlearrowright_i' := \Lambda\alpha\lambda n^N, x^{\S!!\cdots!\alpha}.\S\boxed{\circlearrowright_i''} : \forall\alpha.N \multimap \S\underbrace{!!\cdots!}_{i+1\text{ many}}\alpha \multimap \S\underbrace{!\cdots!}_{i\text{ many}}(!\alpha \otimes L(\alpha))$$

$$\circlearrowright_i'' := \left(\left.\right]\left(n\,!\cdots!(!\alpha \otimes L(\alpha))\,!\boxed{\text{step}}\right)\left[_\S\,!\boxed{\cdots!\boxed{\left.\right]\cdots}\left.\right]x\left[_\S\left[_!\cdots\left[_!\otimes \text{Nil}_\alpha\right.\cdots}\right)\right)$$

$$\text{step} := \lambda p.!\boxed{\cdots!\boxed{\left(\right]\cdots\left.\right]p\left[_!\cdots\left[_!\ \lambda y^{!\alpha}, l^{L(\alpha)}.y \otimes \left(\text{Cons}_\alpha\,\S\boxed{\left.\right]y\left[_!\right.}\,l\right)\right)\cdots}.$$

Now, all ingredients are available to define the iterated iterator term of arbitrary, but constant nesting level i in Light Affine Logic. It makes use of all the $\circlearrowright_i$ terms from above, up to the nesting level we want to have:

$$\circlearrowright_{0\ldots i} := \lambda x^N.\S\boxed{\S\boxed{\left(\right]\left]\left(p_{i+2}^{\S\S}\,x\right)\left[_\S\left[_\S\ \lambda y, x_0, x_1, \ldots, x_i.t\right)\right.}$$

$$: N \multimap \S\underbrace{!\cdots!}_{i\text{ times}}\ \underbrace{L(\cdots L}_{i\text{ times}}(N)\cdots).$$

$$t := \Big(\circlearrowright_0\ L(\cdots L(N)\cdots)\,x_i$$

$$\cdots\left(\circlearrowright_{i-1}\ L(N)\,x_{i-1}\ \left(\circlearrowright_i\ N\,x_i\ \left(\uparrow_N^{\S!^{i+1}}\ y\right)\right)\right)\cdots\Big).$$

Intuitively speaking, the term $(\circlearrowright_{0\ldots i}\ x)$ gives an i-dimensional cube of edge size x with a copy of x in every atomic cell, the "(hyper-) cube of iterated iterators" of dimension i.

Tuples on each nesting level Finally, another possible generalisation is the tuple-wise duplication of each level in the iterated iterator, i.e. some construction of type $L(\cdots L(N^n)^n\cdots)$. In other words, each sub-cube exists n times. This will be needed later if more than one iterator appears in the step of an iteration. This extension $\circlearrowright_{0\ldots i}^n$ is straight forward:

$$\circlearrowleft_i^n := \Lambda\alpha\lambda m^N, x^{\S!!\cdots!\alpha}.\S\boxed{!\cdots!\boxed{\left(\right]\cdots\left.\right](\circlearrowleft_i'\ m\,x)\left[_\S\left[_!\cdots\left[_!\ \lambda l, r.r\right)\right.\right.\right.}\cdots$$

$$: \forall\alpha.N \multimap \S\ \underbrace{!!\cdots!}_{i+1\text{ many}}\ \alpha \multimap \S\ \underbrace{!\cdots!}_{i\text{ many}}(L(\alpha)^n)$$

$$\circlearrowleft_i' := \Lambda\alpha\lambda m^N, x^{\S!!\cdots!\alpha}.\S\boxed{\circlearrowleft_i''} : \forall\alpha.N \multimap \underbrace{\S!!\cdots!}_{i+1\text{ many}}\ \alpha \multimap \underbrace{\S\cdots!}_{i\text{ many}}(!\alpha \otimes L(\alpha)^n)$$

$$\circlearrowleft_i'' := \left(\left.\right](m\,!\cdot\cdot!(!\alpha \otimes L(\alpha)^n)\,!\boxed{\text{step}})\left[_\S\ !\cdots!\boxed{\left.\right]\cdots\left.\right]x\left[_\S\left[_!\cdots\left[_!\ \otimes\ \mathbf{nil}_\alpha^n\right.\right.\right.}\cdots\right)$$

$$\text{step} := \lambda p.!\cdots!\boxed{\left(\right]p\left[_!\ \lambda y^{!\alpha}, l^{L(\alpha)^n}.y \otimes \left(\mathbf{cons}_\alpha^n\,\S\boxed{\left.\right]\cdots\left.\right]y\left[_!\cdots\left[_!\right.\right.\right.}\ l^n\right)\right)}\cdots.$$

And finally again the iterated iterator term, now with products of size n on every level:

$$\circlearrowleft_{0\ldots i}^n := \lambda x^N.\S\boxed{\S\left(\left.\right]\left.\right]\left(p_{i+2}^{\S\S}\,x\right)\left[_\S\left[_\S\ \lambda y, x_0, x_1, \ldots, x_i.t\right)\right.\right.}$$

$$: N \multimap \underbrace{\S!\cdots!}_{i\text{ times}}\underbrace{L(\cdots L}_{i\text{ times}}(N^n)^n\cdots)^n.$$

$$t := \Big(\circlearrowleft_0^n\ L(\cdots L(N^n)^n\cdots)^n\,x_0$$

$$\cdots\Big(\circlearrowleft_{i-1}^n\ L(N^n)^n\,x_{i-1}\ \Big(\circlearrowleft_i^n\ N^n\,x_i\ \Big(p_N^{\S!^{i+1}}\,y\Big)\Big)\Big)\cdots\Big).$$

5.2. Embedding LFPL into Light Affine Logic with Fixed Points

Now, that we know how to create the iterated iterators in LAL, we have to implement the embedding. For that, first take another look at Figure 5.4 to get a picture of the idea. We switch to the variant of LAL which also has fixed point types. In μLAL we can formulate inductive data types *without*

iteration. With some more effort one could probably do something similar in pure LAL, but this should not be our goal here.

> *Remark* 5.1. Of course, Church numerals in LAL can also be used without using all their iteration capabilities, i.e. just for case distinction and no iteration. In this sense, we could also destruct a list Cons_τ by Cons_τ. But, this case distinction will take place on the level of the list, *and not* on the level of the data of type τ. Hence, the Church numerals are not a flat data type that is needed for our purpose. For example, a list of lists $L(L(\tau))$ is not flat because the inner list is on one level higher:
>
> $$\forall\alpha.!(L(\tau) \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha).$$
>
> In order to switch back to pure Light Affine Logic one has to find a flat list data type, with the case distinction and the content of the list being on one level.

5.2.1. Flat Iteration

Light Affine Logic allows a normalisation strategy by levels (compare [AR02] and Section 2.42). The main issue in finding an embedding of LFPL, is to place the impredicative iteration scheme of LFPL (compare the (It)-rule in Definition 2.18) into the levels of μLAL. In Section 5.1.1 and Figure 5.4 we already sketched the basic idea:

The iterated iterators of Section 5.1 can be used to implement LFPL's impredicative iteration scheme, but separated from the actual data, i.e. the lists of type $L(\tau)$ in LFPL. The latter (i.e. the input) will be lifted – once – into one μLAL-level, called the *data level*. In this they will "live" as fixed-point type lists, i.e. without any iteration capability left.

The data level depends on the number of levels that are needed "below" to build the iterated iterators (compare Section 5.1.2 and Figure 5.4). The nested LFPL iterations will be translated into closed μLAL terms of the type $\S\cdots\S(\alpha \multimap \alpha)$, i.e. taking the base case of the iteration on the data level, and producing the result of the iteration on the data level as well:

Definition 5.2 (Flat iteration term)**.** The flat iteration terms It_i^n of width n and depth i are defined as

$$\mathrm{It}_0^n := \lambda l^N.\Lambda\alpha.\lambda f.\S\boxed{\lambda g.\left(\right](l\,\alpha\,f)[_{\S}\ g\right)} : N \multimap \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$$

$$\mathrm{It}_1^n := \lambda l^{L(N)}\Lambda\alpha\lambda f.$$

$$\S\left(\left]\left(l\,\S(\alpha\multimap\alpha)\,!\boxed{\lambda x^{N^n}, p^{\S(\alpha\multimap\alpha)}.\S\boxed{\lambda y.(\,](\,]f[_!\ x)[_{\S}\ (\,]p[_{\S}\ y))}}\right)\right[_{\S}\ \S\boxed{\lambda g.g}\right)$$

$$: L(N) \multimap \forall\alpha.!(N^n \multimap \S^1(\alpha\multimap\alpha)) \multimap \S^2(\alpha\multimap\alpha)$$

$$\vdots$$

$$\mathrm{It}_i^n := \lambda l^{L_n^i(N^n)}\Lambda\alpha\lambda f.$$

$$\S\left(\left]\left(l\,\S^i(\alpha\multimap\alpha)\,!\boxed{\lambda x^{L_n^{i-1}(N^n)}, p^{\S^i(\alpha\multimap\alpha)}.\S^i\boxed{\lambda y.(\,](\,]f[_!\ x)[_{\S^i}(\,]p[_{\S^i}\ y))}}\right)\right[_{\S}\ \S^i\boxed{\lambda g.g}\right)$$

$$: L_n^i(N^n) \multimap \forall\alpha.!(L_n^{i-1}(N^n) \multimap \S^i(\alpha\multimap\alpha)) \multimap \S^{i+1}(\alpha\multimap\alpha)$$

were $L_n^i(N^n)$ denotes $\overbrace{L(\cdots L}^{i}(N^n)^n\cdots)$, i.e. the type of the iterated iterator, and $\S^n\boxed{t}$ denotes $\S\boxed{\cdots\S\boxed{t}\cdots}$ with n boxes, and $]t[_{\S^n}$ denotes $\left]\cdots\right]t[_{\S}\cdots\left[_{\S}\right.$ with n nested holes.

Note that there is no $\cdot^n$ around the outer $L(\cdot)$ in $L_n^i(N^n)$.

The actual data is of type α with enough modalities to be on the data level, e.g. $\alpha := \S\S\S B$ for data level 4 (not $\alpha := \S\S\S\S B$ because α is inside a box in Church numerals). The *flat iteration* maps the data on the data level $\S\cdots\S\alpha$ to the data level $\S\cdots\S\alpha$, in other words: the mapping is *flat*, the reason for the name *flat iteration*.

Moreover, we see that in the step term applied to It_i^n the iterated iterator of type $L_n^{i-1}(N^n)$ is "available". This means that we can use It_{i-1}^n inside this step term, and hence nest flat iterations. Clearly, this can be repeated i times, i.e. a nesting of i requires that we start with It_i^n. In other words, we count the iteration nesting in the original LFPL term, and then use the *flat iteration term* It_i^n for an i that is big enough in order to simulate the LFPL iteration in our embedding.

If we start with the *flat iteration term* It_i^n to simulate the outer iteration

of the LFPL term, we have to use $\S\S^i\alpha$ for the data (for some instantiation for α), i.e. the data level is $i+1$ (plus the levels used to build the iterated iterators in the first place).

5.2.2. Translation of LFPL

With the flat iteration in place, the translation of LFPL into Light Affine Logic with fixed points can be defined. For LFPL-types $\tau \in \mathrm{Ty}_{\mathrm{LFPL}}$ and typed LFPL-terms $t \in \mathrm{Tm}_{\mathrm{LFPL}}$ the translation is denoted with $\phi(\tau)$ and $\Phi_i^n(t)$ where i and n must be chosen big enough for the iteration nesting in t.

Definition 5.3 (Translation of LFPL types into μLAL)**.**

$$\begin{aligned}
\phi : \mathrm{Ty}_{\mathrm{LFPL}} &\to \mathrm{Ty}_{\mu\mathrm{LAL}} \\
\phi(\diamondsuit) &:= \delta \\
\phi(B) &:= \forall\alpha.\alpha \multimap \alpha \multimap \alpha \\
\phi(\sigma \multimap \tau) &:= \phi(\sigma) \multimap \phi(\tau) \\
\phi(\sigma \otimes \tau) &:= \forall\alpha.(\phi(\sigma) \multimap \phi(\tau) \multimap \alpha) \multimap \alpha \\
\phi(\sigma \times \tau) &:= \phi(\sigma) \times \phi(\tau) \\
\phi(L(\sigma)) &:= \mu\alpha.((\delta \otimes \phi(\sigma) \otimes \alpha) + 1)
\end{aligned}$$

for some arbitrary type δ and $\phi(\sigma) \times \phi(\tau)$ as in Equation 3.1 in section 3.1.2.

The translation of the LFPL-types is the canonical (flat) embedding into μLAL. As described before, lists in LFPL are mapped onto the fixed-point type. Hence, they loose the ability to "drive" an iteration, but are flat in the sense that they do not need any modality anymore.

For the translation of the terms, we proceed in two steps: With $\phi_i^n(t)$ we translate a typed LFPL-term $t \in \mathrm{Tm}_{\mathrm{LFPL}}$ to a pseudo μLAL-term. In the second step with $\Phi_i^n(t)$ we put around the necessary §-boxes to get a (typable) μLAL-term.

> *Remark* 5.4. We call $\phi_i^n(t)$ a "pseudo μLAL-term", because it is not an actual μLAL-term yet: it uses $\Big]\cdots\Big]\cdot\Big[_{\S}\cdots\Big[_{\S}$-holes without the necessary

§$\boxed{\cdots \S \boxed{\cdot} \cdots}$ boxes around, which are then added in $\Phi_i^n(t)$. Recall, that the notation §$\boxed{\cdots]\cdot[_{\S}\cdots}$ is just a shorthand for §$\boxed{x =]\cdot[_{\S} \text{ in } \cdots}$.

Definition 5.5 (Translation of LFPL terms into μLAL)**.**

$$
\begin{aligned}
\phi_i^n(x) &:= v_x \\
\phi_i^n(\mathbf{tt}) &:= \Lambda\alpha.\lambda x\lambda y.x \\
\phi_i^n(\mathbf{ff}) &:= \Lambda\alpha.\lambda x\lambda y.y \\
\phi_i^n(\mathbf{cons}_\tau) &:= \lambda d^{\phi(\Diamond)}\lambda x^{\phi(\tau)}\lambda l^{\phi(L(\tau))}.\{(\mathbf{in}_L\, d \otimes x \otimes l)\} \\
\phi_i^n(\mathbf{nil}_\tau) &:= \{(\mathbf{in}_R\, 1)\} \\
\phi_i^n(\otimes_{\tau,\rho}) &:= \lambda l\lambda r.(l \otimes r) \\
\phi_i^n(\lambda x.t) &:= \lambda v_x.\phi_i^n(t) \\
\phi_i^n(\langle t^\tau, s^\sigma\rangle) &:= \langle \phi_i^n(t), \phi_i^n(s)\rangle \\
\phi_i^n((b\, p^{\sigma\times\sigma})) &:= \\
(\phi_i^n(b)\ \phi(\sigma\times\sigma) \multimap \phi(\sigma)\ \lambda p.(\pi_{\phi(\sigma),\phi(\sigma),0}\, p)\ \ \lambda p.(\pi_{\phi(\sigma),\phi(\sigma),1}\, p)\ \ \phi_i^n(p)) & \\
\phi_i^n((p^{\tau\times\sigma}\, \mathbf{tt})) &:= (\pi_{\phi(\tau),\phi(\sigma),0}\, \phi_i^n(p)) \\
\phi_i^n((p^{\tau\times\sigma}\, \mathbf{ff})) &:= (\pi_{\phi(\tau),\phi(\sigma),1}\, \phi_i^n(p)) \\
\phi_i^n((t\, s)) &:= (\phi_i^n(t)\ \phi_i^n(s)) \quad \text{due to } (\multimap^-) \\
\phi_i^n(\left(l^{L(\tau)}\, h^{\Diamond\multimap\tau\multimap\sigma\multimap\sigma}\right)) &:= \\
\lambda g^{\phi(\sigma)}.\left(\left(\left]\right(It_i^n\ m_*^{L_n^i(N^n)}\ \rho\, !\boxed{\text{step}}\right)\Big[_{\S^{i+1}}\ g \otimes \phi_i^n(l)\right)\ \phi(\sigma)\ \lambda s\lambda t.s\right) &
\end{aligned}
$$

with

$$
\begin{aligned}
\rho &:= \phi(\sigma) \otimes \phi(L(\tau)) \\
\text{step} &:= \lambda m^{L_n^{i-1}(N^n)^n}.\S^i\boxed{\lambda p^{\phi(\sigma)\otimes\phi(\tau)}.(p\,\rho\,\lambda s\lambda t.(\{\}t\{\ \phi(\sigma) \multimap \rho\,\text{left right})\ s))} \\
\text{left} &:= \lambda q\lambda s'.\left(q\,\lambda d^{\phi(\Diamond)}\lambda x^{\phi(\tau)}\lambda t'^{\phi(L(\tau))}.((\phi_{i-1}^n(h)\, d\, x\, s')\otimes t')\right) \\
\text{right} &:= \lambda u^1\lambda s'.s' \otimes (\mathbf{in}_R\, 1)
\end{aligned}
$$

and finally

$$\Phi_i^n : \mathrm{Tm}_{\mathrm{LFPL}} \to \mathrm{Tm}_{\mu\mathrm{LAL}}$$
$$\Phi_i^n(t) := \lambda m^{L_n^i(N^n)^n}.\S^{i+1}\boxed{\phi_i^n(t)}$$

with $\pi_{\phi(\tau),\phi(\sigma),0}$, $\pi_{\phi(\tau),\phi(\sigma),1}$, $\langle \phi_i^n(t), \phi_i^n(s)\rangle$ as Equations 3.2, 3.3, 3.4 in Section 3.1.2.

With m_* we denote a projection m_j of a tuple m for some j, such that the linearity constraints are fulfilled.

The presented embedding is mostly canonical in the sense of embedding a Linear System T-like calculus into Linear System F. The iteration case is the only non-obvious one in this definition:

The variable m is a tuple of iterated iterators. It is bound in the step term. Because there might be more than one iteration in an LFPL-term t, there might be more than one free m_* in the translation $\phi_i^n(t)$. But as every LFPL step term h is closed and there are only finitely many of those in a LFPL-term, the parameter n of the translation Φ_i^n can be chosen in a way, that every $m_\star$ can be bound in a linear way as component of the tuple m.

When translating a term t into $\phi_i^n(t)$, some m_* in the outer step term might be free. $\Phi_i^n(t)$ binds these in the tuple $m^{L_n^i(N^n)^n}$. The iterated iterators of Section 5.1.2 can provide a term of such a type.

5.2.3. Example Insertion Sort

In order to illustrate the translation Φ_i^n with a concrete example, the Insertion Sort example is considered again. Insertion Sort is easily writable in LFPL (compare [AS00, Hof99a] and Section 3.3.1). For the time being, let us assume for simplicity that the natural numbers are atomic types in LFPL and μLAL, i.e. denoted by $\mathbb{N}$ in either system with $\phi(\mathbb{N}) = \mathbb{N}$. Otherwise, the translation of $L(\mathbb{N}) = L(L(B))$ would distract from the more interesting parts.

We start with the LFPL-term:

$$\begin{aligned}
\leq &: \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}, \text{ given term}\\
\mathrm{insert} &:= \lambda d^{\Diamond}\lambda x^{\mathbb{N}}\lambda l^{L(\mathbb{N})}.\,((l\,h\,x \otimes \mathbf{nil}_{\mathbb{N}})\ \lambda y\lambda l.\,(\mathbf{cons}_{\mathbb{N}}\,d\,y\,l))\\
h &:= \lambda c^{\Diamond}\lambda y^{\mathbb{N}}\lambda p^{\mathbb{N}\otimes L(\mathbb{N})}.\,(p\,\lambda p_0\lambda p_1.\,((\leq\ y\,p_0)\ \lambda a\lambda b.a \otimes (\mathbf{cons}_{\mathbb{N}}\,c\,b\,p_1)))\\
\mathrm{sort} &:= \lambda l^{L(\mathbb{N})}.\,(l\,\mathrm{insert}\,\mathbf{nil}_{\mathbb{N}})\,.
\end{aligned}$$

Here, for simplicity, $\leq$ is supposed to return a sorted pair of the two inputs (in contrast to the definition of $\leq$ from Section 3.3.1, which returns a boolean as the result of the comparison; but this is a minor change).

The presentation of the translation starts from the outside, i.e. by translating "sort". The nesting level of the iterations in Insertion Sort (compare Section 5.1) is 2 (plus the iteration inside $\leq$, which we take for granted here). Hence, the data level is one higher. This means that all data is within 3 boxes.

$$\Phi_2^n(\mathrm{sort}) = \lambda m^{L_n^2(N^n)}.\S\boxed{\S\boxed{\S\boxed{\phi_2^n(\mathrm{sort})}}}$$

$$\phi_2^n(\mathrm{sort}) = \lambda v_l.\,(\phi_2^n((l\,\mathrm{insert}))\,\phi_2^n(\mathbf{nil}_{\mathbb{N}})) : \phi(L(\mathbb{N})) \multimap \phi(L(\mathbb{N}))$$

$$\phi_2^n((l\,\mathrm{insert})) = \lambda v_l \lambda g^{\phi(L(\mathbb{N}))}.$$
$$\big(\big(\big] (It_2^n\; m_*^{L_n^2(N^n)}\; \tau\; !\boxed{\mathrm{step}_{\mathrm{sort}}}\big)\big[_{\S^3}\; g \otimes \phi_2^n(l))\,\phi(L(\mathbb{N}))\,\lambda s\lambda t.s\big)$$

$$\tau := \phi(L(\mathbb{N})) \otimes \phi(L(\mathbb{N}))$$

$$\mathrm{step}_{\mathrm{sort}} := \lambda m^{L_n^1(N^n)^n}.$$
$$\S\boxed{\lambda p.\,(p\,\tau\,\lambda s\lambda t.\,((\}t\{\,\phi(L(\mathbb{N})) \multimap \tau\,\mathrm{left}_{\mathrm{sort}}\,\mathrm{right}_{\mathrm{sort}})\;s))}$$

$$\mathrm{left}_{\mathrm{sort}} := \lambda q\lambda s'.\,\big(q\,\lambda d\lambda x\lambda t'.((\phi_1^n(\mathrm{insert})\,d\,x\,s') \otimes t')\big)$$

$$\mathrm{right}_{\mathrm{sort}} := \lambda u\lambda s'.(s' \otimes (\mathbf{in}_R\,1)).$$

The step term of the LFPL term "sort" uses one nested iteration to implement "insert". With $i = 2$ for $\Phi_i^n(\mathrm{sort})$ we have the iterated iterators $m^{L_n^1(N^n)^n}$ available to "drive" the insertion (compare the line for $\mathrm{step}_{\mathrm{sort}}$). Hence, for the insertion we will use the *flat iteration* of level 1, i.e. It_1^n which takes $m_0^{L_n^1(N^n)}$ as its argument:

$$\phi_1^n(\mathrm{insert}) = \lambda v_d \lambda v_x \lambda v_l.$$
$$((\phi_1^n((l\,h))\,v_x \otimes \{\mathbf{in}_R 1\})\;\phi(L(\mathbb{N}))\,\lambda v_y \lambda v_l.\{(\mathbf{in}_L\,v_d \otimes v_x \otimes v_l)\})$$

$$\phi_1^n((l\,h)) = \lambda g.\big(\big(\big](It_1^n\; m_*^{L_n^1(N^n)}\; \tau\; !\boxed{\mathrm{step}_{\mathrm{ins}}}\big)\big[_{\S^2}\; g \otimes \phi_1^n(l))\,\phi(\mathbb{N} \otimes L(\mathbb{N}))\,\lambda s\lambda t.s\big)$$

$$\tau := \phi(\mathbb{N} \otimes L(\mathbb{N})) \otimes \phi(L(\mathbb{N}))$$

$$\mathrm{step}_{\mathrm{ins}} := \lambda m^{N^n}.$$
$$\S\boxed{\lambda p.\,\big(p\,\tau\,\lambda s\lambda t.\,\big((\}t\{\,\phi(\mathbb{N} \otimes L(\mathbb{N})) \multimap \tau\,\mathrm{left}_{\mathrm{ins}}\,\mathrm{right}_{\mathrm{ins}}\big)\;s)\big)}$$

$$\begin{aligned}
\text{left}_{\text{ins}} &:= \lambda q \lambda s'.\,\big(q\,\lambda d \lambda x \lambda t'.((\phi_0^n(h)\,d\,x\,s') \otimes t')\big)\\
\text{right}_{\text{ins}} &:= \lambda u^1 \lambda s'.(s' \otimes (\mathbf{in}_R\,1))\\
\phi_0^n(h) &= \Phi_0^n(\lambda c^{\diamondsuit} \lambda y^{\mathbb{N}} \lambda p^{\mathbb{N} \otimes L(\mathbb{N})}.\\
&\qquad \big(p\,\lambda p_0 \lambda p_1.((\leq\ y\,p_0)\,\lambda a \lambda b.(a \otimes (\mathbf{cons}_{\mathbb{N}}\,c\,b\,p_1)))\big)\\
&\quad = \lambda v_c \lambda v_y \lambda v_p.\\
&\qquad \big(p\,\lambda v_{p_0} \lambda v_{p_1}.((\leq\ v_y\,v_{p_0})\,\lambda v_a \lambda v_b.(a \otimes \{(\mathbf{in}_L\,v_c \otimes v_b \otimes v_{p_1})\}))\big).
\end{aligned}$$

As in both step terms there is only one iteration, we can choose n as low as 1. The step_{ins} term provides the iterator m^{N^n} of depth 0. This can be used inside $\leq$ to compute the ordered pair.

Note that the term $\Phi_2^n(\text{sort})$ has a symmetric/flat type. Hence, we could iterate this again, in contrast to the implementation of Insertion Sort in LAL in Section 3.3.1.3.

5.3. Conclusion and Outlook

In this chapter we have introduced an embedding of LFPL-terms into *Light Affine Logic with fixed points*. The main contribution is the insight that one can split LFPL-algorithms into the *impredictively recursive* structure and the *flat non-recursive* part. The former can be simulated by *iterated iterators* in μLAL, and the latter by a canonical embedding of the terms into one level of μLAL. Therefore, iterated iterators capture the impredicative iteration scheme of LFPL. Here, it is essential to spread this scheme over multiple levels of μLAL because of the limited expressivity and normalisation complexity of one single level.

Data size and complexity in the embedding In LFPL the diamond $\diamondsuit$ plays the role of money, and there is no closed term of type $\diamondsuit$ in LFPL. The only way to create a list is by having free variables with $\diamondsuit$ in a positive position in their types. Together with the linear typing of LFPL terms, this idea gives a syntactical way to control the size of computed lists and therefore of the complexity of the definable algorithms.

In the presented embedding into μLAL via $\Phi_i^n(t)$ the money role of the $\diamondsuit$ is not explicit anymore. Instead, $\diamondsuit$ is mapped to an arbitrary type δ. Hence, $\phi(\diamondsuit) = \delta$ is not used anymore to control the complexity.

Which mechanism is taking over this control role in the embedding?

We lift the input data (i.e. the lists) into the data level. There, lists become fixed-point types, i.e. they do not have the capability for iteration. Therefore, they do not contribute to the complexity. Instead, the input data is used – at the same time – to construct the iterated iterators. These do the actual work of iterating and their complexity depends polynomially on the input data length (by correctness of the μLAL-calculus for PTime).

Because we know that during LFPL normalisation no lists are created which are bigger than the input, we also know that the iterated iterators do enough steps to completely destruct the lists on the data level. *This is the missing link between the complexity of the embedding and the original non-size-increasing LFPL term.*

Is the embedding compositional? On the first sight, one cannot naively compose two embedded LFPL-algorithms $\Phi_i^n(t)$ and $\Phi_i^n(s)$. But, we can do this by using $\phi_i^n(t)$ and $\phi_i^n(s)$:

$$\phi_i^n((t\,s)) = (\phi_i^n(t)\,\phi_i^n(s))$$

and then binding the iterated iterators by

$$\Phi_i^n((t\,s)) = \lambda m.\S^{o+1}\boxed{(\phi_i^n(t)\,\phi_i^n(s))}.$$

Compared to $\Phi_{n_t}^{i_t}(t)$ and $\Phi_{n_s}^{i_s}(s)$, we have to use $i := \max\{i_t, i_s\}$ and $n := n_t + n_s$, because, each of $\phi_i^n(t)$ and $\phi_i^n(s)$ can have free $m_\star$ and all must be bound by the tuple m in $\Phi_i^n((t\,s))$. Hence, next to this small adaption of the meta variables i and n,$\Phi_i^n(\cdot)$ is in fact compositional.

Is the embedding satisfactory? In a sense, we have exploited a property of LFPL to make sure that the embedding does, what the original LFPL term did. What we get by this, is an embedding, which keeps the structure of LFPL terms *to some degree*. It is not as structure destructing as a Turing Machine simulation. But still one can argue that the presented embedding is not a completely structural one which would map the $\diamondsuit$ type to some μLAL type, which plays exactly the same role. It is not clear whether something like this could be done. After all, we still have a lot of freedom to choose the δ type in order to make its role explicit in the embedding.

6

Understanding Light Affine Logic as a Variant of System T

In this chapter we explore and try to understand Light Affine Logic as a programming language. The main aim is to make LAL comparable to other polynomial time calculi. We will concentrate on the central properties of the logic when writing down programs.

Light Affine Logic is a variant of higher order propositional logic (System F), or from a programming point of view of the typed polymorphic lambda calculus. While one is not forced to use certain standard encodings (compare [Lag03]) for data types, usually Church encodings are chosen to implement inductive types like numerals or lists. For instance, the completeness proofs for LAL of [AR02, AR00] go this route.

Our approach is to define a variant of System T, called Light Linear T (short $\mathrm{LLT}_!$), which is modelled after the typing rules of LAL. The goal is to get a system which captures the main ideas (e.g. the modalities, stratification and so on) of LAL, but which is formally, as a System T variant, much nearer to LFPL (compare Section 2.2), LT or BC than a higher order propositional logic like LAL, but also polymorphic light lambda calculi without constants like DLAL can be.

Remark 6.1. While being based on System T (instead of the logic System F), this choice seems to restrict the flexibility to choose also non-standard data type encodings. Hence, we have to set some goal how expressive $\mathrm{LLT}_!$ must be in order to consider it as a sensible System T correspondence of LAL. We have a very pragmatic attitude here: we choose the standard arithmetic with Church numerals and the completeness proofs [AR02, AR00] as a measure, in the sense that $\mathrm{LLT}_!$ should admit a very natural formulation of the algorithms used in those contexts. One focus will be the pull-out trick (compare Examples 3.15 and 6.21) which should be seamlessly supported by $\mathrm{LLT}_!$.

Our goal *should not be to allow a complete embedding* of LAL into our $\mathrm{LLT}_!$. From theoretical point of view, the question of the existence of this would be natural. But we do not want to go that route. Of course, compared to a formal embedding of LAL into the new calculus, our goal seems to be very vague. But in our view, the former does not tell much about the practical use of $\mathrm{LLT}_!$ to express algorithms in the system in their natural formulation.

Our motivation is a better understanding of the differences and common properties of the restrictions underlying (D)LAL, BC, LT and the non-size-increasing LFPL.

There are first results in this direction about embeddings (and failed attempts) of BC into LAL [MO04, NM03]. The arguments given are though not very convincing as it is not clear whether their considered "sensible" embeddings are general enough. For example in these works one looks for a correspondence between normal/safe and the modalities ! and § of LAL. It is concluded – mainly by using Beckmann's and Weiermann's example (compare Section 3.1.3 and [BW96]) – that safe recursion is not compatible with LAL's strong normalisation property [Ter01]. In fact, recent work of [Rov08a] seems to solve the problem of embedding BC into the LAL variant Weak Affine Light Typing (WALT) which allows better control over the normalisation order.

The approach of this work is different, as we define a system based on Gödel's T, which is much closer to BC, LT and LFPL. As argued in Remark 6.1 one might think, that this is a major restriction of possible embeddings as there is no polymorphism in the calculus and therefore also no (non-standard) implicitly defined data types. We argue though that *this can give a lot more freedom*: Gödel's T allows the addition of constants and

rewrite rules which have no direct correspondence in the System F world. In Chapter 7 we will pursue this very idea by modifying the iteration constant.

In this chapter we introduce $\mathrm{LLT}_!$ following the style of works about LAL [AR02, AR00, BT04] (compare Section 2.3) – i.e. we introduce a term calculus and a proof net calculus. The former seen as an easy way to describe proof nets without huge graphical illustrations. The model of computation, or better said the actual calculus $\mathrm{LLT}_!$ are the proof nets. The term system is just a notation for these.

The main reason for the choice to use proof nets for the computation is to avoid complications arising from blocked redexes. These would appear in the term calculus as a computation model without the addition of several permutative conversion rules. These problems are easily circumvented by using proof nets which abstract away the difference of terms which only differ in the application of permutative conversions.

