Type Systems

Peter Dybjer
Chalmers University of Technology
Göteborg, Sweden

International Winter School
on Semantics and Applications
Montevideo, July 2003

What are types?

Examples in programming: integers, floating point numbers, types of functions and subroutines, truth values, records, classes, interfaces, ...

Examples in logic: natural numbers, truth values, unary numerical functions, binary numerical functions, propositional functions, truth functions, ...

Syntax or semantics? Parsing vs type-checking. Static semantics ...

Types as sets? Types as algebras? ...

Why types in programming?

- Prevent (certain kinds of) errors.
- Help structuring data and program, eg via abstract data types.
- Code reuse via polymorphism.
- Connection with logic. Propositions as types.

Types in logic - some milestones

1901 Russell's paradox

1910 Russell's ramified type theory, Principia Mathematica

1940 Church's simple theory of types

1958 Gödel's system T of computable functionals of finite type

1957,1968 Propositions as types (Curry, Howard)

1970 Intuitionistic type theory (Scott, Martin-Löf)

Types in computing

1950's High-level imperative programming languages (Fortran, Algol)

1960's Algol's successors, object-oriented languages (Algol 68, Pascal, Simula)

late 1960's Proof assistants based on type theory (Automath)

1970's Typed functional programming languages (ML)

Plan

1. The simply typed lambda calculus: background, syntax, typing rules, à la Church vs à la Curry.

2. Some applied simply typed lambda calculi: Church's simple theory of types, Gödel's T, PCF, typed functional programming languages.

3. Metatheoretic properties of the simply typed lambda calculus: type inference, reduction, conversion, standard model.

4. Intuitionistic type theory: dependent types, propositions as types, typing rules, judgement forms, the logical framework, type checking, rules for inductively defined sets.

Applications

- programming in a typed functional language (ML, Haskell)
- proving in HOL or Isabelle
- programming in a dependently typed functional language (Cayenne, DML)
- proving in a proof assistant based on constructive type theory (Coq, Lego, Agda/Alfa)

Russell's paradox

Let

$$\mathrm{R} = \{x | x \notin x\}$$

Then

$$\mathrm{R} \in \mathrm{R} \iff \mathrm{R} \notin \mathrm{R}$$

What's wrong?

"it is the distinction of logical types that is the key to the whole mystery" (Russell 1903)

Russell's ramified type theory

Distinguish between the types of

- individuals
- propositional functions (ranging over individuals)
- 2nd-order propositional functions (ranging over propositional functions)
- 3rd-order propositional functions (ranging over 2nd-order propositional functions)
- etc

Church's simple theory of types

Types (abstract syntax, add parentheses to disambiguate)

$$A, B ::= \iota \mid o \mid A \to B$$

- Base types:
  - ι is the type of individuals.
  - o is the type of propositions.
- $A \to B$ is the type of functions from A to B. (Church wrote (BA).)

Examples of types

- ι - the type of individuals
- $\iota \to o$ - the type of propositional functions
- $(\iota \to o) \to o$ - the type of 2nd-order propositional functions
- $((\iota \to o) \to o) \to o$ - the type of 3rd-order propositional functions

More examples of types

- $\iota \to \iota$ - the type of unary functions on individuals
- $o \to o$ - the type of unary connectives, eg negation
- $o \to o \to o$ - the type of binary connectives, eg disjunction
- $\iota \to \iota \to o$ - the type of binary relations between individuals, eg equality
- $(A \to A) \to A \to A$ - the type of iterators (Church numerals) of objects of type A.

What is the type of the quantifiers $\forall$, $\exists$?

Functions of several arguments

We use currying (due to Schönfinkel) for representing functions of several arguments. Note that $\rightarrow$ is right associative so that

$$o \rightarrow o \rightarrow o$$

abbreviates

$$o \rightarrow (o \rightarrow o)$$

If we add product types $A \times B$ with elements (a, b) for $a : A$ and $b : B$, then we also have the type of uncurried functions

$$o \times o \rightarrow o$$

General form of types

Note that all types have the form

$$A_1 \to \cdots \to A_n \to \alpha$$

for some base type α and types $A_1, \ldots, A_n$.

Base types. In Church's simple theory of types the base types are the constant types ι and o, but in general we can have other constant base types as well as type variables.

