Introduction to Type Theory

Herman Geuvers

Radboud University Nijmegen, The Netherlands
Technical University Eindhoven, The Netherlands

1 Overview

These notes comprise the lecture "Introduction to Type Theory" that I gave at the Alpha Lernet Summer School in Piriapolis, Uruguay in February 2008. The lecture was meant as an *introduction* to typed λ-calculus for PhD. students that have some (but possibly not much) familiarity with logic or functional programming. The lecture consisted of 5 hours of lecturing, using a beamer presentation, the slides of which can be found at my homepage[1]. I also handed out exercises, which are now integrated into these lecture notes.

In the lecture, I attempted to give an introductory overview of type theory. The problem is: there are so many type systems and so many ways of defining them. Type systems are used in programming (languages) for various purposes: to be able to find simple mistakes (e.g. caused by typing mismatches) at compile time; to generate information about data to be used at runtime, ... But type systems are also used in theorem proving, in studying the the foundations of mathematics, in proof theory and in language theory.

In the lecture I have focussed on the use of type theory for compile-time checking of functional programs and on the use of types in proof assistants (theorem provers). The latter combines the use of types in the foundations of mathematics and proof theory. These topics may seem remote, but as a matter of fact they are not, because they join in the central theme of these lectures:

Curry-Howard isomorphism of formulas-as-types
(and proofs-as-terms)

This isomorphism amounts to two readings of typing judgments

$$M : A$$

- M is a term (program, expression) of the data type A
- M is a proof (derivation) of the formula A

The first reading is very much a "programmers" view and the second a "proof theory" view. They join in the implementation of proof assistants using type systems, where a term (proof) of a type (formula) is sought for interactively between the user and the system, and where terms (programs) of a type can also be used to define and compute with functions (as algorithms).

[1] `url: http:://www.cs.ru.nl/H.Geuvers/Uruguay2008SummerSchool.html/`

For an extensive introduction into the Curry-Howard isomorphism, we refer to [39].

The contents of these notes is as follows.

1. Introduction: what are types and why are they not sets?
2. Simply typed λ-calculus (Simple Type Theory) and the Curry Howard isomorphism
3. Simple Type Theory: "Curry" type assignment, principle type algorithm and normalization
4. Polymorphic type theory: full polymorphism and ML style polymorphism
5. Dependent type theory: logical framework and type checking algorithm

In the course, I have also (briefly) treated higher order logic, the λ-cube, Pure Type Systems and inductive types, but I will not do that here. This is partly because of space restrictions, but mainly because these notes should be of a very introductory nature, so I have chosen to treat lesser things in more detail.

2 Introduction

2.1 Types and sets

Types are not sets. Types are a bit like sets, but types give syntactic information, e.g.

$$3 + (7 * 8)^5 : \mathsf{nat}$$

whereas sets give semantic information, e.g.

$$3 \in \{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)\}$$

Of course, the distinction between syntactical and semantical information can't always be drawn that clearly, but the example should be clear: $3 + (7 * 8)^5$ is of type nat simply because 3, 7 and 8 are natural numbers and $*$ and $+$ are operations on natural numbers. On the other hand, $3 \in \{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)\}$, because there are no positive x, y, z such that $x^n + y^n = z^n$. This is an instance of 'Fermat's last Theorem', proved by Wiles. To establish that 3 is an element of that set, we need a *proof*, we can't just read it off from the components of the statement. To establish that $3 + (7 * 8)^5 : \mathsf{nat}$ we don't need a proof but a *computation*: our "reading the type of the term" is done by a simple computation.

One can argue about what can be "just read off", and what not; a simple criterion may be whether there is an algorithm that establishes the fact. So then we draw the line between "is of type" (:) and "is an element of" ($\in$) as to whether the relation is decidable or not. A further refinement may be given by arguing that a type checking algorithm should be of low complexity (or compositional or syntax directed).

There are very many different type theories and also mathematicians who base their work on set theory use types as a *high level ordering mechanism*, usually in an informal way. As an example consider the notion of *monoid*, which is

defined as a tuple $\langle A, \cdot, e\rangle$, where A is a set, $\cdot$ a binary operation on A and e an element of A, satisfying the monoidic laws. In set theory, such an ordered pair $\langle a, b\rangle$ is typically defined as $\{\{a\}, \{a, b\}\}$, and with that we can define ordered triples, but one usually doesn't get into those details, as they are irrelevant *representation issues*: the only thing that is relevant for an ordered pair is that one has *pairing* and *projection* operators, to create a pair and to take it apart. This is exactly how an ordered pair would be defined in type theory: if A and B are types, then $A \times B$ is a type; if $a : A$ and $b : B$, then $\langle a, b\rangle : A \times B$; if $p : A \times B$, then $\pi_1\, p : A$, $\pi_2\, t : B$ and moreover $\pi_1\langle a, b\rangle = a$ and $\pi_2\langle a, b\rangle = b$. So mathematicians use a kind of high level typed language, to avoid irrelevant representation issues, even if they may use set theory as their foundation. However, this high level language plays a more important role than just a language, as can be seen from the problems that mathematicians study: whether $\sqrt{2}$ is an element of the set π is not considered a relevant question. A mathematician would probably not even considered this as a meaningful question, because *the types don't match*: π isn't a set but a number. (But in set theory, everything is a set.) Whether $\sqrt{2} \in \pi$ depends on the actual representation of the real numbers as sets, which is quite arbitrary, so the question is considered irrelevant.

We now list a number of issues and set side by side how set theory and type theory deal with them.

Collections Sets are "collections of things", where the things themselves are again sets. There are all kinds of ways for putting things together in a set: basically (ignoring some obvious consistency conditions here) one can just put all the elements that satisfy a property together in a set. Types are collections of objects of the same intrinsic nature or the same structure. There are specific ways of forming new types out of existing ones.

Existence Set theory talks about what things *exist*. The infinity axiom states that an infinite set exists and the power set axiom states that the set of subsets of a set exists. This gives set theory a clear *foundational* aspect, apart from its informal use. It also raises issues whether a "choice set" exists (as stated by the axiom of choice) and whether *inaccessible cardinals* exist. (A set X such that for all sets Y with $|Y| < |X|$, $|2^Y| < |X|$.) Type theory talks about how things can be *constructed* (syntax, expressions). Type theory defines a formal *language*. This puts type theory somewhere in between the research fields of software technology and proof theory, but there is more: being a system describing what things can be *constructed*, type theory also has something to say about the *foundations of mathematics*, as it also – just like set theory – describes what exists (can be constructed) and what not.

Extensionality versus intensionality Sets are extensional: Two sets are equal if they contain the same elements. For example $\{n \in \mathbb{N} \mid \exists x, y, z \in \mathbb{N}^+(x^n + y^n = z^n)\} = \{0, 1, 2\}$. So set equality is undecidable. In general it requires a proof to

establish the equality of two sets. Types are intensional[2]. Two types are equal if they have the same *representation*, something that can be verified by simple syntactic considerations. So, $\{n \mid \exists x, y, z : \mathsf{nat}^+(x^n + y^n \neq z^n)\} \quad \neq \quad \{n \mid n = 0 \vee n = 1 \vee n = 2\}$ because these two types don't have the same representation. Of course, one may wonder what types these are exactly, or put differently, what an object of such a type is. We'll come to that below.

Decidability of $:$*, undecidability of* $\in$ Membership is undecidable in set theory, as it requires a proof to establish $a \in A$. Typing (and type checking) is decidable[3]. Verifying whether M is of type A requires purely syntactic methods, which can be cast into a *typing algorithm*. As indicated before, types are about syntax: $3 + (7 * 8)^5 : \mathsf{nat}$, because 3, 7, 8 are of type nat and the operations take objects of type nat to nat. Similarly, $\frac{1}{2}\Sigma_{n=0}^{\infty} 2^{-n} : \mathbb{N}$ is not a typing judgment, because one needs additional information to know that the sum is divisible by 2.

The distinction between syntax and semantics is not always as sharp as it seems. The more we know about semantics (a model), the more we can formalize it and "turn it into syntax". For example, we can turn

$$\{n \in \mathbb{N} \mid \exists x, y, z \in \mathbb{N}^+(x^n + y^n = z^n)\}$$

into a (syntactic) type , with decidable type checking , if we take as its terms pairs

$$\langle n, p\rangle : \{n : \mathsf{nat} \mid \exists x, y, z : \mathsf{nat}^+(x^n + y^n = z^n)\}$$

where p is a proof of $\exists x, y, z \in \mathsf{nat}^+(x^n + y^n = z^n)$. If we have decidable proof checking, then it is decidable whether a given pair $\langle n, p\rangle$ is typable with the above type or not.

In these notes, we will study the formulas-as-types and proof-as-terms embedding, which gives syntactic representation of proofs that can be *type checked* to see whether they are correct and what formula (their type) they prove. So with such a representation, proof checking is certainly decidable. We can therefore summarize the difference between set theory and type theory as the difference between *proof checking* (required to check a typing judgment), which is decidable and *proof finding* (which is required to check an element-of judgment) which is not decidable.

2.2 A hierarchy of type theories

In this paper we describe a numbers of type theories. These could be described in one framework of the λ-cube or Pure Type Systems, but we will not do that here. For the general framework we refer to [5, 4]. This paper should be seen

[2] But the first version of Martin-Löf's type theory is extensional – and hence has undecidable type checking. This type theory is the basis of the proof assistant Nuprl[10].

[3] But there are type systems with undecidable type checking, for example the Curry variant of system F (see Section 5.2). And there are more exceptions to this rule.

(and used) as a very introductory paper, that can be used as study material for researchers interested in type theory, but relatively new to the field.

Historically, untyped λ-calculus was studied in much more depth and detail (see [3]) before the whole proliferation of types and related research took off. Therefore, in overview papers, one still tends to first introduce untyped λ-calculus and then the typed variant. However, if one knows nothing about either subjects, typed λ-calculus is more natural then the untyped system, which – at first sight – may seem like a pointless token game with unclear semantics. So, we start off from the simply typed λ- calculus.

The following diagrams give an overview of the lectures I have given at the Alfa Lernet Summer School. The first diagram describes simple type theory and polymorphic type theory, which comes in two flavors: à la Church and à la Curry. Apart from that, I have treated a weakened version of $\lambda 2$, corresponding to polymorphism in functional languages like ML. This will also be discussed in this paper. A main line of thought is the formulas-as-types embedding, so the corresponding logics are indicated in the left column.

The second diagram deals with the extension with dependent types. In the lectures I have treated all systems in the diagram, but in these notes, only the first row will be discussed: first order dependent type theory $\lambda\mathsf{P}$ and two ways of interpreting logic into it: a *direct encoding* of minimal predicate logic and a *logical framework* encoding of many different logics. The first follows the Curry-Howard version of the formulas-as-types embedding, which we also follow for $\lambda\!\!\rightarrow$ and $\lambda 2$. The second follows De Bruijn's version of the formulas-as-types embedding, where we encode a logic in a context using dependent types. The rest of the diagram is not treated here, but it is treated on the slides[4].

Logic		TT a la Church	Also known as	TT a la Curry
PROP	$\overset{\text{f-as-t}}{\longrightarrow}$	$\lambda\!\!\rightarrow$	STT	$\lambda\!\!\rightarrow$
PROP2	$\overset{\text{f-as-t}}{\longrightarrow}$	$\lambda 2$	system F	$\lambda 2$

					Remarks
PRED	$\overset{\text{f-as-t}}{\longrightarrow}$	$\lambda\mathsf{P}$	LF	$\overset{\text{f-as-t}}{\longleftarrow}$	Many logics
HOL	$\overset{\text{f-as-t}}{\longrightarrow}$	$\lambda\mathsf{HOL}$			language of HOL is STT
HOL	$\overset{\text{f-as-t}}{\longrightarrow}$	CC	Calc. of Constr.		
		PTS			different PTSs for HOL

3 Simple type theory $\boldsymbol{\lambda\!\!\rightarrow}$

In our presentation of the simple type theory, we have just arrow types. This is the same as the original system of [9], except for the fact that we allow type variables, whereas Church starts form two base types ι and o. A very natural

[4] `url: http:://www.cs.ru.nl/H.Geuvers/Uruguay2008SummerSchool.html/`

extension is the one with product types and possibly other type constructions (like sum types, a unit type, ...). A good reference for the simple type theory extended with product types is [27].

Definition 1. *The types of* $\lambda\!\!\rightarrow$ *are*

$$\mathsf{Typ} := \mathsf{TVar} \mid (\mathsf{Typ}\!\rightarrow\!\mathsf{Typ})$$

where TVar *denotes the countable set of type variables.*

Convention 2
- *Type variables will be denoted by* $\alpha, \beta, \gamma, \ldots$. *Types will be denoted by* $\sigma, \tau, \ldots$.
- *In types we let* brackets associate to the right *and we* omit outside brackets*:* $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!(\beta\!\rightarrow\!\gamma)\!\rightarrow\!\alpha\!\rightarrow\!\gamma$ *denotes* $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!((\beta\!\rightarrow\!\gamma)\!\rightarrow\!(\alpha\!\rightarrow\!\gamma))$

Example 1. The following are types: $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!\alpha$, $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!((\beta\!\rightarrow\!\gamma)\!\rightarrow\!(\alpha\!\rightarrow\!\gamma))$. Note the higher order structure of types: we read $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!\alpha$ as the *type of functions that take functions from* α *to* β *to values of type* α.

Definition 3. *The* terms *of* $\lambda\!\!\rightarrow$ *are defined as follows*

- *There are countably many typed variables* $x_1^\sigma, x_2^\sigma, \ldots$, *for every* σ.
- *Application: if* $M : \sigma\!\rightarrow\!\tau$ *and* $N : \sigma$, *then* $(M\,N) : \tau$
- *Abstraction: if* $P : \tau$, *then* $(\lambda x^\sigma.P) : \sigma\!\rightarrow\!\tau$

So the binary application operation is not written. One could write $M \cdot N$, but that is not done in λ-calculus. The λ is meant to *bind* the variable in the *body*: in $\lambda x^\sigma.M$, x^σ is bound in M. We come to that later.

The idea is that $\lambda x^\sigma.M$ is the function $x \mapsto M$ that takes an input argument P and produces the output $M[x := P]$, M *with* P *substituted for* x. This will be made precise by the *β-reduction rule*, which is the computation rule to deal with λ-terms. We come to this in Definition 6.

Convention 4
- *Term variables will be denoted by* $x, y, z, \ldots$. *Terms will be denoted by* $M, N, P, \ldots$.
- *Type annotations on variables will only be written at the λ-abstraction: we write* $\lambda x^\sigma.x$ *instead of* $\lambda x^\sigma.x^\sigma$.
- *In term applications we let* brackets associate to the left *and we* omit outside brackets *and brackets around iterations of abstractions:* $M\,N\,P$ *denotes* $((M\,N)\,P)$ *and* $\lambda x^{\alpha\rightarrow\beta}.\lambda y^{\beta\rightarrow\gamma}.\lambda z^\alpha.xz(yz)$ *denotes* $(\lambda x^{\alpha\rightarrow\beta}.(\lambda y^{\beta\rightarrow\gamma}.(\lambda z^\alpha.((xz)(yz)))))$

Examples 2. For every type σ we have the term $\mathbf{I}_\sigma := \lambda x^\sigma.x$ which is of type $\sigma\!\rightarrow\!\sigma$. This is the *identity combinator* on σ.
For types σ and τ we have the term $\mathbf{K}_{\sigma\tau} := \lambda x^\sigma.\lambda y^\tau.x$ of type $\sigma\!\rightarrow\!\tau\!\rightarrow\!\sigma$. This term, called the K *combinator* takes two inputs and returns the first.

Here are some more interesting examples of typable terms:
$\lambda x^{\alpha\rightarrow\beta}.\lambda y^{\beta\rightarrow\gamma}.\lambda z^\alpha.y(xz) : (\alpha\!\rightarrow\!\beta)\!\rightarrow\!(\beta\!\rightarrow\!\gamma)\!\rightarrow\!\alpha\!\rightarrow\!\gamma$,
$\lambda x^\alpha.\lambda y^{(\beta\rightarrow\alpha)\rightarrow\alpha}.y(\lambda z^\beta.x) : \alpha\!\rightarrow\!((\beta\!\rightarrow\!\alpha)\!\rightarrow\!\alpha)\!\rightarrow\!\alpha$.

To show that a term is of a certain type, we have to "build it up" using the inductive definition of terms (Definition 3). For $\lambda x^{\alpha\to\beta}.\lambda y^{\beta\to\gamma}.\lambda z^{\alpha}.y(xz)$, we find the type as follows:

- If $x : \alpha\to\beta$, $y : \beta\to\gamma$ and $z : \alpha$, then $xz : \beta$,
- so $y(xz) : \gamma$,
- so $\lambda z^{\alpha}.y(xz) : \alpha\to\gamma$,
- so $\lambda y^{\beta\to\gamma}.\lambda z^{\alpha}.y(xz) : (\beta\to\gamma)\to\alpha\to\gamma$,
- so $\lambda x^{\alpha\to\beta}.\lambda y^{\beta\to\gamma}.\lambda z^{\alpha}.y(xz) : (\alpha\to\beta)\to(\beta\to\gamma)\to\alpha\to\gamma$

In λ-calculus (and type theory) we often take a number of λ-abstractions together, writing $\lambda x^{\sigma}y^{\tau}.x$ for $\lambda x^{\sigma}.\lambda y^{\tau}.x$. The conventions about types and applications fit together nicely. If $F : \sigma\to\tau\to\rho$, $M : \sigma$ and $P : \tau$, then

$$F\,M : \tau\to\rho \quad \text{and} \quad F\,M\,P : \rho$$

Given the bracket convention for types, every type of $\lambda\to$ can be written as

$$\sigma_1\to\sigma_2\to\ldots\to\alpha$$

with α a type variable.

The lack of product types is largely circumvented by dealing with functions of multiple arguments by *Currying*: We don't have $F : \sigma\times\tau\to\rho$ but instead we can use $F : \sigma\to\tau\to\rho$, because the latter F is a function that takes two arguments, of types σ and τ, and produces a term of type ρ.

3.1 Computation, free and bound variables, substitution

A λ-term of the form $(\lambda x^{\sigma}.M)P$ is a β- redex (*red*ucible *ex*pression). A redex can be *contracted*:

$$(\lambda x^{\sigma}.M)P \longrightarrow_{\beta} M[x := P]$$

where $M[x := P]$ denotes M with P substituted for x.

As an example, we have $(\lambda x^{\sigma}.\lambda y^{\tau}.x)P \longrightarrow_{\beta} \lambda y^{\tau}.P$. But what if $P = y$? then $(\lambda x^{\sigma}.\lambda y^{\tau}.x)y \longrightarrow_{\beta} \lambda y^{\tau}.y$, which is clearly not what we want, because the *free* y has become *bound* after reduction. The λ is a binder and we have to make sure that free variables don't get bound by a substitution. The solution is to *rename* bound variables before substitution.

Definition 5. *We define the notions of free and bound variables of a term, FV and BV.*

$$\begin{array}{rlrl} FV(x) &= \{x\} & BV(x) &= \emptyset \\ FV(MN) &= FV(M)\cup FV(N) & BV(MN) &= BV(M)\cup BV(N) \\ FV(\lambda x^{\sigma}.M) &= FV(M)\setminus\{x\} & BV(\lambda x^{\sigma}.M) &= BV(M)\cup\{x\} \end{array}$$

$M \equiv N$ *or* $M =_{\alpha} N$ *if* M *is equal to* N *modulo* renaming of bound variables. *A* closed *term is a term without free variables; closed terms are sometimes also called* combinators.

The renaming of bound variable x is done by taking a "fresh" variable (i.e. one that does not yet occur in the term, either free or bound), say y and replace all bound occurrences of x by y and λx by λy.

Examples 3.
- $\lambda x^\sigma.\lambda y^\tau.x \equiv \lambda x^\sigma.\lambda z^\tau.x$
- $\lambda x^\sigma.\lambda y^\tau.x \equiv \lambda y^\sigma.\lambda x^\tau.y$. This equality can be obtained by first renaming y to z, then x to y and then z to y.
- NB we also have $\lambda x^\sigma.\lambda y^\tau.y \equiv \lambda x^\sigma.\lambda x^\tau.x$. This equality is obtained by renaming the second x to y in the second term.

In the last example, we observe that our description of renaming above is slightly too informal. It is not symmetric, as we cannot rename y in the first term to x, and we may at some point not wish to rename with a completely fresh variable, but just with one that is not "in scope". We leave it at this and will not give a completely formal definition, as we think that the reader will be capable of performing α-conversion in the proper way. Fully spelled out definitions can be found in [11, 24, 3].

The general idea of (typed) λ-calculus is that we don't distinguish between terms that are α convertible: we consider terms *modulo α-equality* and we don't distinguish between $\lambda x^\sigma.\lambda y^\tau.x$ and $\lambda x^\sigma.\lambda z^\tau.x$. This implies that all our operations and predicates should be defined on α-equivalence classes, a property that we don't verify for every operation we define, but that we should be aware of.