Structure of this chapter This chapter starts in Section 6.1 with a preliminary motivation of the concept of *levels* as a replacement of the §-modality. In Section 6.2 the $\mathrm{LLT}_!$-calculus is formally introduced with the type system for terms and the proof net calculus for the normalisation, and a number of examples in Subsection 6.2.3 in order to get used to the syntax. Section 6.3 gives a completeness proof using a Turing Machine simulation, showing that every PTime function can be computed by a $\mathrm{LLT}_!$-algorithm. The core of the chapter is the normalisation of $\mathrm{LLT}_!$ in Section 6.4 and its complexity analysis in Subsection 6.4.1. Section 6.5 will motivate and discuss the choice of constants in $\mathrm{LLT}_!$ and the consequences for the calculus. The chapter ends in Section 6.6 with a conclusion on the results, the drawbacks of the calculus and an outlook of possible further work.

6.1. Preliminary Motivation – From Paragraphs to Levels

Before starting with the actual calculus $\mathrm{LLT}_!$, we want to give some intuition of the concept of *levels* which will be used later on. For this section we assume that the reader knows about the modalities of Light Affine Logic, especially about stratification and the role of the paragraph: Light Affine Logic (compare Definition 2.24) has the modalities ! ("bang") and § ("paragraph") in its type system. There are two rules for the former:

$$\frac{\emptyset \vdash t^{\tau}}{\emptyset \vdash !\boxed{t}^{!\tau}}\ (!_0) \qquad \frac{x^{\sigma} \vdash t^{\tau} \quad \Gamma \vdash s^{!\sigma}}{\Gamma \vdash !\boxed{x =]s[_{!}\ \text{in}\ t}^{!\tau}}\ (!_1)$$

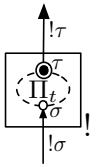
and one rule for the latter:

$$\frac{\overrightarrow{x^{\rho}}, \overrightarrow{y^{\sigma}} \vdash t^{\tau} \quad \overrightarrow{\Gamma \vdash r^{\S\rho}} \quad \overrightarrow{\Sigma \vdash s^{!\sigma}}}{\overrightarrow{\Gamma}, \overrightarrow{\Sigma} \vdash \S\boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_{\S}}, \overrightarrow{]s[_{!}}\ \text{in}\ t}^{\S\tau}}\ (\S)$$

6.1.1. Stratification

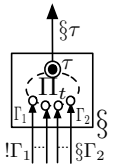
The key observation about the rules above is that they keep a kind of symmetry, in the following sense:

- In $(!_1)$ the variable x^{σ} is free in t^{τ} in the premise. In the conclusion the term is put into a box and gets the type $!\tau$. At the same time, x^{σ} is replaced by a term $s^{!\sigma}$. This symmetry is seen even better in the proof net notation:



  The proof net Π_t is put into a box. It has one input of type σ and the output of type τ. Outside the box the input is turned into $!\sigma$ and the output into $!\tau$. In other words, ! is introduced for the input and output at the same time.

- The same happens in the $(\S)$-rule: the free variables $\overrightarrow{x^{\rho}}, \overrightarrow{y^{\sigma}}$ are replaced with terms $\overrightarrow{]r[_{\S}}$ and $\overrightarrow{]s[_{!}}$ and a box is put around. The box term is typed as $\S\tau$, the subterms $\overrightarrow{r}$ and $\overrightarrow{s}$ as $\overrightarrow{\S\rho}$ and $\overrightarrow{!\sigma}$. In the proof net notation the output of Π_t of type τ is turned into $\S\tau$, and the input contexts Γ_1, Γ_2 into $\S\Gamma_1$ and $!\Gamma_2$:



Hence, again the modalities are introduced for the output and the input contexts at the same time.

The reduction rules of LAL are designed in a way that boxes can be duplicated and merged, but the nesting depth of subproof nets never changes. I.e., one can trace a node x during normalisation of a proof net, and it will always have the same number of boxes around. This stratification property is essential to allow a polynomial normalisation of a proof net.

Stratification of types If the number n of boxes around a node x (of type τ) of a proof net (or a subterm node in a syntax tree) never changes, one could also mark x with this very number directly, i.e. as a label n attached to x. We call this label n *the level of* x. The reduction rules guarantee that the level is constant for a given node during normalisation: it always represents the number of boxes around the node.

We can even go a step further and put the level into the type, i.e. we assign each type τ a level n, e.g. like in τ^n. A node x of type τ^n will be of level n in this setting.

Why do we do this? The level in the types is clearly redundant because the box structure around x already implies the value n. This redundancy though allows us to drop the paragraph box and the paragraph modality from the system altogether: we know that x^{τ^n} is of level n and therefore has n boxes around. Because the !-modality is still explicit and its level is annotated, we know the sequence of the n !- or §-modalities of a type τ^n. For example, $!^2!^4B^7$ means §§!§!§§B.

Without the paragraph How does the stratification property translate to this setting with levels in the types? First of all, we do not need the notation $]r[_{\S}$ anymore to mark that the §-modality is to be removed from the types

of $\overrightarrow{r}$ in the (§)-rule. Hence, the (§)-rule of LAL turns into a much simpler one, which only removes !-modalities from the types of $\overrightarrow{s}$:

$$\frac{\overrightarrow{y^{\sigma^l} \vdash t^{\tau^n}} \quad \overrightarrow{\Sigma \vdash s^{!^m\sigma^{n'}}}}{\overrightarrow{\Sigma} \vdash t[\overrightarrow{y} := \overrightarrow{]s[_{!^m}}]^{\tau^n}} \ (\S')$$

There is no need for the vector notation anymore, because multiple instances of the vector-less rule

$$\frac{y^{\sigma^l} \vdash t^{\tau^n} \quad \Sigma \vdash s^{!^m\sigma^n}}{\Sigma \vdash t[y :=]s[_{!^m}]^{\tau}} \ (\S'')$$

can do the same. Finally, we get rid of this substitution by transforming it into a simple $!^m$-elimination rule (or $]\cdot[_{!^m}$-introduction respectively):

$$\frac{\Sigma \vdash t^{!^m\sigma^n}}{\Sigma \vdash]t[_{!^m}{}^{\sigma^n}} \ (\S''')$$

Level restrictions What are the necessary restrictions in (§‴)? Because a term of type σ^n has one box less around itself than a term of type $!^m\sigma^n$, the restriction $n \geq m+1$ should certainly hold. Why not even $n = m+1$? Imagine the type $!^0B^3$. This corresponds to !§§B in LAL. Similarly, $!^0B^1$ corresponds to !B in LAL. Hence, $n = m+3$ is a reasonable instance of the (§‴)-rule.

Absolute vs. relative levels In Light Affine Logic the boxes (and by the $(!_{0/1})$- and (§)-rules also the types) describe the level *relatively,* in the following sense: a term of type §§B is a boolean value, whose actual value is computed inside two nested §-boxes, e.g. $t := \S\boxed{\S\boxed{\mathbf{tt}}}$, but also e.g.

$$t' := \S\boxed{\left(\lambda x.\S\boxed{]x[_{\S}}\ \S\boxed{\boxed{]}b[_{\S}\boxed{[}_{\S}}\right)}.$$

The level of the boolean in either case is 2, viewing it *on the basis of* t *and* t'. Clearly, the level of the boolean in $t'' := \S\boxed{t}$ is three though.

Conversely, with the concept of levels in the types, the level is a fixed and absolute natural number, that does not depend on the term the considered

node is subterm of. The lowest level is 0, if there is no box around at all.

> *Remark* 6.2. Putting fixed values into the types seems to destroy modularity because functions cannot easily be composed anymore. This is not a problem though because one can type terms with meta variables for the levels, i.e. in the style of typing schemes. For instance, we can say $f^{B^{\mathbf{n}} \multimap B^{\mathbf{n}+1}} \vdash (f\,\mathbf{tt})^{B^{\mathbf{n}+1}}$ for every natural number $\mathbf{n} \in \mathbb{N}$. We use this technique in the examples later on in Section 6.2.3.

6.1.2. Translation of Types

The motivation in the previous subsection suggests that levels and the §-modality are very similar. There should be some kind of translation between types that use the § and those using levels in types.

From levels to §-boxes We can translate types with levels back to those with the §-modality:

$$\Phi^n(!^p(\tau^q)) := \S^{p-n}!(\Phi^{p+1}(\tau^q)) \tag{6.1}$$

$$\Phi^n((\sigma^p \multimap \tau^q)^{\min\{p,q\}}) := \S^{\min\{p,q\}-n}(\Phi^{\min\{p,q\}}(\sigma^q) \multimap \Phi^{\min\{p,q\}}(\tau^q)) \tag{6.2}$$

$$\Phi^n((\sigma^p \otimes \tau^q)^{\min\{p,q\}}) := \S^{\min\{p,q\}-n}(\Phi^{\min\{p,q\}}(\sigma^q) \otimes \Phi^{\min\{p,q\}}(\tau^q)) \tag{6.3}$$

$$\Phi^n(B^p) := \S^{p-n}B \tag{6.4}$$

$$\Phi^n(L^{p+1}(\tau^{q+1})) := \S^{p-n}\forall\alpha.!(\Phi^{p+1}(\tau^{q+1}) \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha). \tag{6.5}$$

The parameter n describes that $\Phi^n(\tau^p)$ should be the translation of τ relative to level n, i.e. $\Phi^2(B^3) = \S B$ and $\Phi^0(B^3) = \S\S\S B$. Clearly, $n \leq p, q$ is necessary for all the cases above.
From Clause 6.1 we get $p + 1 \leq q$ in $!^p(\tau^q)$.
From Clause 6.5 we get $p + 1 \leq q + 1$, i.e $p \leq q$ in $L^p(\tau^q)$.

For $n = 0$ we get back the intuition, that the levels p, q are the number of boxes (or !, § in the types) around a term (or type).

From §-boxes to levels The translation $\Phi^n(\tau^p)$, as given above, is not surjective. It is not, even if we consider those LAL-types only which are built from Church-style lists, booleans, products, functions and !-, §-types: with the pattern on the left side we restrict the function space and the product

space to those types, whose level is the minimum of p and q (Equations 6.2 and 6.3). This is a general design choice that we follow within this chapter.

Remark 6.3. In Light Affine Logic there are the types $\S\sigma \multimap \S\tau$ and $\S(\sigma \multimap \tau)$. The latter is stronger in the sense that the following term (and the corresponding proof net) exists:

$$\lambda f^{\S(\sigma\multimap\tau)}, x^{\S\sigma}.\S\boxed{\left(\right]f[_{\S}\ ]x[_{\S}\left.\right)}^{\S\tau} : \S(\sigma \multimap \tau) \multimap (\S\sigma \multimap \S\tau).$$

With the decision to use the minimum in Equations 6.2 and 6.3 (and later in Definition 6.4), we restrict the types in this chapter to those of the shape $\S(\sigma \multimap \tau)$: a λ-abstraction for this type is inside a box, hence it is assigned the minimum of levels of the input and output type.

Conversely, the level of a λ-abstraction for $\S\sigma \multimap \S\tau$ would not be the minimum of σ and τ, but smaller. Hence, we do not consider those function spaces at all. This choice is kind of arbitrary, but motivated by the reduced number of level annotation in the types: we can drop the level from the $\multimap$ and $\otimes$ type constructors. Moreover, we did not encounter any example which needs the $\S\sigma \multimap \S\tau$-type.

With the given motivation for levels in the types we can now proceed and introduce the calculus formally.

6.2. Syntax

In the following, Light Linear T with ! ($\mathrm{LLT}_!$) is introduced, a linear variant of System T with iteration, explicit sharing via a multiplexer $\overset{n}{\lhd}$ and only one kind of box, namely $!^n\boxed{\cdot}$. The role of the § in LAL will be played by levels on the ground types.

Definition 6.4 (Types)**.** The set $\mathrm{Ty}_{\mathrm{LLT}_!}$ of linear types and the level of a type $\ell(\tau) \in \mathbb{N}_0$ are defined inductively by:

$$\sigma, \tau ::= B^n \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid L^n(\sigma) \mid !^n\sigma$$

$$\ell(\rho) := \begin{cases} n \text{ if } \rho \in \{B^n,\ L^n(\sigma),\ !^n\sigma\} \\ \min\{\ell(\sigma), \ell(\tau)\} \text{ otherwise} \end{cases}$$

with the side condition $\ell(\sigma) \geq n$ for $L^n(\sigma)$ and $\ell(\sigma) > n$ for $!^n\sigma$.

Definition 6.5 (Constants)**.** The set $\mathrm{Cnst}_{\mathrm{LLT}_!}$ of $\mathrm{LLT}_!$ constants:

$$\begin{aligned}
\mathbf{tt}^n, \mathbf{ff}^n &: B^n \\
\mathbf{Case}^n_\sigma &: B^n \multimap \sigma \multimap \sigma \multimap \sigma \text{ with } \ell(\sigma) \geq n \\
\mathbf{Case}^n_{\tau,\sigma} &: L^n(\tau) \multimap (\tau \multimap L^n(\tau) \multimap \sigma) \multimap \sigma \multimap \sigma \text{ with } \ell(\sigma) \geq n \\
\mathbf{cons}^n_\tau &: !^n(\tau \multimap L^{n+1}(\tau) \multimap L^{n+1}(\tau)) \\
\mathbf{nil}^n_\tau &: L^{n+1}(\tau) \\
\otimes^n_{\tau,\rho} &: \tau \multimap \rho \multimap \tau \otimes \rho \text{ with } \ell(\tau \otimes \rho) = n \\
\pi^n_{\tau,\rho,\sigma} &: \tau \otimes \rho \multimap (\tau \multimap \rho \multimap \sigma) \multimap \sigma \text{ with } \ell(\sigma) \geq \ell(\tau \otimes \rho) = n \\
\mathbf{It}^n_{\tau,\sigma} &: L^{n+1}(\tau) \multimap !^n(\tau \multimap \sigma \multimap \sigma) \multimap \sigma \multimap \sigma
\end{aligned}$$

whose level is the level of their type.

Note that the restrictions on types above imply restrictions on the possible types of the constants. E.g., by $\ell(\tau \multimap \sigma \multimap \sigma) > n$ as the restriction for $!^n(\tau \multimap \sigma \multimap \sigma)$ in the type of $\mathbf{It}^n_{\tau,\sigma}$ it follows that $\ell(\tau) > n$ and $\ell(\sigma) > n$ must hold.

We will sometimes leave out the level annotation on the ground type, e.g. write $!^n B$ for $!^n B^{n+1}$, if it is clear from the context.

The importance of the levels will become clearer in Section 6.4. Essentially, it will allow us to put an order on the redexes, such that a polynomial bound can be proved for the number of normalisation steps (see Theorem 6.42).

6.2.1. Terms

The used syntax is in analogy to the presented term system for Light Affine Logic in Section 2.3.2. It will mainly be used as the basis to define (sensible) proof nets later on. For the sake of brevity, we will also liberally use the term syntax to write out examples through out this text although the actual computation will be done on proof nets.

Definition 6.6 (Terms)**.** For a countably infinite set V of variable names the set of (untyped) light linear terms $\mathrm{Tm}_{\mathrm{LLT}_!}$ is defined inductively by:

$$s,t ::= x^\tau \mid c \mid \lambda x^\tau.t \mid (t\,s) \mid !^n\boxed{t} \mid !^n\boxed{x =]s[_{!^n}\ \text{in}\ t} \mid]t[_{!^n} \mid \left(s \overset{n\ x_1}{\lhd_{x_2}} t\right)$$

with types $\tau \in \mathrm{Ty}_{\mathrm{LLT}_!}$, $c \in \mathrm{Cnst}_{\mathrm{LLT}_!}$, $n \in \mathbb{N}_0$ and $x, x_1, x_2 \in V$. Terms which are equal up to the naming of bound variables are identified.

Variables Free and bound variables are defined as for Linear System T in Definition 2.10 plus the additional clauses:

$$\begin{aligned}
\mathrm{FV}(!^n\boxed{t}) &:= \mathrm{FV}(t)\\
\mathrm{FV}(!^n\boxed{x =]s[_{!^n}\ \text{in}\ t}) &:= \mathrm{FV}(s) \cup (\mathrm{FV}(t) \setminus \{x\})\\
\mathrm{FV}(]t[_{!^n}) &:= \mathrm{FV}(t)\\
\mathrm{FV}(\left(s \overset{x_1}{\lhd_{x_2}} t\right)) &:= \mathrm{FV}(s) \cup (\mathrm{FV}(t) \setminus \{x_1, x_2\}).
\end{aligned}$$

Subterms Subterms are defined as for Linear System T in Definition 2.10 with the following additional clauses:

$$\begin{aligned}
t &\lhd_{\mathrm{LLT}_!} !^n\boxed{t}\\
s,t &\lhd_{\mathrm{LLT}_!} !^n\boxed{x =]s[_{!^n}\ \text{in}\ t}\\
t &\lhd_{\mathrm{LLT}_!}]t[_{!^n}\\
s,t &\lhd_{\mathrm{LLT}_!} \left(s \overset{n\ x_1}{\lhd_{x_2}} t\right).
\end{aligned}$$

We call $]s[_{!^n}$ a hole in the box $!^n\boxed{x =]s[_{!^n}\ \text{in}\ t}$.

Notation 6.7. We write $!^n\boxed{t[x :=]s[_{!^n}]}$ instead of $!^n\boxed{x =]s[_{!^n}\ \text{in}\ t}$ as shorthand, but in fact s is not a subterm of t in $!^n\boxed{x =]s[_{!^n}\ \text{in}\ t}$. That is why we can *not* officially use the notation $!^n\boxed{t[x :=]s[_!]}$ which would imply $s \trianglelefteq !^n\boxed{t[x :=]s[_!]}$.

As shorthand we will also use an applicative style (compare Applicative System T in Section 2.1.1.3). We write $\left(p^{\tau\otimes\rho}\,\lambda l^\tau, r^\rho.s^\sigma\right)$ instead of $\left(\pi^n_{\tau,\rho,\sigma}\,p^{\tau\otimes\rho}\,\lambda l, r.s^\sigma\right)$ and the same for the $\mathbf{Case}^n_\sigma$-constant: $\left(b^{B^n}\,l^\sigma\,r^\sigma\right)$ in-

stead of $(\mathbf{Case}^n_\sigma\, b\, l\, r)$. Moreover, we are quite liberal in leaving out types or levels whenever they are clear from the context, or to add types (as upper index) to subterms to make understanding easier for the reader.

The usual projections for products can be easily defined as

$$\pi^n_0 := \lambda p^{\tau\otimes\rho}.\, \left(\pi^n_{\tau,\rho,\tau}\, p\, \lambda l, r.l\right)$$

and

$$\pi^n_1 := \lambda p^{\tau\otimes\rho}.\, \left(\pi^n_{\tau,\rho,\rho}\, p\, \lambda l, r.r\right).$$

Definition 6.8 (Term typing rules)**.** A context is an (unordered) finite *multiset* of type assignments from the variable names V to types, with the context condition that no variable is assigned different types at the same time.

Untyped terms are assigned types using the ternary relation $\vdash$ between a context Γ, a untyped term $t \in \mathrm{Tm}_{\mathrm{LLT}_!}$ and a type $\tau \in \mathrm{Ty}_{\mathrm{LLT}_!}$, denoted $\Gamma \vdash t^\tau$, via the following rules:

$$\frac{}{\Gamma, x^\tau \vdash x^\tau}\ (\mathrm{Var}) \qquad \frac{c \text{ constant of type } \tau}{\Gamma \vdash c^\tau}\ (\mathrm{Const})$$

$$\frac{\Gamma, x^\sigma \vdash t^\tau}{\Gamma \vdash (\lambda x^\sigma.t)^{\sigma\multimap\tau}}\ (\multimap^+) \qquad \frac{\Gamma_1 \vdash t^{\sigma\multimap\tau} \quad \Gamma_2 \vdash s^\sigma}{\Gamma_1, \Gamma_2 \vdash (t\, s)^\tau}\ (\multimap^-)$$

$$\frac{\emptyset \vdash t^\tau}{\emptyset \vdash !^n\boxed{t}^{!^n\tau}}\ (!_0) \qquad \frac{\Gamma \vdash t^{!^n\tau}}{\Gamma \vdash\,]t[^\tau_{!^n}}\ (]\cdot[_{!^n})$$

$$\frac{x^\sigma \vdash t^\tau \quad \Gamma \vdash s^{!^n\sigma}}{\Gamma \vdash !^n\boxed{x =]s[_{!^n}\ \text{in}\ t}^{!^n\tau}}\ (!_1)$$

$$\frac{\Gamma_1, y_1^{!^n\sigma}, y_2^{!^n\sigma} \vdash t^\tau \quad \Gamma_2 \vdash s^{!^n\sigma}}{\Gamma_1, \Gamma_2 \vdash \left(s \mathrel{\overset{n\ y_1}{\triangleleft}_{y_2}} t^\tau\right)^\tau}\ (\mathrm{C})$$

where Γ_1, Γ_2 denotes the multiset union which maintains the context condition.
In $(\multimap^+)$ there must not be x in Γ, in (C) no y_1, y_2 in Γ_1 and in $(!_1)$ no x in Γ.

In $(!_0)$ and $(!_1)$ the term t must not have subterms of level $\leq n$.
The level of a term is the level of its type.

The typing rules are *syntax directed*. Each subterm $s \trianglelefteq t$ of $\Gamma \vdash t^\tau$ appears as the premise of some rule in the typing derivation of t (or in the context in the case of variables).

Fact 6.9. *The restrictions of the* $(!_0)$*- and* $(!_1)$*-rules imply that* t *does not have occurrences*

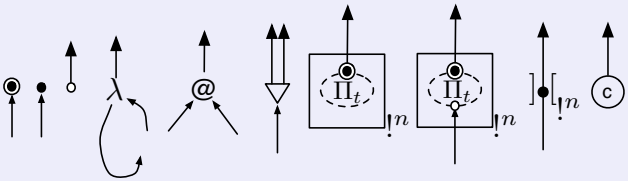
- *of constants of level* $\leq n$ *or*
- *subterms of shape* $]\cdot[_{!i}$ *with* $i \leq n$ *or*
- $!^i\boxed{\cdot}$*-boxes with* $i \leq n$ *or*
- $\left(\cdot \overset{i}{\triangleleft}^{y_1}_{y_2} \cdot\right)$ *with* $i \leq n$.

Definition 6.10 (Level of a subterm)**.** The level of a subterm $s \trianglelefteq t$ is the level $\ell(\sigma)$ of its type σ. The maximal level L_t of a well-typed term t is the maximal level of its subterms.

6.2.2. Proof Nets

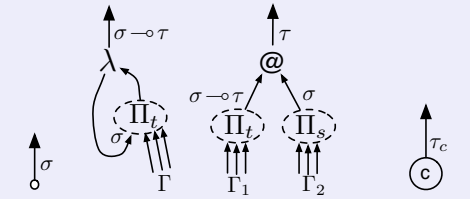
For the definition of proof nets we closely follow – mutatis mutandis – Section 2.3.3 about Light Affine Logic.

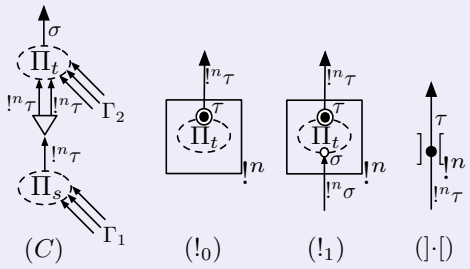
Definition 6.11 (Proof net structure)**.** A proof net structure is a labelled finite directed graph built from links (the edges) and nodes



with their given in- and output degrees and the following properties:

1. Each link is labelled with a type in $\mathrm{Ty}_{\mathrm{LLT}_!}$, satisfying the restrictions given by the type annotations of the nodes' in- and output links with $\sigma, \tau \in \mathrm{Ty}_{\mathrm{LLT}_!}$, τ_c the type of $c \in \mathrm{Cnst}_{\mathrm{LLT}_!}$ and Π_s, Π_t proof net structures in





2. There is exactly one ◉ node.

3. The Π_t in every $!^n$-box is a proof net structure having all the ∘-, ◉-nodes laid out as displayed.

We call

- the ∘-nodes input ports or free variables,
- the •-nodes output ports or weakening ports,
- the ◉-node the principal output port of the proof net structure,
- the ▽-nodes multiplexers,

- the output link of type σ in a λ-node, with another output link $\sigma \multimap \tau$, binding link or binding port,
- and the $]\cdot[_{!^n}$-nodes holes on level n.

The type of a (non-output) node is the type of its (non-binding) output link.
The type of an output node is the type of its input link.
For $(!_0)$ and $(!_1)$ we say that Π_t is the the proof net structure of the box.
A proof net structure is called closed if it has no input ports.
Nodes without a path to the principal node are called garbage. A proof net structure Π' is called cleanup of Π if Π' is created from Π by removing all garbage nodes and closing potential binding links or multiplexer output links with a weakening port.

Remark 6.12. Cleaning up a proof net structure after a reduction – at least in the simple tree-like setting with sharing here – is a simple task which is at worst linear in the proof net size. Hence, we will not study this procedure in detail. Basically, after each reduction where a subproof net Π_s is "unlinked" from a node one can trace back this Π_s to do the cleanup in a very efficient way. Hence, we will assume in the reduction rules further down that the cleanup is always done implicitly when firing a redex, such that we can always assume to have a clean proof net before and after each redex.

The boxes in a proof net create a *nesting structure*, i.e. there are possibly proof nets nested in boxes, and proof nets nested in nested proof nets and so on. Technically the nested proof nets do not directly belong to the graph that a proof net is. Hence, we introduce the concept of nested nodes:

Notation 6.13. We use the term nodes of a proof net structure Π for the graph-theoretic nodes of Π.

We use the term nested nodes of a proof net structure Π

- for nodes of Π
- and for *nested nodes of a proof net structure of a box of* Π.

Definition 6.14 (Subproof net structure)**.** A subproof net structure Π' of a proof net structure Π is either

- derived from a subgraph of Π with the edges which lay only "half" inside connected to additional input and (principal) output ports or
- a *subproof net structure* of a proof net structure Π_t of a box in Π.

Definition 6.15 (Proof net)**.** A proof net structure Π_t is called a proof net if Π_t is the image of the typing derivation $\Gamma \vdash t^\tau$ under the mapping which recursively translates each term typing rule of Definition 6.8 to the corresponding proof net structure rule in Definition 6.11.

The output link coming from the conclusion of the typing rule at the root of $\Gamma \vdash t^\tau$ is marked as principal node $\circledcirc$ in Π_t. Free variables are turned into input links, not appearing bound variables into non-principal output links.

We call a proof net Π a proof net of type τ (denoted with Π^τ) if τ is the type of the principal output node.

A subproof net (short a subnet) of a proof net for the term t^τ is a proof net of a subterm $s \trianglelefteq t$.

Note that the translation of the arrow introduction $(\multimap^+)$ that types $\Gamma \vdash \lambda x^\sigma.t^{\sigma\multimap\tau}$ with x not free in t uses a weakening ports for the "loose end" of the binding link. In the type system of the terms we do not use explicit weakening though for sake of simplicity. In the proof nets weakening is explicit by (non-principal) output nodes.

Fact 6.16. *The restrictions of the typing rules* $(!_0)$ *and* $(!_1)$ *of Definition 6.8 imply that the subnet* Π_t *of a* $!^n$*-box in a proof net* Π *does not have occurrences of*

- *constants of level* $\leq n$ *or*
- $]\cdot[_{!^i}$*-nodes with* $i \leq n$ *or*
- $!^i\boxed{\cdot}$*-boxes with* $i \leq n$ *or*

- *multiplexers* $\overset{i}{\lhd}$ *with* $i \leq n$

in analogy to Fact 6.9 for terms.

Remark 6.17. We have no explicit restriction about nodes of lower level in boxes *for proof nets*. Instead this is a consequence that a proof net is the image of a term into the proof net language.

The translation from terms to proof nets is mostly just the transformation of a term to its parse tree. The only exception is the image of the contraction rule (the multiplexers), which is not faithful because it drops the information about the position of the (C) in the typing derivation (more about this in Section 6.2.4). Hence, not every subproof net structure corresponds to a subterm, but the converse of course holds:

Fact 6.18. *Let* Π_s *be the subproof net of a proof net* Π_t *for the term* t^τ *with* $s \trianglelefteq t$ *as the subterm for* Π_s*. Then* Π_s *is a subproof net structure of* Π_t*.*

Definition 6.19 (Paths and proper paths)**.** Mutatis mutandis like Definition 2.34.

Fact 2.35 and Remarks 2.36, 2.37 about paths apply here as well.

Definition 6.20 (Level of a node, level of a proof net)**.** For a nested node s the level of s is the level of the its type. The maximal level n of nested nodes in Π is denoted by L_Π. The level of a proof net is the level of the principal node.

Each node, which is not input or output node, in a proof net Π for the term t^τ corresponds to a typing rule in the typing derivation of t^τ. Its (non-binding) output link corresponds to a subterm of t therefore, and the other way round. Hence, we get $L_t = L_\Pi$, i.e. both concepts of a level in terms and proof nets are the same.

6.2.3. Examples

In order to get an idea of the syntax, we start with some example terms:

Example 6.21 (Identity)**.** The most basic algorithm we will present is the iteration of the $\mathbf{cons}_\tau$-constant.

First, let us formulate this very naively in Light Affine Logic ($\uparrow$ as the §-lifting, see Remark 6.25):

$$L(\tau) := \forall\alpha.!(\tau \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$$
$$\mathrm{consIt}_{\mathrm{LAL}} := \lambda l^{L(\tau)}.\left(l\, L(\tau)\, !\boxed{\mathrm{step}}\, \S\boxed{\mathrm{Nil}_\tau}\, l\right) : L(\tau) \multimap \S L(\tau)$$
$$\mathrm{step} := \lambda x\lambda t.\Lambda\alpha\lambda f.\S\boxed{\lambda b.\left(]f[_!\ ]\uparrow x[_\S\ \left(](t\,\alpha\, f)[_\S\ b\right)\right)}.$$

The type is not symmetric (i.e. not $L(\tau) \multimap L(\tau)$), hence this algorithm is not the real identity. But one can do better:

$$\mathrm{consIt}'_{\mathrm{LAL}} := \lambda l^{L(\tau)}.\Lambda\alpha\lambda f.\S\boxed{\lambda b.\left(\right]\left(l\,\alpha\,!\boxed{\lambda x\lambda t.\,(]f[_!\ x\,t)}\right)\left[_\S\ b\right)}$$
$$: L(\tau) \multimap L(\tau).$$

The trick here is to pull out the abstractions $\Lambda\alpha\lambda f$ in front of the iteration itself (compare "pull-out" trick in Example 3.15). This works because the step term only uses f once. It is questionable though whether this is still the intended iteration of the $\mathbf{cons}_\tau$ because of the non-local nature of the transformation.

In our LLT!-system, the simple **cons**-iteration can be implemented directly without any trick whatsoever:

$$\mathrm{consIt}_{\mathrm{LLT!}} \quad := \quad \lambda l^{L^1(\tau)}.\left(\mathbf{It}^0_{\tau,L^1(\tau)}\, l\, \mathbf{cons}^0_\tau\, \mathbf{nil}^0_\tau\right).$$

Example 6.22 (Split)**.** The split algorithm, a central part of Merge Sort and in a slightly modified shape with a pivot element in Quick Sort, is non-size-increasing as it only distributes a list into two new lists. In our system LLT! the resulting type is symmetric:

$$\mathrm{split}_{\mathrm{LLT!}} := \lambda l^{L(\tau)}.\left(\pi^1_1\ \left(\mathbf{It}^0_{\tau,\sigma}\, l\, !^0\boxed{s}\,\mathbf{tt}^1 \otimes \mathbf{nil}^0_\tau \otimes \mathbf{nil}^0_\tau\right)\right)$$
$$\sigma := B^1 \otimes L^1(\tau) \otimes L^1(\tau)$$

$$s := \lambda x, p.\, \Big(p\, \lambda b^{B^1}, l, r.\, \big(t\ \big]\mathbf{cons}^0_\tau \big[_{!0}\ x\, l\, r\big)\Big)$$
$$t := \big(b\, \lambda c, x, l, r.(\mathbf{ff}^1 \otimes (c\, x\, l) \otimes r)\, \lambda c, x, l, r.(\mathbf{tt}^1 \otimes l \otimes (c\, x\, r))\big)\,.$$

Hence, one can iterate the $\mathrm{split}_{\mathrm{LLT}_!}$-term in another iteration.

