Martin-Löf's type theory

Hugo Herbelin

February 6, 2020

Abstract

In the 70's, Martin-Löf introduced a kind a formalism now called *dependent type theory*, or *modern type theory*[1], or *Martin-Löf's type theory*, or simply *type theory*, which is both a logic and a programming language. This note gives one presentation of Martin-Löf's type theory, placing it in its historical context and comparing it to other presentation of type theory.

1 Introduction

From intuitionism to the proof-as-programs correspondence via realisability and cut-elimination At the beginning of the 20^{th} century, the mathematician L.E.J. Brouwer [Bro] defended the idea that proofs should be constructive processes, rejecting the law of excluded-middle, and leading to a controversy became famous between a group of formalist mathematicians represented by Hilbert and a group of *intuitionistic* mathematicians represented by Brouwer.

The intuitionistic view became formal with Heyting [Hey28] and Kolmogorov [Kol67] who defined an intuitionistic restriction of first-order logic by simply discarding the excluded-middle.

An highlight of the constructive aspect of intuitionistic proofs was obtained with Kleene's realisability [Kle45]: to any intuitionistic proof of an arithmetic statement can be associated a recursive function which *realises* the statement in the sense that the recursive function will compute a witness for any existential statement $\exists n\, P(n)$. This was modified by Gödel [Göd58] ("Dialectica" functional interpretation) and Kreisel [Kre59] (modified realisability) to use instead an extension of Church's λ-calculus [Chu32] with recursion over numbers in all types (so-called System T). Later, this was extended to second-order and higher-order arithmetic by Girard, using respectively System F and System F_ω [Gir71].

Independently, Gentzen designed in 1935 two new formalisms, *natural deduction* and *sequent calculus* [Gen35], showing in particular that the latter comes with a notion of simplification of proofs called *cut-elimination*. Prawitz showed in 1965 that the former also has a notion of simplification called *normalisation* [Pra65].

Curry had already noticed a correspondence between combinatory logic and axiomatic Hilbert's style systems [Cur34][CFC58, §9E]. A few years after Prawitz, Howard reported an identity of structure between natural deduction and simply-typed λ-calculus, as well as a correspondence between normalisation in natural deduction and normalisation in λ-calculus. This was a key observation for Martin-Löf to develop a formalism which is both a logic and a programming language.

Related textbooks:

- System T, System F, proofs-as-programs correspondence: *Proof and types*, Jean-Yves Girard, Yves Lafont, Paul Taylor (textbook from 1987).
- Realisablity: *Introduction to Constructive Logic and Mathematic*, Thomas Streicher (lecture notes from 2000).

[1] To distinguish it from Russell-Whitehead's ramified type theory in 1908.

- General historical resources on intuitionism: *From Frege to Gödel, A Source Book in Mathematical Logic, 1879-1931*, Jean van Heijenoort, 1977.

Type Theory In 1970, Martin-Löf's introduced a concise typed λ-calculus $\mathsf{Type} : \mathsf{Type}$ usable both as a programming language and a logic. This was shown inconsistent by Girard, thus this first type theory (in the modern sense) was a naive type theory in the same sense as Cantor's set theory was a naive set theory. Martin-Löf then developed along the years new (predicative) type theories, whose key features are: the presence of Π-types, Σ-types and basic data-types (natural numbers, empty type, unit type, ...), the presence of an internal equality called *judgmental equality* or *definitional equality*.

Another lineage comes from de Bruijn who developed in the 60's a proof assistant called *Automath* [dB68]. For this purpose, he developed variants of Church's Higher-Order Logic [Chu40] also based, independently of Howard, on the identity of structure between proofs and programs. This work led to the investigation by Scott of ideas similar to type theory [Sco70]. It also led to Coquand's Calculus of Constructions [Coq85] and Coquand and Paulin-Mohring's Calculus of Inductive of Constructions [CPM90] (the logic of Coq) which can be seen as an *impredicative* variant of Martin-Löf's type theory. All this also led to a general form of type systems for λ-calculus called Pure Type Systems [Ber88, Ter89, Geu93] which are an implementation-oriented presentation of type theory with only Π-types.

Related references:

- Pure Type Systems: *Lambda Calculus with Types*, Henk Barendregt, in Handbook of Logic in Computer Science, volume II, 1991.
- The 1990 view at Martin-Löf's type theory: *Programming in Martin-Löf's Type Theory*, Bengt Nordström, Kent Petersson, and Jan M. Smith, 1990.
- The 2013 view at Martin-Löf's type theory: *Homotopy Type Theory: Univalent Foundations of Mathematics*, the Univalent Foundations Programme, 2013.
- Realisablity in type theory: *Réalisabilité et Paramétricité dans les Systèmes de Types Purs*, PhD, Marc Lasson, 2012.
- General resources: The NLab collaborative wiki on Mathematics, Physics and Philosophy (see url `https://ncatlab.org/nlab`).

2 The different forms of type theory

On the contrary of Zermelo-Fraenkel set theory whose presentation is relatively standard as a first-order predicate logic equipped with axioms, type theory exists in several presentations. These presentations differ in the following respects:

- Which connectives are available?

  For a long time, type theory (and more generally type systems) were seen as extensions of λ-calculus. This is why for instance System F, $\mathsf{Type} : \mathsf{Type}$, the Calculus of Constructions, or more generally Pure Type Systems contain only the Π-type connective (which is a generalisation both of implication, dependent function type, universal quantification).

  Moreover, those of these systems which are impredicative allow a *second-order encoding* of data-types [BB85] which reduces the need for a primitive notion of data-types.

  On the other side, effective programming, or effective proving requires data-types, such as natural numbers, or Boolean values with dependent elimination (i.e. induction) on these types, something which is not available even when a second-order encoding of data-types is available.

The prototypical λ-calculus with a data-type is System T, which itself can be seen as the language of primitive recursion extended to recursion on higher-order types (i.e. not only on $\mathbb{N}$ but also on $\mathbf{N} \to \mathbf{N}$, $(\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$, and so on). Martin-Löf's predicative type theories also comes with primitive data-types, and so does the Calculus of Inductive Constructions, where the word "Inductive" precisely refers to the presence of such data-types.

Nowadays, the usage is to recognize an essential difference between *negative* types (such as Π-type) and *positive* types (such as natural numbers) and to classify types as such. So, nowadays, the terminology *inductive* refers to *recursive* types, possibly mixing positive and negative connectives (e.g. natural numbers, lists, or trees), contrasting with the terminology *coinductive* referring to *corecursive* types (e.g. streams).

- Which form of primitive equality?

  A key feature of type theory is that reasoning is done up to some primitive equality of the theory (compare to strict vs weak equality in ∞-category theory). This primitive equality can however be presented in different ways.

  - Typed vs untyped primitive equality

    Reasoning modulo a primitive equality can be found as early as in Church's Higher Order Logic where this equality, the *β-conversion*, is untyped. In Martin-Löf's work, the conversion is on the other hand typed and called *judgemental equality*, or *definitional equality*.