The simply typed lambda calculus à la Curry

Types

$$A, B ::= \alpha \mid A \to B$$

where α ranges over base types.

Terms à la Curry (untyped lambda terms):

$$f, a, b ::= f\ a \mid \lambda x.\, b \mid x$$

where x ranges over term variables:

$$x ::= \mathbf{x} \mid \mathbf{y} \mid \mathbf{z} \mid \cdots$$

Note $\mathbf{x}$ is a term variable, and x is a metavariable ranging over term variables.

Free variables, open and closed terms

$\mathrm{FV}(a)$ is the set of free term variables of the term a.

A term is closed if $\mathrm{FV}(a) = \emptyset$. It is open otherwise.

Syntactic conventions

Application is left-associative:

$$f\, a_1 \cdots a_n = (\cdots (f\, a_1) \cdots)\, a_n$$

Remove repeated lambdas:

$$\lambda x_1 \cdots x_n . b = \lambda x_1 . \ldots . \lambda x_n . b$$

Typing rules - à la Curry

Contexts:

$$\Gamma ::= x_1 : A_1, \ldots, x_n : A_n$$

Typing rules:

$$\frac{}{\Gamma \vdash x : A}\ (x : A \in \Gamma)$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.\, b : A \to B}$$

A type-assignment system.

Typing substitution

An admissible rule:

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash b[x := a] : B}$$

where $b[x := a]$ is the result of substituting the term a for all $x \in \mathrm{FV}(b)$.

Some typed lambda terms

$$
\begin{array}{lcll}
\mathrm{I} & = & \lambda x.\, x & : \quad \alpha \to \alpha \\
\mathrm{B} & = & \lambda x\, y\, z.\, x\, (y\, z) & : \quad (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \\
\mathrm{K} & = & \lambda x\, y.\, x & : \quad \alpha \to \beta \to \alpha \\
\mathrm{S} & = & \lambda x\, y\, z.\, x\, z\, (y\, z) & : \quad (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma
\end{array}
$$

Remark: we have *polymorphism* - a term can have many types, eg.

$$\mathrm{I} \quad = \quad \lambda x.\, x \quad : \quad (\alpha \to \beta) \to \alpha \to \beta$$

Variables and metavariables

Remark:

$$\lambda x.\, x : \alpha \to \alpha$$

is actually not a term but a term schema: x and α are metavariables ranging over term variables and base types. If ι is a base type and $\mathbf{x}$ is a term variable, then

$$\lambda \mathbf{x}.\, \mathbf{x} : \iota \to \iota$$

is a particular term which is an instance of the term schema above.

Old-fashioned and modern functions

A function in the old-fashioned sense is an expression depending on a free variable:

$$x : A \vdash b : B$$

A function in the modern sense is a λ-abstraction:

$$\vdash \lambda x.\, b : A \to B$$

Typed lambda calculus à la Church

Lambda terms à la Church decorate λ-abstractions with type-labels: we have

$$\lambda x : A.\, b$$

rather than

$$\lambda x.\, b$$

as in the ordinary untyped lambda terms employed in the system à la Church.

The typing rule for abstraction à la Church:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.\, b : A \to B}$$

Some terms à la Church

$$
\begin{array}{lcll}
\mathrm{I} & = & \lambda x : \alpha.\, x & : \quad \alpha \to \alpha \\
\mathrm{B} & = & \lambda x : \beta \to \gamma.\, \lambda y : \alpha \to \beta.\, \lambda z : \alpha.\, x\,(y\,z) & : \quad (\beta \to \gamma) \to (\alpha \to \beta) \\
 & & & \quad \to \alpha \to \gamma \\
\mathrm{K} & = & \lambda x : \alpha.\, \lambda y : \beta.\, x & : \quad \alpha \to \beta \to \alpha
\end{array}
$$

As before, these are really term schemas.

À la Curry vs à la Church

Curry: polymorphic, shorter, used in typed functional programming languages such as ML and Haskell.

Church: monomorphic, denotes unique function with its domain.

Type labels not relevant for computing lambda terms.