The same algorithm, written in traditional Light Affine Logic, is not symmetric (like $\mathrm{consIt}_{\mathrm{LAL}}$ above), or it will get a somehow odd result type, which is different from the expected (and hoped for) pair of lists:

$$L(\tau) := \forall\alpha.!(\tau \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$$
$$\mathrm{split}_{\mathrm{LAL}} := \lambda l^{L(\tau)} \Lambda\alpha, \beta \lambda f.\S \boxed{\lambda a, b.\, \Big(\pi_1\ \Big(\big]\big(l\, B \otimes \alpha \otimes \beta\, !\boxed{s}\big)\big[_\S\ \mathbf{ff} \otimes a \otimes b\Big)\Big)}$$
$$: L(\tau) \multimap \forall\alpha, \beta.$$
$$!((\tau \multimap \alpha \multimap \alpha) \otimes (\tau \multimap \beta \multimap \beta)) \multimap \S(\alpha \multimap \beta \multimap \alpha \otimes \beta)$$
$$s := \lambda x, p.\, \big(p\, \lambda b, q.\, \big(q\, \lambda l, r.\, \big(t\ \big]f\big[_!\ x\, l\, r\big)\big)\big)$$
$$t := (b\, \lambda c, x, l, r.(\mathbf{ff} \otimes ((\pi_0\, c)\ x\, l) \otimes r)\, \lambda c, x, l, r.(\mathbf{tt} \otimes l \otimes ((\pi_1\, c)\ x\, r)))\,.$$

The type $\forall\alpha, \beta.!((\tau \multimap \alpha \multimap \alpha) \otimes (\tau \multimap \beta \multimap \beta)) \multimap \S(\alpha \multimap \beta \multimap \alpha \otimes \beta)$ is some kind of iterator type for two interconnected iterations which can not be used independently in full generality.

Remark 6.23. This example about the split function and its unusual type in LAL is actually a strong hint that a direct embedding of $\mathrm{LLT}_!$ into LAL is not possible. The design of $\mathrm{LLT}_!$ is deeply motivated by the seamless integration of the pull-out trick into the type system. Clearly, the pull-out trick applies to the split algorithm as shown above, and $\mathrm{LLT}_!$ easily types $\mathrm{split}_{\mathrm{LLT}_!}$ therefore. In LAL though, the application of the pull-out trick twice, i.e. for both result lists, cannot be done in a clean way, independently from each other.

Example 6.24 (Arithmetic)**.** The usual unary polynomials with $N := L(B)$ for unary numbers by ignoring the boolean value (compare Section 3.3.2.3 for the LAL version):

$$\mathrm{add}_n := \lambda x^{L^{n+1}(B)}, y^{L^{n+1}(B)}.\, \big(\mathbf{It}^n_{B, L^{n+1}(B)}\, x\, \mathbf{cons}^n_B\, y\big)$$

$$\mathrm{mult}_n := \lambda x^{L^{n+1}(B)}, y^{!^n L^{n+2}(B)}.$$
$$\left(\mathbf{It}^n_{B,L^{n+2}(B)}\, x\, !^n \boxed{\lambda b, p.\, \left(\mathrm{add}_{n+1}\ ]y[_{!^n}\ p\right)} \mathbf{nil}^{n+1}_B\right).$$

Example 6.25 (Lifting)**.** By induction on the type we get coercions for the (data) types

$$\sigma, \tau ::= B^n \mid \sigma \otimes \tau \mid L^n(\sigma) \mid !^n\sigma$$

from τ to the lifted version $\uparrow\tau$, one level higher. I.e., in $\uparrow\tau$ every ground type is lifted, but the levels of the $!^i$ *stay the same*:

$$\uparrow^n_{B^n} := \lambda b^{B^n}.\, \left(\mathbf{Case}^n_{B^{n+1}}\, b\, \mathbf{tt}^{n+1}\, \mathbf{ff}^{n+1}\right) : B^n \multimap B^{n+1}$$
$$\uparrow^n_{\sigma\otimes\tau} := \lambda p.\, \left(p\, \lambda l, r.\, \left(\uparrow^{n'}_\sigma\ l\right) \otimes \left(\uparrow^{n''}_\tau\ r\right)\right) : p^{\sigma\otimes\tau} \multimap p^{\uparrow\sigma\otimes\uparrow\tau}$$
$$\text{where } n = \min\{n', n''\}$$
$$\uparrow^n_{L^n(\sigma)} := \lambda l.\, \left(\mathbf{It}^n_{\sigma,\uparrow L^n(\sigma)}\, l\, !^n \boxed{\mathrm{step}} \mathbf{nil}^n_{\uparrow\sigma}\right) : L^n(\sigma) \multimap L^{n+1}(\uparrow\sigma)$$
$$\mathrm{step} := \lambda x, p.\, \left(\right]\mathbf{cons}^n_{\uparrow\sigma}[_{!^n}\ (\uparrow^n_\sigma\ x)\ p\right)$$
$$\uparrow^n_{!^n\sigma} := \lambda s.!^n \boxed{\uparrow^n_\sigma\ ]s[_{!^n}} :!^n\sigma \multimap !^n(\uparrow\sigma).$$

Remark 6.26. The lifting of the function space $\sigma \multimap \tau$ does not work as easily because σ appears negatively. Hence, one would have to lift the whole term recursively, not only a variable of that type. Otherwise, redexes inside the function definition would be of lower level. Therefore, they could be activated only after the outer beta-reduction takes places. This contradicts the normalisation by levels.

Example 6.27 (!-lifting)**.** As in Light Affine Logic, the lifting into banged types is not possible in such a general way. The reason is that $!^n$-boxes can only have one hole $]\cdot[_{!^n}$, but data types other than B^n have more than one component. Lifting each might be possible (e.g. via $\sigma \multimap !^n\uparrow\sigma$ and $\tau \multimap !^n\uparrow\tau$), but combining them into one data type (e.g. $!^n(\uparrow\sigma\otimes\uparrow\tau)$) will not work, because then the $]\cdot[_{!^n}$-holes of either

component will have to appear as holes of the combined term. The same argument applies to the list type $L^n(\sigma)$.

But fortunately, *some types can be lifted* into a banged version. Especially the common type $L^{n+1}(B^{n+1})$, that is used to represent natural numbers, can be lifted to $!^{n+1}L^{n+3}(B^{n+3})$:

$$\Uparrow^n_{B^n} := \lambda b.\left(\mathbf{Case}^n_{!^n B^{n+1}}\, b\, !\boxed{\mathbf{tt}^{n+1}}\, !\boxed{\mathbf{ff}^{n+1}}\right)$$

$$\Uparrow^n_{L^{n+1}(B^{n+1})} := \lambda l^{L^{n+1}(B^{n+1})}.\left(\mathbf{It}^n_{B^{n+1},\sigma}\, l\, !^n\boxed{\text{step}}\, !^{n+1}\boxed{\mathbf{nil}^{n+2}_{B^{n+3}}}\right)$$

$$\sigma := !^{n+1}L^{n+3}(B^{n+3})$$

$$\text{step} := \lambda x.\,(x\,\lambda p.c_0\,\lambda p.c_1)$$

$$c_0 := !^{n+1}\boxed{(]\mathbf{cons}^{n+2}_{B^{n+3}}[_{!^{n+2}}\ \mathbf{tt}\ ]p[_{!^{n+1}})}$$

$$c_1 := !^{n+1}\boxed{(]\mathbf{cons}^{n+2}_{B^{n+3}}[_{!^{n+2}}\ \mathbf{ff}\ ]p[_{!^{n+1}})}.$$

Note here, that the $!^{n+1}$-boxes in c_0 and c_1 seem to have two holes each, on the first sight. But in fact, the first $]\cdot[_{!^{n+2}}$ around the $\mathbf{cons}^{n+2}_{B^{n+3}}$ is a level $n+2$ hole, which is typed via the $(]\cdot[_{!^{n+2}})$-rule, and not via $(!_1)$.

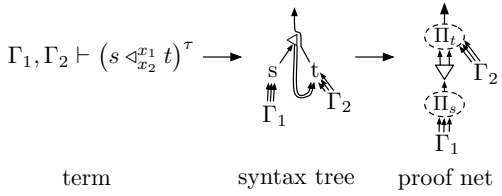
6.2.4. Connection between Terms and Proof Nets

Each typed term can be translated into a proof net according to the rules in Definition 6.15. The multiplexers in the terms are replaced by the proof net multiplexers, as shown in Figure 6.1a.

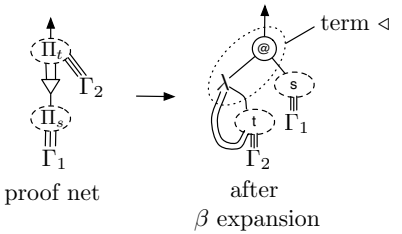
There is also a natural way to translate the proof net multiplexers back to terms by β-expansion, i.e. "abstracting out" the non-linear subnet s (see Figure 6.1c). Though, this translation is not uniquely defined because it is not clear at which position the (C) is to be inserted in the typing derivation.

Consider the proof net on the left in Figure 6.1c. Both syntax trees on the right are valid term representations of the proof net. Hence, proof nets and terms are different, though a proof net can also be seen as the set of those terms which result in the same proof net. In fact, we will compute with the proof net. In this sens, our term syntax is just a way to define proper proof net structures which have the shape of a parse tree with sharing.

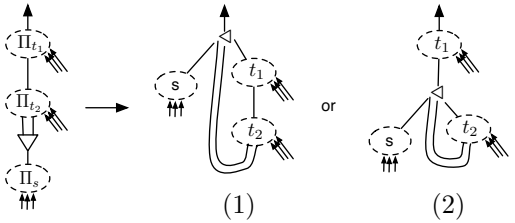
Our model of computation is the proof net, not the lambda term.



(a) From LLT!-terms to proof nets



(b) From proof nets back to LLT!-terms: we drop the multiplexer and do a β-expansion, which gives the picture on the right. As the term system should be linear, the λ- and the @−node are merged into a new $\lhd$-node, which is a binary binder (hence in the figure there are two "binder" lines going down).



(c) Lack of uniqueness in the translation from multiplexers in proof nets to LLT!-terms: both interpretations of the proof net are valid, in the sense of the transformation shown in Figure (b).

Figure 6.1.: Translating multiplexers between LLT!-terms and proof nets

Remark 6.28 (Permutative Conversions). The study of "real" lambda calculus formulations of logics with multiplexers has been done before [Ter01, BT04]. In [BT04] the role of the !-modality in the types is drastically reduced by only allowing ! on the left of an arrow $\multimap$. The resulting calculus DLAL is still complete for PTime and does not have the issues of permutative conversions, but the new type system forces global adaptions of algorithms to fit into the new typing rules.

The main tool of Terui [Ter01] is to add two permutative conversions:

- for two (C) rules in a row
- and for (C) and ($\multimap^-$) in a row, where (C) blocks a beta redex,

namely the following:

$$\dfrac{\dfrac{\Gamma_1, y_1^{!^n\sigma}, y_2^{!^n\sigma} \vdash t^\tau \qquad \dfrac{\Gamma_2, x_1^{!^n\sigma}, x_2^{!^n\sigma} \vdash s^\rho \quad \Gamma_3 \vdash r^{!^n\sigma}}{\Gamma_2, \Gamma_3 \vdash \left(r \overset{n\ x_1}{\lhd_{x_2}} s\right)^{!^n\rho}}\ (\mathrm{C})}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \left(\left(r \overset{n\ x_1}{\lhd_{x_2}} s\right) \overset{n\ y_1}{\lhd_{y_2}} t^\tau\right)^\tau}\ (\mathrm{C})}{\updownarrow \quad (\lhd\lhd\text{-pm})}$$

$$\dfrac{\dfrac{\Gamma_1, y_1^{!^n\sigma}, y_2^{!^n\sigma} \vdash t^\tau \quad \Gamma_2, x_1^{!^n\sigma}, x_2^{!^n\sigma} \vdash s^{!^n\sigma}}{\Gamma_1, \Gamma_2, x_1^{!^n\sigma}, x_2^{!^n\sigma} \vdash \left(s \overset{n\ y_1}{\lhd_{y_2}} t\right)^\tau}\ (\mathrm{C}) \qquad \Gamma_3 \vdash r^{!^n\rho}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \left(r^\rho \overset{n\ x_1}{\lhd_{x_2}} \left(s \overset{n\ y_1}{\lhd_{y_2}} t\right)\right)^\tau}\ (\mathrm{C})$$

and

$$\dfrac{\dfrac{\Gamma_1, x_1^{!^n\rho}, x_2^{!^n\rho} \vdash t^{\sigma\multimap\tau} \quad \Gamma_2 \vdash r^\rho}{\Gamma_1, \Gamma_2 \vdash \left(r \overset{n\ x_1}{\lhd_{x_2}} t\right)^{\sigma\multimap\tau}}\ (\mathrm{C}) \qquad \Gamma_3 \vdash s^\sigma}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \left(\left(r \overset{n\ x_1}{\lhd_{x_2}} t\right)\ s\right)^\tau}\ (\multimap^-)$$

$$\updownarrow \quad (\lhd @\text{-pm})$$

$$\dfrac{\dfrac{\Gamma_1, x_1^{!^n\rho}, x_2^{!^n\rho} \vdash t^{\sigma\multimap\tau} \quad \Gamma_3 \vdash s^\sigma}{\Gamma_1, x_1^{!^n\rho}, x_2^{!^n\rho}, \Gamma_3 \vdash (t\ s)^\tau}\ (\multimap^-) \qquad \Gamma_3 \vdash s^\sigma}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \left(r \overset{n\ x_1}{\lhd_{x_2}} (t\ s)\right)^\tau}\ (C)$$

Both conversions are indeed permutative conversions in the usual sense: observe that in the (C)-rule the type of the conclusion also appears as the type of the first premise. This give raise to the following two blocked redexes which can be activated by applying ($\lhd\lhd$-pm) or respectively ($\lhd$@-pm):

$$\left(\left(r \overset{n\ x_1}{\lhd_{x_2}} !^{\mathbf{n}}\boxed{\mathbf{s}}\right) \overset{n\ y_1}{\lhd_{y_2}} \mathbf{t}\right) \overset{\lhd\lhd\text{-pm}}{\underset{n}{\longrightarrow}} \left(r^{\rho} \overset{n\ x_1}{\lhd_{x_2}} \left(!^{\mathbf{n}}\boxed{\mathbf{s}} \overset{\mathbf{n\ y_1}}{\lhd_{\mathbf{y_2}}} \mathbf{t}\right)\right)$$

$$\left(\left(r \overset{n\ x_1}{\lhd_{x_2}} \lambda\mathbf{xt}\right) \mathbf{s}\right) \overset{\lhd@\text{-pm}}{\underset{n}{\longrightarrow}} \left(r \overset{n\ x_1}{\lhd_{x_2}} (\lambda\mathbf{xt}\,\mathbf{s})\right).$$

In the first case a multiplexer redex is blocked because the box $!^n\boxed{s}$ cannot be duplicated with the multiplexer on the right of it. In the second case the β-redex of λxt and s is blocked. In either case, the $\lhd$-node blocks the redex, while after the permutative conversion the redex is visible and can be fired.

The resulting lambda calculus λLA is shown to be strongly normalising in PTime [Ter01], and there is no reason to expect that the same cannot be done with a lambda version of LLT$_!$. Though, for the sake of simplicity, for our purposes we will stick to the easier proof net calculus which does not have these issues.

6.3. Completeness for Polynomial Time

For the (extensional) completeness proof we use a Turing Machine simulation which is supposed to run polynomially many steps in the size of the input.

Lemma 6.29. *Every boolean function $f : B^k \to B^l$ is expressible in LLT$_!$ as a closed term of type $\underbrace{B^n \otimes \cdots \otimes B^n}_{k\ many} \multimap \underbrace{B^n \otimes \cdots \otimes B^n}_{l\ many}$.*

Proof. Using the $\mathbf{Case}^n_{B^n}$ constant create a big case distinction on the input $s^{(B^n)^k}$. In every branch return the output $f(s)$ build up only by constants $\mathbf{tt}^n$, $\mathbf{ff}^n$ and $\otimes^n_{B^n,B^n}$.

The term can get big, but is still only of constant size for the simulation. □

We code the tape of the Turing Machine as a list $T := L^{n+1}(A \otimes P)$ with

- the alphabet $A := (B^{n+1})^{n_a}$, $n_a \in \mathbb{N}$, and
- a boolean marker $P := B^{n+1}$ for the current position on the tape (**tt** iff the head is at that position).

The number of states of the Turing Machine is finite and can be represented by $S := (B^{n+1})^{n_s}$ for some $n_s \in \mathbb{N}$.

The computation of the transition function is done by an iteration over the tape t^T. For the step term we use the following names for values before the update:

- a^A the old character,
- m_p^P the marker of the previous position,
- m^P the marker of the current position,
- m_n^P the marker of the next position,
- s^S the current state.

The new values after the update are a', m_p', m', m_n', s'. For each position on the tape we can compute the new character a', the new marker m', the current state s and the new state s' with a boolean function

$$f(a, m_p, m, m_n, s) = a' \otimes m' \otimes s \otimes s'^{A \otimes P \otimes S \otimes S}.$$

As the iteration type we use $P \multimap P \otimes T \otimes S \otimes S$, which stands for

$$\lambda m_n \ldots . m \otimes \left(]\mathbf{cons}^n_{A\otimes P}[_{!^n}\ a' \otimes m'\ \mathrm{tl}\right) \otimes s \otimes s' \ldots .$$

With this we get the following transition function:

$$\begin{aligned}
\text{transition} &:= \lambda r^{T\otimes S}.\left(r\,\lambda t, s.\left(\mathbf{It}^n_{A\otimes P, P\multimap P\otimes T\otimes S\otimes S}\,t\,!^n\boxed{\text{step'}}\text{base'}\right)\right)^{T\otimes S}\\
\text{base}' &:= \lambda m_n.\mathbf{ff} \otimes \mathbf{nil}^n_{A\otimes P} \otimes s \otimes (\mathbf{tt} \otimes \cdots \otimes \mathbf{tt})\\
\text{step}' &:= \lambda x^{A\otimes P}, p^{P\multimap P\otimes T\otimes S\otimes S}, m_n.\left(x\,\lambda a, m.\left((p\,m)\ \lambda m_p, \mathrm{tl}, s, s'.\text{result}'\right)\right)\\
\text{result}' &:= \Big(f(a, m_p, m, m_n, s\,\lambda a', m', s, s'.\\
&\qquad\qquad m\otimes(]\mathbf{cons}^n_{A\otimes P}[_{!^n}\ a'\otimes m'\ \mathrm{tl})\otimes s\otimes s'\Big).
\end{aligned}$$

The step has a symmetric type $T \otimes S \multimap T \otimes S$ which is the crucial property for the simulation because it allows iteration. Polynomials are expressible in

$\mathrm{LLT}_!$ by coercions, add_n and mult_n (compare the examples in the previous section).

Moreover, a natural number as input can be translated into a tape, say by $\mathrm{conv}_{in} : L^k(B^k) \multimap T$ (for some levels k and n) with a tape length which is polynomial in the input. The machine head cannot move further than that anyway, so it is big enough to do all computations, and there is never the need to enlarge the tape.

The output can be transformed back into a natural number, say by $\mathrm{conv}_{out} : T \otimes S \multimap L^{n+1}(B^{n+1})$.

Let $\mathrm{prep} : L^1(B^1) \multimap L^n(B^n) \otimes T$ be the preparation of the input of type $L^1(B)$ with

- the left component of the pair as the number of steps the Turing Machine should run,
- the right component of the pair as the starting tape computed via conv_{in}

and $s_0 \in S$ as the starting state. Then we get

$$\begin{aligned} \mathrm{tm} &:= \lambda x^{L^1(B^1)}.\,(\mathrm{conv}_{out}\ ((\mathrm{prep}\, x)\ \lambda z \lambda t_0.\mathrm{sim})) \\ &\quad : L^1(B^1) \multimap L^{n+1}(B^{n+1}) \\ \mathrm{sim} &:= \left(\mathbf{It}^{n-1}_{B^n, T\otimes S}\, z\, !^{n-1} \boxed{\lambda x^{B^n}.\mathrm{transition}}\, t_0 \otimes s_0\right) \end{aligned}$$

as a simulation of Turing Machine which runs polynomially many steps:

Theorem 6.30. *Every function $f : N \to N$ in PTime can be computed by a closed $\mathrm{LLT}_!$ term $t : L^1(B^1) \multimap L^{n+1}(B^{n+1})$ for some level $n \in \mathbb{N}$.*

6.4. Normalisation via Case Distinction Unfolding

For simplicity of the complexity proof, a reduction order should be studied which is as near as possible to the normalisation which takes place in LAL. The main idea there is that normalisation is done by levels, i.e. redexes are assigned a level according the types of terms involved. Redexes with lower levels (in LAL this means redexes which have fewer boxes around) are fired first.

For LAL this is in fact a very bad reduction strategy. But, it can be shown that even this is polynomial in the length of the terms if the maximal level

(i.e. nesting of boxes) is bounded [AR00, AR02], and even more: LAL is strongly normalising in PTime [Ter01].

In the following, reduction by levels for $\mathrm{LLT}_!$ is considered and shown to be sound for PTime. The main point here is *how* and especially *when* iterations are unfolded:

Iteration in LAL In LAL, as a variant of System F, Church numerals or the corresponding type for lists are used for inductive types. Church numerals are "their own" iterator. A Church numeral l applied to an iteration step term f by $(l\, f)$ "executes" the iteration of f. By the type $\forall\alpha.!(\cdots \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ the step term itself (i.e. the content of the box that makes up f) is on a higher level (because of the ! for the step type) and its normalisation will, therefore, not take place until normalisation of the outer level is completely finished. In other words, the substitution of the step terms into the iterator (and the duplication of the step terms that is needed for this) is completely separated from the normalisation of the step terms themselves (i.e. the normalisation of the content of the box of f).

Iteration in $\mathrm{LLT}_!$ How does this translate to the world of $\mathrm{LLT}_!$, i.e. a System T like calculus with $\mathbf{It}^n_{\tau,\sigma}$ and $\mathbf{cons}^n_\tau$? Here, the substitution of the step term into a numeral means that the $\mathbf{cons}^n_\tau$ must be replaced by the actual step term when it comes in contact with the iterator. This is straightforward as we have a flat $\mathbf{Case}^n_{\tau,\sigma}$-constant for lists in the system which can implement the iteration via a long case distinction :

$$\begin{aligned}&\mathrm{CaseIt}^n_{l,0} := \lambda g.g\\ &\mathrm{CaseIt}^n_{l,k+1} :=\\ &\quad \left(\mathbf{Case}^{n+1}_{\tau,\sigma\multimap\sigma}\, l\, \lambda x, l', g.\, (]f_{k+1}[_{!^n}\ x\ \left(\mathrm{CaseIt}^n_{l',k}\, g\right))\ \lambda g.g\right)\end{aligned}$$

for copies f_i of the step term if the length of the list l is bounded by some $k+1$ and known in advance. The needed maximal length $k+1$ of this case construction can be obtained by counting the number of $\mathbf{cons}^n_\tau$-constants and $\overset{n}{\lhd}$ in the list term: each multiplexer which duplicates a $\mathbf{cons}^n_\tau$ vanishes. Hence, the sum of both numbers is a proper upper bound (which we will later call K_n for level n) of the number of $\mathbf{cons}^n_\tau$ that can ever be created during normalisation of level n.

Cuts In the following, the different kinds of cuts will be defined. All the cuts will get assigned a level n which is the minimum of the levels of the types involved. The cut relation is indexed by this level as in $\longmapsto^n_\star$. The lower index $\star$ will mark the kind of redex:

Definition 6.31 (Normalisation/Cut Elimination)**.** The graph rewriting relation between proof nets is split into the following redexes (compare [AR00] and Definition 2.40 for LAL):

1. The linear cuts (see Figure 6.2a):

$$\begin{aligned}
(\lambda x^\sigma.t^\tau\, s) &\longmapsto^n_l t[x := s] \quad \text{with } \ell(\sigma \multimap \tau) = n\\
\left(\pi^n_{\sigma,\tau,\rho}\ \left(\otimes^n_{\sigma,\tau}\, s\, t\right)\right) &\longmapsto^n_l \lambda f.\,(f\, s\, t)\\
(\mathbf{Case}^n_\tau\ \mathbf{tt}^n) &\longmapsto^n_l \lambda x^\tau \lambda y^\tau.x\\
(\mathbf{Case}^n_\tau\ \mathbf{ff}^n) &\longmapsto^n_l \lambda x^\tau \lambda y^\tau.y\\
\left(\mathbf{Case}^n_{\tau,\sigma}\ \mathbf{nil}^{n-1}_\tau\right) &\longmapsto^n_l \lambda f \lambda g.g\\
\left(\mathbf{Case}^n_{\tau,\sigma}\ \left(\right]\mathbf{cons}^{n-1}_\tau\left[_{!^{n-1}}\ v\, l\right)\right) &\longmapsto^n_l \lambda f \lambda g.\,(f\, v\, l)
\end{aligned}$$

2. The iteration cuts for closed l:

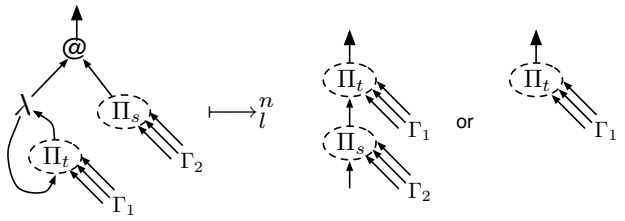
$$(\mathbf{It}^n_{\tau,\sigma}\, l\, f\, g) \longmapsto^n_{i,K_n} \left(\left(f \mathrel{{}^{n}\!\lhd^{f_1}_{f'_1}} \ldots \left(f'_{K_n-1} \mathrel{{}^{n}\!\lhd^{f_{K_n}}_{f'_{K_n}}} \left(\mathrm{CaseIt}^n_{l,K_n}\, g\right)\right)\right)\ g\right).$$

3. The shifting cuts (see Figure 6.2b):

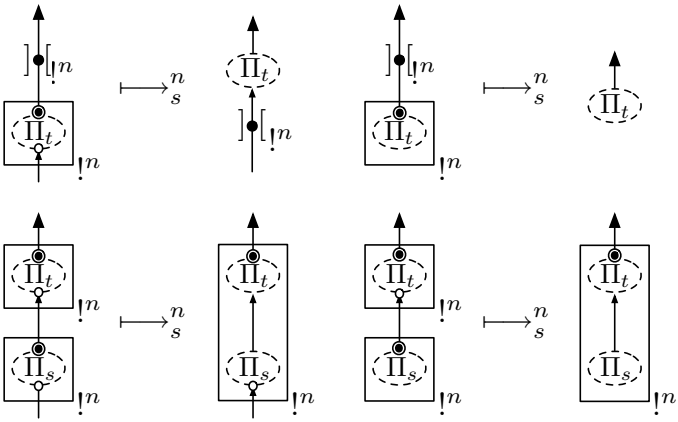
$$\begin{aligned}
\left.\right]!^n\boxed{x = \left.\right]s\left[_{!^n}\ \text{in } t\right.}\left[_{!^n}\right. &\longmapsto^n_s t[x := \left.\right]s\left[_{!^n}\right.]\\
\left.\right]!^n\boxed{t}\left[_{!^n}\right. &\longmapsto^n_s t\\
!^n\boxed{x = \left.\right]!^n\boxed{y = \left.\right]s\left[_{!^n}\ \text{in } v\right.}\left[_{!^n}\ \text{in } t\right.} &\longmapsto^n_s !^n\boxed{y = \left.\right]s\left[_{!^n}\ \text{in } t[x := v]\right.}\\
!^n\boxed{x = \left.\right]!^n\boxed{v}\left[_{!^n}\ \text{in } t\right.} &\longmapsto^n_s !^n\boxed{t[x := v]}.
\end{aligned}$$

4. The multiplexer/contraction cuts (see Figure 6.2c):

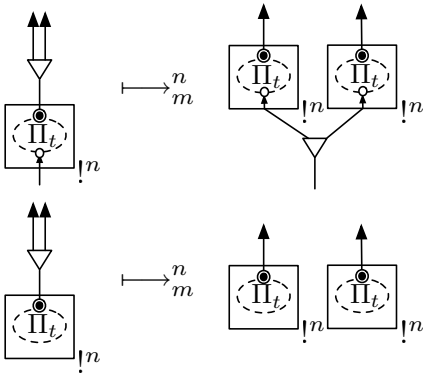
$$\left(!^n\boxed{x = \left.\right]s\left[_{!^n}\ \text{in } v\right.} \mathrel{{}^{n}\!\lhd^{x_1}_{x_2}} t\right)$$



(a) Linear cut $\longmapsto_l^n$ for a β-redex: the left if there is a path from the binding link back to the λ-node, the right otherwise.



(b) Shifting cut $\longmapsto_s^n$



(c) Multiplexer cut $\longmapsto_p^n$

Figure 6.2.: Cuts in LLT!

$$\longmapsto^n_m \left(s \mathrel{{}^{n\,y1}\!\!\lhd_{y_2}} t[x_1 :=!^n \boxed{x =]y_1[_{!^n} \text{ in } v}, x_2 :=!^n \boxed{x =]y_2[_{!^n} \text{ in } v}] \right)$$
$$\left(!^n\boxed{v} \mathrel{{}^{n\,x1}\!\!\lhd_{x_2}} t\right) \longmapsto^n_m t[x_2 :=!^n \boxed{v}, x_2 :=!^n \boxed{v}].$$

The last two kinds are called polynomial redexes and are denoted by $\longmapsto^n_p := \longmapsto^n_s \cup \longmapsto^n_m$, an arbitrary redex by $\longmapsto^n := \longmapsto^n_l \cup \longmapsto^n_{i,K_n} \cup \longmapsto^n_s \cup \longmapsto^n_m$.

We write $\Pi_t \longrightarrow^n \Pi_{t'}$ if $\Pi_s \longmapsto^n \Pi_{s'}$ for some subproof net Π_s of Π_t and $\Pi_{t'}$ is Π_t with Π_s replaced by $\Pi_{s'}$ (and mutatis mutandis $\longrightarrow^n_\star$ for $\longmapsto^n_\star$ redexes) after cleaning up.

A proof net Π_t is called $\longmapsto^n_\star$-normal or $\longrightarrow^n_\star$-normal if there is no $\Pi_{t'}$ with $\Pi_t \longrightarrow^n_\star \Pi_{t'}$. The notation $\xrightarrow{\text{nf}}{}^n_\star$ denotes the relation which puts a term in relation with its normal form(s), i.e.

$$\Pi \overset{\text{nf}}{\sim} \Pi' :\Longleftrightarrow \Pi \sim^* \Pi' \wedge \Pi' \text{ is } \sim\text{-normal}.$$

Note that the redexes in the previous definition are *graph rewriting rules* working on proof nets. *The definition uses the term syntax just for brevity.*

The K_n in the $\longmapsto^n_{i,K_n}$-cut is the maximal length of the list, such that the CaseIt^n_{l,K_n}-term properly simulates the iteration. The complexity proof of Theorem 6.42 will choose a correct value for K_n.

> *Remark* 6.32. The redexes, especially those concerning constants, are defined in a way that they *do not create new redexes on lower levels.* This is the key to make normalisation by levels work. In Section 6.5 we will look at this in more detail.

6.4.1. Complexity of Normalisation

Inside a $!^n$-box no subterms (or nodes in the proof nets) are allowed which have types of level $\leq n$, by the typing rules $(!_0)$ and $(!_1)$. This is the essential property to normalise proof nets because it allows duplication of boxes without creating new nodes or redexes on level $\leq n$.

Lemma 6.33. *A proof net Π_t of a box of level n in a proof net Π is $\longmapsto^k$-normal for every $k \leq n$.*

Proof. By definition of proof nets (or the typing rules $(!_0)$ and $(!_1)$ for the terms) Π_t has no (nested) node of level $\leq n$. Every redex of level $\leq n$ though involves such a (nested) node of the very same level. Hence, Π_t is $\longmapsto^k$-normal for $k \leq n$. □

The normalisation is claimed to be sound for PTime. To make this precise, first a size measure is needed for a proof net Π. This is usually the number of nodes. The same is essentially done here now, but with an independent precise size measure for each level. This will be needed for the analysis of the complexity of the normalisation later on.

Definition 6.34 (Proof net size)**.** The size of level n of a proof net Π is defined as

$$|\Pi|_n := \sum_{v \text{ nested node of } \Pi} |v|_n$$

with:

- $|v|_n := 0$ if $\ell(\tau) \neq n$ for the type τ of v,
- $|v|_n := 0$ for input and output ports,
- $|v|_n := 1$ in all other cases.

The size $|t|_n$ of a term t^τ is the size of the proof net Π_t of t. The size of the whole term is then defined by $|t| = \sum_{i=0}^{N} |t|_i$ if N is the highest level in t (and mutatis mutandis for a proof net Π).

Remark 6.35. An alternative equivalent size measure definition for terms t^τ would be the following, recursively with the size of the subterms:

$$\begin{aligned}
|x^\tau|_n &:= 0 \\
|c^\tau|_n &:= \delta_{n,\ell(\tau)} \\
|\lambda x^\sigma.r|_n &:= |r|_n + \delta_{n,\ell(\tau)} \\
|(r\ s)|_n &:= |r|_n + |s|_n + \delta_{n,\ell(\tau)} \\
\left|!^k\boxed{r}\right|_n &:= |r|_n + \delta_{n,\ell(\tau)} \\
\left|!^k\boxed{x =]s[_{!^k}\ \text{in}\ r}\right|_n &:= |s|_n + |r|_n + \delta_{n,\ell(\tau)}
\end{aligned}$$

$$\left|]r[_{!^k}\right|_n := |r|_n + \delta_{n,\ell(\tau)}$$
$$\left|\left(r \overset{k}{\lhd}_{x_2}^{x_1} s\right)\right|_n := |r|_n + |s|_n + \delta_{n,\ell(\tau)},$$

with

$$\delta_{ij} := \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

as the *Kronecker symbol*.