When reasoning about λ-terms we use concrete terms (and not α-equivalence classes). We will avoid terms like $\lambda x^\sigma.\lambda x^\tau.x$, because they can be confusing. In examples we always rename bound variables such that no clashes can arise.
This is known as the *Barendregt convention*: when talking about a set of λ-terms, we may always assume that all free variables are different from the bound ones and that all bound variables are distinct.

Before reduction or substitution, we rename (if necessary):

$$(\lambda x^\sigma.\lambda y^\tau.x)y \equiv (\lambda x^\sigma.\lambda z^\tau.x)y \longrightarrow_\beta \lambda z^\tau.y$$

Definition 6. *The notions of one-step β-reduction, $\longrightarrow_\beta$, multiple-step β-reduction, $\longrightarrow\!\!\!\!\rightarrow_\beta$, and β-equality, $=_\beta$ are defined as follows.*

$$\begin{aligned}
(\lambda x^\sigma.M)N &\longrightarrow_\beta M[x := N] \\
M \longrightarrow_\beta N &\Rightarrow M\,P \longrightarrow_\beta N\,P \\
M \longrightarrow_\beta N &\Rightarrow P\,M \longrightarrow_\beta P\,N \\
M \longrightarrow_\beta N &\Rightarrow \lambda x^\sigma.M \longrightarrow_\beta \lambda x^\sigma.N
\end{aligned}$$

$\longrightarrow\!\!\!\!\rightarrow_\beta$ is the transitive reflexive closure of $\longrightarrow_\beta$. $=_\beta$ is the transitive reflexive symmetric closure of $\longrightarrow_\beta$.

The type $(\sigma\to\sigma)\to\sigma\to\sigma$ is called the type of *numerals over σ*, nat_σ. The way to encode natural numbers as closed terms of type nat_σ is as follows.

$$c_n := \lambda f^{\sigma\to\sigma}.\lambda x^\sigma.f^n(x)$$

where

$$f^n(x) \text{ denotes } \underbrace{f(\ldots f(f}_{n \text{ times } f} x))$$

So $c_2 := \lambda f^{\sigma\to\sigma}.\lambda x^{\sigma}.f(f\,x)$. These are also known as the *Church numerals*. (For readability we don't denote the dependency of c_n on the type σ, but leave it implicit.) A Church numeral c_n denotes the n-times iteration: it is a higher order function that takes a function $f : \sigma\to\sigma$ and returns the n-times iteration of f.

Example 4. We show a computation with the Church numeral c_2: we apply it to the identity $\mathbf{I}_\sigma$.

$$\begin{aligned}
\lambda z^{\sigma}.c_2\,\mathbf{I}_\sigma\,z &\equiv \lambda z^{\sigma}.(\lambda f^{\sigma\to\sigma}.\lambda x^{\sigma}.f(f\,x))\mathbf{I}_\sigma\,z\\
&\longrightarrow_\beta \lambda z^{\sigma}.(\lambda x^{\sigma}.\mathbf{I}_\sigma(\mathbf{I}_\sigma\,x))z\\
&\longrightarrow_\beta \lambda z^{\sigma}.\mathbf{I}_\sigma(\mathbf{I}_\sigma\,z)\\
&\longrightarrow_\beta \lambda z^{\sigma}.\mathbf{I}_\sigma\,z\\
&\longrightarrow_\beta \lambda z^{\sigma}.z \equiv \mathbf{I}_\sigma
\end{aligned}$$

In the above example, we see that at a certain point there are several ways to reduce: we can contract the inner or the outer redex with $\mathbf{I}_\sigma$. In this case the result is exactly the same. In general there are many redexes within a term that can be reduced, and they may all yield a different result. Often we want to fix a certain method for reducing terms, or we only want to contract redexes of a certain shape. This can be observed in the following example.

Examples 5. Define the $\mathbf{S}$ combinator as follows.

$$\mathbf{S} := \lambda x^{\sigma\to\sigma\to\sigma}.\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.x\,z(y\,z) \;:\; (\sigma\to\sigma\to\sigma)\to(\sigma\to\sigma)\to\sigma\to\sigma$$

Then $\mathbf{S}\,\mathbf{K}_{\sigma\sigma}\,\mathbf{I}_\sigma : \sigma\to\sigma$ and

$$\mathbf{S}\,\mathbf{K}_{\sigma\sigma}\,\mathbf{I}_\sigma \longrightarrow_\beta (\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\,z(y\,z))\mathbf{I}_\sigma$$

There are several ways of reducing this term further:

$$\begin{aligned}
(\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\,z(y\,z))\mathbf{I}_\sigma &\quad \text{is a redex}\\
\mathbf{K}_{\sigma\sigma}\,z &\quad \text{is a redex}
\end{aligned}$$

$$\begin{aligned}
(\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\,z(y\,z))\mathbf{I}_\sigma &\longrightarrow_\beta \lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\,z(\mathbf{I}_\sigma\,z)\\
&\equiv \lambda z^{\sigma}.(\lambda p^{\sigma}\,q^{\sigma}.p)\,z(\mathbf{I}_\sigma\,z)\\
&\longrightarrow_\beta \lambda z^{\sigma}.(\lambda q^{\sigma}.z)\,(\mathbf{I}_\sigma\,z)\\
\textit{Call by Value}\qquad &\longrightarrow_\beta \lambda z^{\sigma}.(\lambda q^{\sigma}.z)z\\
&\longrightarrow_\beta \lambda z^{\sigma}.z
\end{aligned}$$

But also

$$\begin{array}{rll}
(\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\, z(y\, z))\mathbf{I}_{\sigma} & \equiv & (\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.(\lambda p^{\sigma}\, q^{\sigma}.p)\, z(y\, z))\mathbf{I}_{\sigma} \\
& \longrightarrow_{\beta} & (\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.(\lambda q^{\sigma}.z)(y\, z))\mathbf{I}_{\sigma} \\
& \longrightarrow_{\beta} & \lambda z^{\sigma}.(\lambda q^{\sigma}.z)\,(\mathbf{I}_{\sigma}\, z) \\
\textit{Call by Name} & \longrightarrow_{\beta} & \lambda z^{\sigma}.z
\end{array}$$

In the previous example we have seen that the term $\lambda z^{\sigma}.(\lambda q^{\sigma}.z)\,(\mathbf{I}_{\sigma}\, z)$ can be reduced in several ways. *Call-by-name* is the ordinary β-reduction, where one can contract any β-redex. In *call-by-value*, one is only allowed to reduce $(\lambda x.M)N$ if N is a *value*, where a value is an abstraction term or a variable ([34]). So to reduce a term of the form $(\lambda x.M)((\lambda y.N)P)$ "call-by-value", we first have to contract $(\lambda y.N)P$. Call-by-value restricts the number of redexes that is allowed to be contracted, but it does not prescribe which is the next redex to contract. More restrictive variations of β-reduction are obtained by defining a *reduction strategy* which is a recipe that describes for every term which redex to contract. Well-known reduction strategies are *left-most outermost* or *right-most innermost*. To understand these notions it should be observed that redexes can be *contained in another*, e.g. in $(\lambda x.M)((\lambda y.N)P)$ or in $(\lambda x.(\lambda y.N)P)Q$, but they can also be *disjoint*, in which case there's always one to the left of the other. Other reduction strategies select a set of redexes and contract these simultaneously (a notion that should be defined first of course). For example, it is possible to define the simultaneous contraction of all redexes in a term, which is usually called a *complete development*. We don't go into the theory of reduction strategies or developments here, but refer to the literature [3]. Reduction in simple type theory enjoys some important properties that we list here. We don't give any proofs, as they can be found in the standard literature [5].

Theorem 1. *The simple type theory enjoys the following important properties.*

- *Subject Reduction*
  If $M : \sigma$ and $M \longrightarrow_{\beta} P$, then $P : \sigma$.
- *Church-Rosser*
  If M is a well-typed term in $\lambda\!\to$ and $M \longrightarrow\!\!\!\!\rightarrow_{\beta} P$ and $M \longrightarrow\!\!\!\!\rightarrow_{\beta} N$, then there is a (well-typed) term Q such that $P \longrightarrow\!\!\!\!\rightarrow_{\beta} Q$ and $N \longrightarrow\!\!\!\!\rightarrow_{\beta} Q$.
- *Strong Normalization*
  If M is well-typed in $\lambda\!\to$, then there is no infinite β-reduction path starting from M.

Subject reduction states – looking at it from a programmers point of view – that well-typed programs don't go wrong: evaluating a program $M : \sigma$ to a value indeed returns a value of type σ. Church-Rosser states that it doesn't make any difference for the final value *how* we reduce: we always get the same value. Strong Normalization states that no matter how one evaluates, one always obtains a value: there are no infinite computations possible.

3.2 Simple type theory presented with derivation rules

Our definition of $\lambda\!\!\rightarrow$ terms (Definition 3) is given via a standard inductive definition of the terms. This is very close to Church' [9] original definition. A different presentation can be given by presenting the inductive definition of the terms in rule form:

$$\frac{}{x^\sigma : \sigma} \qquad \frac{M : \sigma\!\rightarrow\!\tau \;\; N : \sigma}{MN : \tau} \qquad \frac{P : \tau}{\lambda x^\sigma.P : \sigma\!\rightarrow\!\tau}$$

The advantage is that now we also have a *derivation tree*, a proof of the fact that the term has that type. We can reason over these derivations.

In the above presentations, the set of free variables of a term is a global notion, that can be computed by the function FV. This is sometimes felt as being a bit imprecise and then a presentation is given with *contexts* to explicitly declare the free variables of a term.

$$x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n$$

is a context, if all the x_i are distinct and the σ_i are all $\lambda\!\!\rightarrow$-types. Contexts are usually denoted by Γ and we write $x \in \Gamma$ if x is one of the variables declared in Γ.

Definition 7. *The* derivation rules *of* $\lambda\!\!\rightarrow$ *à la Church are as follows.*

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash M : \sigma\!\rightarrow\!\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x{:}\sigma.P : \sigma\!\rightarrow\!\tau}$$

We write $\Gamma \vdash_{\lambda\!\rightarrow} M : \sigma$ *if there is a derivation using these rules with conclusion* $\Gamma \vdash M : \sigma$*.*

So note that – apart from the context – we now also write the type as a declaration in the λ-abstraction (and not as a superscript): $\lambda x : \sigma.x$ instead of $\lambda x^\sigma.x$. This presents us with a slightly different view on the base syntax: we don't see the variables as being typed (x^σ), but we take the view of a countably infinite collection of untyped variables that we assign a type to in the context (the free variables) or in the λ-abstraction (the bound variables).

To relate this Definition with the one of 3, we state – without proof – the following fact, where we ignore the obvious isomorphism that "lifts" the types in the λ-abstraction to a superscript.

Fact 1. *If* $\Gamma \vdash M : \sigma$*, then* $M : \sigma$ *(Definition 3) and* $FV(M) \subseteq \Gamma$*.*
If $M : \sigma$ *(Definition 3), then* $\Gamma \vdash M : \sigma$*, where* Γ *consists exactly of declarations of all the* $x \in FV(M)$ *to their corresponding types.*

As an example, we give a complete derivation of $\vdash \mathbf{K}_{\sigma\tau} : \sigma \rightarrow \tau \rightarrow \sigma$.

$$\dfrac{\dfrac{x : \sigma, y : \tau \vdash x : \sigma}{x : \sigma \vdash \lambda y{:}\tau.x : \tau \rightarrow \sigma}}{\vdash \lambda x{:}\sigma.\lambda y{:}\tau.x : \sigma \rightarrow \tau \rightarrow \sigma}$$

Derivations of typing judgments tend to get quite broad, because we are constructing a derivation tree. Moreover, this tree may contain quite a lot of duplications. So, when we are looking for a term of a certain type, the tree format may not be the most efficient and easy-to-use. We therefore introduce a *Fitch style representation* of typing derivations, named after the logician Fitch, who has developed a natural deduction system for logic in this format, also called *flag deduction* style [16]. We don't show that these two derivation styles derive the same set of typable terms, because it should be fairly obvious. (And a precise proof involves quite some additional notions and notation.)

Definition 8. *The Fitch style presentation of the rules of $\lambda\!\!\rightarrow$ is as follows.*

1	$x : \sigma$	
2	...	
3	...	
4	$M : \tau$	
5	$\lambda x{:}\sigma.M : \sigma \to \tau$	*abs,* 1, 4

abs-rule

1	...	
2	...	
3	$M : \sigma \to \tau$	
4	...	
5	...	
6	$N : \sigma$	
7	...	
8	$M\,N : \tau$	*app,* 3, 6

app-rule

In a Fitch deduction, a hypothesis is introduced by "raising a flag", e.g. the $x : \sigma$ in the left rule. A hypothesis is discharged when we "withdraw the flag", which happens at line 5. In a Fitch deduction one usually numbers the lines and refers to them in the *motivation* of the lines: the "abs,$1, 4$" and the "app, $3, 6$" at the end of lines 5 and 7.

Some remarks apply.

- It should be understood that one can raise a flag under an already open flag (one can nest flags), but the variable x in the newly raised flag should be *fresh*: it should not be declared in any of the open flags.
- In the app-rule, the order of the M and N can of course be different. Also the terms can be in a "smaller scope", that is: $M : \sigma$ may be higher in the deduction under less flags. Basically the M and N should just be "in scope", where a flag-ending ends the scope of all terms that are under that flag.

We say that a Fitch deduction derives the judgement $\Gamma \vdash M : \sigma$ if $M : \sigma$ is on the last line of the deduction the raised flags together form the context Γ.

Example 6. We show an example of a derivation of a term of type

$$(\alpha{\to}\beta{\to}\gamma){\to}(\alpha{\to}\beta){\to}\alpha{\to}\gamma$$

We show the derivation in two stages, to indicate how one can use the Fitch deduction rules to incrementally construct a term of a given type.

$$
\begin{array}{ll}
1 & \left|\begin{array}{l} x : \alpha{\to}\beta{\to}\gamma \\ \hline \left|\begin{array}{l} y : \alpha{\to}\beta \\ \hline \left|\begin{array}{l} z : \alpha \\ \hline \left|\begin{array}{l} ?? \\ ?? \\ ? : \gamma \end{array}\right. \end{array}\right. \\ \lambda z{:}\alpha.? : \alpha{\to}\gamma \end{array}\right. \\
8 & \left|\begin{array}{l} \lambda y{:}\alpha{\to}\beta.\lambda z{:}\alpha.? : (\alpha{\to}\beta){\to}\alpha{\to}\gamma \end{array}\right. \\
9 & \lambda x{:}\alpha{\to}\beta{\to}\gamma.\lambda y{:}\alpha{\to}\beta.\lambda z{:}\alpha.? : (\alpha{\to}\beta{\to}\gamma){\to}(\alpha{\to}\beta){\to}\alpha{\to}\gamma
\end{array}
$$

$$
\begin{array}{ll}
1 & \left|\begin{array}{l} x : \alpha{\to}\beta{\to}\gamma \\ \hline \left|\begin{array}{l} y : \alpha{\to}\beta \\ \hline \left|\begin{array}{l} z : \alpha \\ \hline \left|\begin{array}{l} x\,z : \beta{\to}\gamma \\ y\,z : \beta \\ x\,z(y\,z) : \gamma \end{array}\right. \end{array}\right. \\ \lambda z{:}\alpha.x\,z(y\,z) : \alpha{\to}\gamma \end{array}\right. \\
8 & \left|\begin{array}{l} \lambda y{:}\alpha{\to}\beta.\lambda z{:}\alpha.x\,z(y\,z) : (\alpha{\to}\beta){\to}\alpha{\to}\gamma \end{array}\right. \\
9 & \lambda x{:}\alpha{\to}\beta{\to}\gamma.\lambda y{:}\alpha{\to}\beta.\lambda z{:}\alpha.x\,z(y\,z) : (\alpha{\to}\beta{\to}\gamma){\to}(\alpha{\to}\beta){\to}\alpha{\to}\gamma
\end{array}
$$

Exercises 1. 1. Construct a term of type $(\delta{\to}\delta{\to}\alpha){\to}(\alpha{\to}\beta{\to}\gamma){\to}(\delta{\to}\beta){\to}\delta{\to}\gamma$
2. Construct two terms of type $(\delta{\to}\delta{\to}\alpha){\to}(\gamma{\to}\alpha){\to}(\alpha{\to}\beta){\to}\delta{\to}\gamma{\to}\beta$
3. Construct a term of type $((\alpha{\to}\beta){\to}\alpha){\to}(\alpha{\to}\alpha{\to}\beta){\to}\alpha$
4. Construct a term of type $((\alpha{\to}\beta){\to}\alpha){\to}(\alpha{\to}\alpha{\to}\beta){\to}\beta$ (Hint: use the previous exercise.)

3.3 The Curry-Howard formulas-as-types correspondence

Using the presentation of $\lambda{\to}$ with derivation rules, it is easier to make the *Curry-Howard formulas-as-types* correspondence precise. The idea is that there are two readings of a judgement $M : \sigma$:

1. term as algorithm/program, type as specification :
   M is a function of type σ
2. type as a proposition, term as its proof :
   M is a proof of the proposition σ

More precisely, the Curry-Howard formulas-as-types correspondence states that there is a natural one-to-one correspondence between typable terms in $\lambda{\to}$

and derivations in *minimal proposition logic*. Looking at it from the logical point of view: the judgement $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \vdash M : \sigma$ can be read as M *is a proof of* σ *from the assumptions* $\tau_1, \tau_2, \ldots, \tau_n$.

Definition 9. *The system of minimal proposition logic* PROP *consists of*

– *implicational propositions, generated by the following abstract syntax:*

$$\mathsf{prop} ::= \mathsf{PropVar} | (\mathsf{prop} \to \mathsf{prop})$$

– *derivation rules (Δ is a set of propositions, σ and τ are propositions)*

$$\frac{\sigma\to\tau \quad \sigma}{\tau}\to\text{-}E \qquad \frac{\begin{array}{c}[\sigma]^j\\ \vdots\\ \tau\end{array}}{\sigma\to\tau}\,[j]\to\text{-}I$$

We write $\Delta \vdash_{\mathrm{PROP}} \sigma$ if there is a derivation using these rules with conclusion σ and non-discharged assumptions *in Δ.*

Note the difference between a context, which is basically a *list*, and a *set* of assumptions. Logic (certainly in natural deduction) is usually presented using a *set* of assumptions, but there is no special reason for not letting the assumptions be a list, or a multi-set.

We now give a precise definition of the formulas-as-types correspondence. For this we take the presentation of PROP with lists (so Δ is a list in the following definition). As a matter of fact the *formulas-as-types* part of the definition is trivial: a proposition in PROP is just a type in $\lambda\!\to$, but the most interesting part of the correspondence is the *proofs-as-terms* embedding, maybe best called the *deductions-as-term* embedding. For PROP, this part is also quite straightforward, but we describe it in detail nevertheless.)

Definition 10. *The* deductions-as-terms *embedding from derivation of* PROP *to term of $\lambda\!\to$ is defined inductively as follows. We associate to a list of propositions Δ a context Γ in the obvious way by replacing σ_i in Δ with $x_i : \sigma_i \in \Gamma$. On the left we give the inductive clause for the derivation and on the right we describe on top of the line the terms we have (by induction) and below the line the term that the derivation gets mapped to.*

$$\frac{}{\Delta \vdash \sigma}\,\sigma \in \Delta \;\rightsquigarrow\; \frac{}{\Gamma \vdash x : \sigma}\,x : \sigma \in \Gamma$$

$$\frac{\sigma\to\tau \quad \sigma}{\tau}\to\text{-}E \rightsquigarrow \frac{\Gamma_1 \vdash M : \sigma\to\tau \quad \Gamma_2 \vdash N : \sigma}{\Gamma_1 \cup \Gamma_2 \vdash M\,N : \tau}$$

$$\frac{\begin{array}{c}[\sigma]^j\\ \vdots\\ \tau\end{array}}{\sigma\to\tau}\,[j]\to\text{-}I \quad\rightsquigarrow\quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma\to\tau}$$

We denote this embedding by $\overline{-}$*, so if* $\mathcal{D}$ *is a derivation from* PROP*,* $\mathcal{D}$ *is a* $\lambda\!\to$*-term.*

For a good understanding we give a detailed example.

Example 7. Consider the following natural deduction derivation PROP and the term in $\lambda\!\to$ it gets mapped to.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\alpha\to\beta\to\gamma]^3 \;\; [\alpha]^1}{\beta\to\gamma} \qquad \dfrac{[\alpha\to\beta]^2 \;\; [\alpha]^1}{\beta}}{\gamma}}{\alpha\to\gamma}\,1}{(\alpha\to\beta)\to\alpha\to\gamma}\,2}{(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}\,3 \quad\mapsto\quad \begin{array}{l}\lambda x{:}\alpha\to\beta\to\gamma.\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz)\\ :(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma\end{array}$$

To create the term on the right, it is best to decorate the deduction tree with terms, starting from the leaves (decorated by variables) and working downwards, finally creating a term for the root node that is the λ-term that corresponds to the whole deduction.

$$\dfrac{\dfrac{\dfrac{\dfrac{[x:\alpha\to\beta\to\gamma]^3 \;\; [z:\alpha]^1}{xz:\beta\to\gamma} \qquad \dfrac{[y:\alpha\to\beta]^2 \;\; [z:\alpha]^1}{yz:\beta}}{xz(yz):\gamma}}{\lambda z{:}\alpha.xz(yz):\alpha\to\gamma}\,1}{\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz):(\alpha\to\beta)\to\alpha\to\gamma}\,2}{\lambda x{:}\alpha\to\beta\to\gamma.\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz):(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}\,3$$

Theorem 2 (Soundness, Completeness of formulas-as-types).