Removing type labels

There is a stripping function $|-|$ which maps terms à la Church to terms à la Curry by removing the type labels:

$$\begin{aligned} |x| &= x \\ |f\,a| &= |f|\,|a| \\ |\lambda x : A.\, b| &= \lambda x.\, |b| \end{aligned}$$

Correspondence between the systems à la Church and à la Curry

If $\Gamma \vdash a' : A$ in the system à la Church, then $\Gamma \vdash |a'| : A$ in the system à la Curry.

If $\Gamma \vdash a : A$ in the system à la Curry, then there is a term a' à la Church, such that $a = |a'|$ and $\Gamma \vdash a' : A$ in the system à la Church.

There is a one-to-one correspondence between type derivations in the two systems.

Why is the pure simply typed lambda calculus interesting?

The pure untyped lambda calculus is interesting because it is simple, and yet a Turing-complete computation model. Great encoding power: natural numbers, data types, general recursion, etc.

The typed lambda calculus is not a Turing-complete computation model. It is better thought of as a "logical framework". You get different interesting "applied" lambda calculi by adding different constants. (We shall look at several examples.)

Its role is similar to that of the first order predicate calculus. It is a logical framework and you get different interesting first order theories by adding different axioms.

Some applied typed lambda calculi

- Church's simple theory of types (1940), the HOL proof assistant.
- Gödel's system T (1958)
- Plotkin's PCF - "LCF as a programming language" (1976)
- "Real" functional programming languages (ML, Haskell)
- Intuitionistic higher order logic, the Isabelle logical framework.

Churchs's simple theory of types

Church had the following primitive constants:

$$\begin{aligned} \neg &: o \to o \\ \vee &: o \to o \to o \\ \forall_A &: (A \to o) \to o \\ \iota_A &: (A \to o) \to A \end{aligned}$$

$\iota_A \phi$ denotes the unique a such that ϕa is true. Other logical constants are definable, eg

$$A \supset B = \neg A \vee B$$

where $\supset$ and $\vee$ are infix constants.

Axioms and inference rules

Some elements of the type o of propositions are classical truths. Church has 11 axioms and 6 inference rules for deriving such truths, eg:

$\forall$-introduction (inference rule VI):

$$\frac{f\,x}{\forall_A\,f}$$

$\forall$-elimination (axiom 5):

$$\forall_A\,f \supset f\,x$$

In both cases $\Gamma \vdash f : A \to o$ and $x : A$ is a fresh variable not in Γ.

Gordon's HOL-system

The HOL-system (Gordon) is essentially an implementation of a version of Church's simple theory of types with type variables and typing à la Curry.

Gödel's system T of computable functionals of finite type

Base type: N.

Constants:

$$\begin{aligned} 0 &: \mathrm{N} \\ \mathrm{Succ} &: \mathrm{N} \to \mathrm{N} \\ \mathrm{R} &: C \to (\mathrm{N} \to C \to C) \to \mathrm{N} \to C \end{aligned}$$

R is a polymorphic constant. If we want a monomorphic system we decorate R with a type label C and get R_C.

R is a constant for primitive recursion

$\mathrm{R}\,d\,e : \mathrm{N} \to C$ denotes a function f defined by primitive recursion with base case d and recursion step e:

$$\begin{aligned} f\,0 &= d \\ f\,(\mathrm{Succ}\,n) &= e\,n\,(f\,n) \end{aligned}$$

Programming in Gödel's T

Recursion equations for factorial:

$$\begin{aligned} \mathrm{fact}\,0 &= 1 \\ \mathrm{fact}\,(\mathrm{Succ}\,n) &= \mathrm{mult}\,(\mathrm{Succ}\,n)\,(\mathrm{fact}\,n) \end{aligned}$$

Factorial in T, using R:

$$\mathrm{fact} = \mathrm{R}\,1\,(\lambda x\,y.\,\mathrm{mult}\,(\mathrm{Succ}\,x)\,y)$$

How to define multiplication mult? Addition?

Primitive recursive functions and primitive recursive functionals

Exercise: show that all primitive recursive functions are definable in System T.

Exercise: show that there are functions in System T which are not primitive recursive.

PCF

Plotkin 1976: "LCF as a programming language". The paper is about the correspondence between denotational and operational semantics, and in particular discusses the full abstraction problem.

PCF is a small functional language with two base types: N and Bool. The language has general recursion expressed using a constant fix for fixed points. We will give two variants of Plotkin's language.