Definition 6.36 (Normalisation measure)**.**
Let Π_t be $\longmapsto^i$-normal for every $i < n$ and $\longmapsto_l^n$-normal. Then the normalisation measure for level n $\nu_n(\Pi_t)$ is defined as

$$\nu^n(\Pi_t) := \left\langle c_W^n(\Pi_t), \ldots, c_0^n(\Pi_t), |\Pi_t|_{\leq n}, |\Pi_t|_{>n} \right\rangle$$

where $c_w(\Pi_t)$ counts the (nested) multiplexers $\overset{n}{\lhd}$ of *weight* w in Π_t, while W is the maximal *weight* of $\overset{n}{\lhd}$ appearing in Π_t.

The number of nested nodes of level $\leq n$ is denoted by $|\Pi_t|_{\leq n}$, the number of nested nodes of level $> n$ by $|\Pi_t|_{>n} := \sum_{k=n+1}^{\infty} |\Pi_t|_k$. The order $\prec$ is the strict lexicographical order on $\nu^n(\Pi_t)$.

The weight $w_\Pi^n(v)$ of a nested multiplexer node v in Π is computed using the following rules (compare Figure 6.3):

- The output link of a $(!_0)$-box has the weight 1.
- The output link of a $(!_1)$-box has the weight of its input link $+1$.
- The output link of a multiplexer has the weight of its input link.
- A multiplexer has the weight of its input link.
- Every other link has the weight 0.

In other words, $w_\Pi^n(\overset{n}{\lhd})$ counts the maximal number of $!^n$-boxes that will be duplicated by that multiplexer when reducing shifting and multiplexer redexes. Compare $w_\Pi^n(\overset{n}{\lhd})$ with $\mathrm{wgt}(\Pi)(a)$, and $\nu^n(\Pi_t)$ with $\varphi_l(\Pi)$ in [AR00].

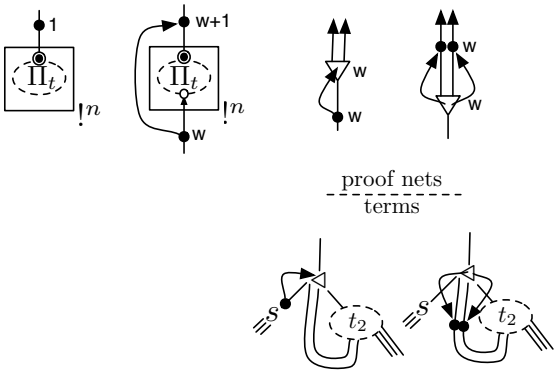


Figure 6.3.: The weight $w^n_\Pi(\cdot)$ for the proof net Π and corresponding term t. The arrows show how the weight propagates from the arrow start to the arrow end.

Remark 6.37. We could also define the weight $w^n_t(s)$ for a term $s \trianglelefteq t$ as:

$$w^n_t(\left(q \overset{n\ x_1}{\triangleleft}_{x_2} r\right)) := w^n_t(q) \tag{6.6}$$

$$w^n_t(x) := w^n_t(m) \text{ if } x \text{ is bound in } t \text{ by a multiplexer } m \tag{6.7}$$

$$w^n_t(!^n\boxed{r}) := 1 \tag{6.8}$$

$$w^n_t(!^n\boxed{x =]q[_{!^n} \text{ in } r}) := w^n_t(q) + 1 \tag{6.9}$$

and $w^n_t(s) := 0$ for all other cases, giving the same weight in the proof net Π_t.

In the term representation of the definition of $w^n_t(\cdot)$ it is not obvious which kind of graph the weight propagation creates on the subterms. The proof net representation in Figure 6.3 is much easier to grasp because the artificial $\triangleleft$-nodes in the term representation do not complicate the structure.

As there is no rule for weight propagation for the λ node, which is the only way to cause loops in proof nets, the arrows of the weight propagation

form trees: the multiplexer trees. Note that there can be many of those trees in a proof net. But no two of them overlap, i.e. a node is at last part of one multiplexer tree.

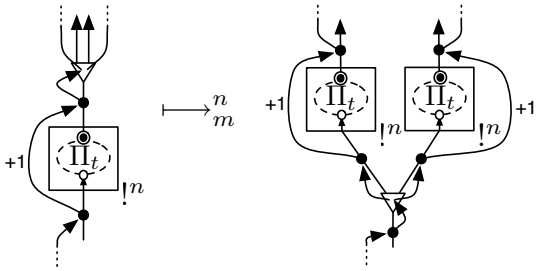


Figure 6.4.: How the multiplexer tree changes during a multiplexer redex $\longmapsto^n_m$.

Lemma 6.38. *The weight of a multiplexer decreases in a* $\longmapsto^n_m$*-cut.*

Proof. The multiplexer tree is transformed as shown in Figure 6.4. The node of the multiplexer moves down and the box moves up by duplicating it. The weight of the multiplexer decreases by one. □

> *Remark* 6.39. The weights of the multiplexers possibly further up in the multiplexer tree in Figure 6.4 do not change because only one of the duplicated boxes contributes to those weights (because the multiplexer tree has no loop).

When reducing a shifting-cut, two boxes will be merged. The multiplexer-cuts duplicate, i.e. create new boxes. Naturally the question arises how big boxes can become during normalisation of both kinds of cuts.

Figure 6.4 and 6.5 show how shifting and multiplexer redexes influence the multiplexer trees *only locally*. The shape of the multiplexer tree essentially stays the same (i.e. the branching structure). Boxes are moved through the tree, away from the root. Hence, only boxes on the same

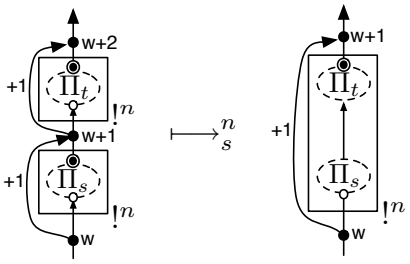


Figure 6.5.: How the multiplexer tree changes during a shifting redex $\longmapsto_s^n$.

branch of the tree will ever have the chance to be merged by shifting redexes. This gives raise the following lemma about the maximal box sizes during $\longmapsto_p^n$-normalisation:

Lemma 6.40. *For $\Pi(\longrightarrow_p^n)^*\Pi'$ and all proof nets Π_t of nested $!^n$-boxes in Π'*

$$|\Pi_t|_k \leq |\Pi|_k$$

holds. Moreover, with $|\Pi_t|_{>n} := \sum_{k=n+1}^{\infty} |\Pi_t|_k$ this implies $|\Pi_t|_{>n} \leq |\Pi|_{>n}$.

Intuitively, this lemma says that copies of boxes due to a $\longmapsto_m^n$-redex will not be able to merge later. This is the key for the PTime normalisation of $\mathrm{LLT}_!$:

Lemma 6.41. *Let Π^τ be a proof net which is $\longmapsto^i$-normal for all $i < n$ and moreover $\longmapsto_l^n$- and $\longmapsto_{i,K_n}^n$-normal. Then $\Pi \xrightarrow{nf}{}_p^n \Pi'$ with $|\Pi'| \in \mathcal{O}(|\Pi|^4)$ in $\mathcal{O}(|\Pi|^3)$ many $\longmapsto_p^n$-steps.*

Proof. **Case** $\Pi \longrightarrow_s^n \Pi'$ (compare Figure 6.2b): Clearly, $|\Pi|_{\leq n}$ decreases by at least one because of the dropped box. Meanwhile, all other components of $\nu^n(\Pi)$ stay the same or decrease as well, i.e. $\nu^n(t') \prec \nu^n(t)$.

Case $\Pi \longrightarrow_m^n \Pi'$ (compare Figure 6.2c) for a multiplexer node s in Π: Then $|\Pi'|_{\leq n}$ increases (by 1 for the extra $!^n$-box) and $d(t)$ increases by the size of the contents of the box. But

$$\nu^n(\Pi') = \left\langle \ldots, c_w^n(\Pi) - 1, c_{w-1}^n(\Pi) + 1, \ldots, \left|\Pi'\right|_{\leq n}, \left|\Pi'\right|_{>n} \right\rangle$$

$$\prec \left\langle \ldots, c^n_w(\Pi), c^n_{w-1}(\Pi), \ldots, |\Pi|_{\leq n}, |\Pi|_{>n} \right\rangle = \nu^n(\Pi)$$

for the weight $w := w^n_\Pi(s)$ independently of the increase of $|\Pi'|_{\leq n}$ and $|\Pi'|_{>n}$, due to the lexicographic ordering.

Now, consider $\Pi\,(\longrightarrow^n_s \overset{\text{nf}}{\cup} \longrightarrow^n_m)\,\Pi'$ and let be $\nu^n(\Pi) = \langle c_W, c_{W-1}, \ldots, c_0, b, d\rangle$. How many m-redexes in Π can be reduced while s-redexes are only reduced if no m-redex is left anymore? The following values are upper bounds for the measure when reducing the m-redexes of highest weight:

- At the beginning with $d := |\Pi|_{>n}$:

  $$\nu^n(\Pi) = \langle c_W, c_{W-1}, \ldots, c_0, b, d\rangle.$$

  By Lemma 6.40 d is an upper bound of the $!^n$-box-sizes in all of the following $\longmapsto^n_s$- and $\longrightarrow^n_m$-normalisations.

- After one m-redex:

  $$\langle c_W - 1, c_{W-1} + 1, \ldots, c_0, b + 1, d + d\rangle.$$

  The $b + 1$ comes from the extra $!^n$-box. By the restrictions on proof nets of boxes, the duplicated $!^n$-boxes do not contain multiplexers of level $\leq n$, any $!^n$-box or any other node of level $\leq n$, which could contribute to b.

- After c_W m-redexes:

  $$\langle 0, c_{W-1} + c_W, \ldots, c_0, b + c_W, c_W \cdot d + d\rangle.$$

- After $c_W + 1$ m-redexes:

  $$\langle 0, c_{W-1} + c_W - 1, \ldots, c_0, b + c_W + 1, (c_W + 1) \cdot d + d\rangle.$$

  Note here that the last component only increases by d because every $!^n$-box is smaller than d by the very choice of d as an upper bound.

- After $c_W + (c_{W-1} + c_W)$ m-redexes:

  $$\langle 0, 0, c_{W-2}+c_{W-1}+c_W, \ldots, c_0, b+c_W+c_{W-1}+c_W, c_W \cdot d+(c_{W-1}+c_W)\cdot d+d\rangle.$$

- After $\sum_{i=k}^{W}(i-k+1)\cdot c_i$ m-redexes:

$$\big\langle \underbrace{0,\dots,0}_{W-k+1}, c_{k-1}+\sum_{i=k}^{W} c_i, \dots, c_0, b+\sum_{i=k}^{W}(i-k+1)\cdot c_i, d\cdot\sum_{i=k}^{W}(i-k+1)\cdot c_i+d\big\rangle.$$

- After $\sum_{i=0}^{W}(i+1)\cdot c_i$ m-redexes:

$$\big\langle \underbrace{0,\dots,0}_{W+1}, b+\sum_{i=0}^{W}(i+1)\cdot c_i, d\cdot\sum_{i=0}^{W}(i+1)\cdot c_i+d\big\rangle.$$

Hence, at this point, less than $b+\sum_{i=0}^{W}(i+1)\cdot c_i$ s-redexes might be left. Altogether this gives an upper bound of

$$\begin{aligned}
&\sum_{i=0}^{W}(i+1)\cdot c_i + b + \sum_{i=0}^{W}(i+1)\cdot c_i \\
&\leq |\Pi| + 2\sum_{i=0}^{W}(i+1)\cdot|\Pi| = |\Pi|\cdot(1+2\sum_{i=1}^{W+1} i) \\
&\leq |\Pi|\,(1+(|\Pi|+1)\cdot(|\Pi|+2) \in \mathcal{O}(|\Pi|^3)
\end{aligned}$$

many s- and m-redexes on level n.

During the reduction of s-redexes, the number of (nested) nodes of level n in the proof net decreases (one $!^n$-box less). During the reduction of m-redexes, it increases by one. But by the analysis above, this increase is polynomial:

$$\begin{aligned}
\big|\Pi'\big| &= \big|\Pi'\big|_{\leq n} + \big|\Pi'\big|_{>n} \\
&\leq b+\sum_{i=0}^{W}(i+1)\cdot c_i + d\cdot\sum_{i=0}^{W}(i+1)\cdot c_i + d \\
&\in \mathcal{O}(|\Pi|^3)+\mathcal{O}(|\Pi|^4) = \mathcal{O}(|\Pi|^4).
\end{aligned}$$

□

The considered normalisation strategy with **c**ase **d**istinction **u**nfolding $\longrightarrow_{\mathrm{cdu}}$ can be expressed as

$$\longrightarrow_{\mathrm{cdu}} := \prod_{n=0}^{N}(\xrightarrow{\mathrm{nf}}{}^{n}_{l}\ \xrightarrow{\mathrm{nf}}{}^{n}_{i,K_n}\ \xrightarrow{\mathrm{nf}}{}^{n}_{l}\ \xrightarrow{\mathrm{nf}}{}^{n}_{p}), \tag{6.10}$$

where N denotes the highest level of types appearing in the proof net. The K_n will be chosen in the following theorem:

Theorem 6.42 (Correctness)**.** *There is a polynomial $P_N(x)$, such that every proof net Π^τ with levels $\leq N$ can be normalised in $P_N(|\Pi|)$ many $\longrightarrow^n$-steps of $\longrightarrow_{cdu}$.*

Proof. The reductions of $\longmapsto^n_l$-redexes decrease the size $|\Pi|$ due to the linearity of the term system and the weight of the constants, which makes the left hand sides of the reduction rules heavier than the right hand sides, for instance:

$$|(\lambda x^\sigma.t^\tau\ s)| = 1 + 1 + |t| + |s| > |t| + |s| > |t[x := s]|$$

or

$$|(\mathbf{Case}^n_\tau\ \mathbf{tt}^n)| = 1 + 1 + 1 = 3 > 2 = |\lambda x \lambda y.x|\,,$$

and similarly for the other linear cuts.

After the linear reductions, the iterations are to be unfolded. At this point, we finally can fix K_n which was left open in $\longmapsto^n_{i,K_n}$ up to now. $L^{n+1}(\tau)$-lists will not get longer than the number of $\mathbf{cons}^n_\tau$-constants which can appear in the term (after possible duplication by the multiplexers). Hence, we choose and *fix* K_n now as the sum of the number of $\mathbf{cons}^n_{\tau'}$ (for any type τ') and $\overset{n}{\lhd}$ nested nodes in Π (i.e. *before* any $\longmapsto^n_{i,K_n}$-redex is fired).

With the chosen K_n, each $\longmapsto^n_{i,K_n}$-redex will, via $\mathrm{CaseIt}^n_{l,K_n}$, introduce several $\mathbf{Case}^{n+1}_{\tau,\sigma}$-constants and $\overset{n}{\lhd}$-nodes in order to get the K_n copies of the step term. For every $\mathbf{cons}^n_\tau$ and every $\overset{n}{\lhd}$ in the proof net Π (i.e. before any $\longmapsto^n_{i,K_n}$-redex was fired), we get another $\overset{n}{\lhd}$ as part of every $\mathrm{CaseIt}^n_{l,K_n}$. With every $\longmapsto^n_{i,K_n}$-redex, we have one $\mathbf{It}^n_{\tau,\sigma}$-node less in the proof net. Both, the number of $\mathbf{It}^n_{\tau,\sigma}$-constants in Π and K_n is bounded by $|\Pi|$. Hence, firing all $\longmapsto^n_{i,K_n}$-redexes gives us a quadratic growth of the proof net size[1].

After the iteration part the new linear redexes of $\mathrm{CaseIt}^n_{l,K_n}$ are normalised with another $\xrightarrow{\mathrm{nf}}{}^n_l$ which decreases the term size. Hence, up to

[1] Note that these bounds are far from optimal in the sense that they can be improved from quadratic to linear by a more careful (and much more involved) analysis of which $\mathbf{cons}^n_\tau$ will interact with which $\mathbf{It}^\sigma_{n,\tau}$. But for our purpose to get just the correctness result, we will stick to this simpler approach.

this point, after $\xrightarrow{\mathrm{nf}}{}^{n}_{l}$ $\xrightarrow{\mathrm{nf}}{}^{n}_{i,K_n}$ $\xrightarrow{\mathrm{nf}}{}^{n}_{l}$, the number of reduction steps and the term size is bounded by $\mathcal{O}(|\Pi|^2)$. With Lemma 6.41 we complete the normalisation with the $\xrightarrow{\mathrm{nf}}{}^{n}_{p}$-redexes, giving a complexity of $\mathcal{O}(|\Pi|^{2\cdot 3})$ to normalise level n altogether. Moreover, the resulting term size will be in $\mathcal{O}(|\Pi|^{2\cdot 4})$ due to Lemma 6.41 and the analysis above for the linear and iteration part of the reduction..

Now consider the normalisation for all levels via $\longrightarrow_{\mathrm{cdu}}$, starting with $\Pi_0 := \Pi$. Every reduction of one level n, starting with Π_n and leading to Π_{n+1}, consists of $\mathcal{O}(|\Pi_n|^{2\cdot 3})$ many reductions of single redexes. Then Π_{n+1} is the start proof net of the reductions of level $n{+}1$ and therefore we have also $|\Pi_{n+1}| \in \mathcal{O}(|\Pi_n|^{2\cdot 4})$. Hence, we get the following upper bound of reduction steps for all levels summed up:

$$\sum_{n=0}^{N} \mathcal{O}(|\Pi_n|^{2\cdot 3}) \leq \sum_{n=0}^{N} \mathcal{O}(|\Pi|^{(2\cdot 4)^{n+1}}) = \mathcal{O}(|\Pi|^{(2\cdot 4)^{N+1}}).$$ □

6.5. Types of Constants

The constants of LLT! are introduced in Section 6.2 without giving a motivation or reasoning about their types. In the following we discuss why the very choice of levels and types is essential. This should give a better understanding of the construction of the LLT!-calculus.

The central constraint driving the typing of constants is the compatibility with normalisation by levels:

Redexes on level n for a certain constant *must not block* redexes of lower levels.

But before looking at the constant types, we need to know the level that should be assigned to an arrow type.

6.5.1. Arrow

It is not obvious that the minimum of the levels of the left and right type in $\sigma \multimap \tau$ is the right choice in Definition 6.4. Hence, we go through the other alternatives in order to see their influence on the calculus.

Given the levels $\ell(\sigma)$ and $\ell(\tau)$, the first case is that both levels are equal. Then the natural choice is to use $\ell(\sigma) = \ell(\tau) = \ell(\sigma \multimap \tau)$.

Otherwise, we have the following cases:

1. $\ell(\sigma) > \ell(\tau)$: Assume $\ell(\sigma \multimap \tau) = \ell(\sigma)$ and consider the term

$$t := \left(\pi^{\ell(\tau)}_{B^{\ell(\tau)}} \left(\lambda x^{\sigma}.\left(\otimes^{\ell(\tau)}_{B^{\ell(\tau)},B^{\ell(\tau)}} y\, z\right)\, s^{\sigma}\right)\right).$$

   The beta redex for λx^{σ} is of level $\ell(\sigma) > \ell(\tau)$ and therefore not fired before level $\ell(\sigma)$ in the normalisation by levels. Hence, in Π_t this redex blocks the projection redex for $\pi^{\ell(\tau)}_{B^{\ell(\tau)}}$, which is on level $\ell(\tau)$, i.e. below $\ell(\sigma)$. Therefore, $\ell(\sigma \multimap \tau) = \ell(\sigma)$ does not satisfy the requirement not to block redexes.
   This counter-example does not work if we choose $\ell(\sigma \multimap \tau) = \ell(\tau)$ instead, because this forces us to fire the beta redex early, in any case before any redex of the body of the abstraction might be a candidate for reduction.

2. $\ell(\sigma) < \ell(\tau)$: Assume $\ell(\sigma \multimap \tau) = \ell(\tau)$ and consider the term

$$t := \left(\lambda x^{\sigma}.\left(\mathbf{Case}^{\ell(\sigma)}\, x^{\sigma}\, y\, z\right)\, s^{\sigma}\right)$$

   with $\sigma = B^{\ell(\sigma)}$. Then the beta redex for λx^{σ} is fired in level $\ell(\sigma \multimap \tau) = \ell(\tau) > \ell(\sigma)$ in Π_t. The inner case distinction redex though (of level $\ell(\sigma)$) is blocked by this. Hence, this choice is not a good one either.
   Again by choosing $\ell(\sigma \multimap \tau) = \ell(\sigma)$ this case cannot happen anymore because it forces the beta reduction to be executed as early as possible.

Hence, we have chosen $\ell(\sigma \multimap \tau) = \min\{\ell(\sigma), \ell(\tau)\}$ in Definition 6.4. With this plus a similar consideration for the product $\sigma \otimes \tau$ we are able to get the following subtype property:

Lemma 6.43. *For every type δ appearing in ρ (negatively or positively) $\ell(\delta) \geq \ell(\rho)$ holds.*

Proof. Induction on the type ρ. □

This lemma is crucial in order to know that the arguments, which are applied to a constant, are of a higher or an equal level as the constant itself.

6.5.2. Case Distinction

The first constant we look at is the case distinction on boolean values:

$$\mathbf{Case}^n_\sigma : B^n \multimap \sigma \multimap \sigma \multimap \sigma \text{ with } \ell(\sigma) \geq n$$

with the following two redexes:

$$(\mathbf{Case}^n_\sigma\, \mathbf{tt}^n) \longmapsto^n_l \lambda x^\sigma \lambda y^\sigma . x$$
$$(\mathbf{Case}^n_\sigma\, \mathbf{ff}^n) \longmapsto^n_l \lambda x^\sigma \lambda y^\sigma . y.$$

The boolean parameter decides which alternative is taken. This redex is of level n and therefore is not fired earlier in the normalisation by levels. Imagine we chose $\ell(\sigma) = n - 1 < n$, e.g. $\sigma = B^{n-1} \otimes B^{n-1}$. Then take the term

$$\left(\pi^{n-1}_\sigma \left(\mathbf{Case}^n_\sigma\, \mathbf{tt} \left(\otimes^{n-1}_{B^{n-1},B^{n-1}} x\, y\right) \left(\otimes^{n-1}_{B^{n-1},B^{n-1}} x'\, y'\right)\right)\right).$$

Clearly, the case distinction blocks the outer projection redex, which is on level $n-1$. The very same situation happens with the list case distinction, i.e.

$$\mathbf{Case}^n_{\tau,\sigma} : L^n(\tau) \multimap (\tau \multimap L^n(\tau) \multimap \sigma) \multimap \sigma \multimap \sigma \text{ with } \ell(\sigma) \geq n$$
$$\mathbf{cons}^n_\tau : !^n(\tau \multimap L^{n+1}(\tau) \multimap L^{n+1}(\tau))$$
$$\mathbf{nil}^n_\tau : L^{n+1}(\tau).$$

In general, the effect of the side condition $\ell(\sigma) \geq n$ is that it forces the level of the constant's type not to be smaller than $\ell(B^n) = n$ (by applying Lemma 6.43). This is the theme to determine the needed side conditions of all the constants.

6.5.3. Product

$$\otimes^n_{\tau,\rho} : \tau \multimap \rho \multimap \tau \otimes \rho \text{ with } \ell(\tau \otimes \sigma) = n$$
$$\pi^n_\sigma : \tau \otimes \rho \multimap (\tau \multimap \rho \multimap \sigma) \multimap \sigma \text{ with } \ell(\sigma) \geq \ell(\tau \otimes \rho).$$

We have $\ell(\tau \otimes \rho) = \min\{\ell(\tau), \ell(\rho)\}$ (by essentially the same argument as for the arrow in Section 6.5.1).

Now assume σ would be arbitrary, especially $\ell(\sigma) < \ell(\tau \otimes \rho)$. We can get the following case with $\sigma = \rho \multimap \delta$:

$$\left(\left(\lambda p^{\tau\otimes\rho}.\left(\pi^n_\sigma\, p\, \lambda l, r.\lambda x^\rho.l^\delta\right)\ \left(\otimes^n_{\tau,\rho}\, l\, r\right)\right)\ t\right).$$

The λp is of level $\ell(\tau\otimes\rho) > \ell(\sigma)$. The inner abstraction $\lambda x.l$ is of level $\ell(\sigma)$, but the beta-redex is blocked by the product calculations. The t can only be substituted into the x after the product introduction and elimination are reduced. Hence, we need the side condition to rule out this example.

Again, like for the case distinctions we make sure with the side condition that the redex is on the level of the constants involved, i.e.

$$\ell(\tau \multimap \rho \multimap \tau\otimes\rho) = n = \ell(\tau\otimes\rho \multimap (\tau\multimap\rho\multimap\sigma)\multimap\sigma).$$

6.5.4. Iteration

$$\mathbf{It}^n_{\tau,\sigma} : L^{n+1}(\tau) \multimap !^n(\tau\multimap\sigma\multimap\sigma)\multimap\sigma\multimap\sigma$$

The reduction for the iteration in Definition 6.31 leads to a reduct with new redexes on the same level:

$$\left(\mathbf{It}^n_{\tau,\sigma}\, l\, f\, g\right) \longmapsto^n_{i,K_n} \left(\left(f \overset{n}{\triangleleft}_{f'_1}^{f_1} \ldots \left(f'_{k-1} \overset{n}{\triangleleft}_{f'_{K_n}}^{f_{K_n}} \left(\mathrm{CaseIt}^n_{l,K_n}\, g\right)\right)\right)\ g\right).$$

In the type of $\mathbf{It}^n_{\tau,\sigma}$ no further side condition is needed because the restriction on $\ell(\sigma)$ and $\ell(\tau)$ is implicit by the side conditions in the definition of the types, leading to

$$\ell(\sigma), \ell(\tau) > \ell(L^{n+1}(\tau) \multimap !^n(\tau\multimap\sigma\multimap\sigma)\multimap\sigma\multimap\sigma) = n.$$

Level of the list At this point, we want to stress again the reason why $\mathbf{It}^n_{\tau,\sigma}$ expects a list of level $n+1$. While by this, the list type lives one level higher than the iterators and the iteration redex, the $\mathbf{cons}^n_\tau : !^n(\tau \multimap L^{n+1}(\tau)\multimap L^{n+1}(\tau))$ for that list type are on the level n. In the iteration reduction the $\mathbf{cons}^n_\tau$ is replaced with a copy of the step term. The step term, as seen in the type of $\mathbf{It}^n_{\tau,\sigma}$ is of a $!^n$-type and therefore lives on level n, in order to be duplicatable with the polynomial reductions.

This fits very well, in fact, to the origin of the $\mathbf{cons}^n_\tau$-constant in LAL, namely the (abstracted) step term of the result list which of course also has a !-type. There, the actual step terms, or more precisely the actual algorithm in the step, also lives on level the above, while the box around is one level below this.

6.6. Conclusion and Outlook

We introduced the calculus $\mathrm{LLT}_!$, a variant of System T with the !-modality and levels in the types. We used the *level* concept in order to implement the *stratification property* without the need of the §-modality. The calculus $\mathrm{LLT}_!$ is shown to be complete and correct for polynomial time.

The goal has been to create a type system with constants which allows typing of typical algorithms from Light Affine Logic with Church numerals in a very natural way in $\mathrm{LLT}_!$. The completeness proof shows how easy and direct the Turing Machine simulation can be implemented. In contrast to the one of Light Affine Logic [AR02], $\mathrm{LLT}_!$ hides a lot of technicalities which must be employed and understood in Light Affine Logic.

One focus, which shows up also in the Turing Machine simulation, has been the seamless integration of the pull-out trick (compare Examples 3.15 and 6.21) in order to archive symmetric types, such that algorithms can be iterated again if the trick can be applied. Especially, the types of the iteration- and the $\mathbf{cons}_\tau$-constants are motivated by this goal. Moreover, this is a key difference to a naive formulation of a System T variant which easily embeds into Light Affine Logic.

The Example 6.22 about the split function suggests though, that $\mathrm{LLT}_!$ *is in fact* more *than a simple definitional calculus* based on LAL, i.e. we *cannot* express every $\mathrm{LLT}_!$-constant as a LAL-term in such a way that we get a direct embedding from $\mathrm{LLT}_!$ into LAL (compare Remark 6.23).

Correctness and unnatural unfolding The correctness proof is very similar to that of [AR02]. The main difference is the unfolding of the iteration constant. The Church numerals of Light Affine Logic make a natural distinction between the duplication of the step term and the actual computations of the step terms. In the normalisation by levels strategy these two steps are clearly separated. In the setting of System T though, this "normalisation by case distinction unfolding" is very unnatural. Therefore, in this chapter it is left open how to switch to a more natural reduction rule of iteration in System T. Chapter 7 though will solve this issue nicely by lifting the iteration constant one level higher.

Going back to the §-modality? We motivated the use of *levels in types* in Section 6.1.2 by arguing that the level is a very similar concept to the §-modality. The choice, whether to stick to levels or §, might be a question of

taste: the § can be seen as a way to implement the stratification property, which is – otherwise – only disturbing in a term. Clearly, it can be argued pro and contra this design decision. It should be possible tough to reformulate $\mathrm{LLT}_!$ without any levels in the types, i.e. using traditional §-boxes and the §-modality like in Light Affine Logic. A sketch of such a system $\mathrm{LLT}_{!\S}$ is shown in Section A.1 in the appendix. The central point, also in such a setting, is the type of the iteration- and the $\mathbf{cons}_\tau$-constant. It allows the formulation of algorithms using the pull-out trick.

7

From LFPL and Light Linear T to Light LFPL

In this chapter we first sketch a step-by-step transition of $\mathrm{LLT}_!$ to a light stratified variant of Hofmann's LFPL which provides the impredicative iteration on each level, *and* which allows a direct embedding of $\mathrm{LLT}_!$ terms as well. The developed ideas are then made formal in the *Light Linear Functional Programming Language* ($\mathrm{LLFPL}_!$), which is proved to be normalisable in polynomial time.

The system $\mathrm{LLT}_!$, as presented up to now in Chapter 6, is very much designed after the normalisation of Light Affine Logic. This is intentional of course because LAL gives the intuition, although this leads to a non-standard unfolding of the iteration constant. The main issue is that $\mathbf{It}^{\sigma}_{n,\tau}$ is a constant on level n, and hence the normalisation must take place on the very same level, although the lists are a level higher on $n+1$. The goal of this chapter is to lift this iteration constant to level $n+1$ as well, in order to allow us to use a standard iteration normalisation, and also to allow us to extend the system by *impredicative iteration*. This will lead to $\mathrm{LLFPL}_!$, which subsumes both iteration schemes, and which solves the problem of $\mathrm{LLT}_!$ to have non-standard reduction rules for the iteration.

Structure of this chapter This chapter starts in with a number of sketches of variations of $\mathrm{LLT}_!$ which approach – step by step – the goal of this chapter: the introduction of Light LFPL: Section 7.1 splits the role of the $\mathbf{cons}^n_\tau$-constant in $\mathrm{LLT}_!$ into a flatly typed $\mathbf{cons}^n_\tau$-constant and a diamond

$\mathbf{d}^n :!^n\diamondsuit^{n+1}$ in $\mathrm{LLT}_{!\diamondsuit}$. Section 7.2 adds a $\diamondsuit^{n+1}$-argument to the iteration steps in the style of LFPL's iteration scheme. Section 7.3 – the core of the chapter – first motivates the necessary lifting of the iteration into the level of the lists. In Subsection 7.3.1 a preliminary type system for a light LFPL with a closed arrow $\multimap_c$ is sketched, before transforming this idea into special iteration typing rules $(L^n(\tau)_0^-)$ and $(L^n(\tau)_1^-)$ in Subsection 7.3.2. Subsection 7.3.3 introduces $\mathrm{LLFPL}_!$ formally by giving type system for terms and a proof net calculus, which is used for normalisation in Subsection 7.3.4. Subsection 7.3.5 finally combines the complexity proofs of LFPL and $\mathrm{LLT}_!$ to show that every (closed) $\mathrm{LLFPL}_!$-proof net can be normalised in polynomial time. The chapter ends in Section 7.4 with a conclusion of the result, the bi-product of an improved $\mathrm{LLT}_!$-iteration reduction rule, and an outlook of applying the ideas to a LAL like system and the conversion into a more traditional type system using the §-modality.

7.1. From $\mathsf{LLT}_!$ to $\mathsf{LLT}_{!\diamondsuit}$

We start with $\mathrm{LLT}_!$, but replace the $\mathbf{cons}_\tau^n$- and $\mathbf{nil}_\tau^n$-constants by

$$\begin{aligned}\mathbf{cons}_\tau^{n+1} &: \diamondsuit^{n+1} \multimap \tau \multimap L^{n+1}(\tau) \multimap L^{n+1}(\tau)\\ \mathbf{nil}_\tau^{n+1} &: L^{n+1}(\tau)\end{aligned}$$

using a new, LFPL inspired, type $\diamondsuit^n$, the diamond, and a corresponding constant

$$\mathbf{d}^n :!^n\diamondsuit^{n+1}.$$

In order to construct a list of level $n+1$ one now needs a $\diamondsuit^{n+1}$-term. Every closed term of this type has to contain the $\mathbf{d}^n$ constant which is of level n, for instance $]\mathbf{d}^n[_{!^n}$.

