Preliminary program for a seminar on

Types, Categories & Logic

Hanno Becker, habecker@math.uni-bonn.de

Overview. *Type theories* are formal calculi that allow for the study of aspects of functional programming, set theory and logic through the following principles:

(1) Data types, sets and propositions are subsumed under the notion of a *type*.
(2) Programs, elements and proofs are subsumed under the notion of a *term*.
(3) The judgements
   - "m is an algorithm producing a value of the data type T",
   - "x is an element of the set A", and
   - "p is a proof of P"

   are subsumed under the *typing judgement* "t is a term of type T".

Here are two informal reasons why one might expect such a wondrous thing:

- Consider assigning to a proposition φ the set of all its proofs $\mathscr{P}(\varphi)$. Then, at least in intuitionistic mathematics, this assignment intertwines the logical connectives $\wedge, \vee, \Rightarrow$ with the set theoretic operations $\times, \sqcup, \mathrm{Maps}(-,-)$: Providing a proof of $\varphi \wedge \psi$ means providing *both* a proof of φ *and* a proof of ψ, i.e. $\mathscr{P}(\varphi \wedge \psi) = \mathscr{P}(\varphi) \times \mathscr{P}(\psi)$. Similarly, providing a proof of $\varphi \vee \psi$ means providing *either* a proof of φ *or* a proof of ψ, i.e. $\mathscr{P}(\varphi \vee \psi) = \mathscr{P}(\varphi) \sqcup \mathscr{P}(\psi)$. Finally, providing a proof of $\varphi \Rightarrow \psi$ means providing a *method* for turning a proof of φ into a proof of ψ, so $\mathscr{P}(\varphi \Rightarrow \psi) = \mathrm{Maps}(\mathscr{P}(\varphi), \mathscr{P}(\psi))$. This suggests that it should be possible to build logic on set theory by identifying a proposition with the set of its proofs (as is indeed done in type theory).
- Algorithms are implicit in our daily mathematical work: When constructing elements of sets we usually don't specify them explicitly, but instead describe a *procedure* for how to compute them. For example, consider the set of natural numbers $\mathbb{N}$ as freely generated by $0 \in \mathbb{N}$ and the successor function $(-)' : \mathbb{N} \to \mathbb{N}$. This means that the canonical elements of $\mathbb{N}$ are precisely $0, 1 := 0', 2 := 0''$ and so on. Further, suppose we define addition $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ inductively by $0 + n := n$ and $m' + n := (m + n)'$, and similarly the multiplication $\cdot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Then, an expression like $5 + 4 \cdot 7$ is rather a *program* for the computation of the canonical element $33 := 0''^{\cdots\prime}$ in $\mathbb{N}$ than this element itself: it is only by repeated application of the definitions of $+$ and $\cdot$ that the expression gets *reduced* to its canonical form. Since the understanding of computation as the successive evaluation of syntactic objects according to fixed rules is characteristic for functional programming, this gives an idea why one should expect a relation between (constructive) mathematics and functional programming.

Moreover, the last point already hints at another similarity between programming, set theory and logic, namely the concept of *computation*:

(1) Algorithms may be *executed* to produce their *return values*.
(2) Element descriptions (like $5+4\cdot7$ above) may be *evaluated* to produce *canonical elements* (like 33 above).
(3) Proofs may be *normalized* (by cut elimination, for example) to produce *normal proofs* (cut-free proofs, for example).

These computational aspects are also treated on equal footing in type theory, under the notion of *term reduction* and *normal terms*.

Finally, many type systems have the property that typing judgements are *decidable* and that terms are *normalizing* in the sense that they reach normal form after sufficiently many reductions. The resulting algorithms for type checking and computation of normal forms have been implemented for many type systems, allowing for *automated proof checking* and *execution of constructive proofs* within these type systems. For example, proving a statement like the existence of square roots of non-negative real numbers in constructive type theory already includes (in fact, it is the same as) an algorithm for computing these square roots, and the type system implementation can then both automatically check the correctness of the algorithm and execute it. Hence "Proof = Construction = Implementation".

Principal goals of the seminar.

– We want to understand the basic theory of type systems, their relation to functional programming, set theory, category theory and logic.
– We want to look at particular type systems that may serve as foundational systems for mathematics alternative to common foundations like ZF set theory.

Structure of the seminar. During the seminar we will study different type systems of varying complexity. For each of them, we will look at the following aspects:

- *Syntax*: Which syntactic layers are present? Are types and terms on the same layer, and are there any further layers such as kinds? Which type forming operations are available?
- *Structure*: What are the typing and reduction rules for the calculus? Is typing decidable? Is the calculus normalizing, i.e. does any well-typed term have a normal form? Is it even strongly normalizing, i.e. does any sequence of reductions terminate?
- *Logic*: What logic can modelled by the calculus through the propositions-as-types principle?
- *Semantics*: Which categorical structure is modelled by the calculus?

Target Audience. Anyone interested in functional programming, logics, set theory, category theory, foundations, automated proof checking and theorem proving.

Prerequisites. Understanding the ideas, as well as the syntactic study of type systems, does not require any special knowledge – in particular, Bachelor students are most welcome to join. For the relations to logical calculi, some accquaintance with propositional and first order predicate logic is helpful, but depending on the audience we might as well recall these. Similarly for the relation to category theory.

Disclaimer. The following is surely only one way to learn the basics of type theory, and most likely not the best – only the best I could come up with based on my current understanding. If anyone has suggestions on how to structure the enormous amount of aspects of type theory differently, I'd be happy to discuss and to modify the program accordingly.

Contact. If you are interested in attending the seminar, just write me an e-mail to `habecker@math.uni-bonn.de`.