    Judgemental equality is easier to build translation to other systems, e.g. to build set-theoretic models of type theory. On the other side, basic meta-theoretical properties are more difficult to establish than with untyped (conversion) equality.

  - Oriented vs equational equality

    A focus can be put on the computational aspects of the equality, so that it organizes as a confluent and terminating rewriting systems, thus a *decidable* equality, as in Church's Higher Order Logic (HOL). Or only the equational theory can be considered as in Martin-Löf's work.

    For the purpose of implementation (Automath - based on a constructive variant similar to F_ω of Church's HOL -, Isabelle/HOL, Coq), a decidable untyped computational primitive equality is generally convenient.

    On the other side, techniques exist for deciding a typed equational primitive equality (so-called Normalisation-by-Evaluation). An advantage of typed equality is that it makes support for η-conversion easier[2].

  - Intensional vs more extensional equality

    The tradition is to include in the primitive equality only β or η-like conversions. Nothing forbids however in general to support arbitrary rewriting systems, as soon as this is decidable. An example of type theory where equaility is defined with arbitrary rewriting systems is the Calculus of Algebraic Constructions. An example of related implementation is CoqMT.

    Variants of type theories with undecidable equality also exist. This is the case e.g. of Martin-Löf's Extensional Type Theory (ETT), constrasting with usual Type Theory which is then contrastingly called Intensional Type Theory (ITT).

  - Implicit vs explicit equality

    Conversely, the primitive equality can in general be replaced by axioms so that no primitive equality is present at all. This is a line of work followed e.g. by Hofmann [Hof95] to embed Extensional Type Theory into Intensional Type Theory. This is also studied e.g. by Geuvers, van Doorn and Wiedijk in the context of Pure Type Systems.

[2] η-reduction for Π-type is costly to implement, so, the natural way to orient η for Π-type is as an expansion but this requires typing. Similarly, η-reduction for the unit type requires typing, otherwise there is no known way to decide the equality of variables $x = y$, which holds if and only if the equality is in an empty of unit type.

- Which level of explicitation of the coercion between terms and types, or of the subtyping: Russell vs Tarski typing?

  - Coercion between terms and types

    Pure Type Systems are generally presented with terms and types living in the same "soup" of expressions, with only typing able to distinguish what is term and what is type. In reference to Russell and Whitehead ramified type theory, this approach is called *à la Russell*.

    On the other side, Martin-Löf's work makes an explicit distinction between terms and types, with a coercion commonly written El (for "elements") or T. In this case, every type has a denotation as a term and the function El maps the denotation as a term into a proper type. This approach is called *à la Tarski*.

    The latter is generally useful for model interpretation, but former generally more convenient for concrete uses, as in proof assistants.

  - Explicit subtyping

    Type theories generally come with a stratified cumulative hierarchy of universes, where cumulative means that the n^{th} universe is included in the $n+1^{\text{th}}$ one. This inclusion can be made explicit à la Tarski, as done e.g. in Palmgren [Pal98], or à la Russell, as originally done in Russell and Whitehead ramified type theory but also in the Calculus of Constructions generalised with universes [Coq85], or in Pure Type Systems with subtyping.

- Model-oriented vs syntax-directed?

  - Syntax-directed sytems

    Most presentations of type theories, e.g. Pure Type Systems or Martin-Löf's work, rely on an explicit rule emphasising that the logic is up to a primitive equality defined as the reflexive-symmetric-transitive closure of β-reduction.

    For the purpose of implementation, it is however convenient to attach the use of the primitive equality to the rules which may really need it, thus presenting the type theory as a type inference function. Little as been written on this approach (see e.g. Pollack), but this approach is important in implementations. In particular, $\Gamma \vdash t : A$ can be seen as describing a function taking as argument a well-formed context Γ and a term t and either returns a type A such that the judgement $\Gamma \vdash t : A$ holds, or fails if no such A exists.

  - Explicit proof terms: Intrinsic vs extrinsic syntax

    The model-theoretic approach of type theory based on Dybjer's Categories with Families [Dyb95] leads to a term-free presentation of type theory where the information is only contained in the derivation tree. This approach is called *intrinsic* and its relation to the term-based approach (constrastingly called *extrinsic*) is an active domain of research these days.

    In particular, the extrinsic syntax based on terms tends to be now perceived as a way to make the intrinsic syntax implementable, with the terms containing the minimal information of a derivation needed to be able to reconstruct a full typing derivation.

3 The standard "mathematical" presentation

There are different presentations of Martin-Löf's type theory. This section describes a version with oriented judgemental equality, and with terms and types living in the same syntactic category à la Russell. For an alternative presentation where terms and types live in different syntactic categories, said à la Tarski, see e.g. Martin-Löf, *Intuitionistic type theory*, 1984.