The iteration constant stays the same:

$$\mathbf{It}_{\tau,\sigma}^n : L^{n+1}(\tau) \multimap !^n(\tau \multimap \sigma \multimap \sigma) \multimap \sigma \multimap \sigma.$$

The reduction rules of the $\mathbf{Case}_\sigma^{n,\tau}$ constant must be changed in the obvious way:

$$\begin{array}{rcl}\left(\mathbf{Case}_{\tau,\sigma}^n\ \mathbf{nil}_\tau^n\right) & \longmapsto_l^n & \lambda f\lambda g.g\\ \left(\mathbf{Case}_{\tau,\sigma}^n\ \left(\mathbf{cons}_\tau^n\ t^{\diamondsuit^n}\ x\ l\right)\right) & \longmapsto_l^n & \lambda f\lambda g.\,(f\ t\ x\ l)\end{array} \tag{7.1}$$

and similarly the iteration unfolding, which has to choose an appropriate K_n as before. Instead of counting the multiplexers and the $\mathbf{cons}_\tau^n$-constants,

we now count the $\trianglelefteq$-maximal $\diamondsuit^{n+1}$-subnets (compare Definition 4.38 for δLFPL). For closed terms this is bounded by the number of multiplexers of level n and $\mathbf{d}^n$ constants.

Remark 7.1. In Equation 7.1 the $\diamondsuit^n$-term t is applied to f. The step term of the iteration though *does not* have an $\diamondsuit^{n+1}$-argument. This is important because otherwise there would be two ways to inject a $\diamondsuit^{n+1}$-term into a $!^n$-box, which would allow typing the "double" function in a symmetric way.

Intuition Compare this new system using $\mathbf{d}^n$ and the flat $\mathbf{cons}_\tau^{n+1}$ with the LLT!-system, which only has $\mathbf{cons}_\tau^n$. Essentially we have split $\mathbf{cons}_\tau^n$ of LLT! into two constants in LLT!◇: the $\mathbf{d}^n$ with its type $!^n\diamondsuit^{n+1}$ makes sure that only one instance can be "injected" into a $!^n$-box. This was the original motivation of the type of $\mathbf{cons}_\tau^n$ in LLT!. With $\mathbf{d}^n$ taking over this functionality, the remaining $\mathbf{cons}_\tau^{n+1}$-constant in LLT!◇ can be typed with a very usual type, like in LFPL, without mentioning the !-modality.

Without giving a technical, formal embedding of LLT! into LLT!◇, it should be clear that we did not lose any expressivity. Translating a LLT!-term into a LLT!◇-term is possible by "replacing" the $\mathbf{cons}_\tau^n$-constants with $\mathbf{d}^n$ and by inserting the "new" $\mathbf{cons}_\tau^{n+1}$ at the places where the $\diamondsuit^{n+1}$-term is now available to construct a list.

Example 7.2 (Arithmetic in LLT!◇)**.** The usual polynomials for unary numbers (compare with the LLT! term in Example 6.24):

$$\begin{aligned}
\mathrm{add}_n &:= \lambda x^{L^{n+1}(B)}, y^{L^{n+1}(B)}.\\
&\qquad \left(\mathbf{It}^n_{B,L^{n+1}(B)}\, x\, !^n \boxed{\lambda b \lambda p.\, (\mathbf{cons}_B^{n+1}\ ]\mathbf{d}^n[_{!^n}\ b\, p)}\, y\right)\\
\mathrm{mult}_n &:= \lambda x^{L^{n+1}(B)}, y^{!^n L^{n+2}(B)}.\\
&\qquad \left(\mathbf{It}^n_{B,L^{n+2}(B)}\, x\, !^n \boxed{\lambda b, p.\, (\mathrm{add}_{n+1}\ ]y[_{!^n}\ p)}\, \mathbf{nil}_B^{n+1}\right).
\end{aligned}$$

Obviously, only one $\mathbf{cons}_B^{n+1}$ can be effectively used in the step term of add_n, because each further $\mathbf{cons}_B^{n+1}$ would need another $\diamondsuit^{n+1}$-subterm. The only way to get this though is by using an instance of $\mathbf{d}^n$ and

another $]\cdot[_{!^n}$-subterm to get it into the $!^n$-box of the step. But due to the restriction of the $(!_1)$-rule, this is impossible a second time.

7.2. From $\mathsf{LLT}_{!\diamondsuit}$ to $\mathsf{LLT}'_{!\diamondsuit}$

The purpose of the $!^n\diamondsuit^{n+1}$ typing of the $\mathbf{d}^n$-constant in $\mathrm{LLT}_{!\diamondsuit}$ is to control the number of "injected" $\diamondsuit^{n+1}$-terms in step terms of level n, based on the "one-hole" restriction of the $(!_1)$-rule. Inspired by LFPL's iteration constant, we can enforce the same by changing the types of the step terms to have an additional $\diamondsuit^{n+1}$-argument (to inject the $\diamondsuit^{n+1}$-term), i.e.

$$\mathbf{It}^n_{\tau,\sigma} \quad : \quad L^{n+1}(\tau) \multimap !^n(\diamondsuit^{n+1} \multimap \tau \multimap \sigma \multimap \sigma) \multimap \sigma \multimap \sigma,$$

and by forbidding the $\mathbf{d}^{n+1} : \diamondsuit^{n+1}$ constant in a $!^n$-box altogether. Note the changed type of $\mathbf{d}^n$ and the therefore different indices compared to $\mathrm{LLT}_{!\diamondsuit}$. This way, the step term can only use one $\diamondsuit^{n+1}$-term, the one given as an argument, to construct a list[1]. The constants must be adapted accordingly:

$$\begin{aligned}
\mathbf{Case}^n_{\tau,\sigma} &: L^n(\tau) \multimap (\diamondsuit^n \multimap \tau \multimap L^n(\tau) \multimap \sigma) \multimap \sigma \multimap \sigma \text{ with } \ell(\sigma) \geq n \\
\mathbf{cons}^n_\tau &: \diamondsuit^n \multimap \tau \multimap L^n(\tau) \multimap L^n(\tau) \\
\mathbf{nil}^n_\tau &: L^n(\tau).
\end{aligned}$$

The reduction rules therefore become

$$\begin{aligned}
\left(\mathbf{Case}^n_{\tau,\sigma}\ \mathbf{nil}^n_\tau\right) &\longmapsto^n_l \lambda f \lambda g.g \\
\left(\mathbf{Case}^n_{\tau,\sigma}\ \left(\mathbf{cons}^n_\tau\ t^{\diamondsuit^n}\ x\, l\right)\right) &\longmapsto^n_l \lambda f \lambda g.\,(f\, t\, x\, l)\,,
\end{aligned}$$

i.e. the case distinction does not forget about the $\diamondsuit^n$ term. The iteration unfolding is adapted accordingly.

It is clear again that the new system is at least as expressive as $\mathrm{LLT}_{!\diamondsuit}$, because the only difference is that now the $\diamondsuit^n$ term is not injected into a box via $]\cdot[_{!^n}$, but via the first argument of the step in analogy to LFPL.

[1] In fact, another possibility is to inject a $!^n\rho$ term, with $\diamondsuit^{n+1}$ positive in ρ, via the $(!_1)$-rule into the step term, e.g. $!^n\boxed{\lambda d \lambda x \lambda l. \ldots .]z[_{!^n}}$ with a free variable $z^{!^n(B \multimap \diamondsuit^{n+1})}$. But there is no closed term of type $!^n(B \multimap \diamondsuit^{n+1})$. Hence, the algorithms will not be a closed term at the end. We will only normalise algorithms without free variables, such that this term would be ruled out.

Remark 7.3. In fact, one does not need to use the "one-hole-per-box" possibility just to inject a $\Diamond^{n+1}$-subterm into a step term anymore in $LLT'_{!\Diamond}$. In other words, the $]\cdot[_{!^n}$-hole, which is allowed according to $(!_1)$, can now be used for other purposes. In $LLT_!$ (or $LLT_{!\Diamond}$) this one-time possibility per $!^n$-box was "used up" for the injection of $\mathbf{cons}^n_\tau$ (or $\mathbf{d}^n$) in every non-trivial step term. Hence, it seems that $LLT'_{!\Diamond}$ is even slightly more expressive than $LLT_!$ or $LLT_{!\Diamond}$.

Now let us see again the standard example:

Example 7.4 (Arithmetic in $LLT'_{!\Diamond}$)**.** The usual polynomials for unary numbers (compare with the $LLT_!$ term in Example 6.24):

$$\mathrm{add}_n := \lambda x^{L^{n+1}(B)}, y^{L^{n+1}(B)}.$$
$$\left(\mathbf{It}^n_{B,L^{n+1}(B)}\, x\, !^n \boxed{\lambda d^{\Diamond^{n+1}} \lambda b \lambda p.\, \left(\mathbf{cons}^{n+1}_B\, d\, b\, p\right)}\, y\right)$$
$$\mathrm{mult}_n := \lambda x^{L^{n+1}(B)}, y^{!^n L^{n+2}(B)}.$$
$$\left(\mathbf{It}^n_{B,L^{n+2}(B)}\, x\, !^n \boxed{\lambda d^{\Diamond^{n+1}} \lambda b, p.\, \left(\mathrm{add}_{n+1}\ ]y[_{!^n}\ p\right)}\, \mathbf{nil}^{n+1}_B\right).$$

Here, the step of add_n gets the $\Diamond^{n+1}$-term from the first argument d. The step can be typed via $(!_0)$ because there is no $]\cdot[_{!^n}$ necessary in the box. The d of the step of mult_n is not used. Of course, it would be possible to apply it to another $\mathbf{cons}^{n+1}_\tau$.

7.3. From $LLT'_{!\Diamond}$ to $LLFPL_!$

The reduction rules for the $\mathbf{Case}^{n+1}_{\tau,\sigma}$ constant in $LLT_{!\Diamond}$ and $LLT'_{!\Diamond}$ only involve terms of level $n+1$, i.e. terms on the same level as the list $L^{n+1}(\tau)$ and the $\mathbf{Case}^n_{\tau,\sigma}$-constant themselves. On the other hand, the corresponding iteration $\mathbf{It}^n_{\tau,\sigma}$-constant of these systems is still on the level n and the reduction rules of $\mathbf{It}^n_{\tau,\sigma}$ involve terms of both levels: $n+1$ *and* n. The final step in our transformation of $LLT_!$ into a light LFPL is to *lift the iteration constant* with case-distinction-unfolding to level $n+1$. This will allow us to use a more standard reduction purely on level $n+1$.

> *Remark* 7.5. For now we always speak about "light LFPL". The final LLFPL$_!$ calculus in Section 7.3.3 will need some more (technical) modifications which we will talk about later.

The central question now is the following:

Why is this lifting to level $n+1$ possible if the step terms are duplicated and therefore must be of a $!^n$-type on the level n below?

Intuition of practical closed terms The key idea to answer the question is to see the $!^n$ boxes as "practically closed terms". This means they are considered being closed up to the possibly existing $]\cdot[_{!^n}$-hole which can be neglected during normalisation.

This intuition is backed up by the typing rules $(!_0)$ and $(!_1)$. The former puts a box around a closed term on level $n+1$, and the later puts the box around a term which is closed up to possible free variables in the one and only $]\cdot[_{!^n}$-hole (with the content on level n). If we normalise by levels though, and are on level $n+1$ reducing an iteration redex, this $]\cdot[_{!^n}$ cannot contribute anymore anyway to the result. Hence, it makes sense to consider a $!^n$-box to be closed on level $n+1$ and higher in this moment of normalisation of the iteration on level $n+1$. Its free variables in the possible $]\cdot[_{!^n}$-hole do not matter anymore.

In the proof net setting, this intuition means that the step Π_t of the iteration, which is in a $!^n$-box, has possibly a proof net below, but otherwise no inputs. We can remove the box around the step and replace it with a $]\cdot[_{!^n}$-node with the former proof net below. Figure 7.1 shows the LLT$_!$ iteration constant next to a potential light LFPL iteration constant that we will use later. The Π_r net on the right will typically be a $!^n$-box. The actual step Π_s has all the properties of a proof net of a box (which in the LLT$_!$-system it really was). But instead the box is removed to make it possible to lift the @-node to level $n+1$.

Relation to LFPL Closed step terms remind a lot of Hofmann's LFPL iteration. There, the steps are duplicated during the iteration (in the words of LLT$_!$: on the same level as the lists), i.e. when an $\mathbf{It}_{\tau,\sigma}$ constant comes into contact with a $(\mathbf{cons}_\tau\, d\, x\, l)$ term. This can only work in a linear system if the steps are closed. But with the upper intuition of "practically closed

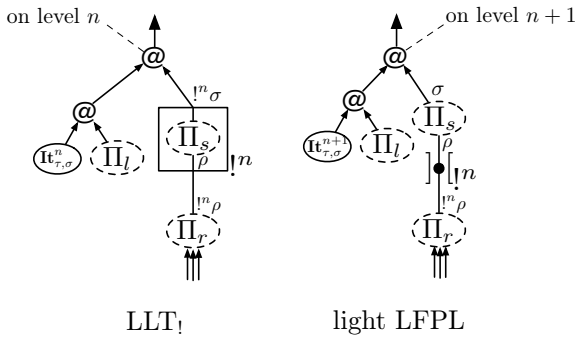


Figure 7.1.: On the left the iteration constant of level n in LLT$_!$ with a boxed step term; on the right the iteration of level $n+1$ in LLFPL$_!$ with a step term on level $n+1$, but with the properties of a term in a box.

terms" we have a very similar situation which leads us to a new iteration constant which has a type on level $n+1$ and its reductions can be done one level $n+1$ as well.

7.3.1. Towards a Light LFPL with $\mathbf{It}^{n+1}_{\tau,\sigma}$-Constant and $\multimap_c$

We will now motivate the technical changes which are necessary to turn $LLT'_{!\diamond}$ into a light LFPL. Note that *the syntax in this subsection will not be the final one* of LLFPL$_!$ yet. We will turn to a more LFPL-like syntax (like the one in Chapter 4 and in [AS00, AS02]) later in Section 7.3.2, in order to be in line with the complexity analysis of δLFPL in Chapter 4.

Lifting the iteration to level $n+1$ As described above, the possible $]s[_{!^n}$-hole in the step term has no influence anymore on the reduction of an iteration on level $n+1$. Therefore, we do not care about it in the following when reducing an iteration redex. Instead, we duplicate the step term and add a $\overset{n}{\lhd}$-multiplexer to duplicate the s in the $]s[_{!^n}$-hole.

$$\left(\mathbf{It}^{n+1}_{\tau,\sigma}\ \left(\mathbf{cons}^{n+1}_{\tau}\ t^{\diamond^{n+1}}\ x\, l\right)\ f[z :=]s[_{!^n}]\ g\right)$$

$$\longmapsto_i^n \left(s \overset{n\ s_1}{\lhd_{s_2}} \left(f[z :=]s_1[_{!^n}]\, t\, x\ \left(\mathbf{It}^{\sigma}_{n+1,\tau}\, l\, f[z :=]s_2[_{!^n}]\, g \right) \right) \right).$$

The reduction creates a multiplexer on level n and makes a copy of the term f which is closed up to $s_{1/2}$. The multiplexer is necessary because s could have free variables. Because the type system is linear, we cannot just copy s syntactically as well. Moreover, using a multiplexer is a constant size increase of the proof net, while duplication of the proof net would be linear in the worst case, for every reduction step!

Doing light iteration What is the type of $\mathbf{It}^{n+1}_{\tau,\sigma}$ and how is the application

$$\left(\mathbf{It}^{n+1}_{\tau,\sigma}\, l\, f[z :=]s[_{!^n}]\right)$$

typed? The step term type should be on level $n+1$ as well (and not on level n), because otherwise the iteration constant would not be on level $n+1$. Hence, a $!^n$-banged type is not possible. But on the other hand, we have to allow light iteration like in $\mathrm{LLT}'_{!\diamond}$. For this we can introduce another arrow $\multimap_c$ into the type system with the following additional rules:

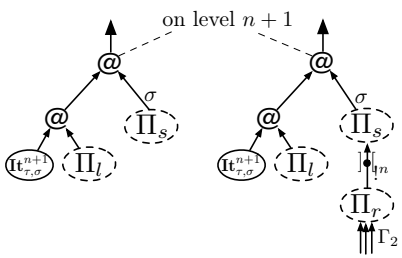
$$\frac{\Gamma \vdash t^{\sigma \multimap_c \tau} \quad \emptyset \vdash s^{\sigma}}{\Gamma \vdash (t\, s)^{\tau}}\ (\multimap^-_{c,0}) \qquad \frac{\Gamma_1 \vdash t^{\sigma \multimap_c \tau} \quad x^{\rho} \vdash s^{\sigma} \quad \Gamma_2 \vdash r^{!^n \rho}}{\Gamma_1, \Gamma_2 \vdash \left(t\, s[x :=]r[_{!^n}]\right)^{\tau}}\ (\multimap^-_{c,1})$$

with no subterm of level $< n+1$ in s, $x \in \mathrm{FV}(s)$ in $(\multimap^-_{c,1})$ and $n+1 = \ell(\sigma \multimap_c \tau)$, or more precisely the corresponding proof net constructions in Figure 7.2, again without (nested) nodes of level $< n+1$ in Π_s.

The resulting application $(t^{\sigma \multimap_c \tau}\, s^{\sigma})$, in fact, is on level $n+1$ now. The step term s is subject of the same restrictions as a term inside a $!^n$-box. In the left case Π_s is a closed step (which corresponds to the $(!_0)$-rule). In the right case Π_s can be a usual (possibly not-closed) step of an $\mathrm{LLT}'_{!\diamond}$-iteration (which corresponds to the $(!_1)$-rule), e.g. by putting the $]\cdot[_{!^n}$ around the $!^n$-box for the step $!^n\boxed{\lambda d \lambda p.s}$ that one is used to in $\mathrm{LLT}'_{!\diamond}$:

$$\left(\mathbf{It}^{n+1}_{\tau,\sigma}\, l\ \right]!^n \boxed{\lambda d \lambda p.s} \Big[_{!^n}\Big).$$

Remark 7.6. These rules play nicely with subject reduction. For $(\multimap^-_{c,0})$ this is clear. For $(\multimap^-_{c,1})$ we have (with the terminology of Figure 7.2):



light LFPL with $\multimap_c$

Figure 7.2.: Two rules to type an $\multimap_c$-application: on the left with a closed Π_s; on the right with a Π_s which has exactly one free variable. In both cases there are no nodes of level $< n+1$ in Π_s.

- If Π_r is a box typed via $(!_0)$, the $]\cdot[_{!^n}$-node above will form a redex with that box. The reduct of this is a closed proof net: Π'_r (the proof net of the box Π_r) below Π_s, linked to Π_s. Hence, this can be typed with $(\multimap^-_{c,0})$ rule.
- If Π_r is a box which is typed via $(!_1)$, the $]\cdot[_{!^n}$-node will form a redex with that box again. Now Π'_r (the proof net of the box Π_r) and Π_s are linked as before, but now with the (one and only) input $]\cdot[_{!^n}$. This is illustrated in Figure 7.3. The reduct on the right side is typable with $(\multimap^-_{c,1})$.

Remark 7.7. The $(\multimap^-_{c,1})$-rule implicitly tells us the following about terms (and in analogy about proof nets):

- If $s[x :=]r[_!]$ is not closed, the free variables must be in r.
- The term $(t\, s)$ typed with $(\multimap^-_{c,1})$ for level $n+1$ cannot be inside another $!^n$-box, because $(!_0)$, $(!_1)$ do not allow $]r[_{!^n}$-subterms because r would be of level n. But this case is outlawed by the side condition of the $(!_0)$- and $(!_1)$-rules.

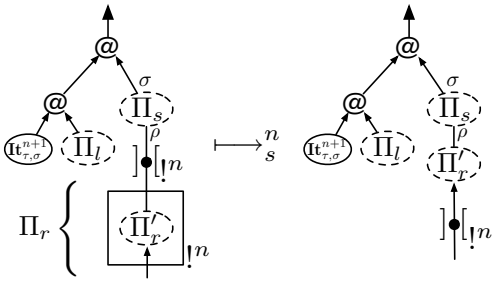


Figure 7.3.: A $(!_1)$-box is merging with a $(\multimap^-_{c,1})$-node.

Why is this important? It is crucial to keep subject reduction for the reduction rules of $\mathbf{It}^{n+1}_{\tau,\sigma}$. If the step term has a $]r[_{!^n}$-subterm ($\Pi_{r'}$-subnet) because the iteration is typed via $(\multimap^-_{c,1})$, the iteration reduction has to introduce copies of r (Π'_r) using $\overset{n}{\lhd}$ multiplexers. This would not be possible if there was another box around the iteration.

Moreover, we want that inside boxes there is "nothing" of level $\leq n$ (to be able to duplicate the box's contents in the $\overset{n}{\underset{m}{\longmapsto}}$-reduction, without creating new subterms of level $\leq n$).

Doing impredicative iteration As we have seen, the new arrow allows us to type the $\mathbf{It}^n_{\tau,\sigma}$-constant for $L^n(\tau)$-lists in a light LFPL with a level n type (instead of one level below in $\mathrm{LLT}_!$ or $\mathrm{LLT}'_{!\diamond}$):

$$\mathbf{It}^{\sigma}_{n,\tau} \quad : \quad L^n(\tau) \multimap (\diamond^n \multimap \tau \multimap \sigma \multimap \sigma) \multimap_c \sigma \multimap \sigma$$

with $\ell(\sigma), \ell(\tau) \geq n$. This is exactly the iteration constant of LFPL (if one only considers one level). Hence, such a light LFPL subsumes LFPL in a trivial way because it allows impredicative iteration. Step terms in LFPL are closed by definition, such that we can use $(\multimap^-_{c,0})$ to type the iteration.

Example 7.8 (Arithmetic with light impredicative iteration)**.** The usual polynomials for unary numbers (compare with the $\mathrm{LLT}_!$ term in Example 6.24):

$$\begin{aligned}
\mathrm{add}_n &:= \lambda x^{L^n(B)}, y^{L^n(B)}.\\
&\qquad \left(\mathbf{It}^n_{B,L^n(B)}\, x\, \lambda d^{\Diamond^n}\, \lambda b \lambda p.\, (\mathbf{cons}^n_B\, d\, b\, p)\ y\right)\\
\mathrm{mult}_n &:= \lambda x^{L^{n+1}(B)}, y^{!^n L^{n+2}(B)}.\\
&\qquad \left(\mathbf{It}^{n+1}_{B,L^{n+1}(B)}\, x\, \lambda d^{\Diamond^{n+1}}\, \lambda b \lambda p.\, \left(\mathrm{add}_{n+2}\ ]y[_{!^n}\ p\right)\ \mathbf{nil}^{n+2}_B\right).
\end{aligned}$$

Here, the step of add_n, as in Example 7.4 before, gets the $\Diamond^n$-term from the first argument d. Moreover, the step is closed. Hence, $(\multimap^-_{c,0})$ is used to type the application to the iteration constant.

In the second term mult_n the step is not closed, but has the shape $s[x :=]r[_{!^n}]^\sigma$ (with $r := y$), such that $(\multimap^-_{c,1})$ must be used.

Note that the multiplication is *not* non-size-increasing and therefore not typable in LFPL. Hence, the levels of a stratified, light LFPL are really necessary in this example.

We can also write the multiplication with a $!^n$-box for the step term. This underlines the similarity with mult_n of $\mathrm{LLT}'_{!\Diamond}$ in Example 7.4 even more. At the same time, it highlights the difference that we put a $]\cdot[_{!^n}$ around the box to lift it to level $n+1$:

$$\begin{aligned}
\mathrm{mult}_n &:= \lambda x^{L^{n+1}(B)}, y^{!^n L^{n+2}(B)}.\\
&\qquad \left(\mathbf{It}^{n+1}_{B,L^{n+1}(B)}\, x\ \right]!^n \boxed{\lambda d^{\Diamond^{n+1}}\, \lambda b \lambda p.\, \left(\mathrm{add}_{n+2}\ ]y[_{!^n}\ p\right)} \Big[_{!^n}\ \mathbf{nil}^{n+2}_B\right).
\end{aligned}$$

7.3.2. Iteration as a Term Construct

The new arrow $\multimap_c$ is only necessary to type the $\mathbf{It}^n_{\tau,\sigma}$-constant in the presented light variant of LFPL. We have not given an introduction rule for the $\multimap_c$-arrow. Therefore, we do not have λ-abstraction for $\multimap_c$.

In order to simplify our type system and to make it more similar to Hofmann's LFPL, we get rid of the $\multimap_c$ immediately again by replacing the $\mathbf{It}^n_{\tau,\sigma}$-constant with two applicative elimination rules for lists (compare Section 2.1.1.3 about Applicative System T):

$$\frac{\Gamma \vdash l^{L^{n+1}(\tau)} \quad \emptyset \vdash t^{\Diamond^{n+1} \multimap \tau \multimap \sigma \multimap \sigma}}{\Gamma \vdash (l\, t)^{\sigma \multimap \sigma}} \; (L^{n+1}(\tau)_0^-)$$

$$\frac{\Gamma_1 \vdash l^{L^{n+1}(\tau)} \quad x^{\rho} \vdash t^{\Diamond^{n+1} \multimap \tau \multimap \sigma \multimap \sigma} \quad \Gamma_2 \vdash r^{!^n \rho}}{\Gamma_1, \Gamma_2 \vdash \left(l\, t[x :=]r[_{!^n}]\right)^{\sigma \multimap \sigma}} \; (L^{n+1}(\tau)_1^-)$$

with no subterm of level $< n+1$ in t, $x \in \mathrm{FV}(t)$ in $(L^{n+1}(\tau)_1^-)$ and $n+1 \leq \ell(\sigma), \ell(\tau)$.

We rewrite our arithmetic example from above a last time to get the LLFPL$_!$ version:

Example 7.9 (Arithmetic with light impredicative iteration)**.** The usual polynomials for unary numbers (compare with the LLT$_!$ term in Example 6.24):

$$\begin{aligned}
\mathrm{add}_n &:= \lambda x^{L^n(B)}, y^{L^n(B)}. \left(x\, \lambda d^{\Diamond^n} \lambda b \lambda p. \left(\mathbf{cons}_B^n\, d\, b\, p\right)\, y\right) \\
\mathrm{mult}_n &:= \lambda x^{L^{n+1}(B)}, y^{!^n L^{n+2}(B)}. \\
&\qquad \left(x\, \lambda d^{\Diamond^{n+1}} \lambda b \lambda p. \left(\mathrm{add}_{n+2}\,]y[_{!^n}\, p\right)\, \mathbf{nil}_B^{n+2}\right).
\end{aligned}$$

The addition algorithm is (up to eta-expansion) the LFPL version of Section 3.3.2.2. The multiplication is essentially the LAL algorithm of Section 3.3.2.3, with the necessary modification to replace the §-modality with the levels of the setting here.

Note that the step terms do not have the $!^n$-box around. Though, again the restriction of the $(L^{n+1}(\tau)_0^-)$- and $(L^{n+1}(\tau)_1^-)$-rules make sure that we could put a box around like in the following alternative formulation:

$$\begin{aligned}
\mathrm{add}_n &:= \lambda x^{L^n(B)}, y^{L^n(B)}. \left(x \right]!^n \boxed{\lambda d^{\Diamond^n} \lambda b \lambda p. \left(\mathbf{cons}_B^n\, d\, b\, p\right)} \Big[_{!^n}\; y\right) \\
\mathrm{mult}_n &:= \lambda x^{L^{n+1}(B)}, y^{!^n L^{n+2}(B)}. \\
&\qquad \left(x \right]!^n \boxed{z =]y[_{!^n}\ \text{ in } \lambda d^{\Diamond^{n+1}} \lambda b \lambda p. \left(\mathrm{add}_{n+2}\, z\, p\right)} \Big[_{!^n}\; \mathbf{nil}_B^{n+2}\right).
\end{aligned}$$

Example 7.10 (Arithmetic and Numerals)**.** Now imagine mult_n is applied to actual numerals, e.g.

$$\left(\mathrm{mult}_0\ \left(\mathbf{cons}^1_B\,\mathbf{d}^1\,\mathbf{tt}\,\mathbf{nil}^1_B\right)\,!^0\boxed{\left(\mathbf{cons}^2_B\,\mathbf{d}^2\,\mathbf{tt}\,\mathbf{nil}^2_B\right)}\right).$$

First note that in the $!^0$-box we could not use $\mathbf{d}^1$. Hence, there is no closed (non-$\mathbf{nil}^1_\tau$) list of type $!^0L^1(\tau)$ in this system, such that on level n we can only duplicate lists of at least level $n+2$. In the example above, the second argument is of type $!^0L^2(B)$ for this very reason.

One can trace back this strange level distance of 2 in the following way: we lifted the iteration *and* the **cons**-constant one level higher to $n+1$, i.e. the $\mathbf{cons}^n_\tau$ of $\mathrm{LLT}_!$ is replaced by $\mathbf{cons}^{n+1}_\tau$ and $\mathbf{d}^{n+1}$ in light LFPL. Hence, the $\mathbf{d}^2$ above corresponds to $\mathbf{cons}^1_\tau$ in $\mathrm{LLT}_!$. In other words, using $\mathbf{d}^1$ and $\mathbf{cons}^1_\tau$ for $L^1(\tau)$ in the example above would correspond to a $!^0L^1(\tau)$ list in $\mathrm{LLT}_!$. But of this type there are no longer lists than length 1 in $\mathrm{LLT}_!$ either. Hence, also here we need at least level 2 to create a list of arbitrary length.

7.3.3. $\mathrm{LLFPL}_!$ Calculus – the Complete Picture

With the intuition in place we can now introduce $\mathrm{LLFPL}_!$ formally:

Definition 7.11 (Types)**.** The set $\mathrm{Ty}_{\mathrm{LLFPL}_!}$ of linear types and the level of a type $\ell(\tau) \in \mathbb{N}_0$ are defined inductively by:

$$\sigma, \tau ::= B^n \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid L^n(\sigma) \mid !^n\sigma \mid \Diamond^n$$

$$\ell(\rho) := \begin{cases} n \text{ if } \rho \in \{B^n,\, L^n(\sigma),\, !^n\sigma, \Diamond^n\} \\ \min\{\ell(\sigma), \ell(\tau)\} \text{ otherwise} \end{cases}$$

with the side condition $\ell(\sigma) \geq n$ for $L^n(\sigma)$ and $\ell(\sigma) > n$ for $!^n\sigma$.

Definition 7.12 (Constants)**.** The set $\mathrm{Cnst}_{\mathrm{LLFPL}_!}$ of $\mathrm{LLFPL}_!$ constants consists of

$\mathbf{tt}^n, \mathbf{ff}^n : B^n$

$$
\begin{aligned}
\mathbf{Case}^n_\sigma &: B^n \multimap \sigma \multimap \sigma \multimap \sigma \text{ with } \ell(\sigma) \geq n \\
\mathbf{Case}^n_{\tau,\sigma} &: L^n(\tau) \multimap (\diamondsuit^n \multimap \tau \multimap L^n(\tau) \multimap \sigma) \multimap \sigma \multimap \sigma \text{ with } \ell(\sigma) \geq n \\
\mathbf{cons}^n_\tau &: \diamondsuit^n \multimap \tau \multimap L^n(\tau) \multimap L^n(\tau) \\
\mathbf{nil}^n_\tau &: L^n(\tau) \\
\mathbf{d}^n &: \diamondsuit^n \\
\otimes^n_{\tau,\rho} &: \tau \multimap \rho \multimap \tau \otimes \rho \text{ with } \ell(\tau \otimes \rho) = n \\
\pi^n_\sigma &: \tau \otimes \rho \multimap (\tau \multimap \rho \multimap \sigma) \multimap \sigma \text{ with } \ell(\sigma) \geq \ell(\tau \otimes \rho) = n
\end{aligned}
$$

whose levels are the levels of their type.

7.3.3.1. Terms

Definition 7.13 (Terms)**.** For a countably infinite set V of variable names the set of (untyped) light linear terms $\mathrm{Tm}_{\mathrm{LLFPL}_!}$ is defined inductively by:

$$
s, t ::= x^\tau \mid c \mid \lambda x^\tau.t \mid (t\ s) \mid !^n\boxed{t} \mid !^n\boxed{x =]s[_{!^n}\ \text{in}\ t} \mid]t[_{!^n} \mid \left(s \lhd^{n\ x_1}_{x_2} t\right)
$$

with types $\tau \in \mathrm{Ty}_{\mathrm{LLFPL}_!}$, $c \in \mathrm{Cnst}_{\mathrm{LLFPL}_!}$, $n \in \mathbb{N}_0$ and $x, x_1, x_2 \in V$. Terms which are equal up to the naming of bound variables are identified.