---

1. Topic – The Untyped λ-Calculus
Syntax & Structure

Terms as programs

Overview: The *untyped λ-calculus* is a formal system for the study of function definition and application. It consists of an underlying language $\mathcal{L}$ and a relation $\sim_{\alpha\beta}$ called *$\alpha\beta$-equivalence* on $\mathcal{L}$, defined by means of relations $\sim_\alpha$ and $\to_\beta$ of *α-convertibility* (renaming of "bound" function argument variables) and *β-reduction* (evaluation of function applications), respectively. Despite its seeming simplicity, it is difficult to say anything about $\sim_{\alpha\beta}$ in the beginning (e.g., it is not even clear that $\sim_{\alpha\beta}$ does not identify all λ-terms), and the first tools for the study of $\mathcal{L}/_{\sim_{\alpha\beta}}$ are the notion of *β-normal forms* (λ-terms that contain no reducible function application) and the *Church-Rosser Theorem* ($\alpha\beta$-equivalent λ-terms admit, up to α-conversion, a common child with respect to β-reduction). For example, this implies that $M \not\sim_{\alpha\beta} N$ if $M, N \in \mathcal{L}$, $M \not\sim_\alpha N$ are in β-normal form. What then remains is the problem to decide whether some $\alpha\beta$-equivalence class $\Phi \in \mathcal{L}/_{\sim_{\alpha\beta}}$ contains a λ-term in β-normal form, and, if yes, if there are *normalization strategies* for finding such β-normal forms from an arbitrary representative $M \in \Phi$ by successive β-reductions. We introduce the *applicative order* and *normal order* evaluation/normalization strategies, the former evaluating arguments first before handing them to functions, the latter evaluating them only if needed in the course of the function evaluation. With respect to a fixed evaluation strategy, a λ-term can now be viewed as a *program* inducing a *computational process*, where (as is characteristical for functional programming) computation is understood as the successive evaluation of terms according to fixed rules. While applicative order evaluation does in general not terminate, even for λ-terms that do have a β-normal form, normal order evaluation always finds the β-normal form provided there is one. With the notion of computation by means of successive normal order evaluation fixed, we can introduce the *Church encoding* of natural numbers in the untyped λ-calculus and define what is means for a partial function $f : \mathbb{N}^k \to \mathbb{N}$ to be *λ-computable* / realizable as a λ-term. Examples (e.g. successor, addition, multiplication, exponentiation, and maybe the more difficult example of the predecessor function)

already hint at the expressive power of the λ-calculus, and indeed λ-computability turns out to be a model of computation equivalent to Turing computability.

References: [HS08, Chapters 1 and 3], [SU98, Chapter I], [Bar84, Chapters 1,2,3], [Pie02, Chapter 5] for the untyped λ-calculus; [AS96] for computation as successive evaluation of terms.

Keywords: Definition of untyped λ-calculus, $\alpha/\beta/\eta$-conversion, normal forms, Church-Rosser theorem, normalization strategies, Church encoding (of booleans, naturals, lists), λ-computability.

---

2. Topic – The Simply Typed λ-Calculus Syntax & Structure

Typing as a way to ensure termination

Overview: In the first topic we discussed that successive normal order evaluation gives both a semi-algorithm for deciding the existence of a β-normal form of a given λ-term as well as means to interpreting a λ-term as a program describing a computational process. *Typing* a λ-term is a way to ensure, from the latter viewpoint, that the process it describes will terminate, or equivalently, from the former viewpoint, that its $\alpha\beta$-equivalence class contains a λ-term in β-normal form; it is achieved by fixing the kind ("type") of input and output for functions instead of leaving it unspecified and opening the possibility for functions that can be applied to themselves as in $\omega := \lambda x.xx \in \mathcal{L}$ [1].

As in the first topic, we introduce the *simply typed λ-calculus* $\lambda_{\to}$ by starting with its underlying formal languages $\mathcal{G}$, $\mathcal{T}$ and $\mathcal{L}$ of *contexts*, *terms* and *types*, respectively, and the $\alpha\beta$-equivalence relation on $\mathcal{L}$. Now, however, there is a third constituent, namely an inductively defined *typing relation* $- \vdash - : - \subset \mathcal{G} \times \mathcal{L} \times \mathcal{T}$; the validity of $\Gamma \vdash t : T$ for $\Gamma \in \mathcal{G}$, $t \in \mathcal{L}$ and $T \in \mathcal{T}$ intuitively reads as "in the context Γ, the term t has type T" and is called a *typing judgement*. A typed λ-term $t \in \mathcal{L}$ is called *well-typed* if there exists a context Γ and a type $T \in \mathcal{T}$ such that $\Gamma \vdash t : T$. The main properties of this calculus are the following: (1) *Type safety*: the typing relation descends to $\mathcal{G} \times (\mathcal{L}/_{\sim_{\alpha\beta}}) \times \mathcal{T}$ (2) *Strong normalization*: Any sequence of successive β-reductions of a well-typed term terminates. In particular, the process w.r.t normal order evaluation that a well-typed λ-term describes, terminates. (3) *Decidability*: The typing relation is decidable.

Finally, strong normalization also comes with an unavoidable restriction of expressiveness: the fact that all computational processes induced by typed λ-terms

[1] To quote the introduction from [Pie02]: "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." Here, the program behavior we exclude by typing λ-terms is the non-termination of the computational processes they describe.

are finite implies that there must be Turing computable functions on the natural numbers that cannot be realized by a typed λ-term, see e.g. [SU98, Section 3.5].

References: [Pie02, HS08, SU98]

Keywords: Definition of typed λ-calculus, type-safety, strong normalization, decidability, realizability of numerical functions.

---

3. Topic – Untyped and typed combinatory Logic Syntax & Structure

A variant of λ-calculus without bound variables

Overview: *Combinatory Logic* arises from the following observation: Considering the structure $(\Lambda, *)$ consisting of the set Λ of λ-terms up to equivalence and its binary operation $* : \Lambda \times \Lambda \to \Lambda$ of concatenation of (classes of) λ-terms, it has the following property of *functional completeness*: For any polynomial expression $f(X_1, ..., X_n) \in \Lambda\langle X_1, ..., X_n\rangle$ in Λ with free variables x_i there is some $\lfloor f \rfloor \in \Lambda$ (the class of $\lambda x_n.\lambda x_{n-1}....\lambda x_1.fx_1...x_n$) such that $f(x_1, ..., x_n) = \lfloor f \rfloor x_1...x_n$ for any $x_1, ..., x_n \in \Lambda$. In other words: any polynomial operation $\Lambda^n \to \Lambda$ on Λ is already given by left multiplication map with some suitable element of Λ (explaining the term 'completeness'). It turns out that, even for an arbitrary structure $(A, \cdot)$, it suffices to require only that the specific polynomials $I(X) := X \in A\langle X\rangle$, $K(X, Y) := X \in A\langle X, Y\rangle$ and $S(X, Y, Z) := (XZ)(YZ) \in A\langle X, Y, Z\rangle$ are realizable as left multiplications, say by $\mathbf{I}$, $\mathbf{K}$ and $\mathbf{S}$ respectively, and in the case of $(\Lambda, *)$, these λ-terms $\mathbf{I}$, $\mathbf{K}$ and $\mathbf{S}$ even generate Λ under $*$ (even $\mathbf{I}$ is unnecessary, but somehow it seems standard to keep it). The *combinator calculus* is now the formal calculus whose set of terms is the free non-associative algebra on the letters $\mathbf{I}, \mathbf{K}, \mathbf{S}$, and whose reduction relation is generated by the local reduction rules $\mathbf{I}x \to x$, $\mathbf{K}xy \to x$ and $\mathbf{S}xyz \to (xz)(yz)$. By the above, it admits a sound map to λ-calculus which is surjective up to term equivalence, so it can be viewed as some weaker, variable-free version of the λ-calculus. Analogous to the latter, it admits a Church-Rosser theorem and a typed variant.