1. *If* $\mathcal{D}$ *is a natural deduction in* PROP *with conclusion* σ *and non-discharged assumption in* Δ*, then*

$$\boldsymbol{x} : \boldsymbol{\Delta} \vdash \overline{\mathcal{D}} : \sigma \text{ in } \lambda\!\to.$$

2. *If* $\Gamma \vdash M : \sigma$ *in* $\lambda\!\to$*, then there is a derivation of* σ *from the* Δ *in* PROP*, where* Δ *is* Γ *without the variable-assignments.*

We don't give the proofs, as they are basically by a straightforward induction. The second part of the Theorem can be strengthened a bit: we can construct a derivation $\mathcal{D}$ out of the term M, as an inverse to the mapping $\overline{-}$. So, the formulas-as-types correspondence constitutes an *isomorphism* between derivations in PROP and well-typed terms in $\lambda\!\to$.

Exercise 2. Add types to the λ-abstractions and give the derivation that corresponds to the term $\lambda x.\lambda y.y(\lambda z.y\,x) : (\gamma\to\varepsilon)\to((\gamma\to\varepsilon)\to\varepsilon)\to\varepsilon$.

In the λ-calculus we have a notion of computation, given by β-reduction: $(\lambda x{:}\sigma.M)P \longrightarrow_\beta M[x := P]$. Apart from that, there is also a notion of η-reduction: $\lambda x{:}\sigma.M\,x \longrightarrow_\eta M$ if $x \notin \mathrm{FV}(M)$. The idea of considering these terms as equal is quite natural, because they behave exactly the same as functions: $(\lambda x{:}\sigma.M\,x)P \longrightarrow_\beta M\,P$. In a typed setting, it is very natural to consider the rule in the opposite direction, because then one can make sure that every term of a function type has a normal form that is a λ-abstraction. Of course this requires a proviso to prevent an infinite η-reduction of the form $x \longrightarrow_\eta \lambda y{:}\sigma.x\,y \longrightarrow_\eta \lambda y{:}\sigma.(\lambda z{:}\sigma.x\,z)y \ldots$

In natural deduction we also have a notion of computation: *cut-elimination* or *detour-elimination*. If one introduces a connective and then immediately eliminates it again, this is called a cut or a detour. A *cut* is actually a *rule* in the sequent calculus representation of logic and the *cut-elimination* theorem in sequent calculus states that the cut-rule is derivable and thus superfluous. In natural deduction, we can eliminate detours, which are often also called cuts, a terminology that we will also use here.

Definition 11. *A* cut *in a deduction in minimal propositional logic is a place where an $\rightarrow$-I is immediately followed by an elimination of that same $\rightarrow$. Graphically ($\mathcal{D}_1$ and $\mathcal{D}_2$ denote deductions):*

$$\dfrac{\dfrac{\begin{array}{c}[\sigma]^1\\ \mathcal{D}_1\\ \tau\end{array}}{\sigma\rightarrow\tau}\,1 \qquad \dfrac{\mathcal{D}_2}{\sigma}}{\tau}$$

Cut-elimination *is defined by replacing the cut in a deduction in the way given below.*

$$\dfrac{\dfrac{\begin{array}{c}[\sigma]^1\\ \mathcal{D}_1\\ \tau\end{array}}{\sigma\rightarrow\tau}\,1 \qquad \dfrac{\mathcal{D}_2}{\sigma}}{\tau} \quad\longrightarrow\quad \begin{array}{c}\mathcal{D}_2\\ \sigma\\ \mathcal{D}_1\\ \tau\end{array}$$

So every occurrence of the discharged assumption $[\sigma]^1$ in $\mathcal{D}_1$ is replaced by the deduction $\mathcal{D}_2$.

It is not hard to prove that eliminating a cut yields a well-formed natural deduction again, with the same conclusion. The set of non-discharged assumptions remains the same or shrinks. (In case there is no occurrence of $[\sigma]^1$ at all; then $\mathcal{D}_2$ is removed and also its assumptions.) That this process terminates is not obvious: if $[\sigma]^1$ occurs several times, $\mathcal{D}_2$ gets copied, resulting in a larger deduction.

Lemma 2. *Cut-elimination in* PROP *corresponds to β-reduction in $\lambda\!\!\rightarrow$: if $\mathcal{D}_1 \longrightarrow_{cut} \mathcal{D}_2$, then $\overline{\mathcal{D}_1} \longrightarrow_\beta \overline{\mathcal{D}_2}$*

The proof of this Lemma is indicated in the following diagram.

$$\dfrac{\dfrac{\begin{array}{c}[x:\sigma]^1\\ \mathcal{D}_1\\ M:\tau\end{array}}{\lambda x{:}\sigma.M:\sigma\!\rightarrow\!\tau}\,1 \qquad \dfrac{\mathcal{D}_2}{P:\sigma}}{(\lambda x{:}\sigma.M)P:\tau} \quad \longrightarrow_\beta \quad \begin{array}{c}\mathcal{D}_2\\ P:\sigma\\ \mathcal{D}_1\\ M[x:=P]:\tau\end{array}$$

To get a better understanding of the relation between cut-elimination and β-reduction, we now study an example.

Example 8. Consider the following proof of $A\!\rightarrow\!A\!\rightarrow\!B, (A\!\rightarrow\!B)\!\rightarrow\!A \vdash B$.

$$\dfrac{\dfrac{\dfrac{A\!\rightarrow\!A\!\rightarrow\!B\ \ [A]^1}{A\!\rightarrow\!B}\ [A]^1}{B} \qquad \dfrac{(A\!\rightarrow\!B)\!\rightarrow\!A \qquad \dfrac{\dfrac{A\!\rightarrow\!A\!\rightarrow\!B\ \ [A]^1}{A\!\rightarrow\!B}\ [A]^1}{B}}{A\!\rightarrow\!B}}{A}}{B}$$

It contains a cut: a $\rightarrow$-I directly followed by an $\rightarrow$-E. We now present the same proof after reduction

$$\dfrac{A\!\rightarrow\!A\!\rightarrow\!B \qquad \dfrac{(A\!\rightarrow\!B)\!\rightarrow\!A \qquad \dfrac{\dfrac{A\!\rightarrow\!A\!\rightarrow\!B\ \ [A]^1}{A\!\rightarrow\!B}\ [A]^1}{B}}{A\!\rightarrow\!B} \qquad (A\!\rightarrow\!B)\!\rightarrow\!A \qquad \dfrac{\dfrac{A\!\rightarrow\!A\!\rightarrow\!B\ \ [A]^1}{A\!\rightarrow\!B}\ [A]^1}{B}}{A}}{B}$$

We now present the same derivations of $A\!\rightarrow\!A\!\rightarrow\!B, (A\!\rightarrow\!B)\!\rightarrow\!A \vdash B$, now with term information

$$\dfrac{\dfrac{\dfrac{p : A{\to}A{\to}B \;\; [x : A]^1}{p\,x : A{\to}B} \qquad [x : A]^1}{p\,x\,x : B}}{\lambda x{:}A.p\,x\,x : A{\to}B} \qquad \dfrac{q : (A{\to}B){\to}A \qquad \dfrac{\dfrac{\dfrac{p : A{\to}A{\to}B \;\; [x : A]^1}{p\,x : A{\to}B} \qquad [x : A]^1}{p\,x\,x : B}}{\lambda x{:}A.p\,x\,x : A{\to}B}}{q(\lambda x{:}A.p\,x\,x) : A}$$

$$(\lambda x{:}A.p\,x\,x)(q(\lambda x{:}A.p\,x\,x)) : B$$

The term contains a β-redex: $(\lambda x{:}A.p\,x\,x)\,(q(\lambda x{:}A.p\,x\,x))$ We now present the reduced proof of $A{\to}A{\to}B, (A{\to}B){\to}A \vdash B$ with term info. For reasons of page size we summarize the derivation of $q(\lambda x{:}A.p\,x\,x) : A$ as $\mathcal{D}$. So

$$\dfrac{\boxed{\mathcal{D}}}{q(\lambda x{:}A.p\,x\,x) : A} \;:=\; \dfrac{q : (A{\to}B){\to}A \qquad \dfrac{\dfrac{\dfrac{p : A{\to}A{\to}B \;\; [x{:}A]^1}{p\,x : A{\to}B} \qquad [x{:}A]^1}{p\,x\,x : B}}{\lambda x{:}A.p\,x\,x : A{\to}B}}{q(\lambda x{:}A.p\,x\,x) : A}$$

This is the sub-derivation that gets copied under cut-elimination (β-reduction).

$$\dfrac{p : A{\to}A{\to}B \qquad \dfrac{\boxed{\mathcal{D}}}{q(\lambda x{:}A.p\,x\,x) : A}}{p(q(\lambda x{:}A.p\,x\,x)) : A{\to}B} \qquad \dfrac{\boxed{\mathcal{D}}}{q(\lambda x{:}A.p\,x\,x) : A}$$

$$p(q(\lambda x{:}A.p\,x\,x))(q(\lambda x{:}A.p\,x\,x)) : B$$

4 Type assignment versus typed terms

4.1 Untyped $\boldsymbol{\lambda}$-calculus

Simple Type Theory is not very expressive: one can only represent a limited number of functions over the nat_σ (see the paragraph after Definition 6) data types in $\lambda{\to}$. We can allow more functions to be definable by relaxing the type constraints. The most flexible system is to have no types at all.

Definition 12. *The terms of the untyped λ-calculus, Λ, are defined as follows.*

$$\Lambda ::= \mathit{Var} \mid (\Lambda\,\Lambda) \mid (\lambda\,\mathit{Var}.\Lambda)$$

Examples are the well-known combinators that we have already seen in a typed fashion: $\mathbf{K} := \lambda x\, y.x$, $\mathbf{S} := \lambda x\, y\, z.x\, z(y\, z)$. But we can now do more: here are some well-known untyped λ-terms $\omega := \lambda x.x\, x$, $\Omega := \omega\, \omega$. The notions of β-reduction and β-equality generalize from the simple typed case, so we don't repeat it here. An interesting aspect is that we can now have *infinite reductions*. (Which is impossible in $\lambda\!\rightarrow$, as that system is Strongly Normalizing.) The simplest infinite reduction is the following *loop*:

$$\Omega \longrightarrow_\beta \Omega$$

A term that doesn't loop but whose reduction path contains infinitely many different terms is obtained by putting $\omega_3 := \lambda x.x\, x\, x$, $\Omega_3 := \omega_3\omega_3$. Then:

$$\Omega_3 \longrightarrow_\beta \omega_3\, \omega_3\, \omega_3 \longrightarrow_\beta \omega_3\, \omega_3\, \omega_3\, \omega_3 \longrightarrow_\beta \ldots$$

The untyped λ-calculus was defined by Church [9] and proposed as a system to capture the notion of *mechanic computation*, for which Turing proposed the notion of Turing machine. An important property of the untyped λ-calculus is that it is *Turing complete*, which was proved by Turing in 1936, see [13]. The power of Λ lies in the fact that you can *solve recursive equations*.

A recursive equation is a question of the following kind:

– Is there a term M such that

$$M\, x =_\beta x\, M\, x?$$

– Is there a term M such that

$$M\, x =_\beta \mathsf{if}\ (\mathsf{Zero}\, x)\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{Mult}\, x\, (M\, (\mathsf{Pred}\, x))?$$

So, we have two expressions on either side of the $=_\beta$ sign, both containing an unknown M and we want to know whether a solution for M exists.

The answer is: **yes**, if we can rewrite the equation to one of the form

$$M =_\beta \boxed{\ldots M \ldots} \tag{1}$$

Note that this is possible for the equations written above. For example the first equation is solved by a term M that satisfies $M =_\beta \lambda x.x\, M\, x$.

That we can solve equation of the form (1) is because every term in the λ-calculus has a *fixed point*. Even more: we have a *fixed point combinator*.

Definition 13. – *The term M is a* fixed point *of the term P if $P\, M =_\beta M$.*
– *The term Y is a* fixed point combinator *if for every term P, $Y\, P$ is a fixed point of P, that is if*

$$P\, (Y\, P) =_\beta Y\, P.$$

In the λ-calculus we have various fixed point combinators, of which the Y-combinator is the most well-known one: $Y := \lambda f.(\lambda x.f(x\, x))(\lambda x.f(x\, x))$.

Exercise 3. Verify that the Y as defined above is a fixed point combinator: $P\,(Y\,P) =_\beta Y\,P$ for every λ-term P.
Verify that $\Theta := (\lambda x\,y.y(x\,x\,y))(\lambda x\,y.y(x\,x\,y))$ is also a fixed point combinator that is even *reducing*: $\Theta\,P \longrightarrow\!\!\!\!\rightarrow_\beta P\,(\Theta\,P)$ for every λ-term P.

The existence of fixed-points is the key to the power of the λ-calculus. But we also need natural numbers and booleans to be able to write programs. In Section 3 we have already seen the *Church numerals*:

$$c_n := \lambda f.\lambda x.f^n(x)$$

where

$$f^n(x) \text{ denotes } \underbrace{f(\ldots f(f}_{n \text{ times } f}\ x))$$

The successor is easy to define for these numerals: $\mathsf{Suc} := \lambda n.\lambda f\,x.f(n\,f\,x)$. Addition can also be defined quite easily, but if we are lazy we can also use the fixed-point combinator. We want to solve

$$\mathsf{Add}\,n\,m := \mathsf{if}\,(\mathsf{Zero}\,n)\,\mathsf{then}\,m\,\mathsf{else}\,\mathsf{Add}\,(\mathsf{Pred}\,n)\,m)$$

where Pred is the predecessor function, Zero is a test for zero and $\mathsf{if}\ldots\mathsf{then}\ldots\mathsf{else}$ is a case distinction on booleans. The booleans can be defined by

$$\begin{aligned}\mathsf{true} &:= \lambda x\,y.x\\ \mathsf{false} &:= \lambda x\,y.y\\ \mathsf{if}\,b\,\mathsf{then}\,P\,\mathsf{else}\,Q &:= b\,P\,Q.\end{aligned}$$

Exercise 4.
1. Verify that the booleans behave as expected:
   $\mathsf{if\,true\,then}\,P\,\mathsf{else}\,Q =_\beta P$ and $\mathsf{if\,false\,then}\,P\,\mathsf{else}\,Q =_\beta Q$.
2. Define a test-for-zero Zero on the Church numerals :$\mathsf{Zero}\,c_0 =_\beta \mathsf{true}$ and $\mathsf{Zero}\,c_{n+1} =_\beta \mathsf{false}$. (Defining the predecessor is remarkably tricky!)

Apart from the natural numbers and booleans, it is not difficult to find encodings of other data, like lists and trees. Given the expressive power of the untyped λ-calculus and the limited expressive power of $\lambda{\to}$, one may wonder why we want types. There are various good reasons for that, most of which apply to the the "typed versus untyped programming languages" issue in general.

Types give a (partial) specification. Types tell the programmer – and a person reading the program — what a program (λ-term) does, to a certain extent. Types only give a very partial specification, like $f : \mathbb{N} \to \mathbb{N}$, but depending on the type system, this information can be enriched, for example: $f : \Pi n : \mathbb{N}.\exists m : \mathbb{N}.m > n$, stating that f is a program that takes a number n and returns an m that is larger than n. In the Chapter by Bove and Dybjer, one can find examples of that type and we will also come back to this theme in this Chapter in Section 6.

"Well-typed programs never go wrong" (Milner). The *Subject Reduction property* guarantees that a term of type σ remains to be of type σ under evaluation. So, if $M : \mathtt{nat}$ evaluates to a value v, we can be sure that v is a natural number.

The type checking algorithm detects (simple) mistakes. Types can be checked at compile time (*statically*) and this is a simple but very useful method to detect simple mistakes like typos and applying functions to the wrong arguments. Of course, in a more refined type system, type checking can also detect more subtle mistakes.

Typed terms always terminate(?) In typed λ-calculi used for representing proofs (following the Curry-Howard isomorphism), the terms are always terminating, and this is seen as an advantage as it helps in proving consistency of logical theories expressed in these calculi. In general, termination very much depends on the typing system. In this paper, all type systems only type terminating (strongly normalizing) λ-terms, which also implies that these systems are not Turing complete. Type systems for programming languages will obviously allow also non-terminating calculations. A simple way to turn $\lambda\!\!\rightarrow$ into a Turing complete language is by adding fixed point combinators (for every type σ) $Y_\sigma : (\sigma\!\rightarrow\!\sigma)\!\rightarrow\!\sigma$ with the reduction rule $Y\,f \rightarrow f(Y\,f)$. This is basically the system PCF, first defined and studied by Plotkin [35].

Given that we want types, the situation with a system like $\lambda\!\!\rightarrow$ as presented in Section 3, is still unsatisfactory from a programmers point of view. Why would the programmer have to write all those types? The compiler should compute the type information for us!

For M an *untyped* term, we want the type system to *assign* a type σ to M (or say that M is not typable). Such a type system is called a *type assignment system*, or also *typing à la Curry* (as opposed to the *typing à la Church* that we have seen up to now).

4.2 Simple type theory à la Church and à la Curry

We now set the two systems side-by-side: $\lambda\!\!\rightarrow$ à la Church and à la Curry.

Definition 14. *In $\lambda\!\!\rightarrow$ à la Curry, the terms are*

$$\Lambda ::= Var \mid (\Lambda\,\Lambda) \mid (\lambda\, Var.\Lambda)$$

In $\lambda\!\!\rightarrow$ à la Church, the terms are

$$\Lambda_{Ch} ::= Var \mid (\Lambda_{Ch}\,\Lambda_{Ch}) \mid (\lambda\, Var{:}\sigma.\Lambda_{Ch})$$

where σ ranges over the simple types, as defined in Definition 1.

These sets of terms are just the *preterms*. The typing rules will select the *well-typed* terms from each of these sets.

Definition 15. *The typing rules of $\lambda\!\!\rightarrow$ à la Church and $\lambda\!\!\rightarrow$ à la Curry are as follows. (The ones for the Church system are the same as the ones in Definition 7.)*
$\lambda\!\!\rightarrow$ (à la Church):

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma \vdash M : \sigma{\rightarrow}\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \quad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x{:}\sigma.P : \sigma{\rightarrow}\tau}$$

$\lambda\!\!\rightarrow$ (à la Curry):

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma \vdash M : \sigma{\rightarrow}\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \quad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma{\rightarrow}\tau}$$

The rules à la Curry can of course also be given in the Fitch style, which is the style we use when giving derivations of typings.

Exercise 5. Give a full derivation of

$$\vdash \lambda x.\lambda y.y(\lambda z.y\,x) : (\gamma{\rightarrow}\varepsilon){\rightarrow}((\gamma{\rightarrow}\varepsilon){\rightarrow}\varepsilon){\rightarrow}\varepsilon$$

in Curry style $\lambda\!\!\rightarrow$

We can summarize the differences between *Typed Terms* and *Type Assignment* as follows:

- With typed terms (typing à la Church), we have terms with type information in the λ-abstraction: $\lambda x{:}\alpha.x : \alpha{\rightarrow}\alpha$. As a consequence:
  - Terms have unique types,
  - The type is directly computed from the type info in the variables.
- With type assignment (typing à la Curry), we assign types to untyped λ-terms: $\lambda x.x : \alpha{\rightarrow}\alpha$. As a consequence:
  - Terms do not have unique types,
  - A *principal type* can be computed (using unification).

Examples 9.
- Typed Terms:

  $$\lambda x{:}\alpha.\lambda y{:}(\beta{\rightarrow}\alpha){\rightarrow}\alpha.y(\lambda z{:}\beta.x)$$

  has only the type $\alpha{\rightarrow}((\beta{\rightarrow}\alpha){\rightarrow}\alpha){\rightarrow}\alpha$
- Type Assignment: $\lambda x.\lambda y.y(\lambda z.x)$ can be assigned the types
  - $\alpha{\rightarrow}((\beta{\rightarrow}\alpha){\rightarrow}\alpha){\rightarrow}\alpha$
  - $(\alpha{\rightarrow}\alpha){\rightarrow}((\beta{\rightarrow}\alpha{\rightarrow}\alpha){\rightarrow}\gamma){\rightarrow}\gamma$
  - ...

  with $\alpha{\rightarrow}((\beta{\rightarrow}\alpha){\rightarrow}\gamma){\rightarrow}\gamma$ being the *principal type*, a notion to be defined and discussed later.