Variables Free and bound variables are defined as for Linear System T in Definition 2.10 plus the cases

$$
\begin{aligned}
\mathrm{FV}(!^n\boxed{t}) &:= \mathrm{FV}(t) \\
\mathrm{FV}(!^n\boxed{x =]s[_{!^n}\ \text{in}\ t}) &:= \mathrm{FV}(s) \cup (\mathrm{FV}(t) \setminus \{x\}) \\
\mathrm{FV}(]t[_{!^n}) &:= \mathrm{FV}(t) \\
\mathrm{FV}(\left(s \lhd^{x_1}_{x_2} t\right)) &:= \mathrm{FV}(s) \cup (\mathrm{FV}(t) \setminus \{x_1, x_2\}).
\end{aligned}
$$

Subterms Subterms are defined as for Linear System T in Definition 2.10 with the following additional clauses:

$$
\begin{aligned}
t &\lhd_{\mathrm{LLFPL}_!} !^n\boxed{t} \\
s, t &\lhd_{\mathrm{LLFPL}_!} !^n\boxed{x =]s[_{!^n}\ \text{in}\ t}
\end{aligned}
$$

$$t \quad \lhd_{\mathrm{LLFPL}_!} \quad]t[_{!^n}$$
$$s, t \quad \lhd_{\mathrm{LLFPL}_!} \quad \left(s \overset{n\ x_1}{\lhd_{x_2}} t\right).$$

The level of a subterm is the level of its type.
We call $]s[_{!^n}$ a hole in the box $!^n\boxed{x =]s[_{!^n} \text{ in } t}$.

Notation 7.14. The same shorthand conventions as in notation 6.7 are used.

Definition 7.15 (Term typing rules)**.** A context is an (unordered) finite *multiset* of type assignments from the variable names V to types $\mathrm{Ty}_{\mathrm{LLFPL}_!}$ with the context condition that no variable is assigned different types at the same time.

Untyped terms are assigned types using the ternary relation $\vdash$ between a context Γ, a untyped term $t \in \mathrm{Tm}_{\mathrm{LLFPL}_!}$ and a type $\tau \in \mathrm{Ty}_{\mathrm{LLFPL}_!}$, denoted $\Gamma \vdash t^\tau$, via the following rules:

$$\frac{}{\Gamma, x^\tau \vdash x^\tau}\ (\mathrm{Var}) \qquad \frac{c \text{ constant of type } \tau}{\Gamma \vdash c^\tau}\ (\mathrm{Const})$$

$$\frac{\Gamma, x^\sigma \vdash t^\tau}{\Gamma \vdash (\lambda x^\sigma.t)^{\sigma \multimap \tau}}\ (\multimap^+) \qquad \frac{\Gamma_1 \vdash t^{\sigma \multimap \tau} \quad \Gamma_2 \vdash s^\sigma}{\Gamma_1, \Gamma_2 \vdash (t\,s)^\tau}\ (\multimap^-)$$

$$\frac{\emptyset \vdash t^\tau}{\emptyset \vdash !^n\boxed{t}^{!^n\tau}}\ (!_0) \qquad \frac{\Gamma \vdash t^{!^n\tau}}{\Gamma \vdash]t[^\tau_{!^n}}\ (]\cdot[_{!^n})$$

$$\frac{x^\sigma \vdash t^\tau \quad \Gamma \vdash s^{!^n\sigma}}{\Gamma \vdash !^n\boxed{x =]s[_{!^n} \text{ in } t}^{!^n\tau}}\ (!_1)$$

$$\frac{\Gamma_1, y_1^{!^n\sigma}, y_2^{!^n\sigma} \vdash t^\tau \quad \Gamma_2 \vdash s^{!^n\sigma}}{\Gamma_1, \Gamma_2 \vdash \left(s \overset{n\ y_1}{\lhd_{y_2}} t^\tau\right)^\tau}\ (\mathrm{C})$$

$$\frac{\Gamma \vdash l^{L^{n+1}(\tau)} \quad \emptyset \vdash t^{\Diamond^{n+1} \multimap \tau \multimap \sigma \multimap \sigma}}{\Gamma \vdash (l\,t)^{\sigma \multimap \sigma}}\ (L^{n+1}(\tau)_0^-)$$

$$\frac{\Gamma_1 \vdash l^{L^{n+1}(\tau)} \quad x^{\rho} \vdash t^{\diamond^{n+1} \multimap \tau \multimap \sigma \multimap \sigma} \quad \Gamma_2 \vdash r^{!^n \rho}}{\Gamma_1, \Gamma_2 \vdash \left(l\, t[x :=]r[_{!^n}]\right)^{\sigma \multimap \sigma}} \; (L^{n+1}(\tau)_1^-)$$

where Γ_1, Γ_2 denotes the multiset union which maintains the context condition.

In $(\multimap^+)$ there must not be x in Γ, in (C) no y_1, y_2 in Γ_1 , in $(!_1)$ no x in Γ and in $(L^{n+1}(\tau)_\tau^-)$ no x in Γ_1, Γ_2.

In $(!_0)$, $(!_1)$, $(L^{n+1}(\tau)_0^-)$ and $(L^{n+1}(\tau)_1^-)$ the term t must not have subterms of level $\leq n$ or occurrences of $\mathbf{d}^i$ with $i \leq n+1$.

The typing rules are essentially *syntax directed*. Each subterm $s \lhd t$ of $\Gamma \vdash t^{\tau}$ (with the exception of the $]r[_{!^n}$ in $(L^{n+1}(\tau)_1^-)$) appears as the premise of some rule in the typing derivation of t (or in the context in the case of variables).

Fact 7.16. *The restriction of the rules* $(!_0)$, $(!_1)$, $(L^{n+1}(\tau)_0^-)$ *and* $(L^{n+1}(\tau)_1^-)$ *implies that* t *does not have occurrences*

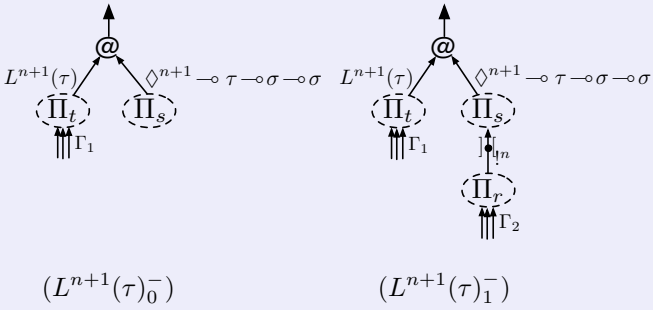
- *of constants of level* $\leq n$ *or*
- *subterms of shape* $]\cdot[_{!^i}$ *with* $i \leq n$ *or*
- $!^i\boxed{\cdot}$*-boxes with* $i \leq n$ *or*
- $\left(\cdot \lhd^{\,i\;\;y_1}_{\;y_2} \cdot\right)$ *with* $i \leq n$ *or*
- $\boldsymbol{d}^i$ *with* $i \leq n+1$ *or*
- *iterations of level* $i < n+1$.

Definition 7.17 (Maximal level)**.** The maximal level L_t of a well-typed term t is the maximal level of its subterms.

7.3.3.2. Proof Nets

For the definition of proof nets we closely follow Section 6.2.2 about $\mathrm{LLT}_!$.

Definition 7.18 (Proof net structure)**.** Mutatis mutandis like Definition 6.15 with the additional typing rules for iterations:



$(L^{n+1}(\tau)_0^-)$ $(L^{n+1}(\tau)_1^-)$

Definition 7.19 (Proof net)**.** Mutatis mutandis like Definition 6.15.

Definition 7.20 (Subproof net structure)**.** Mutatis mutandis like Definition 7.20.

Fact 7.21. *The restrictions of the typing rules* $(!_0)$*,* $(!_1)$*,* $(L^{n+1}(\tau)_0^-)$ *and* $(L^{n+1}(\tau)_1^-)$ *of Definition 7.15 for terms imply that the subnet* Π_t *of a* $!^n$*-box in a proof net* Π *does not have occurrences of*

- *constants of level* $\leq n$ *or*
- $]\cdot[_{!^i}$*-nodes with* $i \leq n$ *or*
- $!^i\boxed{\cdot}$*-boxes with* $i \leq n$ *or*
- *multiplexers* $\overset{i}{\lhd}$ *with* $i \leq n$ *or*
- $\boldsymbol{d}^i$ *with* $i \leq n+1$ *or*
- *iterations of level* $i < n+1$

in analogy to Remark 7.16 for terms.

> *Remark* 7.22. We have no explicit restriction about nodes of lower level in boxes *for proof nets*. Instead this is a consequence that a proof net is the image of a term into the proof net language.

The translation from terms to proof nets, again, is mostly just the transformation of a term to its parse tree. As for $\mathrm{LLT}_!$ we have:

Fact 7.23. *Let Π_s be the subproof net of a proof net Π_t for the term t^τ with $s \trianglelefteq t$ as the subterm for Π_s. Then Π_s is a subproof net structure of Π_t.*

Definition 7.24 (Paths and proper paths)**.** Mutatis mutandis like Definition 2.34.

Fact 2.35 and Remarks 2.36, 2.37 about paths apply here as well.

Definition 7.25 (Level of a node, level of a proof net)**.** Mutatis mutandis like Definition 6.20.

7.3.4. Normalisation

As the definitions above also the normalisation of $\mathrm{LLFPL}_!$ is mostly that of $\mathrm{LLT}_!$ with necessary differences for the new iteration construction and the $\diamondsuit^n$ argument of the $\mathbf{cons}^n_\tau$:

Definition 7.26 (Normalisation/Cut Elimination)**.** The graph rewriting relation between proof nets is split into the following redexes (compare [AR00] and Definition 6.31 for $\mathrm{LLT}_!$):

1. The linear cuts (see Figure 6.2a on page 178) with the λ-restriction *that a $\longmapsto^n_l$-redex is not under a λ-node of level $\leq n$*:

$$
\begin{array}{rcll}
(\lambda x^\sigma.t^\tau\ s) & \longmapsto^n_l & t[x := s] & \text{with } \ell(\sigma \multimap \tau) = n \\
\left(\pi^n_{\sigma,\tau,\rho}\ \left(\otimes^n_{\sigma,\tau}\ s\ t\right)\ f\right) & \longmapsto^n_l & (f\ s\ t) & \\
(\mathbf{Case}^n_\sigma\ \mathbf{tt}^n\ f\ g) & \longmapsto^n_l & f & \\
(\mathbf{Case}^n_\sigma\ \mathbf{ff}^n\ f\ g) & \longmapsto^n_l & g & \\
\left(\mathbf{Case}^n_{\tau,\sigma}\ \mathbf{nil}^n_\tau\ f\ g\right) & \longmapsto^n_l & g & \\
\left(\mathbf{Case}^n_{\tau,\sigma}\ \left(\mathbf{cons}^n_\tau\ d^\diamondsuit\ x\ l\right)\ f\ g\right) & \longmapsto^n_l & (f\ d\ x\ l) &
\end{array}
$$

and

$$\Big(\Big(\mathbf{cons}^{n+1}_{\tau}\, t^{\diamond^{n+1}}\, v\, l\Big)\; f\; g\Big) \longmapsto^{n+1}_{l} (f\, t\, v\; (l\, f\, g))$$

$$\Big(\Big(\mathbf{cons}^{n+1}_{\tau}\, t^{\diamond^{n+1}}\, v\, l\Big)\; f[z :=]r[_{!^n}]\, g\Big) \longmapsto^{n+1}_{l}$$

$$\Big(r \lhd^{n\, r_1}_{r_2} (f[z :=]r_1[_{!^n}]\, t\, v\; (l\, f[z :=]r_2[_{!^n}]\, g))\Big)$$

$$\big(\mathbf{nil}^{n+1}_{\tau}\, f\, g\big) \longmapsto^{n+1}_{l} g$$

for the iteration (see Figure 7.4). The first rule is for a closed f typed via $(L^{n+1}(\tau)^-_0)$, the second for the $(L^{n+1}(\tau)^-_1)$-case.

2. The shifting cuts like in Definition 6.31 (see Figure 6.2b on page 178).

3. The multiplexer/contraction cuts like in Definition 6.31 (see Figure 6.2c on page 178).

The last two kinds are called polynomial redexes and are denoted by $\longmapsto^n_p := \longmapsto^n_s \cup \longmapsto^n_m$, an arbitrary redex by $\longmapsto^n := \longmapsto^n_l \cup \longmapsto^n_s \cup \longmapsto^n_m$.

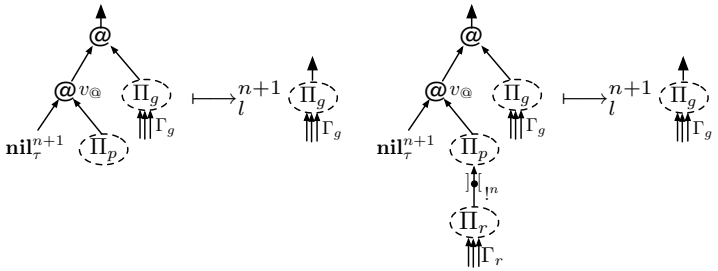
We write $\Pi_t \longrightarrow^n \Pi_{t'}$ if $\Pi_s \longmapsto^n \Pi_{s'}$ for some subproof net Π_s of Π_t and $\Pi_{t'}$ is Π_t with Π_s replaced by $\Pi_{s'}$ (and mutatis mutandis $\longrightarrow^n_\star$ for $\longmapsto^n_\star$ redexes) after cleaning up.

A proof net Π_t is called $\longmapsto^n_\star$-normal or $\longrightarrow^n_\star$-normal if there is no $\Pi_{t'}$ with $\Pi_t \longrightarrow^n_\star \Pi_{t'}$. The notation $\xrightarrow{\text{nf}}{}^n_\star$ denotes the relation which puts a term in relation with its normal form(s), i.e.

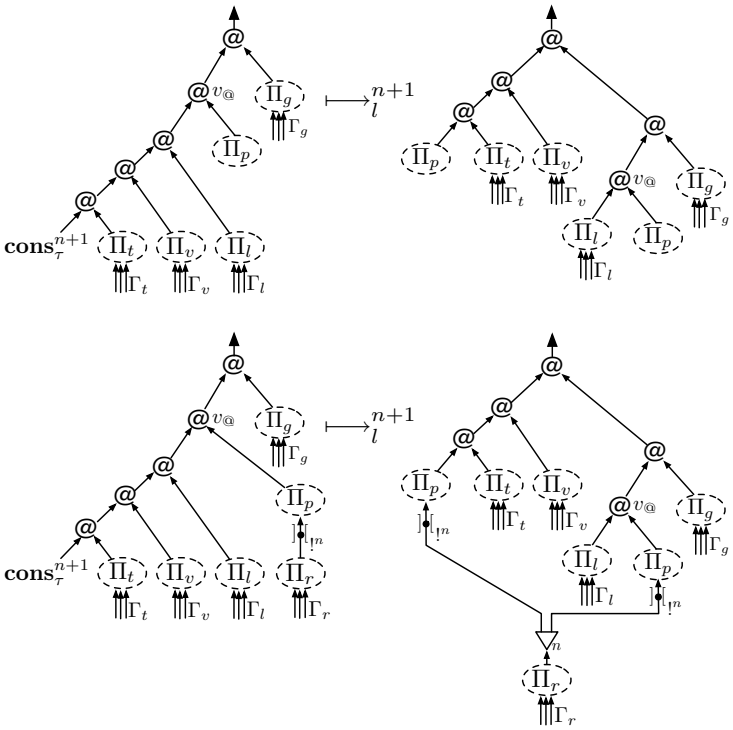
$$\Pi \overset{\text{nf}}{\sim} \Pi' :\Longleftrightarrow \Pi \sim^* \Pi' \wedge \Pi' \text{ is } \sim\text{-normal.}$$

Note the following difference of the normalisation for $LLFPL_!$ defined above and the normalisation of $LLT_!$ in Definition 6.31: in order to allow a length estimate for lists under a λ-abstraction of level $\leq n$ (compare with δLFPL in Chapter 4), we have to forbid normalisation under such a λ-node in $LLFPL_!$.

The $LLFPL_!$-normalisation is a mixture of LFPL's reductions (arbitrary order; $\longmapsto^n_l$ not under λ-nodes of level $\leq n$) and the polynomial reductions of light systems, both phases for each level:



(a) $\mathbf{nil}_\tau^{n+1}$-cases



(b) $\mathbf{cons}_\tau^{n+1}$-cases

Figure 7.4.: The iteration cuts for LLFPL$_!$

$$\longrightarrow_{\text{LLFPL}_!} \quad := \quad \prod_{n=0}^{N} \xrightarrow{\text{nf}}{}^{n}_{l} \xrightarrow{\text{nf}}{}^{n}_{p}$$

with $N \in \mathbb{N}_0$ as the maximal level of (nested) nodes in the proof net.

First, we make sure that the normalisation of one level works by splitting it into the linear and the polynomial reduction:

Lemma 7.27 (Level normalisation)**.** *Let Π_t be $\longmapsto^{i}$-normal for each $i < n$, $\longmapsto^{n}_{l}$-normal and $\Pi_t \longrightarrow^{n}_{p} \Pi'_t$. Then also Π'_t is $\longmapsto^{n}_{l}$-normal.*

Proof. All the $\longmapsto^{n}_{l}$-redexes, with the only exception of the iteration cut, talk only about one level and do not involve any reference to boxes. Moreover, the boxes in $\longmapsto^{n}_{s}$ and $\longmapsto^{n}_{m}$ are of level n, therefore the content is of level $\geq n+1$ and hence not of interest for $\longmapsto^{n}_{l}$-cuts.

The iteration $\longmapsto^{n}_{l}$-redex of level n only references boxes of level $n-1$. Hence, a $\longmapsto^{n}_{s}$- or $\longmapsto^{n}_{m}$-reduction, which merges or duplicates level n boxes, cannot cause a new iteration redex of level n. □

7.3.5. Complexity

The main difference of LLFPL!'s normalisation, compared with the one of LLT!, is the changed iteration cut in the linear part of the normalisation of each level. The iteration phase $\longmapsto^{n}_{i}$ of LLT! for level $n+1$ lists is dropped in LLFPL! and replaced by "flat" level $n+1$ reductions as part of the linear phase. The polynomial part of the reduction is the same in LLT! and LLFPL!, such that we know by Lemma 6.41 that the polynomial reduction $\xrightarrow{\text{nf}}{}^{n}_{p}$ is of complexity $\mathcal{O}(|\Pi|^3)$ and results in a proof net of size $\mathcal{O}(|\Pi|^4)$.

What we have to show now is that the complexity of the linear reduction $\xrightarrow{\text{nf}}{}^{n}_{l}$ is polynomial in the proof net size with a degree which is uniform in the proof net at the very beginning of the normalisation, and of course independent of the input numerals/lists.

Compared to LFPL or δLFPL we do not need free variables in this setting to get terms of type $\diamondsuit^n$, because the constant $\mathbf{d}^n$ is available for this purpose. Hence, the concept of *almost-closeness* (compare Definition 4.6) is not needed here.

Definition 7.28 (λ-abstraction of a level, bound for a level)**.** A λ-node of type $\sigma \multimap \tau$ with $\ell(\sigma \multimap \tau) = n$ is called a λ-abstraction of level n.

A node x in a proof net Π is called bound for level n in Π (short: n-bound) if x is under a λ-*abstraction node* z *of level* n in Π and there is a path from z to x.

A subproof net Π' of Π is called bound for level n in Π (short: n-bound) if there is a node x in Π' that is bound for level n in Π.

If an iteration redex is fired, we know that it is not under λ-*abstraction of level* n (by restrictions on the normalisation rules). Therefore, the list cannot be bound for level n. If a list was bound for level n, a beta-redex on level n might change its length.

Moreover, the λ-restriction in Definition 7.26 also forbids λ-nodes above *of lower level*, i.e. $< n$. This will be necessary in Lemma 7.37 in order to know that the $\diamondsuit^n$-parameter of a $\mathbf{cons}^n_\tau$-constant in the iteration redex is not due to a bound variable of level $< n$.

With this in mind, we define the measures by counting the $\mathbf{d}^n$-nodes:

Definition 7.29 (List length measure)**.** For a proof net Π with $d^n(\Pi)$ we denote the number of (nested) $\mathbf{d}^n$-nodes in Π (and in analogy for terms t, i.e. $d^n(t) := |\{\mathbf{d}^n \trianglelefteq t\}|$).

For a subproof net Π' of Π with a list type $L^n(\tau)$ which is not n-bound in Π, we call $d^n(x)$ the length $l(\Pi') := d^n(\Pi')$ of Π' in Π (and in analogy for a list term t).

Clearly, in this basic setting we get the usual properties of a sensible length measure, e.g.

$$l((\mathbf{cons}^n_\tau\, \mathbf{d}^n\, v\, l)) < l(l)$$

and moreover a bound $l(s) \leq d^n(t)$ for each not n-bound subterm $s \trianglelefteq t$ (and in analogy for proof nets).

Note that we restrict ourselves to lists here which have no bound variables which might be substituted later by a beta-redex of level n. Hence, the substitution property of length measures in Section 4.6 does *not* apply.

We will define the complexity measure in analogy to Definition 4.45 for δLFPL. There, the model of computation *is* the λ-term. Here, we normalise

using proof nets. Hence, we have to adapt the $\vartheta_q(\cdot)$-measure of Chapter 4 to this setting:

Intuition for the complexity measure Each (nested) node v in a proof net Π is assigned a multiplicator $\vartheta^n_\Pi(v) \in \mathbb{N}_0[X]$ for level n, which depends on the $(L^n(\tau)^-_{0/1})$-nodes $v_@$ "effectively above" v in the proof net. Each such node $v_@$ with a list $t^{L^n(\tau)}$ gives raise to

- a factor $X + 1 \in \mathbb{N}_0$ in $\vartheta^n_\Pi(v)$ if $v_@$ is n-bound in Π,
- a factor $l(t) + 1$ in $\vartheta^n_\Pi(v)$ otherwise.

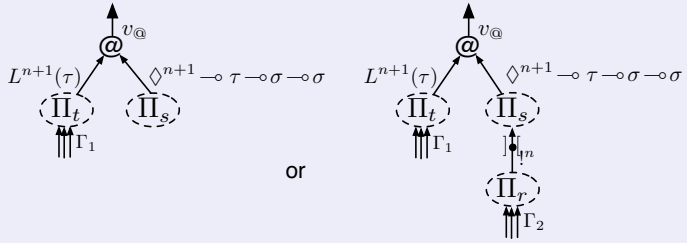
Note that the second case is chosen if $v_@$ is not under λ-abstraction (of any level $\leq n$).

> *Remark* 7.30. The intuition given for δLFPL in Remark 4.46 applies here as well, transferred to the setting with levels. We restrict the first case to "n-bound", instead of just "bound" as in δLFPL in Section 4.45, because the measure $\vartheta^n_\Pi(v)$ is just needed for the normalisation of the linear redexes on level n. It can be that there is a λ-abstraction of level $> n$ above an iteration node. But this has no influence on the level n normalisation, and, in fact, the λ-restriction of level n allows redexes under such a node.

Effectively above We have to define what "effectively above" means: the idea here is that an iteration only duplicates a node x if it is in the proof net Π_s of the $(L^n(\tau)^-_{0/1})$-rule as shown in Definition 7.18. Otherwise, the corresponding factor of the $(L^n(\tau)^-_{0/1})$-node is ignored for this node x. This makes sense because the normalisation of this iteration node will not duplicate any node outside Π_s.

We will make all this formal now:

Definition 7.31 (Effectively above, factor, multiplicator)**.** Let $v_@$ be the @-node as in



$(L^{n+1}(\tau)^-_0)$ $(L^{n+1}(\tau)^-_1)$

in the proof net Π and w a (nested) node *in* Π_s. Then $v_@$ is effectively above w for level $n+1$.

For each such $v_@$ effectively above w, we define the factor for level $n+1$, denoted $\varphi^{n+1}_\Pi(v_@) \in \mathbb{N}_0(X)$, as

- $\varphi^{n+1}_\Pi(v_@) := X + 1$ if $v_@$ is n-bound in Π,
- $\varphi^{n+1}_\Pi(v_@) := l(\Pi_t) + 1$ otherwise.

The multiplicator for level n of a node w in Π, denoted $\vartheta^n_\Pi(w)$, is the product of the factors of nodes which are effectively above w for level n, i.e.

$$\vartheta^n_\Pi(w) := \prod_{v_@ \in ea^n_\Pi(w)} \varphi^n_\Pi(v_@)$$

where $ea^n_\Pi(w)$ is the set of nodes effectively above w in Π.

We call $D^n_\Pi(w) := |ea^n_\Pi(w)|$ the (iteration) nesting depth of w for level n. The maximal $D^n_\Pi(w)$ for all (nested) nodes w of a proof net Π is called the nesting depth of Π for level n, denoted $D^n(\Pi)$.

Lemma 7.32. *For $X \in \mathbb{N}_0$ and every (nested) node v in a proof net Π*

$$\vartheta^n_\Pi(v)(X) \geq 1$$

holds.

Proof. We have $X \geq 0$ by assumption and $l(\Pi_t) \geq 0$ by definition of the length measure. Hence, for every factor $\varphi^n_\Pi(v_@) \geq 1$ holds and therefore

$\vartheta^n_\Pi(v) := \Pi\varphi^n_\Pi(v_@) \geq 1$, especially if there is no $v_@$ effectively above v for level n. □

Compare the multiplicator with Definition 4.45 for δLFPL. Due to the inductive definition of terms there, we gave a recursive definition computing the measure for a whole term. Here, we compute a measure for each node w by looking for the set of nodes effectively above. If a node w is in a step of an iteration (and not in the proof net below the hole in case of $(L^n(\tau)^-_1)$), it will weight more because it might be duplicated depending on the list length of the iteration. This is the same in both definitions.

The distinction between effective and non-effective is not necessary for δLFPL or LFPL, because there every step term is closed. Hence, a (nested) node, which is below an iteration node in the step, is automatically effective, because the right $(L^n(\tau)^-_1)$-case in the figure of Definition 7.31 above does not exist. Moreover, there we only have one level, such that the postfix "for level n" can be dropped as well.

> *Remark* 7.33. Remark 4.46 does apply here as well, as a motivation for the upper definition.

We stress the point that the nodes in a step of $(L^{n+1}(\tau)^-_1)$, which are not in Π_s (e.g. the $]\cdot[_{!^n}$-node and the nodes in Π_r in the figure of Definition 7.31 above) are only counted once, and *not* multiplied by the length of the list (the $(L^{n+1}(\tau)^-_1)$-node is not effective for those). This fits to the reduction rule for the $\mathbf{cons}^{n+1}_\tau$-case of the iteration which does not duplicate the Π_r subnet, but only introduces a multiplexer to get the typing right.

With the multiplicator we define the measure $\vartheta^n(\Pi_t)$ for a complete proof net (which corresponds to $\vartheta_q(t)$ in Definition 4.45 for δLFPL):

Definition 7.34 (Complexity measure)**.** For each proof net Π_t and a level $n \in \mathbb{N}_0$ the complexity measure $\vartheta^n(\Pi_t) \in \mathbb{N}_0[X]$ is defined as:

$$\vartheta^n(\Pi_t) := \sum_{v \text{ nested node of } \Pi_t} w(v) \cdot \vartheta^n_{\Pi_t}(v),$$

with $w(v) := 3$ for $\mathbf{cons}^n_\tau$-nodes and $w(v) := 1$ otherwise.

Remark 7.35. The weight $w(v)$ here is for technical reasons to compensate for the new nodes on the right hand side of the $\mathbf{cons}_\tau^{n+1}$-case in Figure 7.4b on page 214.

In analogy to δLFPL (compare Lemma 4.48), we first consider the beta reduction case. We have to show that the complexity measure goes strictly down when such a redex is fired. For this it is enough to show that for each (nested) node v in the net one of the following holds:

1. $\vartheta^n_\Pi(v)(X)$ stays the same,
2. or $\vartheta^n_\Pi(v)(X)$ goes down,
3. or v disappears during normalisation,

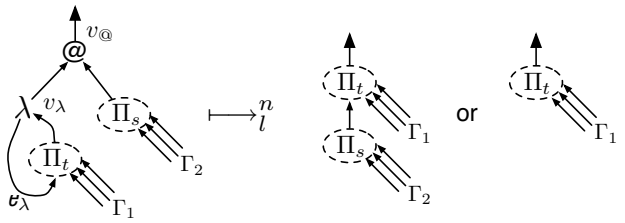
and for at least one node v in the proof net either 2 or 3 is the case, in order to get a strict decrease.

Lemma 7.36 (Substitution complexity)**.** *Let* $\Pi \longrightarrow_l^n \Pi'$ *via a* β*-redex. Then*

$$\vartheta^n(\Pi)(X) > \vartheta^n(\Pi')(X)$$

holds for all $X \geq d^n(\Pi)$*.*

Proof. Assume that $\Pi \longrightarrow_l^n \Pi'$ is the case due to the following β-redex (by definition of the reduction rules $v_@$ is not under λ-abstraction of level $\leq n$):



In either case, the @-node is dropped and by Lemma 7.32 we have $\vartheta^n_\Pi(v_@) \geq 1$. Hence, it is enough to show that the multiplicators of all the other nodes stay the same or decrease for the chosen X.

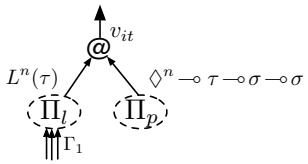
The right case: Clearly, the right case is trivial because then all the nodes in Π_s are dropped (due to the garbage cleanup). If a $(L^n(\tau)^-_{0/1})$-node in

Π_s was effectively above any node w for level n in Π, this w "lost" a factor in its multiplicator $\vartheta^n_{\Pi'}(w)$. Hence, $\vartheta^n_{\Pi'}(w)(X) \leq \vartheta^n_{\Pi}(w)(X)$. Similarly, the possible $\mathbf{d}^n$-nodes in Π_s are dropped which possibly reduces factors of $(L^n(\tau)^-_{0/1})$-nodes above $v_@$ because $l(\Pi_l)$ for those iteration nodes decreased.

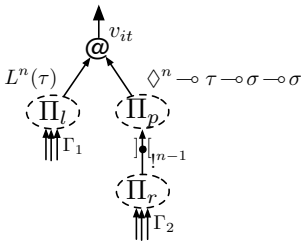
The left case: Now, suppose the left case of the figure happens, i.e. the argument Π_s is placed below Π_t:

First, no node w in Π_s will loose a factor because an iteration node effectively above will still be effectively above.

Can there be a $(L^n(\tau)^-_{0/1})$-node v_{it} in Π_t which is now effectively above a node w in the reduct Π' that was not effectively above before? We can assume, that such a w is not in Π_t and v_{it} is either a $(L^n(\tau)^-_0)$- or $(L^n(\tau)^-_1)$-node:



$(L^n(\tau)^-_0)$



$(L^n(\tau)^-_1)$

In either situation, w must lay inside Π_p in Π', because v_{it} is effectively above w. Moreover, we know that v_{it} is below $v_@$ in Π', because v_{it} is in Π_t. In Π though, there must have been a path, starting with e_λ and ending in v_{it} via the output link of Π_p, because w (which is not in Π_t) cannot be below v_{it} otherwise.

- In the $(L^n(\tau)^-_0)$-case this path is impossible because Π_p is closed.
- In the $(L^n(\tau)^-_1)$-case this path must have gone through the $]\cdot[_{!^{n-1}}$-node of the $(L^n(\tau)^-_1)$-rule in Π. But then Π_s would be completely below this $]\cdot[_{!^{n-1}}$-node in Π'. Hence, w could not be in Π_p. Contradiction.

Finally, if an n-bound $(L^n(\tau)^-_{0/1})$-node in Π is not n-bound anymore in Π', its factors in all the multiplicators turn from $X+1$ to $\ell(\Pi_l)+1$. Hence, due to $X \geq d^n(\Pi) \geq l(\Pi_l)$ for the list proof net of the $(L^n(\tau)^-_{0/1})$-node, every multiplicator $\vartheta^n_\Pi(w)$ with this factor (non-strictly) decreases: $\vartheta^n_\Pi(w) \geq \vartheta^n_{\Pi'}(w)$.

Note that a $(L^n(\tau)^-_{0/1})$-node which is not n-bound in Π, cannot become n-bound in Π'. □

With this result we can tackle the complexity lemma. This will give us the polynomial which bounds the number of steps during the normalisation of the $\longmapsto^n_l$-redexes on one level n:

Lemma 7.37 (Complexity of $\xrightarrow{\text{nf}}{}^n_l$)**.** *Assume a closed proof net* Π *and* $d^n(\Pi) \leq N \in \mathbb{N}$*. If* $\Pi \longrightarrow^n_l \Pi'$*, then*

$$\vartheta^n(\Pi)(N) > \vartheta^n(\Pi')(N).$$

In particular, any reduction sequence starting from Π *has a length of at most* $\vartheta^n(\Pi)(N)$*, and*

$$\vartheta^n(\Pi)(|\Pi|) \leq (|\Pi|+1)^{D^n(\Pi)+1} \in \mathcal{O}(|\Pi|^{D^n(\Pi)+1})$$

holds.