PCF - constants

$$
\begin{array}{rcl}
\text{True} & : & \text{Bool} \\
\text{False} & : & \text{Bool} \\
\text{if} & : & \text{Bool} \to A \to A \to A \\
0 & : & \text{N} \\
\text{Succ} & : & \text{N} \to \text{N} \\
\text{pred} & : & \text{N} \to \text{N} \\
\text{isZero} & : & \text{N} \to \text{Bool} \\
\text{fix} & : & (A \to A) \to A
\end{array}
$$

Programming in PCF

Recursion equations for factorial:

$$\begin{aligned} \mathrm{fact}\,0 &= 1 \\ \mathrm{fact}\,(\mathrm{Succ}\,n) &= \mathrm{mult}\,(\mathrm{Succ}\,n)\,(\mathrm{fact}\,n) \end{aligned}$$

One equation:

$$\mathrm{fact}\,n = \mathrm{if}\,(\mathrm{isZero}\,n)\,1\,(\mathrm{mult}\,n\,(\mathrm{fact}\,(\mathrm{pred}\,n)))$$

Factorial in PCF, using fix:

$$\mathrm{fact} = \mathrm{fix}\,(\lambda f\,n.\,\mathrm{if}\,(\mathrm{isZero}\,n)\,1\,(\mathrm{mult}\,n\,(f\,(\mathrm{pred}\,n))))$$

Plotkin's PCF

We have given a variant of Plotkin's PCF, which differs in the following respects:

- It is formulated à la Church.
- if is only defined for base types: $A ::= \mathrm{Bool} \mid \mathrm{N}$.
- fix has a type label so it is a monomorphic language.
- There are also constants for each natural number.

PCF with a case analysis constant

Replace pred and isZero by definition by cases on 0 and Succ:

$$\mathrm{natcases} : \mathrm{N} \to C \to (\mathrm{N} \to C) \to C$$

Recursion equations:

$$\begin{aligned} \mathrm{natcases}\, 0\, d\, e &= d \\ \mathrm{natcases}\, (\mathrm{Succ}\, n)\, d\, e &= e\, n \end{aligned}$$

Redefine factorial using $\mathrm{natcases}$ and fix!

Define R in PCF using $\mathrm{natcases}$ and fix!

Typed functional programming languages

- Strict: The ML-family (SML, OCAML). Call-by-value.
- Lazy: Miranda, Haskell. Call-by-name.

Essentially, simply typed lambda calculus where base types are generated from type variables and recursive type constructors ("polymorphic datatypes"). Constructors and case analysis constructs for each recursive type.

Other features: modules, classes, ..., integers, strings, floating point numbers, etc, and lots of syntactic sugar.

A recursive type of lambda expressions in Haskell

```
data Exp = Apply Exp Exp | Lambda Var Exp | Var Var
```

This is shorthand for adding Exp to the set of types, and the constructors

$$\begin{aligned} \mathrm{Apply} &: \mathrm{Exp} \to \mathrm{Exp} \to \mathrm{Exp} \\ \mathrm{Lambda} &: \mathrm{Var} \to \mathrm{Exp} \to \mathrm{Exp} \\ \mathrm{Var} &: \mathrm{Var} \to \mathrm{Exp} \end{aligned}$$

to the constants. We also add a constant for case analysis.

Case analysis on Exp

We also need to add a case analysis construct:

$$\begin{aligned}\mathrm{expcases} \quad : \quad & \mathrm{Exp} \to (\mathrm{Exp} \to \mathrm{Exp} \to C) \to (\mathrm{Var} \to \mathrm{Exp} \to C) \\ & \to (\mathrm{Var} \to C) \to C\end{aligned}$$

Recursion equations:

$$\begin{aligned}\mathrm{expcases}\,(\mathrm{Apply}\,f\,a)\,c\,d\,e &= c\,f\,a \\ \mathrm{expcases}\,(\mathrm{Lambda}\,x\,b)\,c\,d\,e &= d\,x\,b \\ \mathrm{expcases}\,(\mathrm{Var}\,x)\,c\,d\,e &= e\,x\end{aligned}$$

Type inference à la Church

The type system à la Church is *monomorphic*, that is, each term has at most one type. This type (if it exists) is easily determined by reading the rules backwards. In the case of an application $f\,a$, first check whether f has a type C and whether a has a type A. If this is the case and if $C = A \to B$ for some B, then the type of $f\,a$ is B. Otherwise it is not typeable.