We will further consider combinatory logic in the next topic only, so we might be rather quick here and add the above to either the previous or the next topic.

References: [HS08],[SU98, Chapter 5], *To Mock a Mockingbird* [Smu84] is a humorous introduction to Combinatory Logic.

Keywords: Functional completeness, untyped and typed combinatory logic, comparison with untyped and typed λ-calculus

---

4. Topic – The Simply Typed λ-Calculus Logic

Propositions as Types, Proofs as Programs

Overview: In this topic we begin discussing the *Curry-Howard correspondence*, the principle that typed calculi can be understood as enhanced logical calculi, with *types corresponding to propositions* and *terms corresponding to proofs*[2]. It is of fundamental importance and will guide us through the rest of the seminar.

Curry-Howard correspondences exist between many kinds of logical and typed calculi. We start by studying its simplest case, namely the relation between logical calculi for *(implicational) intuitionistic propositional logic* and typed combinatory logic and the simply typed λ-calculus – here, *propositional* means that our formulas do not contain quantifiers, *intuitionistic* means that we do not impose the *law of excluded middle* $A \vee \neg A$ resp. its implicational counterpart $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ called *Peirce's law*, and *implicational* means that – instead of $\top, \bot, \wedge, \vee, \Rightarrow$ for full propositional logic – the only logical connective used for the formation of new propositions is the implication $\Rightarrow$. Two prominent calculi for intuitionistic propositional logic are the *natural deduction* and *Hilbert style* proof calculi, which we want to recall, with emphasis on their implicational fragment.

We will see that proof trees in the Hilbert style calculus for implicational intuitionistic propositional logic are essentially the same as terms in *typed combinatory logic*, and that under this correspondence *proof checking is type checking* – in particular, the implicational intuitionistic propositional tautologies (w.r.t. the Hilbert style calculus) are precisely the inhabited types of typed combinatory logic. Similarly (here, however, there doesn't seem to be a canonical bijection between terms and proofs) one can relate proofs in natural deduction style to typing judgements in the simply typed λ-calculus, so that again implicational intuitionistic propositional tautologies (w.r.t. natural deduction) are precisely the inhabited types of the simply typed λ-calculus.

Hence, to summarize, typed combinatory logic and the simply typed λ-calculus may be viewed as alternative calculi for implicational intuitionistic propositional logic, but with an *additional layer of structure*: proofs are now being treated as syntactic objects in their own right, belonging to the underlying language of the calculus. This is the Curry-Howard correspondence. It generalizes to the simply typed λ-calculus with product types and a top type $\top$ (corresponding to the true proposition), which is in a Curry-Howard correspondence to propositional logic over the connectives $\{\top, \wedge, \Rightarrow\}$; considering also sum types and the bottom type $\bot$ corresponding to the false proposition, we obtain a typed λ-calculus in Curry-Howard correspondence to full propositional logic over the connectives $\{\top, \bot, \wedge, \vee, \Rightarrow\}$.

The Curry-Howard correspondence can be seen as a process of *categorification*, as we shall discuss in the next topic. For the moment, we note that it also allows for a syntactic analysis of proofs: for example, the strong normalization of the typed

[2]Is is is also called *Propositions-as-Types* or *Proofs-as-Programs* correspondence

λ-calculus implies that any intuitionistic propositional tautology is witnessed by a λ-term in β-normal form, leading to a *decision algorithm for provability*. Conversely, model theoretic ideas from logic apply to λ-calculus, as we will see in the next topic.

References: [SU98]

Keywords: Hilbert and natural deduction style calculus for intuitionistic propositional logic, Curry-Howard correspondence, decidability of intuitionistic propositional logic.

---

5. Topic – The Simply Typed λ-Calculus 0-categorical and 1-categorical semantics

Types as Objects, Terms as Morphisms

Overview: *Model theory* is the way of studying logical calculi through *denotational semantics*, i.e. by means of interpreting their terms as mathematical objects. For example, we don't prove statements about groups syntactically by providing proof trees for them as first order formulas in the theory of groups, but instead by checking them for all "concrete" groups within an ambient set theory. Similarly, we may check if a propositional formula $\varphi(P_1, ..., P_n)$ is a classical tautology by looking at its truth table instead of trying to derive it using Hilbert or natural deduction style calculi for propositional logic. Each time we do this, the justification for the use of these methods is – apart from the assumed consistency of the ambient set theory – given by a soundness and completeness theorem, such as Goedel's Completeness Theorem for set theory semantics of first order logic or the completeness theorem for the Boolean algebra semantics of classical propositional formulas.

Depending on interest and knowledge of the audience, we might recall these general principles in some detail, and discuss afterwards the case of the *Heyting algebra semantics* for propositional intuitionistic logic, analogous to the truth table semantics for classical propositional logic: For any propositional formula $\varphi(P_1, ..., P_n)$, any Heyting algebra H (a generalized algebra of "truth values", but with $a \vee \neg a \neq 1$ in general) and any tuple $v = (v_1, ..., v_n) \in H^n$ of generalized truth values, it interprets φ as an element $[\![\varphi]\!]_{H,v} \in H$, and φ is an intuitionistic tautology if (completeness) and only if (soundness) $[\![\varphi]\!]_{H,v} = 1$ for all choices of H and v. Here, the crucial point and the reason why Heyting algebras occur is that the poset obtained by strictifying the "syntactic" preorder given by propositional formulas as elements and $\varphi \leq \psi :\Leftrightarrow (\varphi \Rightarrow \psi$ is a tautology) as the comparison relation, is a Heyting algebra, called the *Lindenbaum algebra*.