Proof. We proof the first claim by case distinction on the different kinds of redexes $\longmapsto^n_l$:

- β-redex: shown in Lemma 7.36.
- $\left(\mathbf{Case}^n_{\tau,\sigma}\ \mathbf{nil}^n_\tau\ f\ g\right) \longmapsto^n_l\ g$: only the subnet for g is kept. The other nodes are dropped. Possibly some factors of $(L^n(\tau)^-_{0/1})$-nodes $v_@$ decrease, because of deleted $\mathbf{d}^n$-nodes in the list proof net Π_l of $v_@$ the length measure $l(\Pi_l)$ can decrease.
- $\left(\mathbf{Case}^n_{\tau,\sigma}\ \left(\mathbf{cons}^n_\tau\ d^\diamondsuit\ x\ l\right)\ f\ g\right) \longmapsto^n_l\ (f\ d\ x\ l)$: similar.
- $(\mathbf{Case}^n_\tau\ \mathbf{tt}^n\ f\ g) \longmapsto^n_l\ f$: similar.
- $(\mathbf{Case}^n_\tau\ \mathbf{ff}^n\ f\ g) \longmapsto^n_l\ g$: similar.
- $\left(\pi^n_{\sigma,\tau,\rho}\ (\otimes_{\sigma,\tau}\ s\ t)\ f\right) \longmapsto^n_l\ (f\ s\ t)$: similar.

- Iteration redex: this is the interesting case. Compare the nodes on the left sides with the nodes on the right sides in Figure 7.4 on page 214.
  - $\mathbf{nil}^n_\tau$-cases: similarly to the cases above, most of the nodes are dropped: the dropped $\mathbf{d}^n$ might decrease some factors. The dropped $(L^n(\tau)^-_{0/1})$-nodes $v_@$ might remove factors $\varphi^n_\Pi(v_@)$ from some multiplicators $\vartheta^n_\Pi(w)$.
  - $\mathbf{cons}^n_\tau$-cases: we consider the second $(L^n(\tau)^-_1)$-case (the first is similar, but even simpler). Counting the nodes gives that the right side has one $]\cdot[_{!^{n-1}}$-node, one $\overset{n-1}{\lhd}$-node and the Π_p-copy more (*note:* in the Figure 7.4b $\overset{n+1}{\underset{l}{\longmapsto}}$ is pictured, here we consider $\overset{n}{\underset{l}{\longmapsto}}$).

    Because Π is closed and the redex is not under λ-abstraction for level $\leq n$, we know that Π_t must have at least one $\mathbf{d}^n$-node[2]. Hence, the factor $\varphi^n_{\Pi'}(v_@)$ for the nodes w'_r in the right copy of Π_p in Π' is at least one smaller than it was in Π, i.e. $\vartheta^n_\Pi(v_@)$. Hence, if w is the (nested) node in Π_p in Π, w'_l the corresponding (nested) node in the left copy of Π_p in Π', and w'_r the (nested) node in the right copy, we get:

$$\begin{aligned}
\vartheta^n_\Pi(w)(X) &= \varphi^n_\Pi(v_@)(X) \cdot \prod_{\substack{v \in ea^n_\Pi(w) \\ v \neq v_@}} \varphi^n_\Pi(v)(X) \\
&= (l(\Pi_{(\mathbf{cons}_\tau\ t\ v\ l)}) + 1) \cdot \prod_{\substack{v \in ea^n_\Pi(w) \\ v \neq v_@}} \varphi^n_\Pi(v)(X) \\
&\geq (l(l) + 1 + 1) \cdot \prod_{\substack{v \in ea^n_\Pi(w) \\ v \neq v_@}} \varphi^n_\Pi(v)(X) \\
&\geq \prod_{\substack{v \in ea^n_\Pi(w) \\ v \neq v_@}} \varphi^n_\Pi(v)(X) + (l(l) + 1) \cdot \prod_{\substack{v \in ea^n_\Pi(w) \\ v \neq v_@}} \varphi^n_\Pi(v)(X) \\
&\geq \vartheta^n_{\Pi'}(w'_l)(X) + \vartheta^n_{\Pi'}(w'_r)(X).
\end{aligned}$$

    Because $w(w) = w(w'_l) = w(w'_r)$ holds, we get $w(w) \cdot \vartheta^n_\Pi(w) \geq w(w'_l) \cdot \vartheta^n_{\Pi'}(w'_l) + w(w'_r) \cdot \vartheta^n_{\Pi'}(w'_r)$ as required.

[2] Here, we need that the λ-restriction for level n forbids λ-nodes above of level $\leq n$, not only of level $= n$: there is no term of type $\diamond^n$ with only a free variable of level $> n$; but, there is a term of type $\diamond^n$ with a free variable of level $< n$, e.g. $(x^{B^{n-1} \multimap \diamond^n}\ \mathbf{tt}^{n-1})$.

The dropped $\mathbf{cons}^n_\tau$-node contributes a decrease of $3 \cdot \vartheta^n_\Pi(v_@)$ to the measure $\vartheta^n(\Pi')$, due to $w(\mathbf{cons}^n_\tau) := 3$ in Definition 7.34. The additional $]\cdot[_{!^{n-1}}$- and $\overset{n-1}{\lhd}$-nodes contribute an increase of $1 \cdot \vartheta^n_{\Pi'}(v_@)$ each. With $\vartheta^n_\Pi(v_@) = \vartheta^n_{\Pi'}(v_@)$ this gives a decrease of $1 \cdot \vartheta^n_{\Pi'}(v_@)$ for $\vartheta^n(\Pi')$ altogether, compared to $\vartheta^n(\Pi)$.

For each node w in Π_r the effective nodes $ea^n_\Pi(w)$ stay the same in Π' because no $(L^n(\tau)^-_{0/1})$-node $v_@$ in either copy of Π_p can be effectively above w. Hence, the multiplexer $\overset{n-1}{\lhd}$ and the two nets Π_p cannot lead to an increase of the multiplicator of those w.

We have shown now, that the value $\vartheta^n(\Pi)(N)$ strictly decreases in every linear reduction step. Of course, $\vartheta^n(\Pi)(N)$ cannot be negative, with $\vartheta^n(\Pi)$ as a polynomial with positive coefficients. Hence, $\vartheta^n(\Pi)(N)$ is an upper bound of the possible length of any reduction sequence starting from Π.

A (nested) node v in a proof net Π has at most $D^n_\Pi(v)$ many factors in its multiplicator on level n. For every factor $\varphi^n_\Pi(v_@)$ with the list subnet Π_l of $v_@$ we have by definition and due to $|\Pi| \geq d^n(\Pi) \geq l(\Pi_l)$:

$$\varphi^n_\Pi(v_@)(|\Pi|) \leq |\Pi| + 1.$$

Hence, with $d^n(\Pi) \leq |\Pi|$:

$$\begin{aligned}
\vartheta^n(\Pi)(|\Pi|) &:= \sum_{v \text{ nested node in } \Pi} w(v) \cdot \vartheta^n_\Pi(v)(|\Pi|) \\
&\leq \sum_{v \text{ nested node in } \Pi} w(v) \cdot (|\Pi| + 1)^{D^n(\Pi)} \\
&\leq 3 \cdot |\Pi| \cdot (|\Pi| + 1)^{D^n(\Pi)} \\
&\leq 3 \cdot (|\Pi| + 1)^{D^n(\Pi)+1} \in \mathcal{O}(|\Pi|^{D^n(\Pi)+1}).
\end{aligned}$$

□

Corollary 7.38. *For* $\Pi \xrightarrow{nf}{}^n_l \Pi'$, Π *closed, the size grows polynomially in* $|\Pi|$*, more precisely*

$$\left|\Pi'\right| \in \mathcal{O}(|\Pi|^{D^n(\Pi)+1}).$$

Proof. By Lemma 7.32 and Definition 7.34 we have $|\Pi'| \leq \vartheta^n(\Pi')$. With the previous lemma we get $\vartheta^n(\Pi')(|\Pi|) \leq \vartheta^n(\Pi)(|\Pi|) \in \mathcal{O}(|\Pi|^{D^n(\Pi)+1})$. □

7.3.5.1. Uniformity of the nesting depth

Before the complexity lemma for the linear normalisation can be combined with the polynomial $\xrightarrow{\mathrm{nf}}_p^n$-normalisation on each level, we have to make sure that

1. the degree of $\vartheta^n(\Pi)$ is uniform in Π with fixed $D^n(\Pi)$ in the sense that it only depends on $D^n(\Pi)$, not on the details of Π otherwise,

2. and that the degree of $\vartheta^n(\Pi)$ does not grow under normalisation, especially of lower levels than n.

The first property is part of Lemma 7.32.

In Remark 7.7 we already sketched how we can get the second property. The basic idea is that (non-trivial) nested iterations require the step to be typed with the $(L^n(\tau)_0^-)$-rule instead of $(L^n(\tau)_1^-)$. Then steps are closed from the very beginning and their nesting depth is constant during normalisation.

In order to see the reason for this, first we study three cases which nest iterations:

Case 1. Consider a nested iteration on level n of at least nesting depth 2:

$$\Big(l^{L^n(\tau)}\ \underbrace{\lambda d\lambda x\lambda p.\ldots.]s[_{!^{n-1}}\ldots\big(l'\ \overbrace{\lambda d'\lambda x'\lambda p'.\ldots.]s'[_{!^{n-1}}\ldots}^{\text{inner step}}\big)\ldots}_{\text{outer step}}\Big).$$

The outer step is typed with $(L^n(\tau)_1^-)$, which is fine. The inner step also has a $]\cdot[_{!^{n-1}}$-hole. Therefore, it must be typed with $(L^n(\tau)_1^-)$ as well. But this means that the outer step has two of those $]\cdot[_{!^{n-1}}$ nodes which is not possible by the restriction of the $(L^n(\tau)_1^-)$-rule. For this recall that the proof net of the step of an iteration has the same properties as a proof net of a $!^{n-1}$-box. Hence, it cannot have a (nested) $]\cdot[_{!^{n-1}}$-node.

Case 2. Consider another nesting:

$$\Big(l^{L^n(\tau)}\ \underbrace{\lambda d\lambda x\lambda p.\ldots.]\ldots\big(l'\ \overbrace{\lambda d'\lambda x'\lambda p'.\ldots.]s'[_{!^{n-1}}\ldots}^{\text{inner step}}\big)\ldots[_{!^{n-1}}\ldots}_{\text{outer step}}\Big),$$

i.e. the inner iteration is inside the $]\cdot[_{!^{n-1}}$-hole of the outer step term. This situation is fine. But in order to be a nested iterations in the sense of LFPL, the inner iteration must be in the Π_s-subnet (in the sense of figure of Definition 7.18), such that the outer iteration node $v_@$ is *effectively above* the inner one. Here, this is not the case. In other words, the nesting is only 1.

Case 3. Consider the third nesting:

$$\Big(l^{L^n(\tau)}\ \underbrace{\lambda d\lambda x\lambda p....]...!^{n-1}\boxed{(l'\ \overbrace{\lambda d'\lambda x'\lambda p'....]s'[_{!^{n-1}}...)}^{\text{inner step}}}...[_{!^{n-1}}...}_{\text{outer step}}\Big),$$

i.e. the inner iteration is inside a $!^{n-1}$-box which will eventually merge with the $]\cdot[_{!^{n-1}}$-node of the $(L^n(\tau)^-_1)$-rule of the outer iteration . Here, we have to be more precise in the notation though:

- If $]s'[_{!^{n-1}}$ is supposed to be a subterm (subproof net) of the content of the box (proof net of the box), then this is not typable because of the restriction of the $!_{0,1}$-rules about subterms (nodes).
- If the notation is the shorthand for

$$!^{n-1}\boxed{z =]s'[_{!^{n-1}}\ \text{ in }\ (l'\ \overbrace{\lambda d'\lambda x'\lambda p'....z...)}^{\text{inner step}}},$$

then the inner iteration is, in fact, typed via $(L^n(\tau)^-_0)$.

By the first and the last case we immediately get the following lemma:

Lemma 7.39. *A proof net Π_t of a $!^{n-1}$-node (or the step proof net Π_s of a $(L^n(\tau)^-_1)$-node) has no (nested) $(L^n(\tau)^-_1)$-nodes.*

Corollary 7.40. *For a nesting depth $D^n(\Pi) \geq 2$ the* inner *level n iterations are typed with $(L^n(\tau)^-_0)$.*

We come back to the original question:

Is it possible to increase $D^n(\Pi)$ during the normalisation?

Lemma 7.41. *Let Π' be the result of normalisation by levels up to level $n-1$, started from Π. Then:*

$$D^n(\Pi') \leq D^n(\Pi) + 1.$$

Proof. First, we know that a (nested) node v of the proof net of a (nested) $!^{n-1}$-box $v_!$ in Π' "has always been" a (nested) node of a proof net of a (nested) $!^{n-1}$-box during normalisation. This is clear, because no reduction rule puts a box around a proof net which did not have a box around before the reduction.

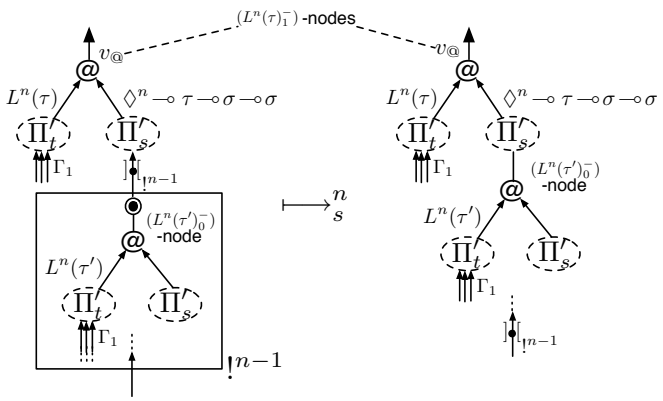
Therefore, every iteration node $v_@$ in the proof net of such a $v_!$, that is a $(L^n(\tau)_0^-)$-node by Lemma 7.39, has always been a $(L^n(\tau)_0^-)$-node during normalisation, i.e. no polynomial cut turned a $(L^n(\tau)_1^-)$-node into such a $(L^n(\tau)_0^-)$-node. This means that the iteration nesting depth of $v_@$ can be at most $D^n(\Pi)$.

We show the following two *invariants during normalisation* of Π:

Every nested $(L^n(\tau)_1^-)$-node has at most $D^n(\Pi)$ factors
and
every $(L^n(\tau)_0^-)$-node in a $!^{n-1}$-box has at most $D^n(\Pi)$ factors.

Both are true by assumption of the lemma at the beginning for Π.

Let Π' be an intermediate reduct and $\Pi' \longrightarrow^i \Pi''$. The only way to increase the number of a factors or a node from Π' to Π'' is by merging an $!^{n-1}$-box with the $]\cdot[_{!^{n-1}}$ of an $(L^n(\tau)_1^-)$-node in Π', like in:



Only the multiplicators of $(L^n(\tau)^-_{0,1})$-nodes w in Π'_s get the additional factor of $v_@$. Because these nodes w are in a box, we know that they are in fact $(L^n(\tau)^-_0)$-nodes, and therefore they have at most $D^n(\Pi)$ factors due to the second invariant. After the reduction w has at most $D^n(\Pi)+1$ factors, as $v_@$ is now possibly effectively above. But w is not in a $!^{n-1}$-box anymore. The number of factors for $v_@$ did not change such that the first invariant is still true for $v_@$.

The number of factors for nodes outside of Π'_s and Π_t did not change because $v_@$ is not effectively above them in Π''. The $(L^n(\tau)^-_{0,1})$-nodes in Π_t are in fact $(L^n(\tau)^-_0)$-nodes such that the invariants do not claim anything about them. Hence, both invariants hold again for Π''. □

Corollary 7.42. *The nesting depth of iterations on any level n of a proof net Π^τ is bounded during normalisation via $\longrightarrow_{LLFPL_!}$. The bound only depends on $D^n(\Pi)$.*

This is the uniformity that is needed to get a polynomial complexity of the normalisation by levels. For every level we can say, just from the original nesting depths, how big $D^n(\Pi')$ will be during normalisation of Π to Π'. It tells us that the nesting depth $D^n(\Pi')$, which will, later on, be a bound of the degree of the complexity polynomial of each level, does not depend on the input (if it is given as numeral with lower nesting than Π).

In other words, it is *not possible* to implement a term whose nesting depth increases depending on the length of some input numeral.

7.3.5.2. Correctness proof

Theorem 7.43 (Correctness)**.** *For every maximal nesting depth D and maximal level L there is a polynomial $P_{L,D}(x)$, such that every closed proof net Π^τ with levels $\leq L$ and $D^n(\Pi) \leq D$ can be normalised in $\mathcal{O}(P_{L,D}(|\Pi|))$ many steps.*

Proof. By Lemma 7.41 and the assumptions we have $D^n(\Pi') \leq D^n(\Pi)+1 \leq D+1$. Let be $D' := D+1$.

By Lemma 7.37 we know that there is a polynomial $p^n_D \in \mathbb{N}_0[x]$ of degree $\leq D'+1$ in the proof net size which bounds the number of possible $\longmapsto^n_l$-normalisation steps on every level n, uniformly for every closed proof net of maximal nesting depth D'.

By Corollary 7.38 the size after $\xrightarrow{\mathrm{nf}}{}^n_l$ is polynomially bounded by $q^n_D \in \mathbb{N}_0[x]$ of degree $\leq D'+1$ in the proof net size.

Applying the polynomial reductions on level n gives a polynomial "blowup" with degree 4 (compare Lemma 6.41) in a cubic number of steps at worst.

Let be Π_n be the proof net which is $\xrightarrow{\mathrm{nf}}{}^i_l \xrightarrow{\mathrm{nf}}{}^i_p$-normal for level $i < n$ and $\Pi_0 := \Pi$. Then, normalising first the linear redexes on level n of Π_n gives Π'_n, and normalising then the polynomial redexes on level n of Π'_n gives $\Pi''_n = \Pi_{n+1}$, altogether in

$$P^n_D(|\Pi_n|) := c \cdot (p^n_D(|\Pi_n|) + q^n_D(|\Pi_n|)^3)$$

many steps in the size of the proof net of the previous level, giving a normalised proof net for level n of size

$$Q^n_D(|\Pi_n|) := d \cdot q^n_D(|\Pi_n|)^4$$

for some constants $c, d \in \mathbb{N}_0$. Hence, we get Π_{n+1} from Π_n in

$$\mathcal{O}(P^n_D(|\Pi_n|))$$

many steps and of size

$$|\Pi_{n+1}| = \mathcal{O}(Q^n_D(|\Pi_n|)).$$

The degree to normalise each level is uniformly (only depending on L and

D), polynomially bounded. Because the maximal level is $\leq L$ we get Π_{L+1} in a polynomial in $|\Pi|$ which is independent of the actual Π, only depending on L and D. □

Example 7.44 (Insertion Sort in LLFPL!)**.** The insertion sort algorithm, the prime example of LFPL:

$$
\begin{aligned}
&<_n : L^n(B) \multimap L^n(B) \multimap L^n(B) \otimes L^n(B) \otimes B\\
&\mathrm{insert}_n := \lambda x^{L^n(B)}, d^{\Diamond^n} l^{L^n(L^n(B))}.\\
&\qquad \left(\pi^n \ \left(l \,\mathrm{step}\, \mathbf{nil}^n_{L^n(B)} \otimes x\right) \ \lambda l, y. \left(\mathbf{cons}^n_{L^n(B)}\, d\, y\, l\right)\right)\\
&\quad \mathrm{step} := \lambda d, x, p. \left(p\, \lambda l, y. \left((<_n \ x\, y) \ \lambda x, y, b. \left(\mathbf{Case}^n_{L^n(L^n(B))}\, c_< \, c_\geq\right)\right)\right)\\
&\qquad : \Diamond^n \multimap L^n(B) \multimap L^n(L^n(B)) \otimes L^n(B) \multimap L^n(L^n(B)) \otimes L^n(B)\\
&\quad c_< := \left(\mathbf{cons}^n_{L^n(B)}\, d\, y\, l\right) \otimes x \qquad c_\geq := \left(\mathbf{cons}^n_{L^n(B)}\, d\, x\, l\right) \otimes y\}\\
&\mathrm{sort}_n := \lambda l^{L^n(L^n(B))}. \left(l \,\mathrm{insert}_n\, \mathbf{nil}^n_{L^n(B)}\right) : L^n(L^n(B)) \multimap L^n(L^n(B)).
\end{aligned}
$$

The type of the sorting function is *symmetric*, as expected from a non-size-increasing algorithm. Hence, we can now iterate sort_n again. Of course, the resulting algorithm is still non-size-increasing and hence has a symmetric type.

Compare Insertion Sort in LLFPL! with Insertion Sort in LAL in Section 3.3.1.3. There, in the latter, the "insert"-term can be expressed with an symmetric type using the pull-out trick, but "sort" *cannot*. This is the consequence of *impredicative iteration* in LLFPL! which is not possible in LAL or LLT!.

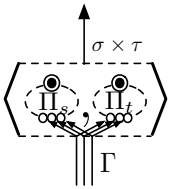
7.4. Conclusion and Outlook

In this chapter we have started with the ideas of LLT! and, step by step, transformed it into a Light LFPL-calculus. The iteration scheme of LLFPL! allows the formulation of LFPL-algorithms in LLFPL! in a trivial way because LLFPL! on one level, without any modalities is basically LFPL. At the same time, LLT!-algorithms can be reformulated to fit into LLFPL!. The latter is based on the key insight that non-size-increasing computation of LFPL with closed step terms is very similar to LLT!'s iteration scheme which requires step terms with the !-type, such that they are essentially closed (up to the possible hole in the box, which cannot contribute to the computation anymore).

While being as expressive as LFPL *and* as expressive as LLT$_!$ independently, in LLFPL$_!$ we can combine algorithms seamlessly from either type system. For example, we can express size-increasing algorithms like multiplication, and still, at the same time, we can type non-size-increasing algorithms in a symmetric way. The prime example for this is the *Insertion Sort* algorithm (compare Example 7.44) which is of type $L^n(L^n(B)) \multimap L^n(L^n(B))$ in this setting. It can be used together with the multiplication algorithm in one term. This would not be possible in either system alone.

Side product of a natural iteration for LLT$_!$ As a side product we have found a way to remove the unnatural "iteration reduction by case distinction unfolding" (compare Section 6.4) which was due to the direct translation of LAL's reduction strategy to the System T situation with levels. Here in LLFPL$_!$, the iteration constant lives on the same level as the lists and hence a normal System T like step-by-step iteration reduction is possible.

Cartesian products In the formulation of LLFPL$_!$, there is no cartesian product type $\sigma \times \tau$, as in LFPL, yet. It is not clear how important this really is (compare Section 3.1.4). In order to add this product, one has to come up with a proof net representation of a $\langle s, t\rangle$-term, which shares the variables (inputs) of s and t. As normalisation inside $\langle s, t\rangle$ is not possible in δLFPL and LFPL (compare Section 4.3), we do not have to take care of blocked redexes inside of this proof net construction either. Hence, some new proof net node like



would probably work, plus the reduction rules for the application to the boolean constants **tt** and **ff**, which would "unpack" the cartesian pair.

Going back to LAL? An interesting question is whether the ideas of LLFPL$_!$ can be translated back to LAL. We originally started with LAL, introduced

the System T variant $\mathrm{LLT}_!$ in Chapter 6, and finally transformed $\mathrm{LLT}_!$ to $\mathrm{LLFPL}_!$. This detour was worthwhile because in System T one has less freedom in the choice of the data types (System F allows many different encodings), but more choice to adapt the types and reduction rules of constants. The latter was the central tool in Sections 7.1-7.3. Though, it is not clear whether something similar works in LAL as well (compare Figure 7.5). Hence, the question is whether this flexibility in the choice of the constants is really essential to implement the core ideas of $\mathrm{LLFPL}_!$.

Going back to the §-modality? In analogy to $\mathrm{LLT}_!$ and the sketch of $\mathrm{LLT}_{!\S}$ in Figure A.1 on page 243, we can try to go back with $\mathrm{LLFPL}_!$ to a more traditional type system with the §-modality. In Section A.2 of the appendix we have sketched how $\mathrm{LLFPL}_{!\S}$ could look like.

The central point in that calculus would be the iteration rules $(L(\tau)_0^-)$ and $(L(\tau)_1^-)$. In $\mathrm{LLFPL}_!$ these two rules are a combination of the $(!_0), (!_1)$-rules and the $(]\cdot[_{!^n})$-rule. The latter is a relict of the (§)-rule of LAL (compare Section 6.1). Hence, the two rules $(L(\tau)_0^-)$ and $(L(\tau)_1^-)$ in $\mathrm{LLFPL}_{!\S}$ will be a combination of $(!_0)$,$(!_1)$ and (§).

Another crucial question is the typing of the $\mathbf{d}$-constant. In $\mathrm{LLFPL}_!$ it is not possible to use the $\mathbf{d}^{n+1}$-constant inside a $!^n$-box. If we use $\mathbf{d} : \S\Diamond$ in the calculus $\mathrm{LLFPL}_{!\S}$, we get the very same effect, namely that we cannot create a list of type $!L(\tau)$, but only $!\S L(\tau)$, for instance by

$$!\Big[\S\Big[\Big(\mathbf{cons}_\tau\ ]\mathbf{d}[_\S\ x\,\mathbf{nil}_\tau\Big)\Big]^{!\S L(\tau)}\Big] \quad .$$

The term

$$!\Big[\Big(\mathbf{cons}_\tau\ ]\mathbf{d}[_\S\ x\,\mathbf{nil}_\tau\Big)\Big]^{!L(\tau)}$$

is not typable. Moreover, the step term of an iteration cannot use the $\mathbf{d}$ freely, but only the one given as the first argument.

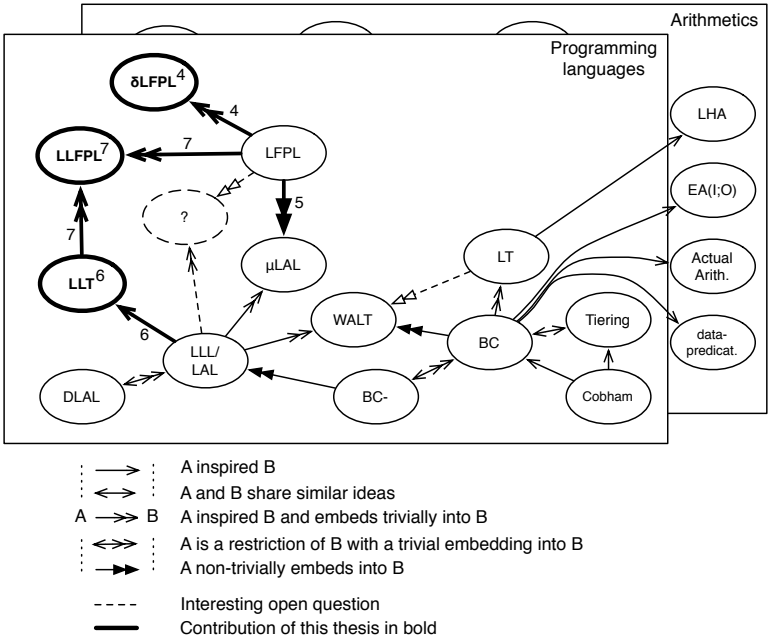


Figure 7.5.: The landscape of polynomial time calculi and arithmetics and the contribution of this thesis (the numbers are the corresponding chapters). Interesting open question: is there a LAL-variant which embeds LFPL, like $\mathrm{LLFPL}_!$ is for $\mathrm{LLT}_!$?

8 Conclusion and Outlook

In this chapter we give some conclusions on the results of this thesis and an outlook with worthwhile further directions of research. In contrast to the "Conclusion and Outlook" section of each technical Chapter 4-7, we step back a few steps here, in order to see the results from a greater distance. For the discussion of details, especially possible technical improvements, we refer to the "conclusion and outlook" of the respective chapter.

8.1. Summary of the results

We have given a detailed description of the contributions of this work in the introduction in Section 1.4. Without repeating ourselves with all the details, the (technical) contributions can be briefly summarised in the following way:

> This dissertation investigates the relationship of *impredicative non-size-increasing iteration* (of LFPL) and *light iteration* (of LAL):
>
> 1. It is shown that light iteration in LAL is expressive enough to allow an (non-trivial) compositional embedding of LFPL into the fixed point variant of LAL.

2. The pull-out trick of LAL is identified as a technique to type certain non-size-increasing algorithms in a way, that they can be iterated again. We introduce the System T-like calculus $\mathrm{LLT}_!$, which seamlessly integrates this technique as a central feature of the iteration scheme. We show that $\mathrm{LLT}_!$ is polynomial time normalisable and also complete for PTime.

3. While $\mathrm{LLT}_!$-iterations of the same level cannot be nested, we generalise $\mathrm{LLT}_!$ to $\mathrm{LLFPL}_!$ which can express *impredicative iterations* of LFPL and *light iteration* of $\mathrm{LLT}_!$, and hence subsumes both systems in one, still being polynomial time normalisable and complete.

In this summary, we intentionally leave out Chapter 4 because its immediate goal is somehow different. Still, it serves the purpose of an analysis of a LFPL-like system, *giving a deep understanding* of the normalisation of LFPL and the impredicative iteration. In this sense, the Chapter 4 is an important step which finally led to the understanding of the relation of LFPL and LAL.

8.1.1. The Bigger Picture

After stepping back, how do these results fit into the bigger picture of implicit complexity research? This thesis studies the two iteration schemes: *light iteration* and *impredicative non-size-increasing iteration*. Both are in the area of type systems for functional programming, i.e. λ-calculi.

Orthogonal iteration schemes The calculus $\mathrm{LLFPL}_!$ shows that the *impredicative iteration* of Hofmann's LFPL and *light iteration* of LAL are not two parallel and highly incompatible concepts. Instead, the iteration of $\mathrm{LLFPL}_!$ integrates both of them *in an orthogonal way*: the impredicative iteration can be used for non-size-increasing computations, on one level without using modalities at all. For size-increasing algorithms light iteration and the levels are available, giving asymmetric types with increasing levels.

From Light Affine Logic's point of view With the pull-out trick, LAL already supports *non-size-increasing* computation *partially*. E.g., the "insertion" function of Insertion Sort can be implemented with a symmetric type.

What is missing, is the ability to express *impredicative iteration nestings*. The contribution of LLFPL$_!$ to LLT$_!$, in this sense, is the lifting of the iteration onto the level of the lists such that those nestings become possible.

From LFPL's point of view Conversely, LFPL supports impredicative iteration. But, it is bound to the restriction to stay on one level, because there are no modalities or levels at all in the system. Inside a step term, no $\Diamond$-term can be "created" without free variables. In LLFPL$_!$ though, $\mathbf{d}^i : \Diamond^i$ can be used freely if the level i is above the level of the current iteration. Hence, from the point of view of LFPL, the calculus LLFPL$_!$ adds the ability to use levels: on each level the terms are still non-size-increasing. But, by going up in the level hierarchy, also size-increasing algorithms can be expressed.

8.2. Open Questions and Outlook

In the following we point out some open questions, which could give a direction of further research. Again, we refer to the "Conclusion and Outlook" sections at the end of each chapter for more detailed and technical discussions.

8.2.1. Going Back to the §-Modality

Both calculi, LLT$_!$ and LLFPL$_!$, are formulated using the concept of levels in order to replace the §-modality in the type system. What is left, is the !-modality and the artifacts of the (§)-rule, namely the $(]\cdot[_{!^n})$-rule. This design choice gives the type system a much different look compared to traditional calculi with the § like LAL. Clearly, an immediate question is whether this is really necessary:

Are there variations of LLT$_!$ and LLFPL$_!$ using the traditional §-modality and (§)-rule?

We think, that this is possible. In the appendix, the type systems LLT$_{!\S}$ (Section A.1) and LLFPL$_{!\S}$ (Section A.2) are sketched. Though, an analysis of these two proposals is still open. We refer to Section 6.6 and Section 7.4 for further explanations of these sketches.

In the same direction we have to ask whether the design choice, to use level instead of §, paid out in some way. Of course, the level concept might

help the intuition to think about boxes in a different way. Moreover, the §-boxes complicate the term (and proof net) structure. In this sense, the levels are a matter of taste whether one prefers levels in types or §-boxes in the terms.

The original hope has been though that *we get more freedom to choose the types of the constants with levels.* But, the sketches of $\mathrm{LLT}_{!\S}$ and $\mathrm{LLFPL}_{!\S}$ suggest that this has not been necessary, in fact, at least not for these two systems: all the constants of $\mathrm{LLT}_!$ and $\mathrm{LLFPL}_!$ can easily be translated into the setting *with the §-modality.*

8.2.2. Strong Normalisation of $\mathrm{LLT}_!$ and $\mathrm{LLFPL}_!$

Terui showed that Light Affine Logic is even polynomial time *strongly normalisable* [Ter01]. This means that the normalisation by levels is not necessary at all, but any reduction strategy will finish in polynomially many steps.