There is an obvious connection between Church and Curry typed $\lambda\!\!\rightarrow$, given by the *erasure map*.

Definition 16. *The* erasure *map* $|-|$ *from* $\lambda\!\!\rightarrow$ *à la Church to* $\lambda\!\!\rightarrow$ *à la Curry is defined by erasing all type information:*

$$\begin{aligned} |x| &:= x \\ |M\,N| &:= |M|\,|N| \\ |\lambda x : \sigma.M| &:= \lambda x.|M| \end{aligned}$$

So, e.g. $|\lambda x{:}\alpha.\lambda y{:}(\beta{\rightarrow}\alpha){\rightarrow}\alpha.y(\lambda z{:}\beta.x))| = \lambda x.\lambda y.y(\lambda z.x))$.

Theorem 3. *If* $M : \sigma$ *in* $\lambda\!\!\rightarrow$ *à la Church, then* $|M| : \sigma$ *in* $\lambda\!\!\rightarrow$ *à la Curry. If* $P : \sigma$ *in* $\lambda\!\!\rightarrow$ *à la Curry, then there is an* M *such that* $|M| \equiv P$ *and* $M : \sigma$ *in* $\lambda\!\!\rightarrow$ *à la Church.*

The proof is by an easy induction on the derivation.

4.3 Principal types

We now discuss the notion of a principal type in $\lambda\!\!\rightarrow$ à la Curry. We will describe an algorithm, the *principal type algorithm*, that, given a closed untyped term M, computes a type σ if M is typable with type σ in $\lambda\!\!\rightarrow$, and "reject" if M is not typable. Moreover, the computed type σ is "minimal" in the sense that all possible types for M are substitution instances of σ.

Computing a principal type for M in $\lambda\!\!\rightarrow$ à la Curry proceeds as follows:

1. Assign a type variable to every variable x in M.
2. Assign a type variable to every *applicative sub-term* of M.
3. Generate a (finite) set of equations E between types that need to hold in order to ensure that M is typable.
4. Compute a "minimal substitution" S, substituting types for type variables, that makes all equations in E hold. (This is a *most general unifier* for E.)
5. With S compute the type of M.

The algorithm described above can fail only if their is no unifying substitution for E. In that case we return "reject" and conclude that M is not typable. An *applicative sub-term* is a term that is not a variable and does not start with a λ. (So it is a sub-term of the form $P\,Q$). One could label all sub-terms with a type variable, but that just adds superfluous overhead. We show how the algorithm works by elaborating an example.

Example 10. We want to compute the principal type of $\lambda x.\lambda y.y(\lambda z.yx)$.

1. Assign type variables to all term variables: $x : \alpha, y : \beta, z : \gamma$.
2. Assign type variables to all applicative sub-terms: $y\,x : \delta$, $y(\lambda z.y\,x) : \varepsilon$. These two steps yield the following situation, where we indicate the types of the variables and applicative sub-terms by super- and subscripts.

$$\lambda x^{\alpha}.\lambda y^{\beta}.\underbrace{y^{\beta}(\lambda z^{\gamma}.\overbrace{y^{\beta}x^{\alpha}}^{\delta})}_{\varepsilon}$$

3. Generate equations between types, necessary for the term to be typable:

$$E = \{\beta = \alpha{\to}\delta, \beta = (\gamma{\to}\delta){\to}\varepsilon\}$$

The equation $\beta = \alpha{\to}\delta$ arises from the sub-term $\overbrace{y^{\beta}x^{\alpha}}^{\delta}$, which is of type δ if β is a function type with domain α and range δ. The equation $\beta = (\gamma{\to}\delta){\to}\varepsilon$ arises from the sub-term $\underbrace{y^{\beta}(\lambda z^{\gamma}.\overbrace{y\,x}^{\delta})}_{\varepsilon}$, which is of type ε if β is a function type with domain $\gamma{\to}\delta$ and range ε.

4. Find a most general substitution (a *most general unifier*) for the type variables that solves the equations:

$$S := \{\alpha := \gamma{\to}\delta,\ \ \beta := (\gamma{\to}\delta){\to}\varepsilon,\ \ \delta := \varepsilon\}$$

5. The *principal type* of $\lambda x.\lambda y.y(\lambda z.yx)$ is now

$$(\gamma{\to}\varepsilon){\to}((\gamma{\to}\varepsilon){\to}\varepsilon){\to}\varepsilon$$

Exercise 6. 1. Compute the principal type for $\mathbf{S} := \lambda x.\lambda y.\lambda z.x\,z(y\,z)$

2. Which of the following terms is typable? If it is, determine the *principal type*; if it isn't, show that the typing algorithm rejects the term.
   (a) $\lambda z\,x.z(x(\lambda y.y\,x))$
   (b) $\lambda z\,x.z(x(\lambda y.y\,z))$
3. Compute the principal type for $M := \lambda x.\lambda y.x(y(\lambda z.x\,z\,z))(y(\lambda z.x\,z\,z))$.

We now introduce the notions required for the principal types algorithm.

Definition 17. – *A* type substitution *(or just substitution) is a map S from type variables to types. As a function, we write it* after the type*, so σS denotes the result of carrying out substitution S on σ.*

– *Most substitutions we encounter are the identity on all but a finite number of type variables, so we often denote a substitution as $[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]$. We view a type substitution as a function that is carried out* in parallel *so $[\alpha := \beta{\to}\beta, \beta := \alpha{\to}\gamma]$ applied to $\alpha{\to}\beta$ results in $(\beta{\to}\beta){\to}\alpha{\to}\gamma$.*

– *We can compose substitutions in the obvious way: $S;T$ is obtained by first performing S and then T.*

– *A* unifier *of the types σ and τ is a substitution that "makes σ and τ equal", i.e. an S such that $\sigma S = \tau S$.*

– *A* most general unifier *(or mgu) of the types σ and τ is the "simplest substitution" that makes σ and τ equal, i.e. an S such that*
  - *$\sigma S = \tau S$*
  - *for all substitutions T such that $\sigma T = \tau T$ there is a substitution R such that $T = S;R$.*

All these notions generalize to lists instead of pairs σ, τ. We say that S *unifies* the list of equations $\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$ if $\sigma_1 S = \tau_1 S, \ldots, \sigma_n S = \tau_n S$, that is: S makes all equations true.

The crucial aspect in the principal type algorithm is the computability of a *most general unifier* for a set of type equations. The rest of the algorithm should be clear from the example and we don't describe it in detail here.

Definition 18. *We define the algorithm U that, when given a list of type equations $E = \langle \sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n \rangle$ outputs a substitution S or "reject" as follows. U looks at the first type equation $\sigma_1 = \tau_1$ and depending on its form it outputs:*

- $U(\langle \alpha = \alpha, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \sigma_2 = \tau_2, \ldots, \sigma_n = \tau_n \rangle)$.
- $U(\langle \alpha = \tau_1, \ldots, \sigma_n = \tau_n \rangle) :=$ *"reject" if* $\alpha \in FV(\tau_1)$, $\tau_1 \neq \alpha$.
- $U(\langle \alpha = \tau_1, \ldots, \sigma_n = \tau_n \rangle) :=$
  $[\alpha := V(\tau_1), U(\langle \sigma_2[\alpha := \tau_1] = \tau_2[\alpha := \tau_1], \ldots, \sigma_n[\alpha := \tau_1] = \tau_n[\alpha := \tau_1] \rangle)]$, *if* $\alpha \notin FV(\tau_1)$, *where V abbreviates* $U(\langle \sigma_2[\alpha := \tau_1] = \tau_2[\alpha := \tau_1], \ldots, \sigma_n[\alpha := \tau_1] = \tau_n[\alpha := \tau_1] \rangle)$.
- $U(\langle \sigma_1 = \alpha, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \alpha = \sigma_1, \ldots, \sigma_n = \tau_n \rangle)$
- $U(\langle \mu \to \nu = \rho \to \xi, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \mu = \rho, \nu = \xi, \ldots, \sigma_n = \tau_n \rangle)$

Theorem 4. *The function U computes the* most general unifier *of a set of equations E. That is,*

- *If $U(E) =$ "reject", then there is no substitution S that unifies E.*
- *If $U(E) = S$, then S unifies E and for all substitutions T that unify E, there is a substitution R such that $T = S; R$ (S is most general).*

Definition 19. *The type σ is a* principal type *for the closed untyped λ-term M if*

- *$M : \sigma$ in $\lambda\!\to$ à la Curry*
- *for all types τ, if $M : \tau$, then $\tau = \sigma S$ for some substitution S.*

Theorem 5 (Principal Types). *There is an algorithm* PT *that, when given a closed (untyped) λ-term M, outputs*

- *A* principal type *σ such that $M : \sigma$ in $\lambda\!\to$ à la Curry.*
- *"reject" if M is not typable in $\lambda\!\to$ à la Curry.*

The algorithm is the one we have described before. We don't give it in formal detail, nor the proof of its correctness, but refer to [5] and [40]. This algorithm goes back to the type inference algorithm for simply typed lambda calculus of Hindley [23], which was independently developed by Milner [29] and extended to the weakly polymorphic case (see Section 5.1). Damas [12] has proved it correct and therefore this algorithm is often referred to as the Hindley-Milner or Damas-Milner algorithm.

If one wants to type an *open term* M, i.e. one that contains free variables, one is actually looking for what is known as a *principal pair*, consisting of a context Γ and a type σ such that $\Gamma \vdash M : \sigma$ and if $\Gamma' \vdash M : \tau$, then there

is a substitution S such that $\tau = \sigma S$ and $\Gamma' = \Gamma S$. (A substitution extends straightforwardly to contexts.) However, there is a simpler way of attacking this problem: just apply the PT algorithm for closed terms to $\lambda x_1 \ldots \lambda x_n.M$ where $x_1, \ldots, x_n$ is the list of free variables in M.

The following describes a list of typical decidability problems one would like to have an algorithm for in a type theory.

Definition 20.

$\vdash M : \sigma$?	Type Checking Problem	*TCP*
$\vdash M : ?$	Type Synthesis *or* Type Assginment Problem	*TSP, TAP*
$\vdash ? : \sigma$	Type Inhabitation Problem	*TIP*

Theorem 6. *For $\lambda\!\!\rightarrow$, all problems defined in Definition 20 are decidable, both for the Curry style and for the Church style versions of the system.*

For Church style, TCP and TSP are trivial, because we can just "read off" the type from the term that has the variables in the λ-abstractions decorated with types. For Curry style, TSP is solved by the PT algorithm. This also gives a way to solve TCP: to verify if $M : \sigma$, we just compute the principal type of M, say τ, and verify if σ is a substitution instance of τ (which is decidable).

In general, one may think that TCP is easier than TSP, but they are (usually) equivalent: Suppose we need to solve the TCP $M\,N : \sigma$. The only thing we can do is to solve the TSP $N :?$ and if this gives answer τ, solve the TCP $M : \tau\!\rightarrow\!\sigma$. So we see that these problems are tightly linked.

For Curry systems, TCP and TSP soon become undecidable if we go beyond $\lambda\!\!\rightarrow$. In the next section we will present the polymorphic λ-calculus, whose Curry style variant has an undecidable TCP.

TIP is decidable for $\lambda\!\!\rightarrow$, as it corresponds to *provability* in PROP, which is known to be decidable. This applies to both the Church and Curry variants, because they have the same inhabited types (as a consequence of Theorem 3). TIP is undecidable for most extensions of $\lambda\!\!\rightarrow$, because TIP corresponds to provability in some logic and provability gets easily undecidable (e.g. already in very weak systems of predicate logic).

As a final remark: if we add a context to the problems in Definition 20, the decidability issues remain the same. For TIP, the problem is totally equivalent since

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash ? : \sigma \iff \vdash ? : \sigma_1\!\rightarrow \ldots \rightarrow\!\sigma_n\!\rightarrow\!\sigma$$

For the Church system, TSP is also totally equivalent:

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M :? \iff \vdash \lambda x_1{:}\sigma_1 \ldots . \lambda x_n{:}\sigma_n.M :?$$

and similarly for TCP.

For the Curry system, the situation is slightly different, because in the TSP $\Gamma \vdash M :?$ the free variables are "forced" to be of specific types, which they are not in $\vdash \lambda \boldsymbol{x}.M :?$. Nevertheless, also if we add a context, TSP and TCP remain decidable and the principal type technique that we have described still works.

4.4 Properties of λ→; Normalization

We now list the most important meta-theoretic properties of $\lambda\!\rightarrow$.

Theorem 7. – *For $\lambda\!\rightarrow$ à la Church:* Uniqueness of types
If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma = \tau$.
– Subject Reduction
If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_\beta N$, then $\Gamma \vdash N : \sigma$.
– Strong Normalization
If $\Gamma \vdash M : \sigma$, then all β-reductions from M terminate.

These are proved using the following more basic properties of $\lambda\!\rightarrow$.

Proposition 1. – Substitution property
If $\Gamma, x : \tau, \Delta \vdash M : \sigma$, $\Gamma \vdash P : \tau$, then $\Gamma, \Delta \vdash M[x := P] : \sigma$.
– Thinning
If $\Gamma \vdash M : \sigma$ and $\Gamma \subseteq \Delta$, then $\Delta \vdash M : \sigma$.

The proof of these properties proceeds by induction on the typing derivation – where we sometimes first have to prove some auxiliary Lemmas that we haven't listed here – except for the proof of Strong Normalization, which was first proved by Tait [38]. As Strong Normalization is such an interesting property and it has an interesting proof, we devote the rest of this section to it. We first study the problem of *Weak Normalization*, stating that every term has (a reduction path to) a normal form.

Definition 21. – *A λ-term M is* weakly normalizing *or WN if there is a reduction sequence starting from M that terminates.*
– *A λ-term M is* strongly normalizing *or SN if all reduction sequences starting from M terminate.*
A type system is WN if all well-typed terms are WN, and it is SN if all well-typed terms are SN.

What is the problem with normalization?

– Terms may get larger under reduction
$(\lambda f.\lambda x.f(fx))P \longrightarrow_\beta \lambda x.P(Px)$, which blows up if P is large.
– Redexes may get multiplied under reduction.
$(\lambda f.\lambda x.f(fx))((\lambda y.M)Q) \longrightarrow_\beta \lambda x.((\lambda y.M)Q)(((\lambda y.M)Q)x)$
– New redexes may be created under reduction.
$(\lambda f.\lambda x.f(fx))(\lambda y.N) \longrightarrow_\beta \lambda x.(\lambda y.N)((\lambda y.N)x)$

To prove WN, we would like to have a reduction strategy that does not create new redexes, or that makes the term shorter in every step. However, this idea is too naive and impossible to achieve. We can define a more intricate notion of "size" of a term and a special reduction strategy that decreases the size of a term at every step, but to do that we have to analyze more carefully what can happen during a reduction. We give the following Lemma about "redex creation", the proof of which is just a syntactic case analysis.

Lemma 3. *There are four ways in which "new" β-redexes can be created in a β-reduction step.*

- Creation
$$(\lambda x. \ldots (x\,P) \ldots)(\lambda y.Q) \longrightarrow_\beta \ldots (\lambda y.Q)P \ldots$$
Here we really create a new redex, by substituting a λ-abstraction for a variable that is in function position.
- Multiplication
$$(\lambda x. \ldots x \ldots x \ldots)((\lambda y.Q)R) \longrightarrow_\beta \ldots (\lambda y.Q)R \ldots (\lambda y.Q)R \ldots$$
Here we copy (possibly many times) an existing redex, thereby creating new ones.
- Hidden redex
$$(\lambda x.\lambda y.Q)R\,P \longrightarrow_\beta (\lambda y.Q[x := R])P$$
Here the redex $(\lambda y.Q)P$ was already present in a hidden form, being "shaded" by the λx; it is revealed by contracting the outer redex.
- Identity
$$(\lambda x.x)(\lambda y.Q)R \longrightarrow_\beta (\lambda y.Q)R$$
This is a different very special case of a "hidden redex": by contracting the identity, the redex $(\lambda y.Q)R$ is revealed.

We now define an appropriate size and an appropriate reduction strategy that proves weak normalization. The proof is originally due to Turing and was first written up by Gandy [17].

Definition 22. *The* height *(or order) of a type $h(\sigma)$ is defined by*

- $h(\alpha) := 0$
- $h(\sigma_1 \to \ldots \to \sigma_n \to \alpha) := max(h(\sigma_1), \ldots, h(\sigma_n)) + 1$.

The idea is that the height of a type σ is at least 1 higher than of any of the domains types occurring in σ. In the definition, we use the fact that we can write types in a "standard form" $\sigma_1 \to \ldots \to \sigma_n \to \alpha$. But it is equivalent to define h directly by induction over the definition of types, as is stated in the following exercise.

Exercise 7. Prove that the definition of h above is equivalent to defining

- $h(\alpha) := 0$
- $h(\sigma \to \tau) := \max(h(\sigma) + 1, h(\tau))$.

Definition 23. *The* height *of a redex $(\lambda x{:}\sigma.P)Q$ is the height of the type of $\lambda x{:}\sigma.P$.*

As an example, we look at the "identity" redex creation case of lemma 3. Note that the height of the redex in $(\lambda x{:}\sigma.x)(\lambda y{:}\tau.Q)R$ is $h(\sigma)+1$ and that the height of the redex in its reduct, $(\lambda y{:}\tau.Q)R$, is $h(\sigma)$. (Note that the type of $\lambda y{:}\tau.Q$) is just σ.) So the created redex has lesser height.

This will be the key idea to our reduction strategy: we will select a redex whose reduction only creates redexes of lesser height.

Definition 24. *We assign a* measure m *to the terms by defining*

$$m(N) := (h_r(N), \#N)$$

where

- $h_r(N) =$ *the maximum height of a redex in* N*,*
- $\#N =$ *the number of redexes of maximum height* $h_r(N)$ *in* N*.*

The measures of terms are ordered in the obvious *lexicographical* way:

$$(h_1, x) <_l (h_2, y) \text{ iff } h_1 < h_2 \text{ or } (h_1 = h_2 \text{ and } x < y).$$

Theorem 8 (Weak Normalization). *If* P *is a typable term in* $\lambda{\to}$*, then there is a terminating reduction starting from* P*.*

Proof. Pick a redex of maximum height $h_r(P)$ inside P that does not contain any other redex of height $hr(P)$. Note that this is always possible: If R_1 and R_2 are redexes, R_1 is contained in R_2 or the other way around. Say we have picked $(\lambda x{:}\sigma.M)N$.

Reduce this redex, to obtain $M[x := N]$. We claim that *this does not create a new redex of height* $h_r(P)$ $(\star)$. This is the important step and the proof is by analyzing the four possibilities of redex creation as they are given in Lemma 3. We leave this as an exercise.

If we write Q for the reduct of P, then, as a consequence of $(\star)$, we find that $m(Q) <_l m(P)$. As there are no infinitely decreasing $<_l$ sequences, this process must terminate and then we have arrived at a normal form.

Exercise 8. Check claim $(\star)$ in the proof of Theorem 8. (Hint: Use Lemma 3.)

Strong Normalization for $\lambda{\to}$ is proved by constructing a *model* of $\lambda{\to}$. We give the proof for $\lambda{\to}$ à la Curry. The proof is originally due to Tait [38], who proposed the interpretation of the $\to$ types as given below. Recently, combinatorial proofs have been found, that give a "measure" to a typed λ-term and then prove that this measure decreases *for all* reduction steps that are possible. See [26].

Definition 25. *The interpretation of* $\lambda{\to}$*-types is defined as follows.*

- $[\![\alpha]\!] := \mathsf{SN}$ *(the set of strongly normalizing* λ*-terms).*
- $[\![\sigma{\to}\tau]\!] := \{M \mid \forall N \in [\![\sigma]\!](MN \in [\![\tau]\!])\}$.

Note that the interpretation of a function type is countable: it is not (isomorphic to) the full function space, but it contains only the functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$ that can be λ-*defined*, i.e. are representable by a λ-term. This set is obviously countable. We have the following closure properties for $[\![\sigma]\!]$.

Lemma 4. *1.* $[\![\sigma]\!] \subseteq \mathsf{SN}$
2. $xN_1 \dots N_k \in [\![\sigma]\!]$ *for all* x, σ *and* $N_1, \dots, N_k \in \mathsf{SN}$.
3. If $M[x := N]\boldsymbol{P} \in [\![\sigma]\!]$, $N \in \mathsf{SN}$, *then* $(\lambda x.M)N\boldsymbol{P} \in [\![\sigma]\!]$.

Proof. All three parts are by induction on the structure of σ. The first two are proved simultaneously. (NB. In the case of $\sigma = \rho{\to}\tau$ for the proof of (1), we need that $[\![\rho]\!]$ is non-empty, which is guaranteed by the induction hypothesis for (2).) For (1) also use the fact that, if $M\,N \in \mathsf{SN}$, then also $M \in \mathsf{SN}$.

Exercise 9. Do the details of the proof of Lemma 4.