Now, through the Curry-Howard correspondence, the soundness of Heyting algebra semantics means that *each* term t of type $\varphi(P_1, ..., P_n)$ in simply typed λ-calculus with product and sum types witnessing the provability of φ implies $[\![\varphi]\!]_{H,v} = 1$ for all H and v; however, all of them do so "in the same way", and the

additional structure gained by having proof terms as parts of the calculus is lost under the Heyting algebra semantics. For example, remember the proofs of the tautology $(X \Rightarrow X) \Rightarrow (X \Rightarrow X)$ are, up to equivalence, parametrized by $\mathbb{N}$ through the Church encoding $n \mapsto \lambda_{f:X\to X}\lambda_{x:X} f^n x$. Naturally, the question arises to find a semantics for the simply typed λ-calculus which assigns meaning both to types and terms, in a "finer" way that does not identify all terms of the same type. The natural solution is to define an algebraic object from the simply typed λ-calculus analogous to the construction of the Lindenbaum algebra, but with each λ-term t satisfying $\vdash t : \varphi \to \psi$ witnessing the implication $\varphi \Rightarrow \psi$ in a different way. In other words, for any two formulas φ and ψ, we don't look at whether or not there is *some* proof of $\varphi \to \psi$, but rather at the *set of proofs*, leading to the notion of a *deductive system* instead of a preorder. Amazingly, "strictifying" this deductive system by identifying λ-terms with respect to a suitable notion of equivalence turns out to give a *Bicartesian Closed Category* (bccc.), culminating in a semantics of simply typed λ-calculus in ccc., as well as to the surprising observation that ccc. naturally arise in logic, whereas non-logic students typically learn about them when studying category theory as means to organize mathematical objects and their morphisms.

To summarize, we noticed by example that "strictifying" proof calculi can produce categorical objects of varying complexity, depending on whether or not proofs where part of the calculus: The classical calculi ignoring proof terms describe Heyting algebras, which can be viewed as Cartesian Closed posets/0-categories, while the typed λ-calculus produces ordinary Cartesian closed $(1-)$categories. In this sense, the λ-calculus is a "categorification" of ordinary proof calculi for intuitionistic propositional logic. It is natural to ask whether this can be pursued further, i.e. if there is an even richer proof calculus in which $\alpha\beta$-reductions between proof terms are part of the calculus, allowing for a 2-categorical strictification, and so on ... depending on time and interest, we might study these extensions as well.

References: [SU98, Chapter 2] for Heyting algebra semantics of intuitionistic propositional logic, [LS86] for relation with cartesian closed categories. [Bae06] might also be a helpful read.

Keywords: Hilbert and natural deduction style calculus for intuitionistic propositional logic, Curry-Howard correspondence, decidability of intuitionistic propositional logic.

---

6. Topic – Dependent Type Systems
Syntax, Structure & Logic

Predicate Logic via Dependent Types

Overview: In this topic, we begin our study of *Dependent Type Theories*, extensions of the simply typed λ-calculus that are in Curry-Howard correspondence

with systems of (higher order) *predicate logic*. The problem in realizing quantified propositions such as $\forall_{x\in\mathbb{R}} x^2 \geq 0$ or $\exists_{x\in\mathbb{R}} x^2 = 7$ as types is that they deal with propositions that are parametrized over sets, like $x^2 \geq 0$ and $x^2 = 7$ depending on the value of $x \in \mathbb{R}$. Abstracting from propositions and sets as types, we are lead to study type systems allowing for *dependent types*, i.e. type expressions that may contain free variables for terms of other types. If we want to emphasize the name of a free term variable x of type A a type expression B may contain, we might also write $B(x)$, but the added "(x)" is *not* part of the syntax. Also, note that while the intuition blurrs that distinction, *formally* B is *not* to be considered as a type valued function on A – it's like the difference between a λ-term $\lambda x.f\ (f\ x)$ with free variable f and its associated λ-closure $\lambda f.\lambda x.f\ (f\ x)$. *Informally*, a dependent type $B(x)$ over A may set-theoretically be thought of as a map $p : B \to A$, with the type $B(x)$ being given by the fiber $p^{-1}(x)$ once x is fixed (indeed this intuition will later become the formal semantics of dependent types). Now, given a dependent type $B(x)$ over A we are interested in forming its Π-type (or *dependent product type*) $\prod_{x:A} B(x)$ and Σ-type (or *dependent sum type*) $\sum_{x:A} B(x)$: the terms of $\prod_{x:A} B(x)$ are functions assigning to any $x : A$ a term of $B(x)$, and the terms of $\sum_{x:A} B(x)$ are pairs (x, y) consisting of some $x : A$ and $y : B(x)$. Hence, if $B(x)$ is a proposition like $x^2 \geq 0$ above, producing an element of $\prod_{x:A} B(x)$ means to give a proof of $B(x)$ for all $x : A$ – hence, $\prod_{x:A} B(x)$ can be viewed (as the set of proofs of) as the universally quantified proposition $\forall_{x\in A} B(x)$. Similarly, for $B(x)$ a proposition, $\sum_{x:A} B(x)$ is (the set of proofs of) the existentially quantified proposition $\exists_{x\in A} B(x)$. Intuitively, if $B(x)$ comes from a map $p : B \to A$, then $\prod_{x:A} B(x)$ is the set of sections of p, while $\sum_{x:A} B(x)$ is the total space B itself. Finally, if B does not contain x, then $\prod_{x:A} B(x) \equiv A \to B$ and $\sum_{x:A} B(x) \equiv A \times B$.

There exist many ways to define extensions of the simply typed λ-calculus by dependent types, and it seems that in order not to get lost in comparing minor differences between them, we should agree on some of them and then discuss these in detail – which ones is mainly up to the taste of the speaker. To get an impression of the larger structural differences, there are at least the following three different versions of dependent type systems: Firstly, there is *Martin-Löf Type Theory* [ML80] which, among others, adds a new kind of judgement $\Gamma \vdash A$ `Type` to the calculus, signifying that in a certain context Γ, A is a type. Then the dependent sum and product types can be introduced through the inference rules that $\Gamma \vdash \prod_{x:A} B(x)$ `Type` and $\Gamma \vdash \sum_{x:A} B(x)$ `Type` provided $\Gamma, x : A \vdash B(x)$ `Type`. With this approach, dependent types have to be built in through adding axiomatic judgements like $x : A, y : A \vdash \mathrm{Eq}_{A,x,y}$ `Type` for the dependent equality type on a type A, but they do not exist as type-valued functions within the system. Another possibility which treats dependent types in this way is the system λLF: here, one adds another layer of *kinds* on top of terms and types, together with a *kinding relation* between types and kinds analogous to the typing relation between terms and types. The kind of ordinary "independent" types is $*$, while the kind of a type depending on the values of the types $A_1, ..., A_n$ is $A_1 \to ... \to A_n \to *$. With this approach, a dependent type like Eq_A is indeed a type, and its kind $A \to A \to *$