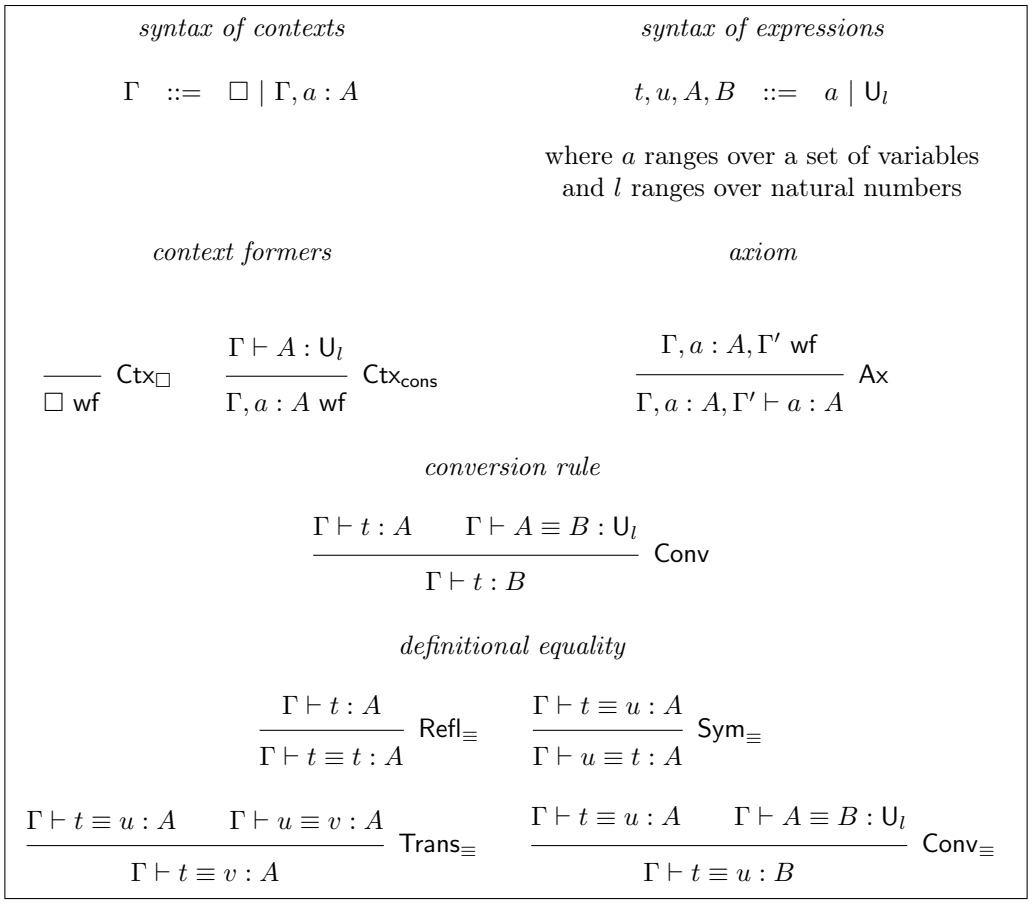

Figure 1: Core structure of type theory

3.1 Core type theory

[Version note: to simplify, the reference to $\triangleright$ has been removed; the current version does not make any more a difference between $\triangleright$ and $\equiv$; otherwise said, we do not highlight anymore that some reduction are worth being oriented in a particular direction: everything is equational.]

The core, connective-free infrastructure is presented on Figure 1. We adopt a view where types and terms live in the same soup of raw expressions. Types correspond to the subset of terms whose type is some universe U_l.

3.2 Universes

Extension with a hierarchy of universe is obtained with the rules on Figure 2. For a presentation of the hierarchy of universes in the context of type theory à la Tarski, i.e. for a presentation distinguishing between terms and types, see Eric Palmgren, *On Universes in Type Theory*, 1998.

3.3 Identity type

Extension with an identity type is obtained with the rules on Figure 3. What we write $\mathsf{subst}\, t\, \mathsf{in}\, u$ is also commonly written $J\, t\, u$ (the J operator).

type former

$$\frac{}{\Gamma \vdash \mathsf{U}_l : \mathsf{U}_{l+1}}\ \mathsf{U}$$

Figure 2: Universes in type theory

extended syntax of expressions

$$t, u, v, A, B, p, q \quad ::= \quad \ldots \mid t =_A u \mid \mathsf{refl}\, t \mid \mathsf{subst}\, p\, \mathsf{in}\, v$$

type former

$$\frac{\Gamma \vdash A : \mathsf{U}_l \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t =_A u : \mathsf{U}_l}$$

introduction rule

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{refl}\, t : t =_A t}$$

elimination rule

$$\frac{\Gamma \vdash p : t =_A u \qquad \Gamma, a : A, b : t =_A a \vdash P : \mathsf{U}_l \qquad \Gamma \vdash v : P[t/a][\mathsf{refl}\, t/b]}{\Gamma \vdash \mathsf{subst}\, p\, \mathsf{in}\, v : P[u/a][p/b]}$$

β-conversion rules

$$\frac{\Gamma \vdash t : A \qquad \Gamma, a : A, b : t =_A a \vdash B : \mathsf{U}_l \qquad \Gamma \vdash v : B[t/a][\mathsf{refl}\, t/b]}{\Gamma \vdash \mathsf{subst}\, (\mathsf{refl}\, t)\, \mathsf{in}\, v \equiv v : B[t/a][\mathsf{refl}\, t/b]}$$

congruence rules

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash u \equiv u' : A}{\Gamma \vdash (t =_A u) \equiv (t' =_{A'} u') : \mathsf{U}_l}$$

$$\frac{\Gamma \vdash t \equiv t' : A}{\Gamma \vdash \mathsf{refl}\, t \equiv \mathsf{refl}\, t' : t =_A t}$$

$$\frac{\Gamma \vdash p \equiv p' : t =_A u \qquad \Gamma, a : A, b : t =_A a \vdash B : \mathsf{U}_l \qquad \Gamma \vdash v \equiv v' : B[t/a][\mathsf{refl}\, t/b]}{\Gamma \vdash \mathsf{subst}\, p\, \mathsf{in}\, v \equiv \mathsf{subst}\, p'\, \mathsf{in}\, v' : B[u/a][p/b]}$$

Figure 3: Identity type

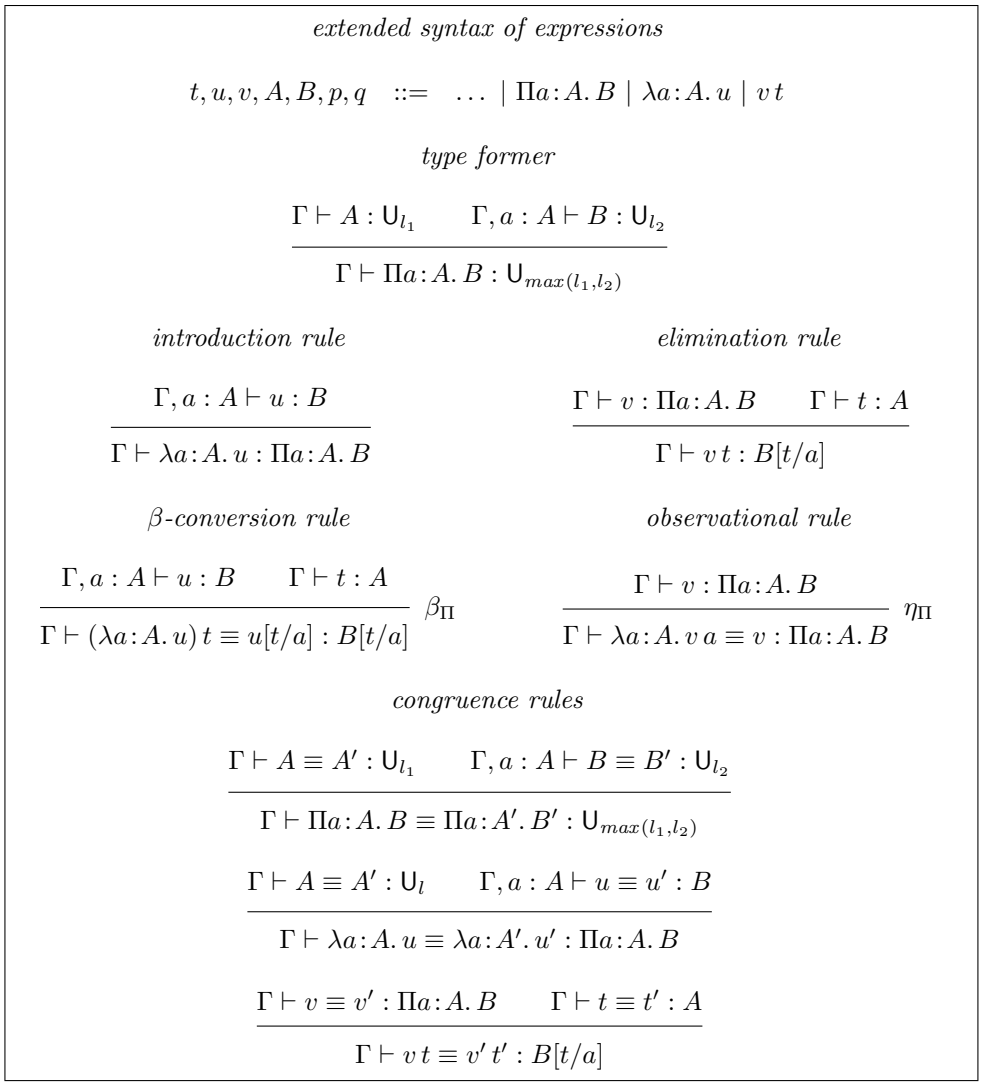

Figure 4: Typing and computational rules for Π

3.4 Dependent function type

Extension with a dependent function type is obtained with the rules on Figure 4. The arrow type, implication and universal quantification can be seen as particular cases of the type of dependent functions so, we shall occasionally use the following syntactic abbreviations:

arrow type	$A \to B$	$\triangleq$	$\Pi a\!:\!A.\, B$	for a fresh variable
implication	$A \Rightarrow B$	$\triangleq$	$\Pi a\!:\!A.\, B$	for a fresh variable
universal quantifier	$\forall a : A.B$	$\triangleq$	$\Pi a\!:\!A.\, B$	

3.5 Dependent sum type

Extension with a dependent sum type is obtained with the rules on Figure 5. The product of types, conjunction and existential quantifier can be seen as particular cases of the type of dependent sums so, we shall

extended syntax of expressions

$$t, u, v, A, B, p, q \quad ::= \quad \ldots \mid \Sigma a{:}A.\, B \mid \langle t, u\rangle \mid v.1 \mid v.2$$

type former

$$\frac{\Gamma \vdash A : \mathsf{U}_{l_1} \qquad \Gamma, a : A \vdash B : \mathsf{U}_{l_2}}{\Gamma \vdash \Sigma a{:}A.\, B : \mathsf{U}_{max(l_1,l_2)}}$$

introduction rule

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/a]}{\Gamma \vdash \langle t, u\rangle : \Sigma a{:}A.\, B}$$

elimination rules

$$\frac{\Gamma \vdash v : \Sigma a{:}A.\, B}{\Gamma \vdash v.1 : A} \qquad \frac{\Gamma \vdash v : \Sigma a{:}A.\, B}{\Gamma \vdash v.2 : B[v.1/a]}$$

β-conversion rules

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/a]}{\Gamma \vdash \langle t, u\rangle.1 \equiv t : A}\ \beta^1_\Sigma \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/a]}{\Gamma \vdash \langle t, u\rangle.2 \equiv u : B[t/a]}\ \beta^2_\Sigma$$

observational rule

$$\frac{\Gamma \vdash v : \Sigma a{:}A.\, B}{\Gamma \vdash \langle v.1, v.2\rangle \equiv v : \Sigma a{:}A.\, B}\ \eta_\Sigma$$

congruence rules

$$\frac{\Gamma \vdash A \equiv A' : \mathsf{U}_{l_1} \qquad \Gamma, a : A \vdash B \equiv B' : \mathsf{U}_{l_2}}{\Gamma \vdash \Sigma a{:}A.\, B \equiv \Sigma a{:}A'.\, B' : \mathsf{U}_{max(l_1,l_2)}}$$

$$\frac{\Gamma \vdash t \equiv t' : A \qquad \Gamma, a : A \vdash u \equiv u' : B}{\Gamma \vdash \langle t, u\rangle \equiv \langle t', u'\rangle : \Sigma a{:}A.\, B}$$

$$\frac{\Gamma \vdash v \equiv v' : \Sigma a{:}A.\, B}{\Gamma \vdash v.1 \equiv v'.1 : A} \qquad \frac{\Gamma \vdash v \equiv v' : \Sigma a{:}A.\, B}{\Gamma \vdash v.2 \equiv v'.2 : B[v.1/a]}$$

Figure 5: Typing and computational rules for Σ

occasionally use the following syntactic abbreviations:

product type	$A \times B$	$\triangleq$	$\Sigma a{:}A.\, B$	for a fresh variable
conjunction	$A \wedge B$	$\triangleq$	$\Sigma a{:}A.\, B$	for a fresh variable
existential quantifier	$\exists a : A.B$	$\triangleq$	$\Sigma a{:}A.\, B$	

3.6 Natural numbers

Extension with Peano natural numbers is obtained with the rules on Figure 6.

3.7 Streams

Extension with streams (infinite lists) is obtained with the rules on Figure 7.

extended syntax of expressions

$$t, u, v, A, B, p, q \quad ::= \quad \ldots \mid \mathbb{N} \mid 0 \mid \mathsf{succ}\, t \mid \mathsf{rec}\, [0 \mapsto t \mid \mathsf{succ}\, a \mapsto_b u]\, v$$

type former

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathsf{U}_0}$$

introduction rules

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\, t : \mathbb{N}}$$

elimination rule

$$\frac{\Gamma, a : \mathbb{N} \vdash B : \mathsf{U}_l \qquad \Gamma \vdash t : B[0/a] \qquad \Gamma, a : \mathbb{N}, b : B[n/a] \vdash u : B[\mathsf{succ}\, n/a] \qquad \Gamma \vdash v : \mathbb{N}}{\Gamma \vdash \mathsf{rec}\, [0 \mapsto t \mid \mathsf{succ}\, a \mapsto_b u]\, v : B[v/a]}$$

β-conversion rules

$$\frac{\Gamma, a : \mathbb{N} \vdash B : \mathsf{U}_l \qquad \Gamma \vdash t : B[0/a] \qquad \Gamma, a : \mathbb{N}, b : B[n/a] \vdash u : B[\mathsf{succ}\, n/a]}{\Gamma \vdash \mathsf{rec}\, [0 \mapsto t \mid \mathsf{succ}\, a \mapsto_b u]\, 0 \equiv t : B[0/a]}\ \beta^0_{\mathbb{N}}$$

$$\frac{\Gamma, a : \mathbb{N} \vdash B : \mathsf{U}_l \qquad \Gamma \vdash t : B[0/a] \qquad \Gamma, a : \mathbb{N}, b : B[n/a] \vdash u : B[\mathsf{succ}\, n/a] \qquad \Gamma \vdash v : \mathbb{N}}{\Gamma \vdash \mathsf{rec}\, [0 \mapsto t \mid \mathsf{succ}\, a \mapsto_b u]\, \mathsf{succ}\, v \equiv u[v/a][\mathsf{rec}\, [0 \mapsto t \mid \mathsf{succ}\, a \mapsto_b u]\, v/b] : B[\mathsf{succ}\, v/a]}\ \beta^{\mathsf{succ}}_{\mathbb{N}}$$