The original normalisation proofs for LAL [AR00, AR02] though were also based on normalisation by levels. We essentially followed them to prove polynomial time normalisation for $\mathrm{LLT}_!$ and $\mathrm{LLFPL}_!$. Hence, we have to ask the question:

Can $\mathrm{LLT}_!$ and $\mathrm{LLFPL}_!$ be normalised in polynomial time with an arbitrary reduction strategy?

For $\mathrm{LLT}_!$ For the former, i.e. $\mathrm{LLT}_!$, this is very improbable: the normalisation of $\mathrm{LLT}_!$ uses the unfolding of the iteration (compare Section 6.4) to simulate the substitution of the step term of Church numerals. Hence, it is not clear how strong normalisation would look like in this setting because at some point the number K_n must be chosen. At least in our proof, this number depends on the normalisation of the lower levels. Hence, it is not clear which K_n should be chosen in order to make the $\mathrm{CaseIt}^n_{l,k}$-term long enough.

For $\mathrm{LLFPL}_!$ In the case of $\mathrm{LLFPL}_!$ (or $\mathrm{LLFPL}_{!\S}$) the normalisation rules do not depend on *normalisation by levels.* Hence, here the question about strong normalisation is much more interesting. The main differences of $\mathrm{LLFPL}_!$, compared to LAL, are the impredicative iteration and the restriction that reductions under λ-abstraction are not allowed (the *λ-restriction*). The latter is necessary in order to get a good enough size measure estimate

(compare Section 7.3.4). Though, it is not clear whether the λ-restriction is really needed for polynomial time normalisation of LFPL. Maybe, a better, more precise measure of the length of lists is enough to drop the λ-restriction.

Is the λ-restriction necessary in LFPL and LLFPL$_!$?

For LLT$_!$ again, with iteration of LLFPL$_!$ One can also view LLFPL$_!$ as a more natural formulation of LLT$_!$ if one restricts LLFPL$_!$ to predicative iterations. Then, the reduction rules of the iteration constant do not use the *case distinction unfolding* of LLT$_!$, but instead very System T-like reduction rules. In this setting, the upper question about strong normalisation makes sense again. Moreover, this setting might be easier than the impredicative setting of LLFPL$_!$ for answering the question whether the λ-restriction is still necessary.

8.2.3. Ideas of LLFPL$_!$ in LAL

In Chapter 6 we turned LAL into a variant of System T with constants. We argued that this gives us more freedom in the choice of the constants (compared to a System F like calculus), especially the iteration constant. We exploited this freedom in Chapter 7 to lift the iteration into the level of the lists, leading to LLFPL$_!$.

Clearly, we had to pay for this freedom in the loss of the flexibility in choosing other *non-standard encodings* of data types, and in the loss of *polymorphism*. Especially the latter can be seen as a step backwards. Hence, we ask whether we can transfer the ideas of *light impredicative iteration* of LLFPL$_!$ back to LAL:

Is there a variant of LAL which allows impredicative non-size-increasing iteration?

This is maybe one of the most interesting further directions of this work. But at this time, it is not clear how this would look like. A guess is that there must be some kind of duplication of closed (or *practically-closed*) terms that is done without having a box around them. This would allow the duplication of step terms in a way that does not depend on the number of multiplexers in the term as in LAL.

8.2.4. DLLFPL

By restricting the occurrences of the !-modality to the left position of an arrow, Terui and Baillot [BT04] manage to define a λ-calculus variant DLAL of LAL, which does not need permutative conversions. The type system is simplified a lot in comparison to LAL. Of course, we can ask whether this can also be applied to LLFPL$_!$ or LLT$_{!\S}$ in a similar way.

Is there a Dual Light Linear Functional Programming Language?

8.2.5. Divide and Conquer

The calculi in the implicit complexity community mostly stick to *primitive recursion* or *primitive iteration*. We, with LLT$_!$ and LLFPL$_!$, are no exception. Many algorithms, which are used in practice, are not primitive recursive in their natural formulation. Clearly, a formulation of QuickSort, for instance, in LLT$_!$ is "somehow" possible, but not in an elegant way. Hence, we can ask for an extension of the calculi with *divide-and-conquer iteration*:

Can one add a divide-and-conquer iteration to LLT$_!$ or LLFPL$_!$?

The QuickSort algorithm is even non-size-increasing. Therefore, it could be, especially, an interesting question whether we can add divide-and-conquer capabilities to LLFPL$_!$.

8.2.6. BC + LLFPL, WALT

Recently, Roversi managed to embed full safe recursion into his LAL-variant WALT [Rov08a]. He essentially extends the control over the normalisation order when reducing the polynomial redexes. Can one learn anything from his work to do something similar in our setting of LLFPL$_!$?

Can one extend LLFPL$_!$ to allow the embedding of BC?

8.2.7. Complete Terms and Light Iteration

Another intuition in the direction of BC and LLFPL$_!$, which might be worthwhile to investigate, is the following: The central idea in the calculi which use predicative recursion (compare Section 1.3.1) is that recursion can only be done over those lists (or numbers) which are completely "available",

i.e. which can be computed by normalisation because all necessary data is given. In LT by Bellantoni and Schwichtenberg [SB01], this idea is explicitly visible in the definition of a *complete term*: a term is complete if all free variables are complete. This means that such a term can be computed up to a numeral if the values of the free variables are given.

How can this fit to light systems? Let t^τ be a (light) complete term of level n' which has only complete free variables of level $n \leq n' = \ell(\tau)$. Then this t could be used as a level n term because the numeral value of such a complete term can be computed by normalisation. In other words, this numeral could be "pulled down" from level $n' > n$ to level n after the normalisation has turned it into a numeral at higher level. I.e., a numeral of level $n' > n$ would then be handled like a numeral of level n. Clearly, this is a very vague idea. It would allow a combination of predicativity and a light iteration.

Can one combine the idea of predicativity with light iteration by reducing the levels of certain terms which can be computed to a numeral?

A

Calculi with the Traditional §-Modality

A.1. Sketch of $\mathrm{LLT}_{!\S}$

The following definition sketch a $\mathrm{LLT}_{!\S}$-calculus which uses the traditional §-modality and the corresponding §-boxes.

Definition A.1 (Types)**.** The set $\mathrm{Ty}_{\mathrm{LLT}_{!\S}}$ of linear types is defined inductively by:

$$\sigma, \tau \quad ::= \quad B \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid L(\sigma) \mid !\sigma \mid \S\sigma.$$

Definition A.2 (Constants, Typing)**.** The set $\mathrm{Cnst}_{\mathrm{LLT}_{!\S}}$ of $\mathrm{LLT}_{!\S}$ constants:

$$\begin{aligned}
\mathbf{tt}, \mathbf{ff} &: B \\
\mathbf{Case}_{\sigma} &: B \multimap \sigma \multimap \sigma \multimap \sigma \\
\mathbf{Case}_{\tau,\sigma} &: L(\tau) \multimap (\tau \multimap L(\tau) \multimap \sigma) \multimap \sigma \multimap \sigma \\
\mathbf{cons}_{\tau} &: !(\tau \multimap L(\tau) \multimap L(\tau)) \\
\mathbf{nil}_{\tau} &: L(\tau) \\
\otimes_{\tau,\rho} &: \tau \multimap \rho \multimap \tau \otimes \rho
\end{aligned}$$

$$\pi_\sigma : \tau \otimes \rho \multimap (\tau \multimap \rho \multimap \sigma) \multimap \sigma$$
$$\mathbf{It}_{\tau,\sigma} : \S L(\tau) \multimap !(\tau \multimap \sigma \multimap \sigma) \multimap \S(\sigma \multimap \sigma).$$

Replace the $(]\cdot[_{!^n})$-rule of LLT$_!$ with the (§)-rule

$$\frac{\overrightarrow{x^\rho}, \overrightarrow{y^\sigma} \vdash t^\tau \quad \overrightarrow{\Gamma \vdash r^{\S\rho}} \quad \overrightarrow{\Sigma \vdash s^{!\sigma}}}{\overrightarrow{\Gamma}, \overrightarrow{\Sigma} \vdash \S \boxed{\overrightarrow{x}, \overrightarrow{y} = \overrightarrow{]r[_\S}, \overrightarrow{]s[_!} \text{ in } t}^{\S\tau}} \ (\S)$$

while the other rules are like those of LLT$_!$ mutatis mutandis without the levels in the types.

A.2. Sketch of LLFPL$_{!\S}$

The following definitions sketch a LLFPL$_{!\S}$-calculus which uses the traditional §-modality and the corresponding §-boxes. Special attention is due for the **d**-constant and the iteration rules.

Definition A.3 (Types)**.** The set $\mathrm{Ty}_{\mathrm{LLFPL}_{!\S}}$ of linear types is defined inductively by:

$$\sigma, \tau \quad ::= \quad B \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid L(\sigma) \mid !\sigma \mid \S\sigma \mid \Diamond.$$

Definition A.4 (Constants, Typing)**.** The set $\mathrm{Cnst}_{\mathrm{LLFPL}_{!\S}}$ of LLFPL$_{!\S}$ constants:

$$\begin{aligned}
\mathbf{tt}, \mathbf{ff} &: B \\
\mathbf{Case}_\sigma &: B \multimap \sigma \multimap \sigma \multimap \sigma \\
\mathbf{Case}_{\tau,\sigma} &: L(\tau) \multimap (\Diamond \multimap \tau \multimap L(\tau) \multimap \sigma) \multimap \sigma \multimap \sigma \\
\mathbf{cons}_\tau &: \Diamond \multimap \tau \multimap L(\tau) \multimap L(\tau) \\
\mathbf{nil}_\tau &: L(\tau) \\
\mathbf{d} &: \S\Diamond \\
\otimes_{\tau,\rho} &: \tau \multimap \rho \multimap \tau \otimes \rho \\
\pi_\sigma &: \tau \otimes \rho \multimap (\tau \multimap \rho \multimap \sigma) \multimap \sigma.
\end{aligned}$$

Replace the $(]\cdot[_{!^n})$-rule of LLFPL$_!$ with the (§)-rule

$$\frac{\overrightarrow{x^\rho}, \overrightarrow{y^\sigma} \vdash t^\tau \quad \overrightarrow{\Gamma \vdash r^{\S\rho}} \quad \overrightarrow{\Sigma \vdash s^{!\sigma}}}{\overrightarrow{\Gamma}, \overrightarrow{\Sigma} \vdash \S\boxed{\overrightarrow{x}, \overrightarrow{y} =]\overrightarrow{r[_\S},]\overrightarrow{s[_!} \text{ in } t}^{\S\tau}} \ (\S)$$

and the iterations rules with

$$\frac{\Gamma \vdash l^{L(\tau)} \quad \emptyset \vdash t^{\Diamond \multimap \tau \multimap \sigma \multimap \sigma}}{\S\Gamma \vdash \S\boxed{(l\, t)}^{\S(\sigma \multimap \sigma)}} \ (L(\tau)_0^-)$$

$$\frac{\Gamma_1 \vdash l^{L(\tau)} \quad x^\rho \vdash t^{\Diamond \multimap \tau \multimap \sigma \multimap \sigma} \quad \Gamma_2 \vdash r^{!\rho}}{\S\Gamma_1, \Gamma_2 \vdash \S\boxed{x =]r[_! \text{ in } (l\, t)}^{\S(\sigma \multimap \sigma)}} \ (L(\tau)_1^-)$$

while the other rules are like those of LLFPL$_!$ mutatis mutandis without the levels in the types.

Index

W

List of Figures

Bibliography

[ABHS04] Klaus Aehlig, Ulrich Berger, Martin Hofmann, and Helmut Schwichtenberg. An arithmetic for non-size-increasing polynomial-time computation. *Theoretical Computer Science*, 318(1-2):3–27, 2004.

[ABT07] Vincent Atassi, Patrick Baillot, and Kazushige Terui. Verification of Ptime reducibility for system F terms via Dual Light Affine Logic. *Logical Methods in Computer Science*, 3(4), 2007.

[AJ05] Klaus Aehlig and Jan Johannsen. An elementary fragment of second-order lambda calculus. *ACM Transactions on Computational Logic (TOCL)*, 6(2):468–480, 2005.

[AL94] Andrea Asperti and Cosimo Laneve. Interaction Systems I - The theory of optimal reductions. *Mathematical Structures in Computer Science*, 4(4):457–504, 1994.

[AL96] Andrea Asperti and Cosimo Laneve. Interaction Systems II - The Practice of Optimal Reductions. *Theoretical Computer Science*, 159(2):191–244, 1996.

[AM04] Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In *Proceedings of Computer*

Science Logic, Lecture Notes in Computer Science, volume 3210, pages 190–204, 2004.

[AM05] Toshiyasu Arai and Georg Moser. Proofs of Termination of Rewrite Systems for Polytime Functions. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, volume 3821, pages 529–540, 2005.

[AR00] Andrea Asperti and Luca Roversi. Light Affine Logic. *arXiv*, cs/0006010, 2000.

[AR02] Andrea Asperti and Luca Roversi. Intuitionistic Light Affine Logic. *ACM Transactions on Computational Logic*, 3(1):137–175, 2002.

[AS00] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *Proceedings of Logic in Computer Science*, pages 84–91, 2000.

[AS02] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *ACM Transactions on Computational Logic*, 3(3):383–401, 2002.

[Bai06] Patrick Baillot. Linear logic, lambda-calculus and polynomial time complextixy. *GEOCAL Winter School, Marseille*, 2006.

[Bar92] Henk Barendregt. Lambda calculi with types. *Handbook of Logic in Computer Science*, 1992.

[Bar96] Andrew Barber. Dual Intuitionistic Linear Logic. 1996.

[BB05] Henk Barendregt and Erik Barendsen. Introduction to Lambda Calculus. *manuscript*, 2005.

[BC92] Stephen Bellantoni and Stephen Arthur Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[BCdNM03] Olivier Bournez, Felipe Cucker, Paulin Jacobé de Naurois, and Jean-Yves Marion. Safe Recursion Over an Arbitrary Structure: PAR, PH and DPH. *Electronic Notes in Theoretical Computer Science*, 90(1), 2003.

[BCL07] Patrick Baillot, Paolo Coppola, and Ugo Dal Lago. Light Logics and Optimal Reduction: Completeness and Complexity. In *Proceedings of Logic in Computer Science*, pages 421–430, 2007.

[Bec01] Arnold Beckmann. Exact Bounds for Lengths of Reductions in Typed Lambda-Calculus. *The Journal of Symbolic Logic*, 66(3):1277–1285, 2001.

[Bel92] Stephen Bellantoni. Predicative Recursion and Computational Complexity. Ph.D. thesis, University of Toronto, 1992.

[Ben01] Ralph Benzinger. Automated Complexity Analysis of NuPRL Extracts. *Journal of Functional Programming*, 11(1):3–31, 2001.

[Ben04] Ralph Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79–103, 2004.

[BH02] Stephen Bellantoni and Martin Hofmann. A New "Feasible" Arithmetic. *Journal of Symbolic Logic*, 67(1):104–116, 2002.

[BJdM97] Richard S Bird, Geraint Jones, and Oege de Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5):541–547, 1997.

[BM04] Patrick Baillot and Virgile Mogbil. Soft lambda-calculus: A Language for Polynomial Time Computation. In *Proceedings of Foundations of Software Science and Computation Structures*, volume 2987, 2004.

[BMM05] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretation: a way to control resources. *Theoretical Computer Science*, 2005.

[BNS00] Stephen Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1-3):17–30, 2000.

[BT04] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *Proceedings of Logic in Computer Science*, pages 266–275, 2004.

[BT05] Patrick Baillot and Kazushige Terui. A Feasible Algorithm for Typing in Elementary Affine Logic. In *Proceedings of Typed Lambda Calculi and Applications, Lecture Notes of Computer Science*, volume 3461, pages 55–70, 2005.

[BT07] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus (long version). *manuscript*, 2007.

[BW96] Arnold Beckmann and Andreas Weiermann. A term rewriting characterization of the polytime functions and related complexity classes. *Archive for Mathematical Logic*, 36(1):11–30, 1996.

[BW00] Arnold Beckmann and Andreas Weiermann. Characterizing the elementary recursive functions by a fragment of Gödel's T. *Archive for Mathematical Logic*, 39(7):475–491, 2000.

[Cas96a] Vuokko-Helena Caseiro. An Equational Characterization of the Poly-time Functions on any Constructor Data Structure. *citeseer*, 1996.

[Cas96b] Vuokko-Helena Caseiro. Criticality Conditions on Equations to Ensure Poly-time Functions. 1996.

[Cas97] Vuokko-Helena Caseiro. Equations for Defining Poly-time Functions. Ph.D. thesis, University of Oslo, 1997.

[Chu40] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[Cob65] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of Logic, Methodology, and Philosophy of Science*, volume II, pages 24–30, 1965.

[Col01] Loïc Colson. Functions versus algorithms. *Current Trends in Theoretical Computer Science*, pages 343–362, 2001.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–522, 1985.

[DC92] Vincent Danos and Roberto Di Cosmo. The linear logic primer. *manuscript*, 1992.

[DJ03] Vincent Danos and Jean-Baptiste Joinet. Linear logic and elementary time. *Information and Computation*, 183(1):123–137, 2003.

[dRG06] Simona Ronchi della Rocca and Marco Gaboardi. Soft Linear Lambda-Calculus and Intersection Types. *GEOCAL Workshop, Marseille*, 2006.

[Ehr04] Thomas Ehrhard. Linear Logic in Computer Science. *Cambridge University Press, New York, NY, USA*, 2004.

[GAL92a] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. Linear logic without boxes. In *Proceedings of Logic in Computer Science*, pages 223–234, 1992.

[GAL92b] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of Principles of Programming Languages*, pages 15–26, 1992.

[Geu05] Herman Geuvers. Lambda Cube and Pure Type Systems. *Types Summer School, Göteborg*, 2005.

[Gir72] Jean-Yves Girard. Interpétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Ph.D. thesis, Université de Paris VII, 1972.

[Gir95] Jean-Yves Girard. Linear logic: its syntax and semantics. In *Proceedings of the Workshop on Linear Logic, Ithaca*, pages 1–42, 1995.

[Gir98] Jean-Yves Girard. Light Linear Logic. *Information and Computation*, 143(2):175–204, 1998.

[Gli05] Johan Glimming. System F. *manuscript*, 2005.

[GLT88] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1988.

[Göd58] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[Gue99] Stefano Guerrini. A general theory of sharing graphs. *Theoretical Computer Science*, 227(1-2):99–151, 1999.

[Gue03] Stefano Guerrini. Coherence for sharing proof-nets. *Theoretical Computer Science*, 294(3):379–409, 2003.

[Gue04] Stefano Guerrini. Proof Nets and the Lambda-Calculus. *London Mathematical Society Lecture Note Series*, 316:65–118, 2004.

[Hof98] Martin Hofmann. A Mixed Modal/Linear Lambda Calculus with Applications to Bellantoni-Cook Safe Recursion. In *Proceedings of Computer Science Logic, Lecture Notes of Computer Science*, volume 584, pages 275–294, 1998.

[Hof99a] Martin Hofmann. Linear Types and Non-Size-Increasing Polynomial Time Computation. In *Proceedings of Logic in Computer Science*, pages 464–473, 1999.

[Hof99b] Martin Hofmann. Type Systems For Polynomial-time Computation. Habilitation thesis, Technische Universität Darmstadt, 1999.

[Hof00a] Martin Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(1):31–42, 2000.

[Hof00b] Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 1-3:113–166, 2000.

[Hof02] Martin Hofmann. The strength of non-size increasing computation. In *Proceedings of Principles of Programming Languages*, pages 260–269, 2002.

[Hof06] Martin Hofmann. Non-Size-Increasing Computation. *GEOCAL Winter School, Marseille*, 2006.

[KJ05] Lars Kristiansen and Neil D Jones. The Flow of Data and the Complexity of Algorithms. *Computability in Europe, Lecture Notes in Computer Science*, 3526:263–274, 2005.

[Kri08] Lars Kristiansen. Implicit Characterisations of Complexity Classes and recursion in higher types. *NoCOST workshop, Paris*, 2008.

[KV03] Lars Kristiansen and Paul J Voda. The Surprising Power of Restricted Programs and Gödel's Functionals. In *Proceedings of Computer Science Logic, Lecture Notes in Computer Science*, volume 2803, pages 345–358, 2003.

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004.

[Lag03] Ugo Dal Lago. On the Expressive Power of Light Affine Logic. In *Proceedings of the Italian Conference of Theoretical Computer Science, Lecture Notes of Computer Science*, volume 2841, pages 216–227, 2003.

[Lag06a] Ugo Dal Lago. Context semantics, linear logic and computational complexity. In *Proceedings of Logic in Computer Science*, pages 169–178, 2006.

[Lag06b] Ugo Dal Lago. Context Semantics, Linear Logic and Implicit Complexity. *GEOCAL Winter School, Marseille*, 2006.

[Lag06c] Ugo Dal Lago. Semantic Frameworks for Implicit Computational Complexity. Ph.D. thesis, Università degli Studi di Bologna, 2006.

[LB06] Ugo Dal Lago and Patrick Baillot. On light logics, uniform encodings and polynomial time. *Mathematical Structures in Computer Science*, 16(4):713–733, 2006.

[Lei91] Daniel Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93–113, 1991.

[Lei94a] Daniel Leivant. A foundational delineation of poly-time. *Information and Computation*, 110(2):391–420, 1994.

[Lei94b] Daniel Leivant. Predicative Recurrence in Finite Types. In *Proceedings of Logical Foundations of Computer Science, Lecture Notes in Computer Science*, volume 813, pages 227–239, 1994.

[Lei95a] Daniel Leivant. Intrinsic Theories and Computational Complexity. In *Proceedings of Logical and Computational Complexity, Lecture Notes in Computer Science*, volume 960, pages 117–194, 1995.

[Lei95b] Daniel Leivant. Ramified Recurrence and Computational Complexity I: Word Recurrence and Poly-time. In *Proceedings of Feasible Mathematics II, Birkhäuser*, pages 320–342, 1995.

[Lei98] Daniel Leivant. Applicative control and computational complexity. In *Proceedings of Computer Science Logic, Lecture Notes in Computer Science*, volume 1683, pages 82–95, 1998.

[Lei99] Daniel Leivant. Ramified recurrence and computational complexity III: Higher type recurrence and elementary complexity. *Annals of Pure and Applied Logic*, 96(1-3):209–229, 1999.

[Lei01] Daniel Leivant. Termination Proofs and Complexity Certification. In *Proceedings of Theoretical Aspects of Computer Software, Lecture Notes in Computer Science*, volume 2215, pages 183–200, 2001.

[Lei02] Daniel Leivant. Intrinsic reasoning about functional programs I: first order theories. *Annals of Pure and Applied Logic*, 114(1-3):117–153, 2002.

[LH05] Ugo Dal Lago and Martin Hofmann. Quantitative Models and Implicit Complexity. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, volume 3821, pages 189–200, 2005.

[LM93] Daniel Leivant and Jean-Yves Marion. Lambda Calculus Characterizations of Poly-Time. *Fundamenta Informaticae*, 19(1/2):167–184, 1993.

[LM95] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In *Proceedings of Computer Science Logic, Lecture Notes in Computer Science*, volume 933, pages 486–500, 1995.

[LM98] Daniel Leivant and Jean-Yves Marion. Ramified Recurrence and Computational Complexity IV: Predicative functionals and Poly-space. *unpublished*, 1998.

[LMR04] Ugo Dal Lago, Simone Martini, and Luca Roversi. Higher-Order Linear Ramified Recurrence. In *Proceedings of Types For Proofs And Programs, Jouy-en-Josas, Lecture Notes in Computer Science*, volume 3085, pages 178–193, 2004.

[Mai02] Harry G. Mairson. From Hilbert Spaces to Dilbert Spaces: Context Semantics Made Simple. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, volume 2556, pages 2–17, 2002.

[Mai03] Harry G. Mairson. From Hilbert space to Dilbert space: Context semantics as a language for games and flow analysis. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2003.

[Mar01] Jean-Yves Marion. Actual Arithmetic and Feasibility. In *Proceedings of Computer Science Logic, Lecture Notes in Computer Science*, volume 2142, pages 115–129, 2001.

[Mar06a] Jean-Yves Marion. Ramification. *GEOCAL Winter School, Marseille*, 2006.

[Mar06b] Jean-Yves Marion. Rewriting systems and quasi-interpretation. *GEOCAL Winter School, Marseille*, 2006.

[Mat98] Ralph Matthes. Monotone fixed-point types and strong normalization. In *Proceedings of Computer Science Logic, Lecture Notes in Computer Science*, volume 1584, pages 298–312, 1998.

[Mat99] Ralph Matthes. Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types. Ph.D. thesis, LMU München, 1999.

[Maz08] Damiano Mazza. Linear Logic by Levels and bounded time complexity. *NoCOST workshop, Paris*, 2008.

[MdPR] Maria Emilia Maietti, Valeria de Paiva, and Eike Ritter. Normalization bounds in rudimentary linear lambda calculus. *unpublished*.

[Men88] Paul Francis Mendler. Inductive Definition in Type Theory. Ph.D. thesis, Cornell University, 1988.

[Miq05a] Alexandre Miquel. Normalisation of Second Order Arithmetic. *Types Summer School, Göteborg*, 2005.

[Miq05b] Alexandre Miquel. System F. *Types Summer School, Göteborg*, 2005.

[MO00] Andrzej S Murawski and C.-H. L. Ong. Can safe recursion be interpreted in light logic. In *Proceedings of the Workshop on Implicit Computational Complexity, Santa Barbara*, 2000.

[MO04] Andrzej S Murawski and C.-H. L. Ong. On an interpretation of safe recursion in light affine logic. *Theoretical Computer Science*, 318(1-2):197–223, 2004.

[MR67] Albert R Meyer and Dennis M Ritchie. The complexity of loop programs. *ACM Annual Conference*, pages 465–469, 1967.

[Nee04a] Peter Neergaard. BC-epsilon-minus: A Recursion-Theoretic Characterization of. *unpublished*, 2004.

[Nee04b] Peter Neergaard. Complexity Aspects of Programming Language Design. Ph.D. thesis, Brandeis University, 2004.

[Nel97] Edward Nelson. Ramified Recursion and Intuitionism. *manuscript*, 1997.

[NM02] Peter Neergaard and Harry G. Mairson. LAL is square: Representation and expressiveness in light affine logic. In *Proceedings of the Workshop on Implicit Computational Complexity, Copenhagen*, 2002.

[NM03] Peter Neergaard and Harry G. Mairson. How light is safe recursion? compositional translations between languages of polynomial time. *Unpublished Notes*, 2003.

[Nor05] Bengt Nordström. Types, Propositions and Problems - an introduction to type theoretical ideas. *Types Summer School, Göteborg*, 2005.

[OW05a] Geoffrey E Ostrin and Stanley S Wainer. Complexity in Predicative Arithmetic. In *Proceedings of New Computational Paradigms, Conference on Computability in Europe, Lecture Notes in Computer Science*, volume 3526, pages 378–384, 2005.

[OW05b] Geoffrey E Ostrin and Stanley S Wainer. Elementary arithmetic. *Annals of Pure and Applied Logic*, 133(1-3):275–292, 2005.

[Rey74] John C Reynolds. Towards a theory of type structure. In *Proceedings of the Symposium on Programming, Lecture Notes in Computer Science*, volume 19, pages 408–423, 1974.

[Rov02] Luca Roversi. Light Linear Logic and Programming Languages. *slides*, page 21, 2002.

[Rov03] Luca Roversi. An introduction to intuitionistic light affine logic. *Mini (Doctoral) School Chambéry/Turin of Theoretical Computer Science*, 2003.

[Rov08a] Luca Roversi. Weak Affine Light Typing: Polytime intensional expressivity, soundness and completeness. *arXiv*, 0712.4222, 2008.

[Rov08b] Luca Roversi. Weak Affine Light Typing (WALT). *NoCOST workshop, Paris*, 2008.

[Rus03] Bertrand Russell. Appendix B: The Doctrine of Types. *Principles of Mathematics, Cambridge University Press*, pages 523–528, 1903.

[Rus08] Bertrand Russell. Mathematical Logic as Based on the Theory of Types. *American Journal of Mathematics*, 30:222–262, 1908.

[SB01] Helmut Schwichtenberg and Stephen Bellantoni. Feasible Computation with Higher Types, 2001.

[Sch05] Stefan Schimanski. A quasi-linear typed term system for PTIME computation, 2005.

[Sch06] Helmut Schwichtenberg. An arithmetic for polynomial-time computation. *Theoretical Computer Science*, 318(1-2):3.27, 2006.

[Sha06] Jatin Shah. Fundamental issues in representing NP-complete problems. Ph.D. thesis, Yale University, 2006.

[Str97] Thomas Strahm. Polynomial Time Operations in Explicit Mathematics. *Journal of Symbolic Logic*, 62(2):575–594, 1997.

[Str04] Thomas Strahm. A proof-theoretic characterization of the basic feasible functionals. *Theoretical Computer Science*, 329(1-3):159–176, 2004.

[Tai67] William W Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.

[Ter] Kazushige Terui. A Translation of LAL into DLAL, a short note. *unpublished*.

[Ter01] Kazushige Terui. Light Affine Calculus and Polytime Strong Normalization. In *Proceedings of Logic in Computer Science*, 2001.

[Ter02] Kazushige Terui. Light Logic and Polynomial Time Computation. Ph.D. thesis, Keio University, 2002.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[Tur04] D. A Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, 1989.

[Wei98] Andreas Weiermann. How is it that infinitary methods can be applied to finitary mathematics? Gödel's T: a case study. *Journal of Symbolic Logic*, 63(4):1348–1370, 1998.

Acknowledgments

I am thankful for the support from the Graduiertenkolleg "Logic in Computer Science" which was financed by the Deutsche Forschungsgemeinschaft and the state of Bavaria. Being part of it gave me a very broad and inspiring view on the subject and many areas around it. I have to thank especially my supervisor Prof. Schwichtenberg for giving me the chance to be part of this environment and the logic group in Munich, and for his endless patience in listening to my woolly thoughts.

Moreover, I want to thank my second adviser, Prof. Hofmann from the Computer Science department, for his support, the year of funding, but more importantly for his help when meeting him, through his immediate comprehension of my new vague ideas and his ability to ask the right questions.

I also have to send my thanks to Frank Stamm, who gave me – a long, long ago, still in Clausthal – this survey paper by some M. Hofmann [Hof00a] about polynomial time programming languages, which should eventually become my Ph.D. topic.

When thinking about my Ph.D. time in a few years' time, the first thing which will most likely come to my mind will be the office with Diana and Freiric. They were not only very good colleagues, open to my cryptic explanations and drawings of boxes and lambda terms on the board, long

before they were remotely presentable to another audience. But more importantly, during all that time, they and also Dan – Diana's husband – became good friends, with whom one could talk about things outside the University, and who always had encouraging words for me when necessary during yet another Ph.D. crisis.

I owe many thanks to the good colleagues and friends in the working group, Basil, Bogomil, Dominik, Florian, Klaus, Luca, Markus, Martin, Pierre, Sebastian, Simon, Trifon and everybody I forgot, making the time in Munich what it was, especially with our local self-help group "the Underground seminar" for disoriented Ph.D. students.

I want to thank Klaus Thiel for his endless good mood, energy to make up plans for the weekends and evenings which eventually brought us all much nearer, turning colleagues into friends. I remember a lot of barbecues, Döners, beers, Oktoberfest visits, trips, movies and pizzas, discussion about the meaning of life, intuitionism, truth and the pros and contras of being a vegetarian. Our group climate would not have been what it was without him.

I have to thank Luca for inviting me to his meeting with the Italians on a lonely evening, promising me that there will be some Germans as well. Meeting my Clelia there changed my life and made me smile uncountable times for no apparent reason when sitting in the office and working on this text. Without her, I might not be where I am. Moreover, without her patience, especially during the last months, this work would not have been possible.

Last but not least, I have to thank my family: my parents, my sisters and my brother for always keeping the faith and encouraging me to go my way. I was always happy to see each of them again, and the other way around, when travelling the long way back to Wolfenbüttel, and I feel lucky to have such a family.

Finally, I wish to thank everybody who contributed comments to earlier versions of this text, especially Dan and Diana for their many, many remarks and for fighting my run-on sentences and missing commas.

Curriculum Vitae

Birthday	08/04/1979, Wolfenbüttel, Germany
1998	Abitur, Gymnasium Große Schule, Wolfenbüttel
1998	Software development, Pinnacle Systems, Braunschweig
1998-1999	Military service, PzGrenKp 4/332, Wesendorf
1999	Software development, Pinnacle Systems, Braunschweig
1999-2004	Undergraduate studies in Computer Science, TU Clausthal
2000-2004	Undergraduate studies in Mathematics, TU Clausthal
2000-2001	Software Development, Caldera International, Erlangen
2001	Erasmus semester, Queen's University of Belfast, Northern Ireland
2004	Diploma in Computer Science, with honour, TU Clausthal
2004	Diploma in Mathematics, with honour, TU Clausthal
2004-2006	Fellow of the Post Graduate Program Graduiertenkolleg "Logic in Computer Science", LMU
2006-2007	Research Assistant, Dept. of Computer Science, LMU
2007-2008	Research Assistant, Dept. of Mathematics, LMU
2009-	Software Development, DFS Deutsche Flugsicherung, Langen