Proposition 2.

$$\left.\begin{array}{l} x_1{:}\tau_1, \dots, x_n{:}\tau_n \vdash M : \sigma \\ N_1 \in [\![\tau_1]\!], \dots, N_n \in [\![\tau_n]\!] \end{array}\right\} \Rightarrow M[x_1 := N_1, \dots, x_n := N_n] \in [\![\sigma]\!]$$

Proof. By induction on the derivation of $\Gamma \vdash M : \sigma$, using (3) of the Lemma 4

Corollary 1 (Strong Normalization for $\lambda{\to}$). $\lambda{\to}$ *is SN*

Proof. By taking $N_i := x_i$ in Proposition 2. (Note that $x_i \in [\![\tau_i]\!]$ by Lemma 4.) Then $M \in [\![\sigma]\!] \subseteq \mathsf{SN}$.

Exercise 10. Verify the details of the Strong Normalization proof. That is, prove Proposition 2 in detail by checking the inductive cases.

In the Strong Normalization proof, we have constructed a model that has the special nature that the interpretation of the function space is countable. If one thinks about semantics in general, one of course can also take the full set-theoretic function space as interpretation of $\sigma{\to}\tau$. We elaborate a little bit on this point, mainly as a reference for a short discussion in the Section on polymorphic λ-calculus.

We say that $\lambda{\to}$ *has a simple set-theoretic model*. Given sets $[\![\alpha]\!]$ for type variables α, define

$$[\![\sigma{\to}\tau]\!] := [\![\tau]\!]^{[\![\sigma]\!]} \ \text{(set theoretic function space } [\![\sigma]\!] \to [\![\tau]\!])$$

Now, if any of the base sets $[\![\alpha]\!]$ is infinite, then there are higher and higher infinite cardinalities among the $[\![\sigma]\!]$, because the cardinality of $[\![\sigma{\to}\tau]\!]$ is always strictly larger than that of $[\![\sigma]\!]$.

There are smaller models, e.g.

$$[\![\sigma{\to}\tau]\!] := \{f \in [\![\sigma]\!] \to [\![\tau]\!] | f \text{ is definable}\}$$

where *definability* means that it can be constructed in some formal system. This restricts the collection to a *countable* set. As an example we have seen in the SN proof the following interpretation

$$[\![\sigma{\to}\tau]\!] := \{f \in [\![\sigma]\!] \to [\![\tau]\!] | f \text{ is } \lambda\text{-definable}\}$$

The most important thing we want to note for now is that in $\lambda{\to}$ we have a lot of freedom in choosing the interpretation of the $\to$-types. In the polymorphic λ-calculus, this is no longer the case.

5 Polymorphic Type Theory

Simple type theory $\lambda{\to}$ is not very expressive: we can only define generalized polynomials as functions [37] and we don't have a clear notion of data types. Also, in simple type theory, we cannot 'reuse' a function. For example, $\lambda x{:}\alpha.x : \alpha{\to}\alpha$ and $\lambda x{:}\beta.x : \beta{\to}\beta$, which is twice the identity in slightly different syntactic form. Of course, in the Curry version we have $\lambda x.x : \alpha{\to}\alpha$ and $\lambda x.x : \beta{\to}\beta$, but then still we can't have *the same term* $\lambda x.x$ being of type $\alpha{\to}\alpha$ and of type $\beta{\to}\beta$ *at the same time*. To see what we mean with that, consider the following term that we can type

$$(\lambda y.y)(\lambda x.x)$$

In the Church version, this would read, e.g.

$$(\lambda y{:}\sigma{\to}\sigma.y)(\lambda x{:}\sigma.x)$$

which shows that we can type this term with type $\sigma{\to}\sigma$. To type the two identities with the same type at the same time, we would want to type the following (of which the term above is a one-step reduct):

$$(\lambda f.f\ f)(\lambda x.x).$$

But this term is not typable: f should be of type $\sigma{\to}\sigma$ and of type $(\sigma{\to}\sigma){\to}\sigma{\to}\sigma$ at the same time, which we can't achieve in $\lambda{\to}$.

We want to define functions that can treat types *polymorphically*. We add types of the form $\forall\alpha.\sigma$.

Examples 11.
- $\forall\alpha.\alpha{\to}\alpha$
  If $M : \forall\alpha.\alpha{\to}\alpha$, then M can map any type to itself.
- $\forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha$
  If $M : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha$, then M can take two inputs (of arbitrary types) and return a value of the first input type.

There is a weak and a strong version of polymorphism. The first is present in most functional programming languages, therefore also called *ML style polymorphism*. The second allows more types and is more immediate if one takes a logical view on types. We first treat the weak version.

5.1 Typed λ-calculus with weakly polymorphic types

Definition 26. *In* weak $\lambda 2$ *(the system with* weak polymorphism*) we have the following additional types*

$$\mathsf{Typ}_w := \forall\alpha.\mathsf{Typ}_w | \mathsf{Typ}$$

where Typ *is the collection of* $\lambda\!\!\rightarrow$*-types as defined in Definition 1.*

So, the weak polymorphic types are obtained by adding $\forall\alpha_1 \ldots \forall\alpha_n.\sigma$ for σ a $\lambda\!\!\rightarrow$-type.

We can formulate polymorphic λ-calculus in Church and in Curry style. As for $\lambda\!\!\rightarrow$, the two systems are different in the type information that occurs in the terms, but now the difference is larger: in polymorphic λ-calculus we also have abstractions over types.

Definition 27. *The terms of* weak $\lambda 2$ *à la Church are defined by*

$$\Lambda_2^{ch} ::= Var \mid (\Lambda_2^{ch}\ \Lambda_2^{ch}) \mid (\lambda\, Var{:}\mathsf{Typ}.\Lambda_2^{ch}) \mid (\lambda\mathsf{TVar}.\Lambda_2^{ch}) \mid \Lambda_2^{ch}\ \mathsf{Typ}$$

The terms of the Curry version of the calculus are of course just Λ. This means that in the Curry version we will not record the abstractions over type variables. This is made precise in the following rules.

Definition 28. *The* Derivation rules *for* weak $\lambda 2$ *(ML-style polymorphism) in Church style are as follows*

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \quad \frac{\Gamma, x:\sigma \vdash M:\tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma\!\rightarrow\!\tau}\ \textit{if}\ \sigma,\tau \in \mathsf{Typ} \quad \frac{\Gamma \vdash M:\sigma\!\rightarrow\!\tau \quad \Gamma \vdash N:\sigma}{\Gamma \vdash M\,N:\tau}$$

$$\frac{\Gamma \vdash M:\sigma}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.\sigma}\ \alpha \notin FV(\Gamma) \quad \frac{\Gamma \vdash M:\forall\alpha.\sigma}{\Gamma \vdash M\tau : \sigma[\alpha := \tau]}\ \textit{if}\ \tau \in \mathsf{Typ}$$

Definition 29. *The* derivation rules *for* weak $\lambda 2$ *(ML-style polymorphism) in Curry style are as follows.*

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \quad \frac{\Gamma, x:\sigma \vdash M:\tau}{\Gamma \vdash \lambda x.M : \sigma\!\rightarrow\!\tau}\ \textit{if}\ \sigma,\tau \in \mathsf{Typ} \quad \frac{\Gamma \vdash M:\sigma\!\rightarrow\!\tau \quad \Gamma \vdash N:\sigma}{\Gamma \vdash M\,N:\tau}$$

$$\frac{\Gamma \vdash M:\sigma}{\Gamma \vdash M : \forall\alpha.\sigma}\ \alpha \notin FV(\Gamma) \quad \frac{\Gamma \vdash M:\forall\alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]}\ \textit{if}\ \tau \in \mathsf{Typ}$$

Examples 12. 1. In $\lambda 2$ à la Curry: $\lambda x.\lambda y.x : \forall\alpha.\forall\beta.\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$.

2. In $\lambda 2$ à la Church we have the following, which is the same term as in the previous case, but now with type information added: $\lambda\alpha.\lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$.
3. In $\lambda 2$ à la Curry: $z : \forall\alpha.\alpha\!\rightarrow\!\alpha \vdash z\,z : \forall\alpha.\alpha\!\rightarrow\!\alpha$.

4. In $\lambda 2$ à la Church, we can annotate this term with type information to obtain: $z : \forall\alpha.\alpha{\to}\alpha \vdash \lambda\alpha.z\,(\alpha{\to}\alpha)\,(z\,\alpha) : \forall\alpha.\alpha{\to}\alpha$.
5. We do *not* have $\vdash \lambda z.z\,z : \ldots$

We can make flag deduction rules for this system as follows.

$$\begin{array}{l|l|l}
1 & \boxed{\begin{array}{l} x : \sigma \\ \hline \ldots \\ \ldots \\ M : \tau \end{array}} & \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \boxed{\begin{array}{l} \ldots \\ \ldots \\ M : \sigma \to \tau \\ \ldots \\ \ldots \\ N : \sigma \\ \ldots \\ M\,N : \tau \end{array}} \\
2 & & \\
3 & & \\
4 & & \\
5 & \lambda x{:}\sigma.M : \sigma \to \tau & \\
\end{array}$$

if $\sigma, \tau \in \mathsf{Typ}$

$$\begin{array}{l|l|l}
1 & \boxed{\begin{array}{l} \alpha \\ \hline \ldots \\ \ldots \\ M : \sigma \end{array}} & \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \boxed{\begin{array}{l} \ldots \\ \ldots \\ M : \forall\alpha.\sigma \\ \ldots \\ \ldots \\ M\,\tau : \sigma[\alpha := \tau] \end{array}} \\
2 & & \\
3 & & \\
4 & & \\
5 & \lambda\alpha.M : \forall\alpha.\sigma & \\
\end{array}$$

if α fresh

The "freshness" condition in the rule means that α should *not occur free above the flag*. In terms of contexts, this means precisely that the type variable that we intend to abstract over does not occur in the context Γ. The freshness condition excludes the following wrong flag deduction.

$$\text{WRONG} \quad \begin{array}{l|l}
1 & \boxed{\begin{array}{l} f : \alpha{\to}\beta \\ \hline \boxed{\begin{array}{l} \alpha \\ \hline \boxed{\begin{array}{l} x : \alpha \\ \hline f\,x : \beta \end{array}} \end{array}} \\ \lambda x.f\,x : \alpha{\to}\beta \end{array}} \\
2 & \\
3 & \\
4 & \\
5 & \\
6 & \lambda x.f\,x : \forall\alpha.\alpha{\to}\beta
\end{array}$$

Of course, we should not be able to derive

$$f : \alpha{\to}\beta \vdash \lambda x.f\,x : \forall\alpha.\alpha{\to}\beta$$

Examples 13. Here are the first four examples of 12, now with flag deductions. In the first row we find the derivations in the Church systems, in the second row

the ones in the Curry system.

$$
\begin{array}{l|l}
1 & \alpha \\
\hline
2 & \quad \beta \\
\hline
3 & \quad\quad x : \alpha \\
\hline
4 & \quad\quad\quad y : \beta \\
\hline
5 & \quad\quad \lambda y{:}\beta.x : \beta{\to}\alpha \\
6 & \quad \lambda x{:}\alpha.\lambda y{:}\beta.x : \alpha{\to}\beta{\to}\alpha \\
7 & \quad \lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\beta.\alpha{\to}\beta{\to}\alpha \\
8 & \lambda\alpha.\lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha
\end{array}
\qquad
\begin{array}{l|l}
1 & z : \forall\alpha.\alpha{\to}\alpha \\
\hline
2 & \quad \alpha \\
\hline
3 & \quad\quad z\,\alpha : \alpha{\to}\alpha \\
4 & \quad\quad z\,(\alpha{\to}\alpha) : (\alpha{\to}\alpha){\to}\alpha{\to}\alpha \\
5 & \quad\quad z\,(\alpha{\to}\alpha)(z\,\alpha) : \alpha{\to}\alpha \\
6 & \lambda\alpha.z\,(\alpha{\to}\alpha)(z\,\alpha) : \forall\alpha.\alpha{\to}\alpha
\end{array}
$$

$$
\begin{array}{l|l}
1 & \alpha \\
\hline
2 & \quad \beta \\
\hline
3 & \quad\quad x : \alpha \\
\hline
4 & \quad\quad\quad y : \beta \\
\hline
5 & \quad\quad \lambda y.x : \beta{\to}\alpha \\
6 & \quad \lambda x.\lambda y.x : \alpha{\to}\beta{\to}\alpha \\
7 & \quad \lambda x.\lambda y.x : \forall\beta.\alpha{\to}\beta{\to}\alpha \\
8 & \lambda x.\lambda y.x : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha
\end{array}
\qquad
\begin{array}{l|l}
1 & z : \forall\alpha.\alpha{\to}\alpha \\
\hline
2 & \quad \alpha \\
\hline
3 & \quad\quad z : \alpha{\to}\alpha \\
4 & \quad\quad z : (\alpha{\to}\alpha){\to}\alpha{\to}\alpha \\
5 & \quad\quad z\,z : \alpha{\to}\alpha \\
6 & z\,z : \forall\alpha.\alpha{\to}\alpha
\end{array}
$$

In the types, $\forall$ only occurs on the outside. Therefore, in programming languages (that usually follow a Curry style typing discipline) it is usually left out and all type variables are *implicitly universally quantified over*. This means that one doesn't write the $\forall$, so the types are restricted to Typ and hence we don't have the last two rules. That any type variable can be instantiated with a type, is then made formal by changing the variable rule into

$$\frac{}{\Gamma, x{:}\forall\boldsymbol{\alpha}.\sigma \vdash x : \sigma[\boldsymbol{\alpha} := \boldsymbol{\tau}]}\ \text{if } \boldsymbol{\tau} \subseteq \mathsf{Typ}$$

This should be read as that σ doesn't contain any $\forall$ anymore. So we have universal types only in the context (as types of declared variables) and only at this place we can instantiate the $\forall\alpha$ with types, which must be simple types. It can be proven that this system is equivalent to $\lambda 2$ à la Curry in the following sense. Denote by $\vdash_v$ the variant of the weak $\lambda 2$ rules à la Curry where we don't have rules for $\forall$ and the adapted rule for variables just described. Then

$$\Gamma \vdash_v M : \sigma \Rightarrow \Gamma \vdash M : \sigma$$
$$\Gamma \vdash M : \forall\boldsymbol{\alpha}.\sigma \Rightarrow \Gamma \vdash_v M : \sigma[\boldsymbol{\alpha} := \boldsymbol{\tau}]$$

This is proved by induction on the derivation, using a Substitution Lemma that holds for all type systems that we have seen so far and that we therefore state here in general.

Lemma 5 (Substitution for types). *If $\Gamma \vdash M : \sigma$, then $\Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$, for all systems à la Curry defined so far.*
For the Church systems we have types in the terms. Then the Substitution Lemma states: If $\Gamma \vdash M : \sigma$, then $\Gamma[\alpha := \tau] \vdash M[\alpha := \tau] : \sigma[\alpha := \tau]$.

For all systems, this Lemma is proved by a straightforward induction over the derivation.

With weak polymorphism, type checking is still decidable: the principal types algorithm can be extended to incorporate *type schemes*: types of the form $\forall \boldsymbol{\alpha}.\sigma$. We have observed that weak polymorphism allows terms to have many (polymorphic) types, but we cannot abstract over variables of these types. This is allowed with *full polymorphism*, also called *system F style polymorphism*.

5.2 Typed λ-calculus with full polymorphism

Definition 30. *The types of $\lambda 2$ with* full (system F-style) polymorphism *are*

$$\mathsf{Typ}_2 := \mathsf{TVar} \mid (\mathsf{Typ}_2 \to \mathsf{Typ}_2) \mid \forall \alpha.\mathsf{Typ}_2$$

1. *The derivation rules for $\lambda 2$ with* full (system F-style) polymorphism *in Curry style are as follows. (Note that σ and τ range over Typ_2.)*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \quad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha.\sigma}\ \alpha \notin FV(\Gamma) \quad \frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]}$$

2. *The derivation rules for $\lambda 2$ with* full (system F-style) polymorphism *in Church style are as follows. (Again, note that σ and τ range over Typ_2.)*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \tau} \quad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \lambda \alpha.M : \forall \alpha.\sigma}\ \alpha \notin FV(\Gamma) \quad \frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M\tau : \sigma[\alpha := \tau]}$$

So now, $\forall$ can also occur deeper in a type. We can write flag deduction rules for the full $\lambda 2$ in the obvious way, for both the Curry and the Church variant of the system. We now give some examples that are only valid with full polymorphism.

Examples 14. – $\lambda 2$ à la Curry: $\lambda x.\lambda y.x : (\forall \alpha.\alpha) \to \sigma \to \tau$.
– $\lambda 2$ à la Church: $\lambda x{:}(\forall \alpha.\alpha).\lambda y{:}\sigma.x\tau : (\forall \alpha.\alpha) \to \sigma \to \tau$.

Here are the flag deductions that prove the two typings in the Examples.

1	$x : \forall\alpha.\alpha$
2	$\quad y : \sigma$
3	$\quad x\,\tau : \tau$
4	$\lambda y{:}\sigma.x\,\tau : \sigma\to\tau$
5	$\lambda x{:}\forall\alpha.\alpha.\lambda y{:}\sigma.x\,\tau : (\forall\alpha.\alpha)\to\sigma\to\tau$

1	$x : \forall\alpha.\alpha$
2	$\quad y : \sigma$
3	$\quad x : \tau$
4	$\lambda y.x : \sigma\to\tau$
5	$\lambda x.\lambda y.x : (\forall\alpha.\alpha)\to\sigma\to\tau$

In $\lambda 2$ we use the following abbreviations for types: $\bot := \forall\alpha.\alpha$, $\top := \forall\alpha.\alpha\to\alpha$. The names are derived from the behavior of the types. From a term of type $\bot$, we can create a term of any type: $\lambda x{:}\bot.x\sigma : \bot\to\sigma$ for any type σ. So $\bot$ is in some sense the "smallest type". Also, $\bot$ is empty: there is no closed term of type $\bot$. On the other hand, $\top$ is the type with one canonical closed element: $\lambda\alpha.\lambda x{:}\alpha.x : \top$. We can now also type a term like $\lambda x.x\,x$.

Examples 15.
- In Curry $\lambda 2$: $\lambda x.xx : \bot\to\bot$, $\lambda x.xx : \top\to\top$
- In Church $\lambda 2$: $\lambda x{:}\bot.x(\bot\to\bot)x : \bot\to\bot$, $\lambda x{:}\top.x\,\top\,x : \top\to\top$.
- In Church $\lambda 2$: $\lambda x{:}\bot.\lambda\alpha.x(\alpha\to\alpha)(x\alpha) : \bot\to\bot$, $\lambda x{:}\top.\lambda\alpha.x(\alpha\to\alpha)(x\alpha) : \top\to\top$.

We show two typings in the previous example by a flag deduction.

1	$x : \bot$
2	$\quad \alpha$
3	$\quad x : \alpha\to\alpha$
4	$\quad x : \alpha$
5	$\quad x\,x : \alpha$
6	$x\,x : \forall\alpha.\alpha$
7	$\lambda x{:}\bot.x\,x : \bot\to\bot$

1	$x : \top$
2	$\quad x\,\top : \top\to\top$
3	$\quad x\,\top\,x : \top$
4	$\lambda x{:}\top.x\,\top\,x : \top\to\top$

Exercises 11.
1. Verify using a flag deduction that in Church $\lambda 2$: $\lambda x{:}\top.\lambda\alpha.x(\alpha\to\alpha)(x\alpha) : \top\to\top$.
2. Verify using a flag deduction that in Curry $\lambda 2$: $\lambda x.xx : \top\to\top$
3. Find a type in Curry $\lambda 2$ for $\lambda x.x\,x\,x$
4. Find a type in Curry $\lambda 2$ for $\lambda x.x\,x\,(x\,x)$

With full polymorphism, type checking becomes undecidable [41] for the Curry version of the system. For the Church version it is clearly still decidable, as we have all necessary type information in the term.

Definition 31. *We define the* erasure map *from $\lambda 2$ à la Church to $\lambda 2$ à la Curry as follows.*

$$\begin{array}{llll} |x| & := x & & \\ |\lambda x{:}\sigma.M| & := |\lambda x.M| & |\lambda\alpha.M| & := |M| \\ |MN| & := |M|\,|N| & |M\sigma| & := |M| \end{array}$$

We have the following proposition about this erasure map, that relates the Curry and Church systems to each other.

Proposition 3. *If $\Gamma \vdash M : \sigma$ in $\lambda 2$ à la Church, then $\Gamma \vdash |M| : \sigma$ in $\lambda 2$ à la Curry.*
If $\Gamma \vdash P : \sigma$ in $\lambda 2$ à la Curry, then there is an M such that $|M| \equiv P$ and $\Gamma \vdash M : \sigma$ in $\lambda 2$ à la Church.