observational rule

$$\frac{\Gamma, a : \mathbb{N} \vdash E[a] : A \qquad \Gamma \vdash v : \mathbb{N}}{\Gamma \vdash \mathsf{rec}\, [0 \mapsto E[0] \mid \mathsf{succ}\, a \mapsto_b E[\mathsf{succ}\, a]]\, v \equiv E[v] : A}\ \eta_{\mathbb{N}}$$

where $E[a]$ is made only from elimination rules applied to a

congruence rules

$$\frac{\Gamma \vdash t \equiv t' : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\, t \equiv \mathsf{succ}\, t' : \mathbb{N}}$$

$$\frac{\Gamma \vdash t \equiv t' : B[0/a] \qquad \Gamma, a : \mathbb{N}, b : B[n/a] \vdash u \equiv u' : B[\mathsf{succ}\, n/a] \qquad \Gamma \vdash v \equiv v' : \mathbb{N}}{\Gamma \vdash \mathsf{rec}\, [0 \mapsto t \mid \mathsf{succ}\, a \mapsto_b u]\, v \equiv \mathsf{rec}\, [0 \mapsto t' \mid \mathsf{succ}\, a \mapsto_b u']\, v' : B[v/a]}$$

Figure 6: Typing and computational rules for $\mathbb{N}$

extended syntax of expressions

$$t, u, v, A, B, p, q \quad ::= \quad \ldots \mid \mathsf{Stream}\ A \mid t.\mathsf{hd} \mid t.\mathsf{tl} \mid \{\mathsf{hd} \mapsto t; \mathsf{tl} \mapsto_s u\}_c^v$$

type former

$$\frac{\Gamma \vdash A : \mathsf{U}_l}{\Gamma \vdash \mathsf{Stream}\ A : \mathsf{U}_l}$$

introduction rule

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, c : C \vdash t : A \qquad \Gamma, s : C \to \mathsf{Stream}\ A, c : C \vdash u : \mathsf{Stream}\ A \qquad \Gamma \vdash v : C}{\Gamma \vdash \{\mathsf{hd} \mapsto t; \mathsf{tl} \mapsto_s u\}_c^v : \mathsf{Stream}\ A}$$

elimination rules

$$\frac{\Gamma \vdash t : \mathsf{Stream}\ A}{\Gamma \vdash t.\mathsf{hd} : A} \qquad \frac{\Gamma \vdash t : \mathsf{Stream}\ A}{\Gamma \vdash t.\mathsf{tl} : \mathsf{Stream}\ A}$$

β-conversion rules

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, c : C \vdash t : A \qquad \Gamma, s : C \to \mathsf{Stream}\ A, c : C \vdash u : \mathsf{Stream}\ A \qquad \Gamma \vdash v : C}{\Gamma \vdash (\{\mathsf{hd} \mapsto t; \mathsf{tl} \mapsto_s u\}_c^v).\mathsf{hd} \equiv t[v/c] : A}\ \beta^{\mathsf{hd}}_{\mathsf{Stream}}$$

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, c : C \vdash t : A \qquad \Gamma, s : C \to \mathsf{Stream}\ A, c : C \vdash u : \mathsf{Stream}\ A \qquad \Gamma \vdash v : C}{\Gamma \vdash (\{\mathsf{hd} \mapsto t; \mathsf{tl} \mapsto_s u\}_c^v).\mathsf{tl} \equiv u[v/c][\{\mathsf{hd} \mapsto t; \mathsf{tl} \mapsto_s u\}_c^x / s\, x] : \mathsf{Stream}\ A}\ \beta^{\mathsf{tl}}_{\mathsf{Stream}}$$

observational rule

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, a : C \vdash t : \mathsf{Stream}\ A}{\Gamma \vdash \{\mathsf{hd} \mapsto t.\mathsf{hd}; \mathsf{tl} \mapsto_s t.\mathsf{tl}\}_a^a \equiv t : \mathsf{Stream}\ A}\ \eta_{\mathsf{Stream}}$$

congruence rules

$$\frac{\Gamma \vdash A \equiv A' : \mathsf{U}_l}{\Gamma \vdash \mathsf{Stream}\ A \equiv \mathsf{Stream}\ A' : \mathsf{U}_l} \qquad \frac{\Gamma \vdash t \equiv t' : \mathsf{Stream}\ A}{\Gamma \vdash t.\mathsf{hd} \equiv t'.\mathsf{hd} : A} \qquad \frac{\Gamma \vdash t \equiv t' : \mathsf{Stream}\ A}{\Gamma \vdash t.\mathsf{tl} \equiv t'.\mathsf{tl} : A}$$

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma \vdash t \equiv t' : A \qquad \Gamma, s : C \to \mathsf{Stream}\ A, c : C \vdash u \equiv u' : \mathsf{Stream}\ A \qquad \Gamma \vdash v \equiv v' : C}{\Gamma \vdash \{\mathsf{hd} \mapsto t; \mathsf{tl} \mapsto_s u\}_c^v \equiv \{\mathsf{hd} \mapsto t'; \mathsf{tl} \mapsto_s u'\}_c^{v'} : \mathsf{Stream}\ A}$$

Figure 7: Typing and computational rules for streams

3.8 Generic positive types

We give a syntax for arbitrary forms of (non-recursive) positive type, as a (non-recursive) generalization of the type $\mathbb{N}$. For that purpose, we introduce a couple of auxiliary structures.

We introduce a class of positive types, denoted by the letter P and we reuse for that purpose the notation $\otimes$ of linear logic, but this time in a dependent form (i.e. the type on the right can depend on the inhabitant of the type of the left), and in an intuitionistic setting (i.e. with contraction and weakening allowed).

We introduce a class w of inhabitants of such positive types and a class ρ of patterns for matching inhabitants of such positive types. These patterns can be declared in the context.

A positive type has the form $(c_1 : P_1 \oplus ... \oplus c_n : P_n)^w_{\rho:P}$ where w are the parameters of the type and the c_i are the names of constructors (assumed all distinct).

A constructor of this type has the form $c_i\, w$. A destructor has the form $\mathsf{case}\ t\ \mathsf{of}\ [c_1\rho \mapsto t|...|c_n\rho \mapsto t]$.

Substitution of ρ by w is as expected. Note that the axiom rule needs to be generalized so as to extract variables of a pattern.

3.9 Generic negative types

We give a syntax for arbitrary forms of (non-recursive) negative type, as a (non-recursive) generalization of the type $\mathsf{Stream}\, A$.

A negative type has the form $\{d_1 : A \,\&\, ... \,\&\, d_n : A\}^w_{\rho:P}$ where w are the parameters of the type and the d_i are the names of destructors (assumed all distinct). A constructor of this type has the form $\{d_1 \mapsto t; ...; d_n \mapsto t\}^v_c$. A destructor has the form $t\,.$

3.10 Recursive types

We give a syntax for recursive types, i.e. for types defined as smallest type generated by its constructors. Recursion is expected to occur only in *strictly positive* position, as e.g. in $\mu X.(1 \oplus X)$ (which is isomorphic to $\mathbb{N}$) or $\mu X.(1 \oplus A \otimes X)_{A:=\mathbb{N}}$ (which denotes the type of lists of natural numbers), or $\mu X.\{\mathsf{hd} : \mathbb{N} \,\&\, \mathsf{tl} : (1 \oplus X)\}$ (which denotes the negative presentation of lists).