signifies its dependence on two values of A. However, in λLF it is not possible to define new dependent types through λ-abstractions on the type level as is done with ordinary λ-abstractions on the term level. This, finally, is possible in the pure type system λP which exhibits the structural novelty that there is no syntactic distinction between terms and types anymore. Instead, there is only one syntactic category of types, the typing relation holds between two types, and there is only one syntactic construct for λ-abstraction and formation of dependent product types. The hierarchy of terms, types and kinds present in λLF is restored in λP through the introduction of two special types $*$ and $\Box$ called *sorts* measuring the "size" of a type: inhabitants of $*$ are the types from λLF, while those of $\Box$ are the kinds in λLF. Sorts also allow for the distinction between ordinary "dependent terms" (the elements of function types $A \to B$ for types A, B in λLF) and "dependent types" (the types of function kinds like $A \to *$ in λLF): given B, the type $\prod_{x:A} B$ is well-typed only if firstly $A : *$ and secondly either $x : A \vdash B : *$ or $x : A \vdash B : \Box$; in the former case, B is a type depending on A, and we put $\prod_{x:A} B : *$; in the latter case, B is a kind, and we put $\prod_{x:A} B : \Box$. Considering other possible combinations leads to the pure type systems we study in the next topic. In the setup of λLF, for example, the type of type valued functions on a type A is $A \to * :\equiv \prod_{x:A} *$, which is well-typed of type $\Box$ by the second of the above typing rules; similarly, type valued functions with several arguments $A_1, ..., A_n$ have type $A_1 \to ... \to A_n \to *$, which again is of type $\Box$. One could then introduce the greater-or-equal type $\geq_{\mathbb{R}}$ on a type $\mathbb{R}$ of reals by the typing axiom $\geq_{\mathbb{R}} : \mathbb{R} \to \mathbb{R} \to *$, and realize our example proposition from above as the type $\forall_{x:\mathbb{R}} \geq_{\mathbb{R}} x^2\ 0$, provided x^2 is defined (note that the concatentation of x^2 and 0 are mandatory to make $\geq_{\mathbb{R}} x^2\ 0$ a type expression depending on x, and are not to be confused with the notation $B(x)$)

The above dependent type systems allow for sound realizations of (fragments of) first-order predicate logic as types, sound in the sense that provable propositions become inhabited types, but concerning the faithfulness and the precise relations between the systems, I haven't found much material so far. Apart from the relation to predicate logic, the structural properties of the systems should be discussed.

Keywords: Variants of dependent type sytems and their structural properties. Curry-Howard correspondence with predicate logic.

References: [ML80] for Martin-Löf Type theory, [Pie04, Section 2.1], [Pie02, Section 30.5] for the idea of dependent types in general, [Pie04, Section 2.2-2.5] for precise definitions and structural properties of λLF, [Pie04, Section 2.7] for pure type systems, in particular λP, and relation between λLF and λP. See also [SU98, Chapter 10], but note that what is called λP there is a mixture of λLF and λP in the sense of [Pie04]. [nlae, nlaa], [HS08, Chapter 13] might be helpful as well.

0-categorical Semantics

Interpreting intuitionistic predicate logic in fibered Heyting algebras

Overview: We begin our study of the semantics of dependent type systems, i.e. we investigate which categorical structures they model, just as Heyting algebras were modelled by intuitionistic propositional logic and cartesian closed categories were modelled by the simply typed λ-calculus with products. The semantics of dependent type systems is a large topic, and we shall start with the classical *semantics of first order predicate logic*, and proceed to the semantics of their type theoretic "categorification" afterwards.

A signature of a first-order theory is a triple $(\mathcal{S}, \mathcal{R}, \mathcal{F})$ of sets, where $\mathcal{S}$ is arbitrary and called the set of *sorts*, and where $\mathcal{R}$ and $\mathcal{F}$ are sets of finite sequences in $\mathcal{S}$ called *relations* and *functions*, respectively. Intuitively, the elements of $\mathcal{S}$ are the domains of discourse of the theory, while if $r = (s_1, ..., s_n) \in \mathcal{R}$ or $f = (s_1, ..., s_n, t) \in \mathcal{F}$ for $s_i, t \in \mathcal{S}$, then we think of r as an n-ary relation between the sorts $s_1, ..., s_n$ and of f as a function $s_1 \times ... \times s_n \to t$. A signature is interpreted in sets by assigning to each sort a "concrete" set $[\![s_i]\!] \in \mathtt{Set}$, to each relation symbol $r = (s_1, ..., s_n) \in \mathcal{R}$ a relation $[\![r]\!] \subset [\![s_1]\!] \times ... \times [\![s_n]\!]$ on the $[\![s_i]\!]$, and to each function symbol $f = (s_1, ..., t_n) \in \mathcal{F}$ a map $[\![s_1]\!] \times ... \times [\![s_n]\!] \to [\![t]\!]$. Given such an interpretation, it can then be extended to any propositional first-order formula: Suppose $\Gamma := \{(x_i, s_i)\}$ is a context of free variables x_i over sorts s_i and $\varphi(x_1, ..., x_n)$ is a well-formed first-order formula in the context Γ, then its interpretation $[\![\varphi(x_1, ..., x_n)]\!]_\Gamma$ can inductively defined as a subset of $[\![\Gamma]\!] := [\![S_1]\!] \times ... \times [\![S_n]\!]$. In particular, a closed formula is interpreted as a subset of the singleton, the empty set corresponding to the truth value "false" and the singleton itself to the truth value "true".

We want to work out the essence of this interpretation, both to understand the relation with the semantics of propositional logic and to be able to extend it to interpretations that are not necessarily set-valued. To begin, for a fixed context Γ the interpretation function $[\![-]\!]_\Gamma$ from above assigns truth values in the Boolean algebra $\mathcal{P}([\![\Gamma]\!])$ to each propositional first-order formula over Γ. This is already reminiscent of the truth-value interpretation of propositional logic; the difference, however, is that here we are interpreting formulas in Boolean algebras that vary with the formulas themselves, and that moreover the definition of $[\![-]\!]_\Gamma$ involves switching between these Boolean algebras whenever a quantifier is to be interpreted: Namely, suppose again that $\varphi(x_1, ..., x_n)$ is a well-formed propositional first-order formula in the context Γ, and put $\Gamma' := \Gamma \setminus \{(x_1, s_1)\}$. Then $[\![\exists_{x_1:s_1}\varphi(x_1, ..., x_n)]\!]_{\Gamma'}$ is the image of $[\![\varphi(x_1, ..., x_n)]\!]_\Gamma \subset [\![\Gamma]\!]$ under the projection $[\![\pi_{\Gamma,\Gamma'}]\!] : [\![\Gamma]\!] \cong [\![s_1]\!] \times [\![\Gamma']\!] \to [\![\Gamma']\!]$, while $[\![\forall_{x_1:s_1}\varphi(x_1, ..., x_n)]\!]_{\Gamma'}$ consists of those $(v_2, ..., v_n) \in [\![\Gamma']\!]$ for which $[\![\pi_{\Gamma,\Gamma'}]\!]^{-1}(v_2, ..., v_n) \subseteq [\![\varphi(x_1, ..., x_n)]\!]_\Gamma$. Despite of our ad-hoc set-theoretic descrition, the two maps $\mathcal{P}([\![\Gamma]\!]) \to \mathcal{P}([\![\Gamma']\!])$ used here are actually completely determined by the functor $[\![\pi_{\Gamma,\Gamma'}]\!]^* : \mathcal{P}([\![\Gamma']\!]) \to \mathcal{P}([\![\Gamma]\!])$ (where again we view posets as 0-categories): namely, the image functor $\mathcal{P}([\![\Gamma]\!]) \to \mathcal{P}([\![\Gamma']\!])$ used in interpreting $\exists$-quantification is the left-adjoint to $[\![\pi_{\Gamma,\Gamma'}]\!]^*$, while the assignment used in interpreting $\forall$-quantification is right-adjoint to $[\![\pi_{\Gamma,\Gamma'}]\!]^*$. Finally,