The proof is by straightforward induction on the derivations. We don't give the details. In the Examples of 12, 14 and 15, we can see the Proposition at work: an erasure of the Church style derivation gives a Curry style derivation. The other way around: any Curry style derivation can be "dressed up" with type information to obtain a Church style derivation. The undecidability of type checking in $\lambda 2$ à la Curry can thus be rephrased as: we cannot algorithmically reconstruct the missing type information in an untyped term. In Example 15 we have seen two completely different ways to "dress up" the term $\lambda x.x\,x$ to make it typable in $\lambda 2$-Church. This is a general pattern: there are many possible ways to add typing information to an non-typable term to make it typable in $\lambda 2$-Church.

We have opposed "weak polymorphism" to "full polymorphism" and we referred to the first also as ML-style polymorphism. It is the case that polymorphism in most functional programming languages is weaker than full polymorphism, and if we restrict ourselves to the pure λ-calculus, it is just the weak polymorphic system that we have described. However, to regain some of the "full polymorphism", ML has additional constructs, like *let polymorphism*.

$$\frac{\Gamma \vdash M : \sigma \;\; \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash \mathsf{let}\, x = M \,\mathsf{in}\, N : \tau} \text{ for } \tau \text{ a } \lambda{\to}\text{-type}, \sigma \text{ a } \lambda 2\text{-type}$$

We can see a term $\mathsf{let}\, x = M \,\mathsf{in}\, N$ as a β-redex $(\lambda x{:}\sigma.N)M$. So the let-rule allows the formation of a β-redex $(\lambda x{:}\sigma.N)M$, for σ a polymorphic type, while we cannot form the abstraction term $\lambda x{:}\sigma.N : \sigma{\to}\tau$. So it is still impossible to have a universal quantifier deeper inside a type, though we can have a polymorphic term as a subterm of a well-typed term. For the extension of the principal type algorithm to include weak polymorphism with lets we refer to [30].

Exercise 12. 1. Type the term $(\lambda f.f\,f)(\lambda x.x)$ in (full) $\lambda 2$-Curry.
2. Type the term $\mathsf{let}\, f = \lambda x.x \,\mathsf{in}\, f\,f$ in weak $\lambda 2$-Curry with the let-rule as given above.

5.3 Meta-theoretic Properties

We now recall the decidability issues of Definition 20 and look into them for $\lambda 2$.

Theorem 9. *Decidability properties of the (weak and full) polymorphic λ-calculus*

- *TIP is decidable for the weak polymorphic λ-calculus, undecidable for the full polymorphic λ-calculus.*
- *TCP and TSP are equivalent.*

– *TCP*	*à la Church*	*à la Curry*
ML-style	*decidable*	*decidable*
System F-style	*decidable*	*undecidable*

With full polymorphism (system F), untyped terms contain too little information to compute the type [41]. TIP is equivalent to provability in logic. For full $\lambda 2$, this is *second order intuitionistic proposition logic* (to be discussed in the next Section), which is known to be undecidable. For weak $\lambda 2$, the logic is just a very weak extension of PROP which is decidable.

5.4 Formulas-as-types for full $\lambda 2$

There is a formulas-as-types isomorphism between full system-F style $\lambda 2$ and second order proposition logic, PROP2.

Definition 32. *Derivation rules of* PROP2*:*

$$\frac{\begin{array}{c}\tau_1 \dots \tau_n \\ \vdots \\ \sigma\end{array}}{\forall\alpha.\sigma}\ \forall\text{-}I,\ \textit{if}\ \alpha \notin FV(\tau_1,\dots,\tau_n) \qquad \frac{\forall\alpha.\sigma}{\sigma[\alpha := \tau]}$$

NB This is constructive second order proposition logic: *Peirce's law*

$$\forall\alpha.\forall\beta.((\alpha\to\beta)\to\alpha)\to\alpha$$

is not derivable (so there is no closed term of this type). The logic only has implication and universal quantification, but the other connectives are now definable.

Definition 33. *Definability of the other intuitionistic connectives.*

$$\begin{aligned}\bot &:= \forall\alpha.\alpha\\ \sigma\wedge\tau &:= \forall\alpha.(\sigma\to\tau\to\alpha)\to\alpha\\ \sigma\vee\tau &:= \forall\alpha.(\sigma\to\alpha)\to(\tau\to\alpha)\to\alpha\\ \exists\alpha.\sigma &:= \forall\beta.(\forall\alpha.\sigma\to\beta)\to\beta\end{aligned}$$

Proposition 4. *All the standard constructive deduction rules (elimination and introduction rules) are derivable using the definitions in 33*

Example 16. We show the derivability of the $\wedge$-elimination rule by showing how to derive σ from $\sigma \wedge \tau$:

$$\dfrac{\dfrac{\forall\alpha.(\sigma\to\tau\to\alpha)\to\alpha}{(\sigma\to\tau\to\sigma)\to\sigma} \quad \dfrac{\dfrac{[\sigma]^1}{\tau\to\sigma}}{\sigma\to\tau\to\sigma}\,1}{\sigma}$$

There is a formulas-as-types embedding of PROP2 into $\lambda 2$ that maps deductions to terms. It can be defined inductively, as we have done for PROP and $\lambda\!\!\rightarrow$ in Definition 10. We don't give the definitions, but illustrate it by an example.

Example 17. The definable $\wedge$-elimination rule in PROP2 under the deductions-as-terms embedding yields a λ-terms that "witnesses" the construction of an object of type σ out of an object of type $\sigma \wedge \tau$.

$$\dfrac{\dfrac{M : \forall\alpha.(\sigma\!\rightarrow\!\tau\!\rightarrow\!\alpha)\!\rightarrow\!\alpha}{M\sigma : (\sigma\!\rightarrow\!\tau\!\rightarrow\!\sigma)\!\rightarrow\!\sigma} \quad \dfrac{\dfrac{[x:\sigma]^1}{\lambda y{:}\tau.x : \tau\!\rightarrow\!\sigma}}{\lambda x{:}\sigma.\lambda y{:}\tau.x : \sigma\!\rightarrow\!\tau\!\rightarrow\!\sigma}\,1}{M\sigma(\lambda x{:}\sigma.\lambda y{:}\tau.x) : \sigma}$$

So the following term is a "witness" for the $\wedge$-elimination.

$$\lambda z{:}\sigma\!\wedge\!\tau.z\,\sigma\,(\lambda x{:}\sigma.\lambda y{:}\tau.x) : (\sigma \wedge \tau)\!\rightarrow\!\sigma$$

Exercise 13. Prove the derivability of some of the other logical rules:

1. Define $\mathsf{inl} : \sigma \rightarrow \sigma \vee \tau$
2. Define pairing : $\langle -, - \rangle : \sigma \rightarrow \tau \rightarrow \sigma \times \tau$
3. Given $f : \sigma\!\rightarrow\!\rho$ and $g : \tau\!\rightarrow\!\rho$, construct a term $\mathsf{case}\ f\ g : \sigma \vee \tau\!\rightarrow\!\rho$

5.5 Data types in $\lambda 2$

In $\lambda\!\!\rightarrow$ we can define a type of "natural numbers over a type σ": $(\sigma\!\rightarrow\!\sigma)\!\rightarrow\!\sigma\!\rightarrow\!\sigma$. In $\lambda 2$, we can define this type polymorphically.

$$\mathsf{Nat} := \forall\alpha.(\alpha\!\rightarrow\!\alpha)\!\rightarrow\!\alpha\!\rightarrow\!\alpha$$

This type uses the encoding of natural numbers as Church numerals

$$n \mapsto c_n := \lambda f.\lambda x.f(\ldots(fx)) \quad n\text{-times } f$$

- $0 := \lambda\alpha.\lambda f{:}\alpha\!\rightarrow\!\alpha.\lambda x{:}\alpha.x$
- $S := \lambda n{:}\mathsf{Nat}.\lambda\alpha.\lambda f{:}\alpha\!\rightarrow\!\alpha.\lambda x{:}\alpha.f(n\,\alpha\,x\,f)$

Proposition 5. *Over the type* Nat*, functions can be defined by* iteration*: if $c : \sigma$ and $g : \sigma\!\rightarrow\!\sigma$, then there is a function*

$$\mathsf{It}\,c\,g : \mathsf{Nat}\!\rightarrow\!\sigma$$

satisfying

$$\mathsf{It}\,c\,g\,0 = c$$
$$\mathsf{It}\,c\,g\,(S\,x) = g(\mathsf{It}\,c\,g\,x)$$

Proof. $\mathsf{It}\,c\,g$: $\mathsf{Nat}\!\rightarrow\!\sigma$ is defined as $\lambda n{:}\mathsf{Nat}.n\,\sigma\,g\,c$. It is left as a (quite ease) exercise to see that this satisfies the equations.

The function It acts as an iterator: it takes a "begin value" c and a map g and n times iterates g on c, where n is the natural number input. So $\mathsf{It}\,c\,g\,n = g(\ldots(g\,c))$, with n times g.

Examples 18. 1. Addition

$$\mathsf{Plus} := \lambda n{:}\mathsf{Nat}.\lambda m{:}\mathsf{Nat}.\mathsf{It}\,m\,S\,n$$

or if we unfold the definition of It: $\mathsf{Plus} := \lambda n{:}\mathsf{Nat}.\lambda m{:}\mathsf{Nat}.n\,\mathsf{Nat}\,S\,m$, which says: iterate the $+1$ function n times on begin value m.

2. Multiplication

$$\mathsf{Mult} := \lambda n{:}\mathsf{Nat}.\lambda m{:}\mathsf{Nat}.\mathsf{It}\,0\,(\lambda x{:}\mathsf{Nat}.\mathsf{Plus}\,m\,x)\,n$$

The predecessor is notably difficult to define! The easiest way to define it is by first defining *primitive recursion* on the natural numbers. This means that if we have $c : \sigma$ and $f : \mathsf{Nat}{\to}\sigma{\to}\sigma$, we want to define a term $\mathsf{Rec}\,c\,f : \mathsf{Nat}{\to}\sigma$ satisfying

$$\begin{aligned}\mathsf{Rec}\,c\,f\,0 &= c\\ \mathsf{Rec}\,c\,f\,(S\,x) &= f\,x\,(\mathsf{Rec}\,c\,f\,x)\end{aligned}$$

(Note that if we can define functions by primitive recursion, the predecessor is just $P := \mathsf{Rec}\,0\,(\lambda x\,y : \mathsf{Nat}.x)$.)

It is known that primitive recursion can be encoded in terms of iteration, so therefore we can define the predecessor in $\lambda 2$. However, the complexity (in terms of the number of reduction steps) of this encoding is very bad. As a consequence:

$$\mathsf{Pred}(n+1) \longrightarrow\!\!\!\!\!\rightarrow_\beta n$$

in a number of steps of $O(n)$.

Exercise 14. 1. Complete the details in the proof of Proposition 5

2. Verify in detail that addition and multiplication as defined in Example 18 behave as expected.
3. Define the data type $\mathsf{Three} := \forall\alpha{:}*.\alpha{\to}\alpha{\to}\alpha{\to}\alpha$.
   (a) Give three different closed inhabitants of the type Three in $\lambda 2$ à la Church: $\mathsf{one}, \mathsf{two}, \mathsf{three} : \mathsf{Three}$.
   (b) Define a function $\mathsf{Shift} : \mathsf{Three} \to \mathsf{Three}$ that does the following

$$\begin{aligned}\mathsf{Shift}\,\mathsf{one} &=_\beta \mathsf{two}\\ \mathsf{Shift}\,\mathsf{two} &=_\beta \mathsf{three}\\ \mathsf{Shift}\,\mathsf{three} &=_\beta \mathsf{one}\end{aligned}$$

Apart from the natural numbers, many other algebraic data types are definable in $\lambda 2$. Here is the example of lists over a base type A.

$$\mathsf{List}_A := \forall\alpha.\alpha{\to}(A{\to}\alpha{\to}\alpha){\to}\alpha$$

The representation of lists over A as terms of this type uses the following encoding

$$[a_1, a_2, \ldots, a_n] \mapsto \lambda x.\lambda f. f a_1 (f a_2 (\ldots (f a_n x))) \quad n\text{-times } f$$

We can now define the constructors Nil (empty list) and Cons (to "cons" an element to a list) as follows.

- $\mathsf{Nil} := \lambda\alpha.\lambda x{:}\alpha.\lambda f{:}A{\to}\alpha{\to}\alpha.x$
- $\mathsf{Cons} := \lambda a{:}A.\lambda l{:}\mathsf{List}_A.\lambda\alpha.\lambda x{:}\alpha.\lambda f{:}A{\to}\alpha{\to}\alpha. f\, a (l\, \alpha\, x\, f)$

Note that the definition of Cons conforms with the representation of lists given above.

Proposition 6. *Over the type* List_A *we can define functions by* iteration*: if* $c : \sigma$ *and* $g : A{\to}\sigma{\to}\sigma$*, then there is a function*

$$\mathsf{It}\, c\, g : \mathsf{List}_A{\to}\sigma$$

satisfying

$$\mathsf{It}\, c\, g\, \mathsf{Nil} = c$$
$$\mathsf{It}\, c\, g\, (\mathsf{Cons}\, a\, l) = g\, a\, (\mathsf{It}\, c\, g\, l)$$

Proof. $\mathsf{It}\, c\, g : \mathsf{List}_A{\to}\sigma$ is defined as $\lambda l{:}\mathsf{List}_A. l\, \sigma\, c\, g$. This satisfies the equations in the proposition. Basically, we have, for $l = [a_1, \ldots, a_n]$, $\mathsf{It}\, c\, g\, l = g\, a_1 (\ldots (g\, a_n\, c))$ (n times g).

Example 19. A standard function one wants to define over lists is the "map" function, which given a function on the carrier type, extends it to a function on the lists over this carrier type. Given $f : \sigma{\to}\tau$, $\mathsf{Map}\, f : \mathsf{List}_\sigma{\to}\mathsf{List}_\tau$ should apply f to all elements in a list. It is defined by

$$\mathsf{Map} := \lambda f{:}\sigma{\to}\tau.\mathsf{It}\, \mathsf{Nil} (\lambda x{:}\sigma.\lambda l{:}\mathsf{List}_\tau.\mathsf{Cons}(f\, x) l).$$

Then

$$\begin{aligned} \mathsf{Map}\, f\, \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{Map}\, f\, (\mathsf{Cons}\, a\, k) &= \mathsf{It}\, \mathsf{Nil} (\lambda x{:}\sigma.\lambda l{:}\mathsf{List}_\tau.\mathsf{Cons}(f\, x) l)\, (\mathsf{Cons}\, a\, k) \\ &= (\lambda x{:}\sigma.\lambda l{:}\mathsf{List}_\tau.\mathsf{Cons}(f\, x) l) a (\mathsf{Map}\, f\, k) \\ &= \mathsf{Cons}(f\, a)(\mathsf{Map}\, f\, k) \end{aligned}$$

This is exactly the recursion equation for Map that we would expect.

Many more data-types can be defined in $\lambda 2$. The product of two data-types is defined in the same way as the (logical) conjunction: $\sigma{\times}\tau := \forall\alpha.(\sigma{\to}\tau{\to}\alpha){\to}\alpha$. we have already seen how to define projections and pairing (the $\wedge$-elimination

and $\wedge$-introduction rules). The disjoint union (or sum) of two data-types is defined in the same way as the logical disjunction: $\sigma{+}\tau := \forall\alpha.(\sigma{\to}\alpha){\to}(\tau{\to}\alpha){\to}\alpha$. We can define inl, inr and case. The type of binary trees with nodes in A and leaves in B can be defined as follows.

$$\mathsf{Tree}_{A,B} := \forall\alpha.(B{\to}\alpha){\to}(A{\to}\alpha{\to}\alpha{\to}\alpha){\to}\alpha$$

and we define $\mathsf{leaf} : B{\to}\mathsf{Tree}_{A,B}$ by $\mathsf{leaf} := \lambda b{:}B.\lambda\alpha.\lambda l{:}B{\to}\alpha.\lambda j{:}A{\to}\alpha{\to}\alpha{\to}\alpha.\, l\, b$.

Exercises 15.
- Define a function of type $\mathsf{List}_A{\to}\mathsf{Nat}$ that computes the length of a list.
- Define $\mathsf{join} : \mathsf{Tree}_{A,B} \to \mathsf{Tree}_{A,B} \to A \to \mathsf{Tree}_{A,B}$ that takes two trees and a node label and builds a tree.
- Define a function $\mathsf{TreeSum} : \mathsf{Tree}_{\mathsf{Nat},\mathsf{Nat}}{\to}\mathsf{Nat}$ that computes the sum of all leaves and nodes in a binary tree.
- Give the *iteration scheme* over binary trees and show that it is definable in $\lambda 2$.

5.6 Meta-theory of $\lambda 2$; Strong Normalization

Theorem 10.
- *For $\lambda 2$ à la Church: Uniqueness of types*
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma = \tau$.
- *Subject Reduction*
  If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.
- *Strong Normalization*
  If $\Gamma \vdash M : \sigma$, then all β-reductions from M terminate.

The third property is remarkably complicated and we address it in detail below. The first two are proved by induction on the derivation, using some other meta-theoretic properties, that one also proves by induction over the derivation (the Substitution Property we have already seen for types; this is the same property for terms):

Proposition 7.
- Substitution property
  If $\Gamma, x : \tau, \Delta \vdash M : \sigma$, $\Gamma \vdash P : \tau$, then $\Gamma, \Delta \vdash M[x := P] : \sigma$.
- Thinning
  If $\Gamma \vdash M : \sigma$ and $\Gamma \subseteq \Delta$, then $\Delta \vdash M : \sigma$.

We now elaborate on the proof of Strong Normalization of β-reduction for $\lambda 2$. The proof is an extension of the Tait proof of SN for $\lambda{\to}$, but it needs some crucial extra ingredients that have been developed by Girard [21]. We motivate these additional notions below.

In $\lambda 2$ à la Church there are two kinds of β-reductions:

- Kind 1, term-applied-to-term: $(\lambda x{:}\sigma.M)P \longrightarrow_\beta M[x := P]$
- Kind 2, term-applied-to-type: $(\lambda\alpha.M)\tau \longrightarrow_\beta M[\alpha := \tau]$

The second kind of reductions does no harm: (i) there are no infinite β-reduction paths with solely reductions of kind 2; (ii) if $M \longrightarrow_\beta N$ with a reduction of kind 2, then $|M| \equiv |N|$. So, if there is an infinite β-reduction in $\lambda 2$-Church, then there is one in $\lambda 2$-Curry and we are done if we prove SN for $\lambda 2$ à la Curry.

Recall the model construction in the proof for $\lambda\!\!\rightarrow$:

- $[\![\alpha]\!] := \mathsf{SN}$.
- $[\![\sigma\!\rightarrow\!\tau]\!] := \{M \mid \forall N \in [\![\sigma]\!](MN \in [\![\tau]\!])\}$.

So, now the question is: How to define $[\![\forall\alpha.\sigma]\!]$? A natural guess would be to define $[\![\forall\alpha.\sigma]\!] := \mathit{\Pi}_{X\in U}[\![\sigma]\!]_{\alpha:=X}$, where U is some set capturing all possible interpretations of types. This $\mathit{\Pi}$-set is a set of functions that take an element X of U (an interpretation of a type) and yield an element of the interpretation of σ where we assign X to α.

But now the problem is that $\mathit{\Pi}_{X\in U}[\![\sigma]\!]_{\alpha:=X}$ gets too big: if there is any type with more than one element (something we would require for a model), then $\mathrm{card}(\mathit{\Pi}_{X\in U}[\![\sigma]\!]_{\alpha:=X}) > \mathrm{card}(U)$. The cardinality of the interpretation of $\forall\alpha.\sigma$ would be larger than the set it is a member of and that's impossible. So we cannot interpret the $\forall$ as a $\mathit{\Pi}$-set (or a union for the same reason).

Girard has given the solution to this problem: $[\![\forall\alpha.\sigma]\!]$ should be very small:

$$[\![\forall\alpha.\sigma]\!] := \bigcap_{X\in U} [\![\sigma]\!]_{\alpha:=X}$$

This conforms with the idea that $\forall\alpha.\sigma$ is the type of terms that act *parametrically* on a type. A lot of literature has been devoted to the nature of polymorphism, for which the terminology *parametricity* has been introduced, intuitively saying that a function operates on types *without looking into them*. So a parametric function cannot act differently on Nat and Bool. This implies, e.g. that there is only one parametric function from α to α: the identity. See [1] for more on parametricity.

The second important novelty of Girard is the actual definition of U, the collection of all *possible interpretations* of types. U will be defined as SAT, the collection of *saturated sets* of (untyped) λ-terms.

Definition 34. *$X \subset \mathit{\Lambda}$ is* saturated *if*

- $xP_1 \dots P_n \in X$ *(for all $x \in \mathsf{Var}$, $P_1, \dots, P_n \in \mathsf{SN}$)*
- $X \subseteq \mathsf{SN}$
- *If $M[x := N]\boldsymbol{P} \in X$ and $N \in \mathsf{SN}$, then $(\lambda x.M)N\boldsymbol{P} \in X$.*

The definition of saturated sets basically arises by taking the closure properties that we *proved* for the interpretation of $\lambda\!\!\rightarrow$-types as a *definition* of the collection of possible interpretation of $\lambda 2$-types.