We restricted the rules to the case of recursion on a variable $X : \mathsf{U}_l$. This could be extended to mutual recursion on a tuple of type variable. This could be extended as well to a recursion on arities, i.e. on variables X of type $\Pi a_1 : A_1. ... \Pi a_n : A_n.\, \mathsf{U}_l$ (in which case, one would also provide an instance for the arity).

The property of x guarded from a in t informally means that the recursive call x can be applied to an element of $\mu X.A$ which comes by steps of destruction of a without ever using $\mathsf{enter}\,.$

Note that the observational rule is one among other variants.

3.11 Co-recursive types

We give a syntax for co-recursive types, i.e. for types defined as greatest type generated by its constructors.

The property of x guarded in t informally means that the path to the occurrences of x in t never meet an $\mathsf{out}\,.$

Note that the observational rule is one among other variants.

References

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[Ber88] Stefano Berardi. Towards a mathematical analysis of the coquand–huet calculus of constructions and the other systems in barendregt's cube. Technical Report 66-WSK-05, Department of Computer Science, CMU, and Dipartimento Matematica, Universitá di Torino, 1988.

extended syntax of expressions

$$\begin{array}{lll} t,u,v,A,B,p,q & ::= & \ldots \mid (c_1 : P \oplus \ldots \oplus c_n : P)^w_{\rho:P} \mid \mathsf{case}\ t\ \mathsf{of}\ [c_1\rho \mapsto t | \ldots | c_n\rho \mapsto t] \mid c_i w \\ P & ::= & 1 \mid (a : A) \otimes P \\ w & ::= & () \mid (t, w) \\ \rho & ::= & () \mid (a, \rho) \\ \Gamma & ::= & \ldots \mid \rho : P \end{array}$$

type formers

$$\frac{}{\Gamma \vdash_P 1 : \mathsf{U}_l} \qquad \frac{\Gamma \vdash A : \mathsf{U}_l \qquad \Gamma, a : A \vdash_P P : \mathsf{U}_l}{\Gamma \vdash_P (a : A) \otimes P : \mathsf{U}_l} \qquad \frac{\Gamma \vdash_P P : \mathsf{U}_l \qquad \Gamma, \rho : P \vdash_P P_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \Gamma \vdash w : P \qquad \text{names } d_i \text{ disjoint}}{\Gamma \vdash (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^w_{\rho:P} : \mathsf{U}_l}$$

typing rules for instances

$$\frac{}{\Gamma \vdash_p () : 1} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash_p w : P[t/a]}{\Gamma \vdash_p (t, w) : (a : A) \otimes P}$$

typing rules for patterns

$$\frac{}{\Gamma \vdash_{pat} () : 1} \qquad \frac{\Gamma \vdash A : \mathsf{U}_l \qquad \Gamma, a : A \vdash_{pat} \rho : P}{\Gamma \vdash_{pat} (a, \rho) : (a : A) \otimes P} \qquad \frac{\Gamma\ \mathsf{wf} \qquad \Gamma \vdash_{pat} \rho : P}{\Gamma, \rho : P\ \mathsf{wf}}$$

introduction rules

$$\frac{\Gamma \vdash w' : P \qquad \Gamma, \rho : P \vdash P_j : \mathsf{U}_l \quad (1 \le j \le n) \qquad \Gamma \vdash w : P_i[w'/\rho]}{\Gamma \vdash c_i w : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^{w'}_{\rho:P}}\ (1 \le i \le n)$$

elimination rule

$$\frac{\Gamma \vdash v : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^w_{\rho:P} \qquad \Gamma, a : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^w_{\rho:P} \vdash B : \mathsf{U}_l \qquad \Gamma, \rho : P_i[w/\rho] \vdash t_i : B[c_i\,\rho/a] \quad (1 \le i \le n)}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ [c_1\rho \mapsto t_1 | \ldots | c_n\rho \mapsto t_n] : B[v/a]}$$

β-conversion rules

$$\frac{\begin{array}{l}\Gamma \vdash_P P : \mathsf{U}_l \qquad \Gamma, \rho' : P \vdash_P P_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \Gamma \vdash w' : P \qquad d_i \text{ disjoint} \\ \Gamma, a : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^{w'}_{\rho:P'} \vdash B : \mathsf{U}_l \qquad \Gamma, \rho : P_i[w'/\rho'] \vdash t_i : B[c_i\,\rho/a] \quad (1 \le i \le n) \qquad \Gamma \vdash_p w : P_i[w'/\rho']\end{array}}{\Gamma \vdash \mathsf{case}\ c_i\ w\ \mathsf{of}\ [c_1\rho \mapsto t_1 | \ldots | c_n\rho \mapsto t_n] \equiv t_i[w/\rho] : B[c_i\,w/a]}\ \beta^i_{pos}$$

observational rule

$$\frac{\Gamma \vdash v : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^w_{\rho:P} \qquad \Gamma, a : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^w_{\rho:P} \vdash E[a] : A}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ [c_1\rho \mapsto E[c_1\,\rho] | \ldots | c_n\rho \mapsto E[c_n\,\rho]] \equiv E[v] : B[v/a]}\ \eta_{pos}$$

where $E[a]$ is made only from elimination rules applied to a

congruence rules

$$\frac{\Gamma \vdash P \equiv P' : \mathsf{U}_l \qquad \Gamma, \rho : P \vdash P_i \equiv P'_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \Gamma \vdash w \equiv w' : P}{\Gamma \vdash (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^w_{\rho:P} \equiv (c_1 : P'_1 \oplus \ldots \oplus c_n : P'_n)^{w'}_{\rho:P'} : \mathsf{U}_l}$$

$$\frac{\Gamma \vdash P : \mathsf{U}_l \qquad \Gamma, \rho : P \vdash P_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \Gamma \vdash w'' : P \qquad \Gamma \vdash w \equiv w' : P_i[w''/\rho]}{\Gamma \vdash c_i\,w \equiv c_i\,w' : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^{w''}_{\rho:P}}$$

$$\frac{\Gamma \vdash v \equiv v' : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^w_{\rho':P} \qquad \Gamma, a : (c_1 : P_1 \oplus \ldots \oplus c_n : P_n)^w_{\rho':P} \vdash B : \mathsf{U}_l \qquad \Gamma, \rho : P_i[w/\rho'] \vdash t_i \equiv t'_i : B[c_i\,\rho/a] \quad (1 \le i \le n)}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ [c_1\rho \mapsto t_1 | \ldots | c_n\rho \mapsto t_n] \equiv \mathsf{case}\ v'\ \mathsf{of}\ [c_1\rho \mapsto t'_1 | \ldots | c_n\rho \mapsto t'_n] : B[v/a]}$$