in forming the interpretation of an atomic formula like $r(f(x), g(y))$ (in a context like $\Gamma := \{(x, s_1), (y, s_2)\}$, and with function symbols $f = (s_1, s_1')$, $g = (s_2, s_2')$ and a relation symbol $r = (s_1', s_2')$) we need to be able to pull back the relation $[\![r(a, b)]\!]_{\Gamma'} \subseteq [\![\Gamma']\!]$, $\Gamma' := \{(a, s_1'), (b, s_2')\}$, to $[\![\Gamma]\!]$ along $[\![f]\!] \times [\![g]\!] : [\![\Gamma]\!] \to [\![\Gamma']\!]$.

Hence, summing up informally, an interpretation of first-order formulas over a given signature needs the following:

(1) A "category of contexts" $\mathcal{C}$ with finite limits in which contexts and function symbols of the signature can be interpreted as objects and morphisms, respectively. In the set-theoretic interpretation above, we have $\mathcal{C} = \mathtt{Set}$.
(2) For each object X of $\mathcal{C}$ a Boolean algebra $\mathcal{D}_X$ of "predicates/propositions over X", in which propositional formulas can be interpreted. In the set-theoretic interpretation, $\mathcal{D}_X := \mathcal{P}(X)$ is the algebra of subsets of X.
(3) Pullback morphisms $u^* : \mathcal{D}_Y \to \mathcal{D}_X$ for any morphism $u : X \to Y$ in $\mathcal{C}$, such that pullbacks along projection morphisms admit left and right adjoints. In set-theoretic semantics, $u^* : \mathcal{P}(Y) \to \mathcal{P}(X)$ is given by $u^*(A) := u^{-1}(A)$.

Formally, this goes under the fancy name of a *first-order hyperdoctrine*. Part (2) und (3) can be formalized by either saying that one has a functor $\mathcal{C}^{\mathrm{op}} \to \mathtt{Cat}$ sending X to $\mathcal{D}_X$, where $\mathtt{Cat}$ is the category of categories, or by employing the notion of a Grothendieck fibration $\mathcal{D} \to \mathcal{C}$. In our case it doesn't matter which approach to choose since the fibers $\mathcal{D}_X$ are skeletal, but in general the approach using Grothendieck fibrations should be used if the assignment $X \mapsto \mathcal{D}_X$ is only to be functorial up to coherent isomorphism of functors.

Now recall that in the semantics for intuitionistic propositional logic, we noticed more than that interpretation may take place in any Heyting algebra: The intuitionistic propositional calculus itself gave rise to a Heyting algebra, the Lindenbaum algebra, carrying the universal interpretation through which any other interpretation factored by means of a unique morphism of Heyting algebras. Here, the situation is completely analogous: We may form a first-order "Lindenbaum" hyperdoctrine from any first-order signature, and this hyperdoctrine carries the universal interpretation. For the Lindenbaum hyperdoctrine, the category $\mathcal{C}$ is taken to be the *context category* of our signature, i.e. the cartesian category freely generated by the sorts and function symbols. For a context Γ, the fiber $\mathcal{D}_X$ is the Heyting algebra $\mathcal{H}_\Gamma$, defined as the poset of equivalence classes of well-formed propositional first-order formulas in context Γ. The pullback functors are given by substitution, while its left and right adjoints in case of projections are given by existential and universal quantification, respectively; explicitly, the latter boils down to the following well-known and intuitive principles: provability of $(\exists_x \phi(x)) \Rightarrow \psi$ is equivalent to provability $\forall_x(\phi(x) \Rightarrow \psi)$ (or $\phi(x) \Rightarrow \psi$ if universal quantification is implicit), while provability of $\psi \Rightarrow \forall_x \phi(x)$ is equivalent to provability of $\forall_x(\psi \Rightarrow \phi(x))$ (or $\psi \Rightarrow \phi(x)$ if universal quantification is implicit).

In our original example of set-valued semantics again, the hyperdoctrine was given by $\mathcal{C} = \mathtt{Set}$ and $\mathcal{D}_X = \mathcal{P}(X)$, the poset of subobjects (to be thought of as predicates/truth values) on X. The question of which categories $\mathcal{C}$ give rise to a first-order hyperdoctrine by setting $\mathcal{D}_X := \mathrm{Sub}_{\mathcal{C}}(X)$ leads to the notion of a

Heyting category (or *logos*). Examples include *elementary topoi* like categories of G-sets for a group G, or categories of (set-valued) sheaves over an arbitrary site; the corresponding semantics of first-order logic in topoi is called *Kripke-Joyal semantics* and can be used to formalize *Cohen forcing* in the context of topos theory, which we might also touch upon shortly.

How does the above relate to Martin-Löf type theories? The point is that when constructing an ML type theory from a first-order signature as in the previous topic, a judgement $\mathrm{WF}(\Gamma, \varphi)$ of the form "In the variable context Γ, the formula φ is a well-formed proposition" gives rise to a judgement $\Gamma \vdash \varphi$ `Type` in that ML type theory. Now, we assembled all judgements $\mathrm{WF}(\Gamma, \varphi)$ into a hyperdoctrine, in such a way that the predicate associated with $\mathrm{WF}(\Gamma, \varphi)$ is the top one in the corresponding Heyting algebra provided φ is provable in context Γ. Translating to ML type theory, we therefore have constructed a truth-value interpretation for some judgements $\Gamma \vdash R$ `Type` in such a way that a typing judgement $\Gamma \vdash t : R$ would prove that the truth value attached to $\Gamma \vdash R$ `Type` was the top one. However, there are two things to note:

(1) As in the treatment of intuitionistic propositional logic through Heyting algebras, this construction is *proof neglecting* – we can't distinguish different t, t' that both have type R in context Γ.
(2) Not all judgements $\Gamma \vdash R$ `Type` in ML type theory arise from the first-order judgements $\mathrm{WF}(\Gamma, \varphi)$ above. For example, in ML type theory, the context Γ might contain elements of higher order types as well, or even elements of dependent type.