Definition 35. *Let $\rho : \mathsf{TVar} \to SAT$ be a* valuation *of type variables. Define $[\![\sigma]\!]_\rho$ by:*

- $[\![\alpha]\!]_\rho := \rho(\alpha)$
- $[\![\sigma{\to}\tau]\!]_\rho := \{M | \forall N \in [\![\sigma]\!]_\rho (MN \in [\![\tau]\!]_\rho)\}$
- $[\![\forall\alpha.\sigma]\!]_\rho := \cap_{X \in SAT} [\![\sigma]\!]_{\rho,\alpha:=X}$

Proposition 8.

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \sigma \Rightarrow M[x_1 := P_1, \ldots, x_n : P_n] \in [\![\sigma]\!]_\rho$$

for all valuations ρ *and* $P_1 \in [\![\tau_1]\!]_\rho, \ldots, P_n \in [\![\tau_n]\!]_\rho$

The proof is by induction on the derivation of $\Gamma \vdash M : \sigma$.

Corollary 2. $\lambda 2$ *is SN*

Proof. Take P_1 to be x_1, ..., P_n to be x_n.

Exercise 16. Verify the details of the proof of the Proposition.

We end this section with some remarks on semantics. In the section on $\lambda{\to}$ we have seen that the SN proof consists of the construction of a model, where the interpretation of the function type is "small" (not the full function space). But for $\lambda{\to}$, there are also models where the function type is the full set-theoretic function space. These are often referred to as *set-theoretical models* of type theory.

Theorem 11 (Reynolds[36]). $\lambda 2$ *does not have a non-trivial set-theoretic model.*

This is a remarkable theorem, also because there are no requirements for the interpretation of the other constructs, only that the model is sound. The proof proceeds by showing that if $[\![\sigma{\to}\tau]\!] := [\![\tau]\!]^{[\![\sigma]\!]}$ (the set theoretic function space), then $[\![\sigma]\!]$ is a singleton set for every σ. We call such a model *trivial*, as all types are interpreted as the empty set or the singleton set. This is a sound interpretation, corresponding with the interpretation of the formulas of PROP2 in a classical way (suing a truth table semantics). It is also called the *proof irrelevance semantics* as all proofs of propositions are identified. As said, it is a sound model, but not a very interesting one, certainly from a programmers point of view, because all natural numbers are identified.

So we can rephrase Reynolds result as: in an interesting $\lambda 2$-model, $[\![\sigma{\to}\tau]\!]$ must be small.

6 Dependent Type Theory

In the paper by Bove and Dybjer, we can see "Dependent Types at Work" and that paper also gives some of the history and intuitions behind it. In this Section I will present the rules. The problem with dependent types is that "everything depends on everything", so we can't first define the types and then the terms. We will have two "universes": **type** and **kind**. Dybjer and Bove used "Set" for

what I call **type**: the universe of types. (We can't have **type** : **type**, so to type **type** we need to have another universe: **type** : **kind**.)

we define first order dependent type theory, $\lambda\mathsf{P}$. This system is also known as LF (Logical Framework, [22]) The judgements of the system are of the form

$$\Gamma \vdash M : B$$

where

- Γ is a context
- M and B are terms taken from the set of *pseudo-terms*.

Definition 36. *The set of pseudo-terms is defined by*

$$\mathsf{T} ::= \mathsf{Var} \mid \mathbf{type} \mid \mathbf{kind} \mid (\mathsf{TT}) \mid (\lambda x{:}\mathsf{T}.\mathsf{T}) \mid \Pi x{:}\mathsf{T}.\mathsf{T},$$

Furthermore, there is an *auxiliary* judgement

$$\Gamma \vdash$$

to denote that Γ is a *correct context*.

Definition 37. *The* derivation rules *of* $\lambda\mathsf{P}$ *are (*$\mathbf{s}$ *ranges over* $\{\mathbf{type}, \mathbf{kind}\}$*):*

$$(\mathit{base})\ \emptyset \vdash \quad (\mathit{ctxt})\ \frac{\Gamma \vdash A : \mathbf{s}}{\Gamma, x{:}A \vdash}\ \text{if } x \text{ not in } \Gamma \quad (\mathit{ax})\ \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{type} : \mathbf{kind}}$$

$$(\mathit{proj})\ \frac{\Gamma \vdash}{\Gamma \vdash x : A}\ \text{if } x{:}A \in \Gamma \quad (\Pi)\ \frac{\Gamma, x{:}A \vdash B : \mathbf{s} \ \ \Gamma \vdash A : \mathbf{type}}{\Gamma \vdash \Pi x{:}A.B : \mathbf{s}}$$

$$(\lambda)\ \frac{\Gamma, x{:}A \vdash M : B \ \ \Gamma \vdash \Pi x{:}A.B : \mathbf{s}}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}\ (\mathit{app})\ \frac{\Gamma \vdash M : \Pi x{:}A.B \ \ \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$(\mathit{conv})\ \frac{\Gamma \vdash M : B \ \ \Gamma \vdash A : \mathbf{s}}{\Gamma \vdash M : A} A =_{\beta\eta} B$$

In this type theory, we have a new phenomenon, which is the *Π-type*:

$$\Pi x{:}A.B(x) \simeq \text{the type of functions } f \text{ such that}$$
$$f\,a : B(a) \text{ for all } a{:}A$$

The Π-type is a generalization of the well-known function type: if $x \notin \mathrm{FV}(B)$, then $\Pi x{:}A.B$ is just $A{\to}B$. So, we will use the arrow notation $A{\to}B$ as an abbreviation for $\Pi x{:}A.B$ in case $x \notin \mathrm{FV}(B)$. The Π rule allows to form two forms of function types in $\lambda\mathsf{P}$

$$(\Pi)\ \frac{\Gamma \vdash A : \mathbf{type} \ \ \Gamma, x{:}A \vdash B : \mathbf{s}}{\Gamma \vdash \Pi x{:}A.B : \mathbf{s}}$$

- With $\mathbf{s} = \mathbf{kind}$, we can form $A{\to}A{\to}\mathbf{type}$ and $A{\to}\mathbf{type}$.
- With $\mathbf{s} = \mathbf{type}$ and $P : A{\to}\mathbf{type}$, we can form $A{\to}A$ and $\Pi x{:}A.P\,x{\to}P\,x$.

6.1 Formulas-as-types: minimal predicate logic into λP

Following the methods for $\lambda\!\!\rightarrow$ and $\lambda 2$, we can embed logic into first order dependent type theory following the Curry-Howard formulas-as-types embedding. For $\lambda\mathsf{P}$, we can embed *minimal first order predicate logic*, the predicate logic with just implication and universal quantification and the intuitionistic rules for these connectives. The idea is to represent *both the domains and the formulas* of the logic as types. In predicate logic we need to interpret a *signature* for the logic, that tells us which constants, functions and relations there are (and, in case of many-sortedness, which are the domains). This is not difficult but it involves some overhead when giving a precise formal definition. So, we don't give a completely formal definition but just an example.

Example 20. Minimal first order predicate logic over one domain with one constant, one unary function and two unary and one binary relation is embedded into $\lambda\mathsf{P}$ by considering the context

$$\Gamma := A : \mathbf{type}, a : A, f : A{\rightarrow}A, P : A{\rightarrow}\mathbf{type}, Q : A{\rightarrow}\mathbf{type}, R : A{\rightarrow}A{\rightarrow}\mathbf{type}.$$

Implication is represented as $\rightarrow$ and $\forall$ is represented as Π:

$$\forall x{:}A.P\,x \mapsto \Pi x{:}A.P\,x$$
$$\forall x{:}A.R\,x\,x{\rightarrow}P\,x \mapsto \Pi x{:}A.R\,x\,x{\rightarrow}P\,x$$

the intro and elim rules are just λ-abstraction and application, both for implication and universal quantification.

The terms of type A act as the first order terms of the language: a, $f\,a$, $f(f\,a)$ etc. The formulas are encoded as terms of type **type**: $P\,a$, $R\,a\,a$ are the closed atomic formulas and with $\rightarrow$, Π and variables we build the first order formulas from that.

In $\lambda\mathsf{P}$, we can give a precise derivation that the context Γ is correct: $\Gamma \vdash$. These derivations are quite lengthy, because in a derivation tree the same judgment is derived several times in different branches of the tree. Therefore such derivations are best given in *flag style*. It should be clear by now how we turn a set of derivations rules into a flag format. We will usually omit derivations of the correctness of a context, but for completeness we here give one example, in flag format. We give a precise derivation of the judgment

$$A : \mathbf{type}, P : A{\rightarrow}\mathbf{type}, a : A \vdash P\,a{\rightarrow}\mathbf{type} : \mathbf{kind}$$

NB. we use the $\rightarrow$-formation rule as a degenerate case of the Π-formation rule (if $x \notin \mathrm{FV}(B)$).

$$\frac{\Gamma \vdash A : \mathbf{type}\ \ \Gamma \vdash B : \mathbf{s}}{\Gamma \vdash A \rightarrow B : \mathbf{s}}\ {\rightarrow}\text{-form}$$

1	$\mathbf{type} : \mathbf{kind}$	
2	$A : \mathbf{type}$	ctxt-proj, 1
3	$A \to \mathbf{type} : \mathbf{kind}$	$\to$-form, 2, 1
4	$P : A \to \mathbf{type}$	ctxt-proj, 3
5	$a : A$	ctxt-proj, 2
6	$P\,a : \mathbf{type}$	app, 4, 5
7	$P\,a \to \mathbf{type} : \mathbf{kind}$	$\to$-form, 6, 1

Example 21. We illustrate the use of application and abstraction to encode elimination and introduction rules of the logic. take Γ to be the context of Example 20.

$$\Gamma \vdash \lambda z{:}A.\lambda h{:}(\Pi x, y{:}A.R\,x\,y).h\,z\,z : \Pi z{:}A.(\Pi x, y{:}A.R\,x\,y){\to}R\,z\,z$$

This term is a proof of $\forall z{:}A.(\forall x, y{:}A.R(x, y)){\to}R(z, z)$. The first λ encodes a $\forall$-introduction, the second λ an implication-introduction.

Example 22. We now show how to construct a term of type $(\Pi x{:}A.P\,x{\to}Q\,x){\to}(\Pi x{:}A.P\,x){\to}\Pi x{:}A.Q\,x$ in the context Γ. We do this by giving a derivation in "flag style", where we omit derivations of the well-formedness of types and contexts. We write σ for $(\Pi x{:}A.P\,x{\to}Q\,x){\to}(\Pi x{:}A.P\,x){\to}\Pi x{:}A.Q\,x$.

1	$A : \mathbf{type}$	
2	$P : A \to \mathbf{type}$	
3	$Q : A \to \mathbf{type}$	
4	$h : \Pi x{:}A.P\,x{\to}Q\,x$	
5	$g : \Pi x{:}A.P\,x$	
6	$x : A$	
7	$h\,x : P\,x \to Q\,x$	app, 4, 6
8	$g\,x : P\,x$	app, 5, 6
9	$h\,x(g\,x) : Q\,x$	app, 7, 8
10	$\lambda x{:}A.h\,x(g\,x) : \Pi x{:}A.Q\,x$	λ-rule, 6, 9
11	$\lambda g{:}\Pi x{:}A.P\,x.\lambda x{:}A.h\,x(g\,x) : (\Pi x{:}A.P\,x) \to \Pi x{:}A.Q\,x$	λ-rule, 5, 10
12	$\lambda h{:}\Pi x{:}A.P\,x{\to}Q\,x.\lambda g{:}\Pi x{:}A.P\,x.\lambda x{:}A.h\,x(g\,x) : \sigma$	λ-rule, 4, 11

So:

$$\Gamma \vdash \lambda h{:}\Pi x{:}A.P\,x{\to}Q\,x.\lambda g{:}\Pi x{:}A.P\,x.\lambda x{:}A.h\,x(g\,x) : \sigma$$

Exercise 17. 1. Find terms of the following types (NB $\to$ binds strongest)

$$(\Pi x{:}A.P\,x{\to}Q\,x){\to}(\Pi x{:}A.P\,x){\to}\Pi x{:}A.Q\,x$$

and

$$(\Pi x{:}A.P\,x{\to}\Pi z.R\,z\,z){\to}(\Pi x{:}A.P\,x){\to}\Pi z{:}A.R\,z\,z).$$

2. Find a term of the following type and write down the context in which this term is typed.

$$(\Pi x{:}A.P\,x{\to}Q){\to}(\Pi x{:}A.P\,x){\to}Q$$

What is special about your context? (It should somehow explicitly state that the type A is not empty.)

The representation that we have just described is called the *direct encoding* of logic in type theory. This is the formulas-as-types embedding originally due to Curry and Howard and described first in formal detail in [25]. Apart from this, there is the *LF encoding* of logic in type theory. This is the formulas-as-types embedding as it was invented by De Bruijn in his Automath project [31]. We describe it now.

6.2 LF embedding of logic in type theory

For $\lambda{\to}$, $\lambda 2$ and $\lambda\mathsf{P}$ we have seen *direct* representations of logic in type theory. Characteristics of such an encoding are:

– Connectives each have a counterpart in the type theory:

implication	$\sim$ $\to$-type
universal quantification	$\sim$ $\forall$-type

– Logical rules have their direct counterpart in type theory:

$\to$-introduction	$\sim$ λ-abstraction
$\to$- elimination	$\sim$ application
$\forall$-introduction	$\sim$ λ-abstraction
$\forall$-elimination	$\sim$ application

– the context declares a *signature*, *local variables* and *assumptions*.

There is another way of interpreting logic in type theory, due to De Bruijn, which we call the *logical framework* representation of logic in type theory. The idea is to use type theory as a framework in which various logics can be encoded by choosing an appropriate context. Characteristics of the LF encoding are:

– Type theory is used as a *meta system* for encoding ones own logic.
– The context is used as a *signature for the logic*: one chooses an appropriate context Γ_L in which the logic L (including its proof rules) is declared.
– The type system is a meta-calculus for dealing with *substitution* and *binding*.

We can put these two embeddings side by side by looking at the trivial proof of A implies A.

	proof	formula
direct embedding	$\lambda x{:}A.x$	$A{\to}A$
LF embedding	$\mathsf{imp\_intr}\,A\,A\,\lambda x{:}T\,A.x$	$T(A \Rightarrow A)$

For the LF embedding of minimal proposition logic into $\lambda\mathsf{P}$, we need the following context.

$$\begin{aligned} \Rightarrow &: \mathsf{prop}\to\mathsf{prop}\to\mathsf{prop} \\ \mathsf{T} &: \mathsf{prop}\to\mathbf{type} \\ \mathsf{imp\_intr} &: (A,B:\mathsf{prop})(\mathsf{T}\,A\to\mathsf{T}\,B)\to\mathsf{T}(A\Rightarrow B) \\ \mathsf{imp\_el} &: (A,B:\mathsf{prop})\mathsf{T}(A\Rightarrow B)\to\mathsf{T}\,A\to\mathsf{T}\,B. \end{aligned}$$

The idea is that prop is the type of names of propositions and that T "lifts" a name φ to the type of its proofs $T\,\varphi$. The terms $\mathsf{imp\_intr}$ and $\mathsf{imp\_el}$ encode the introduction and elimination for implication.

Exercise 18. Verify that $\mathsf{imp\_intr}\,A\,A\,\lambda x{:}T\,A.x : T(A\Rightarrow A)$ in the context just described.

In the following table we summarize the difference between the two encodings

Direct embedding	LF embedding
One type system : One logic	One type system : Many logics
Logical rules $\sim$ type theoretic rules	Logical rules $\sim$ context declarations

Apart from this, a direct embedding aims at describing a formulas–as-types *isomorphism* between the logic and the type theory, whereas the LF idea is to provide a system for enabling a formulas–as–types embedding for many different logics. For $\lambda\!\to$ and $\lambda 2$ there is indeed a one-one correspondence between deductions in logic and typable terms in the type theory. For the case of $\lambda\mathsf{P}$ and minimal predicate logic, this is not so obvious, as we have identified the domains and the formulas completely: they are all of type $\mathbf{type}$. This gives rise to types of the form $\Pi x{:}A.P\,x\to A$ and allows to form predicates over formulas, like $B : (\Pi x{:}A.R\,x\,x)\to\mathbf{type}$, that don't have a correspondence in the logic. It can nevertheless be shown that the direct embedding of PRED into $\lambda\mathsf{P}$ is complete, but that requires some effort. See [18] for details.

Now, we show some examples of logics in the logical framework LF – which is just $\lambda\mathsf{P}$. Then we exhibit the properties of LF that make this work.

Minimal propositional logic in $\lambda\mathsf{P}$ Fix the signature (context) of minimal propositional logic.

$$\begin{aligned} \mathsf{prop} &: \mathbf{type} \\ \mathsf{imp} &: \mathsf{prop}\to\mathsf{prop}\to\mathsf{prop} \end{aligned}$$

As a notation we introduce

$$A\Rightarrow B \text{ for } \mathsf{imp}\,A\,B$$

The type prop is the type of 'names' of propositions. A term of type prop can not be inhabited (proved), as it is not a type. We 'lift' a name $p : \mathsf{prop}$ to the type of its proofs by introducing the following map:

$$\mathsf{T} : \mathsf{prop}\to\mathbf{type}.$$

The intended meaning of $\mathsf{T}p$ is 'the type of proofs of p'. We interpret 'p is valid' by '$\mathsf{T}p$ is inhabited'. To derive $\mathsf{T}p$ we also encode the logical derivation rules by adding to the context

$$\begin{aligned}\mathsf{imp\_intr} &: \Pi p, q : \mathsf{prop}.(\mathsf{T}p \to \mathsf{T}q) \to \mathsf{T}(p \Rightarrow q),\\ \mathsf{imp\_el} &: \Pi p, q : \mathsf{prop}.\mathsf{T}(p \Rightarrow q) \to \mathsf{T}p \to \mathsf{T}q.\end{aligned}$$

$\mathsf{imp\_intr}$ takes two (names of) propositions p and q and a term $f : \mathsf{T}\,p \to \mathsf{T}\,q$ and returns a term of type $\mathsf{T}(p \Rightarrow q)$ Indeed $A \Rightarrow A$ is now valid: $\mathsf{imp\_intr}A\,A(\lambda x{:}\mathsf{T}\,A.x) : \mathsf{T}(A \Rightarrow A)$

Exercise 19. Construct a term of type $\mathsf{T}(A \Rightarrow (B \Rightarrow A))$ in the context with $A, B : \mathsf{prop}$.

Definition 38. *Define Σ_{PROP} to be the signature for minimal proposition logic, PROP, as just constructed.*

Now, why would this be a "good" encoding? Are all derivations represented as terms in $\lambda\mathsf{P}$? And if a type is inhabited, is the associated formula then provable? We have the following desired properties of the encoding.

Definition 39. – Soundness *of the encoding states that*

$$\vdash_{PROP} A \Rightarrow \Sigma_{PROP}, a_1{:}\mathsf{prop}, \ldots, a_n{:}\mathsf{prop} \vdash p : \mathsf{T}\,A \text{ for some } p.$$

where $\{a, \ldots, a_n\}$ is the set of proposition variables in A.
– Adequacy *(or completeness) states the converse:*

$$\Sigma_{PROP}, a_1{:}\mathsf{prop}, \ldots, a_n{:}\mathsf{prop} \vdash p : \mathsf{T}\,A \Rightarrow \vdash_{PROP} A$$

Proposition 9. *The LF encoding of* PROP *in $\lambda\mathsf{P}$ is sound and adequate.*

The proof of soundness is by induction on the derivation of $\vdash_{\mathrm{PROP}} A$. Adequacy also holds, but it is more involved to prove. One needs to define a canonical form of terms of type $\mathsf{T}\,A$ (the so called *long $\beta\eta$-normal-form*) and show that these are in one-one correspondence with proofs. See [22] for details.