$+$ judgemental equality and congruence rules for $\Gamma \vdash w : P$

Figure 8: General typing and computational rules for positive types

extended syntax of expressions

$$\begin{array}{lll} t, u, v, A, B, p, q & ::= & \ldots \mid \{d_1 : N \,\&\, \ldots \,\&\, d_n : N\}^w_{\rho:P} \mid \{d_1 \mapsto t; \ldots; d_n \mapsto t\}^v_c \mid d_i\, t \\ N & ::= & A \mid \Pi a{:}A.\, N \qquad \text{(i.e. a distinguished subset of the grammar of terms)} \end{array}$$

type formers

$$\frac{\Gamma \vdash_p w : P \qquad \Gamma, \rho : P \vdash N_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \text{names } d_i \text{ disjoint}}{\Gamma \vdash \{d_1 : N_1 \,\&\, \ldots \,\&\, d_n : N_n\}^w_{\rho:P}}$$

introduction rule

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma \vdash_p w : P \qquad \Gamma, \rho : P \vdash N_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \Gamma, c : C \vdash t_i : N_i[w/\rho] \quad (1 \le i \le n) \qquad \Gamma \vdash v : C}{\Gamma \vdash \{d_1 \mapsto t_1; \ldots; d_n \mapsto t_n\}^v_c : \{d_1 : N_1 \,\&\, \ldots \,\&\, d_n : N_n\}^w_{\rho:P}}$$

elimination rules

$$\frac{\Gamma \vdash t : \{d_1 : N_1 \,\&\, \ldots \,\&\, d_n : N_n\}^w_{\rho:P}}{\Gamma \vdash d_i\, t : N_i[w/\rho]} \; (1 \le i \le n)$$

β-conversion rules

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma \vdash_p w : P \qquad \Gamma, \rho : P \vdash N_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \Gamma, c : C \vdash t_i : N_i[w/\rho] \quad (1 \le i \le n) \qquad \Gamma \vdash v : C}{\Gamma \vdash d_i\, \{d_1 \mapsto t_1; \ldots; d_n \mapsto t_n\}^v_c \equiv t_i[v/c] : N_i[w/\rho]}\; \beta^i_{neg}$$

observational rule

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, c : C \vdash t : \{d_1 : N_1 \,\&\, \ldots \,\&\, d_n : N_n\}^w_{\rho:P}}{\Gamma \vdash \{d_1 \mapsto d_i\, t; \ldots; d_n \mapsto d_n\, t\}^c_c \equiv t : \{d_1 : N_1 \,\&\, \ldots \,\&\, d_n : N_n\}^w_{\rho:P}}\; \eta_{neg}$$

congruence rules

$$\frac{\Gamma \vdash_p w \equiv w' : P \qquad \Gamma, \rho : P \vdash N_i \equiv N'_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \Gamma \vdash P \equiv P' : \mathsf{U}_l}{\Gamma \vdash \{d_1 : N_1 \,\&\, \ldots \,\&\, d_n : N_n\}^w_{\rho:P} \equiv \{d_1 : N'_1 \,\&\, \ldots \,\&\, d_n : N'_n\}^{P'}_{w'::} \mathsf{U}_l}$$

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma \vdash_p w : P \qquad \Gamma, \rho : P \vdash N_i : \mathsf{U}_l \quad (1 \le i \le n) \qquad \Gamma, c : C \vdash t_i \equiv t'_i : N_i[w/\rho] \quad (1 \le i \le n) \qquad \Gamma \vdash v \equiv v' : C}{\Gamma \vdash \{d_1 \mapsto t_1; \ldots; d_n \mapsto t_n\}^v_c \equiv \{d_1 \mapsto t'_1; \ldots; d_n \mapsto t'_n\}^{v'}_c : \{d_1 : N_1 \,\&\, \ldots \,\&\, d_n : N_n\}^w_{\rho:P}}$$

$$\frac{\Gamma \vdash t \equiv t' : \{d_1 : N_1 \,\&\, \ldots \,\&\, d_n : N_n\}^w_{\rho:P}}{\Gamma \vdash d_i\, t \equiv d_i\, t' : N_i[w/\rho]}$$

Figure 9: General typing and computational rules for negative types

extended syntax of expressions

$$\begin{array}{lcl} t, u, v, A, B, p, q & ::= & \ldots \mid \mu X.A \mid \mathsf{enter}\, t \mid \mathsf{fix}\, f\, [\mathsf{enter}\, a \mapsto t]\, \mathsf{in}\, f\, t \\ P & ::= & \ldots \mid (a : X) \otimes P \\ N & ::= & \ldots \mid X \end{array}$$

type former

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A : \mathsf{U}_l \qquad A \text{ is a } (\ldots \oplus \ldots) \text{ or } \{\ldots \,\&\, \ldots\} \text{ type}}{\Gamma \vdash \mu X.A : \mathsf{U}_l}$$

introduction rule

$$\frac{\Gamma \vdash t : A[\mu X.A/X]}{\Gamma \vdash \mathsf{enter}\, t : \mu X.A}$$

elimination rule

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A : \mathsf{U}_l \qquad \Gamma, b : \mu X.A \vdash P : \mathsf{U}_l \qquad \Gamma \vdash v : \mu X.A \qquad \Gamma, f : \Pi b{:}\mu X.A.\, P, a : A[\mu X.A/X] \vdash t : P[\mathsf{enter}\, a/b] \qquad x \text{ guarded from } a \text{ in } t}{\Gamma \vdash \mathsf{fix}\, f\, [\mathsf{enter}\, a \mapsto t]\, \mathsf{in}\, f\, v : P[v/b]}$$

β-conversion rule

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A : \mathsf{U}_l \qquad \Gamma, b : \mu X.A \vdash P : \mathsf{U}_l \qquad \Gamma \vdash v : A[\mu X.A/X] \qquad \Gamma, f : \Pi b{:}\mu X.A.\, P, a : A[\mu X.A/X] \vdash t : P[\mathsf{enter}\, a/b] \qquad x \text{ guarded from } a \text{ in } t}{\Gamma \vdash \mathsf{fix}\, f\, [\mathsf{enter}\, a \mapsto t]\, \mathsf{in}\, f\, (\mathsf{enter}\, v) \equiv t[v/a][\mathsf{fix}\, f\, [\mathsf{enter}\, a \mapsto t]\, \mathsf{in}\, f\, a/f a] : P[\mathsf{enter}\, v/b]}$$

observational rule

$$\frac{\Gamma \vdash t : \mu X.A \qquad \Gamma, a : \mu X.A \vdash E[a] : B}{\Gamma \vdash \mathsf{fix}\, f\, [\mathsf{enter}\, a \mapsto E[\mathsf{enter}\, a]]\, \mathsf{in}\, f\, t \equiv E[t] : B[t/a]}$$

congruence rules

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A \equiv A' : \mathsf{U}_l}{\Gamma \vdash \mu X.A \equiv \mu X.A' : \mathsf{U}_l}$$