The first point leads to categorifying the fibers of our hyperdoctrine, the second to enriching the base category. Both will be discussed in the next topic.

Summarizing, the point of this topic is that intuitionistic first order logic gives a first example of the general principle that

Logical calculi model algebraic structures that are fibered over categories of contexts.

In fact, we could have observed this already in the example of the natural deduction calculus for intuitionistic propositional logic: there, we defined the partial order on the Lindenbaum algebra by strictifying the preorder $\varphi \leq \psi :\Leftrightarrow (\emptyset \vdash \varphi \Rightarrow \psi)$, in which provability is with respect to the empty context. However, we could also have done this in some arbitrary context Γ: assuming without loss of generality that Γ consists of a single propositional type τ only, this would lead to the preorder $\varphi \leq_\Gamma \psi :\Leftrightarrow (\Gamma \vdash \varphi \Rightarrow \psi)$. However, we have $\varphi \leq_\Gamma \psi \Leftrightarrow (\emptyset \vdash (\tau \wedge \varphi) \Rightarrow \psi) \Leftrightarrow (\tau \wedge \varphi) \leq_\emptyset \psi$, so the strictification of this preorder is just the restriction of the original Lindenbaum algebra to $\{[\psi] \mid \psi \leq \tau\}$. Hence, we *do* have a family of Heyting algebras fibered over contexts already in the case of intuitionistic propositional logic, but it is just the "simple fibration" in the sense of [Jac99, Section 1.3] associated to the Lindenbaum algebra for the empty context.

Keywords: Semantics of first-order logic in sets, Heyting categories and general first-order hyperdoctrines. Kripke-Joyal semantics for topoi.

References: For the general idea see [Jac99, Chapter 0]. (First-order) Hyperdoctrines are discussed in [See83, Law06, nlab, nlac] and [Jac99, Chapter 4]. [Jac99, Chapter 1] and [Vis05] contain introductions to fibered categories. [MLM94, Section VI.6] or [nlad] for the Kripke Joyal semantics.

---

8. Topic – Intuitionistic Type Systems
1-categorical semantics

Locally cartesian closed categories

---

9. Topic – The λ-Cube and Pure Type Systems
Syntax & Structure

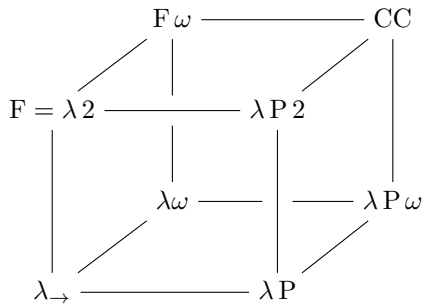
Inductive Types via Polymorphism

Overview: In the last topic we considered dependent types, i.e. *types depending on terms*. Further, right from the beginning of the seminar we are also considering *terms depending on terms*, i.e. functions. It is natural to ask whether one can also build systems which allow for *terms depending on types* and *types depending on types*. In the theory of programming languages, the former is commonly known as *parametric polymorphism*: the possibility of having values taking on many different types depending on one or more type parameters, an example being the identity function id which can be viewed as being of type $A \to A$ for any type A. Types depending on types are in turn known as *type operators* in functional programming, an example being the operator `List` sending a type A to the type `List` A of lists over A.

The minimal extension of the simply typed λ-calculus providing polymorphic types is Girard's *System F / polymorphic λ-calculus* $\lambda 2$; as for dependent types, there are several ways to define it: One can either take the simply typed λ-calculus and add to it λ-abstractions and function types with type parameters, or one can define the syntax of System F to be exactly the same as the one for $\lambda \mathrm{P}$, but now declaring a dependent product $\prod_{x:A} B$ as being well-typed only if $A : *$ and $x : A \vdash B : *$ or $A : \Box$ and $x : A \vdash B : *$. It turns out that System F is still strongly normalizing, with a greatly increased expressive power compared to simply typed λ-calculus in that it is now possible to realize *inductive data types* such as natural numbers, lists or trees inside System F.

The minimal extension λ_ω of the simply typed λ-calculus providing type operators can be constructed in the same way, by declaring $\prod_{x:A} B$ well-typed if either $A : *$ and $x : A \vdash B : *$ or $A : \Box$ and $x : A \vdash B : \Box$.

In general, we may construct a type system from any set $\mathcal{S}$ of sorts, a set $\mathcal{A} \subset \mathcal{S}^2$ of "axioms" and a set $\mathcal{R} \subset \mathcal{S}^3$ of "rules" as follows: Firstly, the syntax is the same as for $\lambda\mathrm{P}$, System F and λ_ω. Secondly, for any $(s, s') \in \mathcal{A}$ one introduces the typing judgement $\vdash s : s'$ as an axiom. Thirdly and finally, we set $\prod_{x:A} B : s''$ if $A : s$ and $x : A \vdash B : s'$ and $(s, s', s'') \in \mathcal{R}$. Type systems constructed in this way are called *pure type systems*, and they provide a common generalization for the systems $\lambda_\to$, λ_ω, System F and $\lambda\mathrm{P}$ we have been studying so far. Moreover, the formalism allows combining these systems to more complex ones: for example, taking $\mathcal{S} := \{*, \Box\}$, $\mathcal{A} := \{(*, \Box)\}$ and $\mathcal{R} := \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$ gives *System* F_ω, a type system allowing for the formation of polymorphic types and type operators. The 8 possible pure type systems based on $\mathcal{S} := \{*, \Box\}$, $\mathcal{A} := \{(*, \Box)\}$ and using only rules (s, s', s'') with $s' = s''$ are the corners of the so-called λ-*cube*, at the outermost corner of which one has the *Calculus of Constructions (CoC)*, in which any combination of types/terms depending on types/terms is realizable:



All the systems of the λ-cube are decidable and strongly normalizing, but, as we shall see, this is not true for arbitrary type systems.

Keywords: Pure type systems, System F, λ-cube

References: [HS08, Chapter 13], [SU98, Chapter 13.5-13.8], [Pie04, Chapter 2.7] for definition and structural properties of pure type systems in general, [Pie04, Chapter 2.6] and the original article [CH88] for the calculus of constructions.