Minimal predicate logic over one domain A in $\lambda\mathsf{P}$ Signature:

$$\begin{aligned}\mathsf{prop} &: \mathbf{type},\\ A &: \mathbf{type},\\ \mathsf{T} &: \mathsf{prop} \to \mathbf{type}\\ \mathsf{f} &: A \to A,\\ \mathsf{R} &: A \to A \to \mathsf{prop},\\ \Rightarrow &: \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop},\\ \mathsf{imp\_intr} &: \Pi p, q : \mathsf{prop}.(\mathsf{T}p \to \mathsf{T}q) \to \mathsf{T}(p \Rightarrow q),\\ \mathsf{imp\_el} &: \Pi p, q : \mathsf{prop}.\mathsf{T}(p \Rightarrow q) \to \mathsf{T}p \to \mathsf{T}q.\end{aligned}$$

Now we encode $\forall$ by observing that $\forall$ takes a $P : A{\to}\mathsf{prop}$ and returns a proposition, so:

$$\forall : (A{\to}\mathsf{prop}){\to}\mathsf{prop}$$

Universal quantification $\forall x{:}A.(Px)$ is then translated by $\forall(\lambda x{:}A.(Px))$

Definition 40. *The signature: Σ_{PRED} is defined by adding to the above the following intro and elim rules for $\forall$.*

$$\begin{aligned}\forall &: (A{\to}\mathsf{prop}){\to}\mathsf{prop},\\ \forall\_\mathsf{intr} &: \Pi P{:}A{\to}\mathsf{prop}.(\Pi x{:}A.\mathsf{T}(Px)){\to}\mathsf{T}(\forall P),\\ \forall\_\mathsf{elim} &: \Pi P{:}A{\to}\mathsf{prop}.\mathsf{T}(\forall P){\to}\Pi x{:}A.\mathsf{T}(Px).\end{aligned}$$

The proof of

$$\forall z{:}A(\forall x, y{:}A.Rxy) \Rightarrow Rzz$$

is now mirrored by the proof-term

$$\begin{aligned}\forall\_\mathsf{intr}[\_](\ &\lambda z{:}A.\mathsf{imp\_intr}[\_][\_](\lambda h{:}\mathsf{T}(\forall x, y{:}A.Rxy).\\ &\forall\_\mathsf{elim}[\_](\forall\_\mathsf{elim}[\_]hz)z) \ )\end{aligned}$$

For readability, we have replaced the instantiations of the Π-type by $[\_]$.
This term is of type

$$\mathsf{T}(\forall(\lambda z{:}A.\mathsf{imp}(\forall(\lambda x{:}A.(\forall(\lambda y{:}A.\mathsf{R}xy))))(\mathsf{R}zz)))$$

Exercise 20. Construct a proof-term that mirrors the (obvious) proof of $\forall x(P\,x \Rightarrow Q\,x) \Rightarrow \forall x.P\,x \Rightarrow \forall x.Q\,x$

Proposition 10. *We have soundness and adequacy for minimal predicate logic:*

$$\vdash_{\mathrm{PRED}} \varphi \Rightarrow \Sigma_{\mathrm{PRED}}, x_1{:}A, \ldots, x_n{:}A \vdash p : \mathsf{T}\varphi, \text{ for some } p,$$

where $\{x_1, \ldots, x_n\}$ is the set of free variables in φ.

$$\Sigma_{\mathrm{PRED}}, x_1{:}A, \ldots, x_n{:}A \vdash p : \mathsf{T}\varphi \Rightarrow \vdash_{\mathrm{PRED}} \varphi$$

6.3 Meta-theory of λP

Proposition 11.
- *Uniqueness of types*
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma{=_\beta}\tau$.
- *Subject Reduction*
  If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_\beta N$, then $\Gamma \vdash N : \sigma$.
- *Strong Normalization*
  If $\Gamma \vdash M : \sigma$, then all β-reductions from M terminate.

The proofs are by induction on the derivation, by first proving auxiliary lemmas like Substitution and Thinning. SN can be proved by defining a reduction preserving map from λP to λ→. Then, an infinite reduction path in λP would give rise to an infinite reduction path in λ→, so SN for λP follows from SN for λ→. See [22] for details. We now come back to the decidability questions of Definition 20.

Proposition 12. *For* $\lambda\mathsf{P}$*:*

- *TIP is undecidable*
- *TCP/TSP is decidable*

The undecidability of TIP follows from the fact that provability in minimal predicate logic is undecidable. A more straightforward proof is given in [7], by interpreting the halting problem for register machines as a typing problem (in a specific context) in $\lambda\mathsf{P}$.

We will expand on the decidability of TCP below. It is shown by defining two algorithms simultaneously: one that does type synthesis $\Gamma \vdash M :?$ and one that does *context checking*: $\Gamma \vdash ?$. This is to mirror the two forms of judgment in $\lambda\mathsf{P}$.

Remark 1. One can also introduce a *Curry variant* of $\lambda\mathsf{P}$. This is done in [2]. A related issue is whether one can type an *untyped* λ-term in $\lambda\mathsf{P}$. So, given an M, is there a context Γ, a type A and a term P such that $\Gamma \vdash P : A$ and $|P| \equiv M$. Here, $|-|$ is the erasure map defined by

$$\begin{array}{ll} |x| & := x \\ |\lambda x{:}\sigma.M| := |\lambda x.M| & |MN| := |M|\,|N| \end{array}$$

The answer to this question is yes, because an untyped term is $\lambda\mathsf{P}$-typable iff it is typable in $\lambda{\to}$. But there is a little snag: if we *fix* the context Γ, the problem becomes undecidable, as was shown in [14], where Dowek gives a context Γ and a term M such that $\exists P, A(\Gamma \vdash P : A \wedge |P| = M)$ is equivalent to the Post correspondence problem.

6.4 Type Checking for $\lambda\mathsf{P}$

We define algorithms Ok$(-)$ and Type_$(-)$ simultaneously:

- Ok$(-)$ takes a context and returns 'accept' or 'reject'
- Type_$(-)$ takes a context and a term and returns a term or 'reject'.

Definition 41. *The type synthesis algorithm* Type_$(-)$ *is* sound *if*

$$\mathrm{Type}_\Gamma(M) = A \;\Rightarrow\; \Gamma \vdash M : A \textit{ (for all } \Gamma \textit{ and } M\textit{)}$$

The type synthesis algorithm Type_$(-)$ *is* complete *if*

$$\Gamma \vdash M : A \;\Rightarrow\; \mathrm{Type}_\Gamma(M) =_\beta A \textit{ (for all } \Gamma \textit{ and } M\textit{)}$$

Completeness only makes sense if we have uniqueness of types: only then it makes sense to check if the type that is given to us is convertible to the type computed by Type. In case we don't have uniqueness of types, one would let Type_$(-)$ compute a set of possible types, one for each β-equivalence class.

Definition 42.

$$
\begin{aligned}
\mathrm{Ok}(<>) &= \textit{`accept'}\\
\mathrm{Ok}(\Gamma, x{:}A) &= \mathsf{if}\ \mathrm{Type}_\Gamma(A) \in \{\mathbf{type}, \mathbf{kind}\}\ \mathsf{then}\ \mathrm{Type}_\Gamma(A)\ \mathsf{else}\ \textit{`reject'},\\
\mathrm{Type}_\Gamma(x) &= \mathsf{if}\ \mathrm{Ok}(\Gamma)\ \mathsf{and}\ x{:}A \in \Gamma\ \mathsf{then}\ A\ \mathsf{else}\ \textit{`reject'},\\
\mathrm{Type}_\Gamma(\mathbf{type}) &= \mathsf{if}\ \mathrm{Ok}(\Gamma)\ \mathsf{then}\ \mathbf{kind}\ \mathsf{else}\ \textit{`reject'},\\
\mathrm{Type}_\Gamma(MN) &= \mathsf{if}\ \mathrm{Type}_\Gamma(M) = C\ \mathsf{and}\ \mathrm{Type}_\Gamma(N) = D\\
&\quad \mathsf{then}\ \ \mathsf{if}\ C \twoheadrightarrow_\beta \Pi x{:}A.B\ \mathsf{and}\ A =_\beta D\\
&\qquad \mathsf{then}\ B[x := N]\ \mathsf{else}\ \textit{`reject'}\\
&\quad \mathsf{else}\ \ \textit{`reject'},\\
\mathrm{Type}_\Gamma(\lambda x{:}A.M) &= \mathsf{if}\ \mathrm{Type}_{\Gamma,x:A}(M) = B\\
&\quad \mathsf{then}\qquad \mathsf{if}\ \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathbf{type}, \mathbf{kind}\}\\
&\qquad\qquad \mathsf{then}\ \Pi x{:}A.B\ \mathsf{else}\ \textit{`reject'}\\
&\quad \mathsf{else}\ \textit{`reject'},\\
\mathrm{Type}_\Gamma(\Pi x{:}A.B) &= \mathsf{if}\ \mathrm{Type}_\Gamma(A) = \mathbf{type}\ \mathsf{and}\ \mathrm{Type}_{\Gamma,x:A}(B) = s\\
&\quad \mathsf{then}\ s\ \mathsf{else}\ \textit{`reject'}
\end{aligned}
$$

Proposition 13. *The type checking algorithm is sound:*

$$
\begin{aligned}
\mathrm{Type}_\Gamma(M) = A &\Rightarrow \Gamma \vdash M : A\\
\mathrm{Ok}(\Gamma) = \textit{`accept'} &\Rightarrow \Gamma \vdash
\end{aligned}
$$

The proof is by simultaneous induction on the computation of Type and Ok.

For completeness, we need to prove the following simultaneously, which we would prove by induction on the derivation.

$$
\begin{aligned}
\Gamma \vdash A : \mathbf{s} &\Rightarrow \mathrm{Type}_\Gamma(A) = \mathbf{s}\\
\Gamma \vdash M : A &\Rightarrow \mathrm{Type}_\Gamma(M) =_\beta A\\
\Gamma \vdash &\Rightarrow \mathrm{Ok}(\Gamma) = \text{`accept'}
\end{aligned}
$$

The first slight strengthening of completeness is not a problem: in case the type of A is **type** or **kind**, $\mathrm{Type}_\Gamma(A)$ returns exactly **type** or **kind** (and not a term $=_\beta$-equal to it). The problem is the λ-rule, where $\mathrm{Type}_{\Gamma,x:A}(M) = C$ and $C =_\beta B$ and we know that $\mathrm{Type}_\Gamma(\Pi x{:}A.B) = \mathbf{s}$, but we need to know that $\mathrm{Type}_\Gamma(\Pi x{:}A.C) = \mathbf{s}$, because that is the side condition in the Type algorithm for the $\lambda x{:}A.M$ case.

The solution is to change the definition of Type a little bit. This is motivated by the following Lemma, which is specific to the type theory $\lambda\mathsf{P}$.

Lemma 6. *The derivable judgements of $\lambda\mathsf{P}$ remain exactly the same if we replace the λ-rule by*

$$
(\lambda')\ \frac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash A : \mathbf{type}}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}
$$

The proof is by induction on the derivation. Now we can in the λ-case of the definition of Type replace the side condition

$$\mathsf{if}\ \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathbf{type}, \mathbf{kind}\}$$

by

$$\mathsf{if}\ \mathrm{Type}_\Gamma(A) \in \{\mathbf{type}\}$$

Definition 43. *We adapt the definition of* Type *in Definition 42 by replacing the λ-abstraction case by*

$$\begin{array}{rl}\mathrm{Type}_\Gamma(\lambda x{:}A.M) = & \mathsf{if}\ \mathrm{Type}_{\Gamma,x:A}(M) = B \\ & \quad \mathsf{then} \quad \mathsf{if}\ \mathrm{Type}_\Gamma(A) = \mathbf{type} \\ & \qquad\qquad \mathsf{then}\ \Pi x{:}A.B\ \mathsf{else}\ \textit{`reject'} \\ & \quad \mathsf{else}\ \textit{`reject'},\end{array}$$

Then soundness still holds and we have the following.

Proposition 14.

$$\begin{aligned}\Gamma \vdash A : \mathbf{s} &\Rightarrow \mathrm{Type}_\Gamma(A) = \mathbf{s} \\ \Gamma \vdash M : A &\Rightarrow \mathrm{Type}_\Gamma(M) =_\beta A \\ \Gamma \vdash &\Rightarrow \mathrm{Ok}(\Gamma) = \textit{`accept'}\end{aligned}$$

As a consequence of soundness and completeness we find that

$$\mathrm{Type}_\Gamma(M) = \text{`reject'} \quad \Rightarrow \quad M \text{ is not typable in } \Gamma$$

Completeness implies that Type terminates correctly on *all well-typed terms*. But we want that Type terminates on *all pseudo terms*: we want to assure that on a non-typable term, Type returns 'reject', which is not guaranteed by Soundness and Completeness.

To prove that Type_(−) terminates on all inputs, we need to make sure that $\mathrm{Type}_\Gamma(M)$ and $\mathrm{Ok}(\Gamma)$ are called on arguments of lesser size. As "size" we take the sum of the lengths of the context Γ and the term M, and then all cases are decreasing, apart from λ-abstraction and application. In the λ-abstraction case, Type is called on a pseudo-term $\Pi x{:}A.B$ that is not necessarily smaller. But our replacement of the side condition in Type for the λ-abstraction case in Definition 43 solves this problem.

In the case of application, the function Type is called on smaller inputs, but the algorithms requires β-equality and β-reduction checking:

$$\begin{array}{rl}\mathrm{Type}_\Gamma(MN) = & \mathsf{if}\ \mathrm{Type}_\Gamma(M) = C\ \mathsf{and}\ \mathrm{Type}_\Gamma(N) = D \\ & \quad \mathsf{then}\ \ \mathsf{if}\ C \longrightarrow\!\!\!\!\!\rightarrow_\beta \Pi x{:}A.B\ \mathsf{and}\ A =_\beta D \\ & \qquad\quad \mathsf{then}\ B[x := N]\ \mathsf{else}\ \text{`reject'} \\ & \quad \mathsf{else}\ \ \text{`reject'},\end{array}$$

So, we need to decide β-reduction and β-equality, which, for pseudo-terms is undecidable. The solution is that Type will only check β-equality (and reduction) for *well-typed* terms, and we know that $\lambda\mathsf{P}$ is SN and CR, so this is decidable.

Proposition 15. *Termination of* Type *and* Ok*: For all pseudo-terms M and pseudo-contexts Γ,* $\mathrm{Type}_\Gamma(M)$ *terminates (in either a pseudo-term A or 'reject') and* $\mathrm{Ok}(\Gamma)$ *terminates (in either 'accept' or 'reject').*

The proof is by induction on the size of the inputs, using the Soundness (Proposition 13) and decidability of β-equality for well-typed terms in the application case.

7 Conclusion and further reading

In this paper we have introduced various type theories by focussing on the Curry-Howard formulas-as-types embedding and by highlighting some of the programming aspects related to type theory. As stated in the Introduction, we could have presented the type systems à la Church in a unified framework of the "λ-cube" or "Pure Type Systems", but for expository reasons we have refrained from doing so. In the PTS framework, one can study the systems $F\omega$, a higher order extension of system F, $\lambda\mathsf{HOL}$, a type systems for higher order (predicate) logic, and the Calculus of Constructions, a higher order extension of $\lambda\mathsf{P}$. Also one can generalize these systems to the logically inconsistent systems λU and $\lambda\star$ where **type** is itself a type. (These systems are computationally not inconsistent so still interesting to study.) We refer to [4, 5] and the references in those papers for further reading.

In another direction, there are various typing constructs that can be added to make the theories more powerful. The most prominent one is the addition of a scheme for inductive types, which also adds an induction and a well-founded recursion principle for every inductive type. Examples are the Calculus of Inductive Constructions (CIC) and Martin-Löf type theory. The latter is introduced in the paper by Bove and Dybjer and a good reference is [32]. CIC is the type theory implemented in the proof assistant Coq and a good reference for both the theory and the tool is [6].

In the direction of programming, there is a world of literature and a good starting point is [33]. We have already mentioned PCF [35], which is the simplest way to turn $\lambda\!\!\rightarrow$ into a Turing complete language. The next thing to add are algebraic data types, which are the programmer's analogue to inductive types. Using these one can define and compute with data types of lists, trees etc. as a primitive in the language (as opposed to coding them in $\lambda 2$ as we have done in Section 5.5). Other important concepts to add, especially to the type systems à la Curry, are overloading and subtyping. A good reference for further reading is [33].

References

1. E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott, Functorial polymorphism. *Theoretical Computer Science* 70, 35-64, 1990.

2. S. van Bakel, L. Liquori, S. Ronchi Della Rocca and P. Urzyczyn, Comparing Cubes of Typed and Type Assignment Systems. *Ann. Pure Appl. Logic* 86(3): 267-303, 1997.
3. H. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, vol. 103 of Studies in Logic and the Foundations of Mathematics, North Holland, 1981.
4. H. Barendregt and H. Geuvers, Proof Assistants using Dependent Type Systems, Chapter 18 of the *Handbook of Automated Reasoning* (Vol 2), eds. A. Robinson and A. Voronkov, Elsevier, pp. 1149 – 1238, 2001.
5. H. Barendregt, Lambda Calculi with Types, *Handbook of Logic in Computer Science*, Volume 1, Abramsky, Gabbay, Maibaum (Eds.), Clarendon, 1992.
6. Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art : the Calculus of Inductive Constructions*, EATCS Series: Texts in Theoretical Computer Science, Springer 2004, 469 pp.
7. M. Bezem and J. Springintveld, A Simple Proof of the Undecidability of Inhabitation in λP. *J. Funct. Program.* 6(5): 757-761, 1996.
8. C. Böhm and A. Berarducci, Automatic synthesis of typed λ-programs on term algebras, *Theoretical Computer Science* 39, pp. 135154, 1985.
9. A. Church, A formulation of the simple theory of types, *Journal of Symbolic Logic* 5, pp. 5668, 1940.
10. R.L Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki and S.F. Smith, *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall, NJ, 1986.
11. H.B. Curry, R. Feys, and W. Craig, *Combinatory Logic*, Vol. 1. North–Holland, Amsterdam, 1958.
12. L. Damas and R. Milner, Principal type-schemes for functional programs, *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207-212, ACM, 1982.
13. M. Davis, editor, *The Undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problems And Computable Functions*, Raven Press, New York, 1965.
14. G. Dowek, The Undecidability of Typability in the Lambda-Pi-Calculus *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, LNCS 664, pp. 139 - 145, 1993.
15. G. Dowek, Collections, sets and types, *Mathematical Structures in Computer Science* 9, pp. 1-15, 1999.
16. F. Fitch, *Symbolic Logic, An Introduction*, The Ronald Press Company, 1952.
17. R.O. Gandy, An early proof of normalization by A.M. Turing. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 453455. Academic Press, 1980.
18. H. Geuvers and E. Barendsen, Some logical and syntactical observations concerning the first order dependent type system λP, *Math. Struc. in Comp. Sci.* vol. 9-4, 1999, pp. 335 – 360.
19. J.-Y. Girard, P. Taylor, and Y. Lafont, *Proofs and types*. Cambridge tracts in theoretical computer science, no. 7. Cambridge University Press.
20. J.-Y. Girard, The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159192, 1986.
21. J.-Y. Girard, Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types, *Proceedings of the Second Scandinavian Logic Symposium*, pp. 63–92. Studies in Logic and the Foundations of Mathematics, Vol. 63, North-Holland, Amsterdam, 1971.

22. R. Harper, F. Honsell and G. Plotkin, A Framework for Defining Logics, *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science*, LICS'87, Ithaca, NY, USA, 22–25 June 1987
23. J.R. Hindley, The principal type-scheme of an object in combinatory logic, *Transactions of the American Mathematical Society* 146 (1969), 29–60.
24. J.R. Hindley and J.P. Seldin, *Introduction to combinators and lambda-calculus*. London Mathematical Society Student Texts. Cambridge University Press, 1986.
25. W. Howard, The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.
26. F. Joachimski, R. Matthes, Short proofs of normalization for the simply- typed lambda-calculus, permutative conversions and Gödel's T. *Arch. Math. Log.* 42(1): 59-87, 2003.
27. J. Lambek and P. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986.
28. P. Martin-Löf, *Intuitionistic type theory*, Bibliopolis, 1984.
29. R. Milner, Robin, A Theory of Type Polymorphism in Programming, *JCSS* 17: 348-375, 1978.
30. A. Mycroft, Polymorphic type schemes and recursive definitions, *International Symposium on Programming*, LNCS 167, pp. 217-228, Springer 1984.
31. R. Nederpelt, H. Geuvers and R. de Vrijer (eds.) *Selected Papers on Automath*, Studies in Logic, Vol. 133, North-Holland, 1994.
32. B. Nordström, K. Petersson and J. Smith, *Programming in Martin-Löf's Type Theory, An Introduction*, Oxford University Press, 1990.
33. Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002.
34. G. Plotkin, Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comp. Sci.* 1(2): 125-159, 1975.
35. G. Plotkin, LCF considered as a programming language, *Theor. Comp. Sci.* 5, pp. 223255, 1977.
36. J.C. Reynolds, Polymorphism is not Set-Theoretic. In G. Kahn, D.B. MacQueen, G. Plotkin (Eds.): *Semantics of Data Types*, International Symposium, Sophia-Antipolis, France, June 27-29, LNCS 173, Springer 1984, pp. 145-156.
37. H. Schwichtenberg, Definierbare Funktionen im λ-Kalkül mit Typen, *Archiv für Mathematische Logik und Grundlagenforschung* 17, pp. 113114, 1976.
38. W.W. Tait, Intensional interpretation of functionals of finite type. *J. Symbol. Logic*, v. 32, n. 2, pages 187–199, 1967.
39. P. Urzyczyn and M. Sørensen, *Lectures on the Curry-Howard Isomorphism*, Volume 149 of Studies in Logic and the Foundations of Mathematics, Elsevier, 2006.
40. M. Wand, A simple algorithm and proof for type inference,*Fundamenta Informaticae* X, pp. 115122, 1987.
41. J.B Wells, Typability and type-checking in the second-order λ-calculus are equivalent and undecidable, *Proceedings of the 9th Annual Symposium on Logic in Computer Science*, Paris, France, IEEE Computer Society Press, pp. 176185, 1994.