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A : \mathsf{U}_l \qquad \Gamma, a : \mu X.A \vdash P : \mathsf{U}_l \qquad \Gamma \vdash v : \mu X.A \qquad \Gamma, f : \Pi a{:}\mu X.A.\, P, a : A[\mu X.A/X] \vdash t : P \qquad x \text{ guarded from } a \text{ in } t}{\Gamma \vdash \mathsf{fix}\, f\, [\mathsf{enter}\, a \mapsto t]\, \mathsf{in}\, f\, v \equiv \mathsf{fix}\, f\, [\mathsf{enter}\, a \mapsto t']\, \mathsf{in}\, f\, v' : P[v/a]}$$

Figure 10: General typing and computational rules for recursive types

extended syntax of expressions

$$t, u, v, A, B, p, q \quad ::= \quad \ldots \mid \nu X.A \mid \mathsf{out}\, t \mid \mathsf{cofix}\, f\, c\, =\, \{\mathsf{out} \mapsto t\}\, \mathsf{in}\, f\, v$$

type formers

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A : \mathsf{U}_l \qquad A \text{ is a } (\ldots \oplus \ldots) \text{ or } \{\ldots \,\&\, \ldots\} \text{ type}}{\Gamma \vdash \nu X.A : \mathsf{U}_l}$$

introduction rule

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A : \mathsf{U}_l \qquad \Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, f : C \to \nu X.A, c : C \vdash t : A[\nu X.A/X] \qquad \Gamma \vdash v : C \qquad x \text{ guarded in } t}{\Gamma \vdash \mathsf{cofix}\, f\, c\, =\, \{\mathsf{out} \mapsto t\}\, \mathsf{in}\, f\, v : \nu X.A}$$

elimination rule

$$\frac{\Gamma \vdash t : \nu X.A}{\Gamma \vdash \mathsf{out}\, t : A[\nu X.A/X]}$$

β-conversion rule

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A : \mathsf{U}_l \qquad \Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, f : C \to \nu X.A, c : C \vdash t : A[\nu X.A/X] \qquad \Gamma \vdash v : C \qquad x \text{ guarded in } t}{\Gamma \vdash \mathsf{out}\, (\mathsf{cofix}\, f\, c\, =\, \{\mathsf{out} \mapsto t\}\, \mathsf{in}\, f\, v) \equiv t[v/c][\mathsf{cofix}\, f\, c\, =\, \{\mathsf{out} \mapsto t\}\, \mathsf{in}\, f\, c/f\, c] : A[\nu X.A/X]}\ \beta_\nu$$

observational rule

$$\frac{\Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, c : C \vdash t : \nu X.A \qquad \Gamma \vdash v : C}{\Gamma \vdash \mathsf{cofix}\, f\, c\, =\, \{\mathsf{out} \mapsto \mathsf{out}\, t\}\, \mathsf{in}\, f\, v \equiv t[v/c] : \nu X.A}\ \eta_{neg}$$

congruence rules

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A \equiv A' : \mathsf{U}_l}{\Gamma \vdash \nu X.A \equiv \nu X.A' : \mathsf{U}_l}$$

$$\frac{\Gamma, X : \mathsf{U}_l \vdash A : \mathsf{U}_l \qquad \Gamma \vdash C : \mathsf{U}_l \qquad \Gamma, f : C \to \nu X.A, c : C \vdash t \equiv t' : A[\nu X.A/X] \qquad \Gamma \vdash v \equiv v' : C \qquad x \text{ guarded in } t}{\Gamma \vdash \mathsf{cofix}\, f\, c\, =\, \{\mathsf{out} \mapsto t\}\, \mathsf{in}\, f\, v \equiv \mathsf{cofix}\, f\, c\, =\, \{\mathsf{out} \mapsto t'\}\, \mathsf{in}\, f\, v' : \nu X.A}$$

Figure 11: General typing and computational rules for co-recursive types

[Bro] Luitzen Egbertus Jan Brouwer. On the significance of the principle of excluded middle in mathematics, especially in function theory. In *From Frege to Gödel: A Source Book in Mathematical Logic*. Harvard University Press. original version in German " Begründung der Funktionenlehre unabhängig vom logischen Satz vom ausgeschlossenen Dritten", 1923.

[CFC58] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume 1. North-Holland, 1958. §9E.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 2:33, 346–366, 1932.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[Coq85] Thierry Coquand. *Une théorie des Constructions*. Dissertation, University Paris 7, January 1985.

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Cur34] Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20:584–590, 1934.

[dB68] Nicolaas de Bruijn. Automath, a language for mathematics. Technical Report 66-WSK-05, Technological University Eindhoven, November 1968.

[Dyb95] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210,405–431, 1935. English Translation in [Sza69], "Investigations into logical deduction", pages 68-131.

[Geu93] Herman Geuvers. *Logics and Type Systems*. Ph.D. thesis, Katholieke Universiteit Nijmegen, September 1993.

[Gir71] Jean-Yves Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Second Scandinavian Logic Symposium*, number 63 in Studies in Logic and the Foundations of Mathematics, pages 63–92. North Holland, 1971.

[Göd58] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12(3):280–287, December 1958.

[Hey28] Arend Heyting. Zur intuitionistischen axiomatik der projektive geometrie. *Mathematische Annalen*, 98:491–538, 1928.

[Hof95] Martin Hofmann. *Extensional concepts in intensional type theory*. Phd, University of Edinburgh, 1995.

[Kle45] Stephen C. Kleene. On the interpretation of intuitionistic number theory. *The Journal of Symbolic Logic*, 10(4):109–124, 1945.

[Kol67] Andrej N. Kolmogorov. On the principle of the excluded middle. In *From Frege to Gödel: A Source Book in Mathematical Logic*, pages 414–447. Harvard University Press, 1967. original version in Russian "O principe tertium non datur" in Mat. Sbornik 32:646–667, 1925.

[Kre59] Georg Kreisel. Interpretation of analysis by means of functionals of finite type. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, 1959.

[Pal98] Erik Palmgren. On universes in type theory. In *Twenty Five Years of Constructive Type Theory*, pages 191–204. Oxford University Press, 1998.

[Pra65] Dag Prawitz. *Natural Deduction, a Proof-Theoretical Study*. Almqvist and Wiksell, Stockholm, 1965.

[Sco70] Dana Scott. Constructive validity. In Michel Laudet, Daniel Lacombe, Louis Nolin, and Marcel-Paul Schützenberger, editors, *Symposium on Automatic Demonstration*, pages 237–275, Berlin, Heidelberg, 1970. Springer Berlin Heidelberg.

[Sza69] Manfred E. Szabo, editor. *The Collected Works of Gerhard Gentzen*. North Holland, Amsterdam, 1969.

[Ter89] J. Terlouw. Een nadere bewijstheoretische analyse van GSTTs. in Dutch, April 1989.