---

10. Topic – Paradoxes in Type Theory

Impredicativity, Axiom of Choice & Classical Logic

Overview: We have seen that type systems can be viewed as logics, with types corresponding to propositions and terms to proofs. Naturally, the question of *(in)consistency* arises: Is any proposition provable? Translated to type systems,

this is the question of whether all types, or at least those we think of as corresponding to propositions, are inhabited. For example, for the simply typed λ-calculus we know that it is consistent as its inhabited types correspond to the tautologies in implicational propositional intuitionistic first order logic, and we know the latter to be consistent (alternatively, we saw that one might employ the strong normalization to check that, e.g., the type $((A \to B) \to A) \to A$ is not inhabited). More generally, all systems in the λ-cube are consistent since their strong normalization implies that the type $\prod_{X:*} X$ is not inhabited (?).

However, despite these positive results, relaxing the typing constraints for λ-abstraction and dependent products too much quickly yields an inconsistent system: For example, the final pure type system λ_{fin} with $\mathcal{S} := \{\texttt{Type}\}, \mathcal{A} := \{\texttt{Type} : \texttt{Type}\}$ and $\mathcal{R} := \{(\texttt{Type}, \texttt{Type}, \texttt{Type})\}$ is inconsistent, as can be seen through *Girard's paradox*. The situation resembles the naive approach to set theory in which the assumption that there is a set `Set` of all sets allows forming Russel's contradictory set $\{x \in \texttt{Set} \mid x \notin x\}$ – namely, in the final PTS λ_{fin} above the type `Type` behaves like a type of all types, in particular satisfying `Type` : `Type`. Now typing judgements are meta-mathematical statements *about* λ_{fin} instead of propositions *inside* λ_{fin}, so Russel's paradox cannot be directly formalized inside λ_{fin} – however, it turns out that a variant of it can, namely the *Burali-Forti Paradox* arising from the assumption that there is a set of all well-ordered sets, see [Hur95]. This is what we want to discuss in the first part of this topic. An implementation of Hurken's variant of the paradox in the Calculus of Constructions (under the assumption that $*$ "retracts" to a proper proposition, corresponding to the axiom `Type` : `Type` in λ_{fin}) can be found in the Coq library `Coq.Logic.Hurkens`.

Finally, we want to discuss variants of *Diaconescu's Theorem* [Dia75]: In set theory, the theorem states that in the *intuitionistic* first order theory of sets, the ZFC axioms already imply the law of excluded middle [GM78], while topos theoretically it states that any topos satisfying the internal axiom of choice is Boolean [MLM94, Exercise VI.16, p. 346]. We want to discuss in particular the type theoretic versions.

Keywords: Girard's paradox, Burali-Forti Paradox, Diaconescu's Theorem

References: [Hur95, BB96, Stu, Coq94, Dia75, GM78, MLM94]

---

11. Topic – Coq
Calculus of Inductive Constructions

Depending on interest, a description will be added later.

---

12. Topic – Homotopy Type Theory
An outlook

Depending on interest, a description will be added later.

References

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[Bae06] John Baez. The n-category caf'e: Categorifying cccs: Computation as a process, 2006. Post at `https://golem.ph.utexas.edu/category/2006/08/categorifying_cccs_seeing_comp.html`.

[Bar84] H. P. Barendregt. *The lambda calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, revised edition, 1984. Its syntax and semantics.

[BB96] F. Barbanera and S. Berardi. Proof-irrelevance out of excluded-middle and choice in the calculus of constructions. *Journal of Functional Programming*, 6(3):519–525, 1996.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Inform. and Comput.*, 76(2-3):95–120, 1988.

[Coq94] Thierry Coquand. A new paradox in type theory. In *Logic, Methodology and Philosophy of Science IX : Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science*, pages 7–14. Elsevier, 1994.

[Dia75] Radu Diaconescu. Axiom of choice and complementation. *Proc. Amer. Math. Soc.*, 51:176–178, 1975.

[GM78] N. Goodman and J. Myhill. Choice implies excluded middle. *Z. Math. Logik Grundlag. Math.*, 24(5):461, 1978.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-calculus and combinators, an introduction*. Cambridge University Press, Cambridge, 2008.

[Hur95] A. J. C. Hurkens. A simplification of girard's paradox. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications: Proc. of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA-95)*, pages 266–278. Springer, Berlin, Heidelberg, 1995.

[Jac99] Bart Jacobs. *Categorical logic and type theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1999.

[Law06] F. William Lawvere. Adjointness in foundations. *Repr. Theory Appl. Categ.*, (16):1–16, 2006. Reprinted from Dialectica **23** (1969).

[LS86] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1986.

[ML80] Per Martin-Löf. An intuitionistic theory of types, 1980. Available online at `http://www.csie.ntu.edu.tw/~b94087/ITT.pdf`.

[MLM94] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic*. Universitext. Springer-Verlag, New York, 1994. A first introduction to topos theory, Corrected reprint of the 1992 edition.

[nlaa] nlab. Dependent type theory. Online at `http://ncatlab.org/nlab/show/dependent+type+theory`.

[nlab] nlab. First-order hyperdoctrine. Online at `http://ncatlab.org/nlab/show/first-order+hyperdoctrine`.

[nlac] nlab. Hyperdoctrine. Online at `http://ncatlab.org/nlab/show/hyperdoctrine`.

[nlad] nlab. Kripke-joyal semantics. Online at `http://ncatlab.org/nlab/show/Kripke-Joyal+semantics`.

[nlae] nlab. Martin-löf dependent type theory. Online at `http://ncatlab.org/nlab/show/Martin-L%C3%B6f+dependent+type+theory`.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, 2002.

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[See83] Robert A. G. Seely. Hyperdoctrines, natural deduction and the Beck condition. *Z. Math. Logik Grundlag. Math.*, 29(6):505–542, 1983.

[Smu84] Raymond Smulliyan. *To Mock a Mockingbird*. Oxford University Press, 1984.

[Stu] Aaron Stump. On Coquand's "An Analysis of Girard's Paradox". Available online at `homepage.cs.uiowa.edu/~astump/papers/qual.ps.gz`.

[SU98] Morten Heine B. Sørensen and Pawel Urzyczyn. Lectures on the Curry-Howard Isomorphism, 1998.

[Vis05] Angelo Vistoli. Grothendieck topologies, fibered categories and descent theory. In *Fundamental algebraic geometry*, volume 123 of *Math. Surveys Monogr.*, pages 1–104. Amer. Math. Soc., Providence, RI, 2005.

E-mail address: `habecker@math.uni-bonn.de`