The abstraction case $\lambda x : A.\, b$. Determine the type of b in the context $x : A$!

Thus, in general we need to do type inference of open expressions: Given a context Γ and a term a determine the unique type A such that $\Gamma \vdash a : A$.

Type inference à la Curry

The type system à la Curry is *polymorphic*, that is, each term may have several types. If we assume that we have type variables in our system, then each closed term a which is typeable (has some type) has a principal type A; each of its other types $a : B$ is an instance of A, that is, there are type variables $\alpha_1, \ldots, \alpha_n$ and types $A_1, \ldots, A_n$, such that

$$B = A[\alpha_1 := A_1, \ldots, \alpha_n := A_n]$$

for some type variables $\alpha_1, \ldots, \alpha_n$ and types $A_1, \ldots, A_n$.

The principal type is obtained by reading the typing rules backwards and using unification of type expressions.

Type inference algorithm

Consider first the case of a *closed* application $f\,a$. We assume that we have inferred that A is the principal type of a, and C is the principal type of f, and A and C have disjoint sets of type variables. Then we try to unify C and $A \to \alpha$ (for a fresh type variable α). If successful we get a most general unifying substitution σ, such that $C\,\sigma = (A \to \alpha)\,\sigma$. Then the principal type of $f\,a$ is $\alpha\,\sigma$.

To infer the principal type of an abstraction $\lambda x.\,b$, we infer the principal type of the expression b which may contain free occurrences of the variable x. So in general we need to do type inference for open expressions.

Type inference for open expressions

In general we need to infer the principal typing of an open expression a which depends on free variables in $\{x_1, \ldots, x_n\} \supseteq \mathrm{FV}(a)$. Such a principal typing is a sequence of types $A_1, \ldots, A_n, A$, such that

$$x_1 : A_1, \ldots, x_n : A_n \ \vdash \ a : A$$

such that any other typing

$$x_1 : B_1, \ldots, x_n : B_n \ \vdash \ a : B$$

is an instance of it, that is, B_1 is an instance of A_1, ..., B_n is an instance of A_n, and B is an instance of A.

The principal typing of an open application

Let

$$x_1 : C_1, \ldots, x_n : C_n \ \vdash \ f : C$$

$$x_1 : A_1, \ldots, x_n : A_n \ \vdash \ a : A$$

be the respective principal typings of the expressions f and a with free variables $x_1, \ldots, x_n$. (We assume disjoint type variables occurring in $A_1, \ldots, A_n, A$ and in $C_1, \ldots, C_n, C$.) Then the principal typing of $f\,a$ is

$$x_1 : A_1\sigma, \ldots, x_n : A_n\sigma \ \vdash \ f\,a : \alpha\sigma$$

where α is a fresh type variable and there is a most general unifying substitution σ such that $A_1\sigma = C_1\sigma, \ldots, A_n\sigma = C_n\sigma$ and $C\sigma = (A \to \alpha)\sigma$.

An algorithm using principal pairs

Remark. The above is the most straightforward algorithm. A better algorithm is obtained by determining the principal typing of an expression a in a context Γ. See eg lecture notes on Types by A. Pitts:

`http://www.cl.cam.ac.uk/Teaching/2002/Types/`

β-reduction

Contracting a β-redex:

$$(\lambda x.\, b)\, a \to_\beta b[x := a]$$

As before, $b[x := a]$ is the result of substituting the term a for all $x \in \mathrm{FV}(b)$.
We perform α-conversion to avoid capture of variables.

Inductive definition of β-reduction

$$(\lambda x.\, b)\, a \to_\beta b[x := a]$$

Contextual closure (two application rules and the ξ-rule).

$$\frac{f \to_\beta f'}{f\, a \to_\beta f'\, a} \qquad \frac{a \to_\beta a'}{f\, a \to_\beta f\, a'} \qquad \frac{b \to_\beta b'}{\lambda x.\, b \to_\beta \lambda x.\, b'}$$

Reflexive and transitive closure:

$$f \to_\beta f \qquad \frac{f \to_\beta f' \qquad f' \to_\beta f''}{f \to_\beta f''}$$

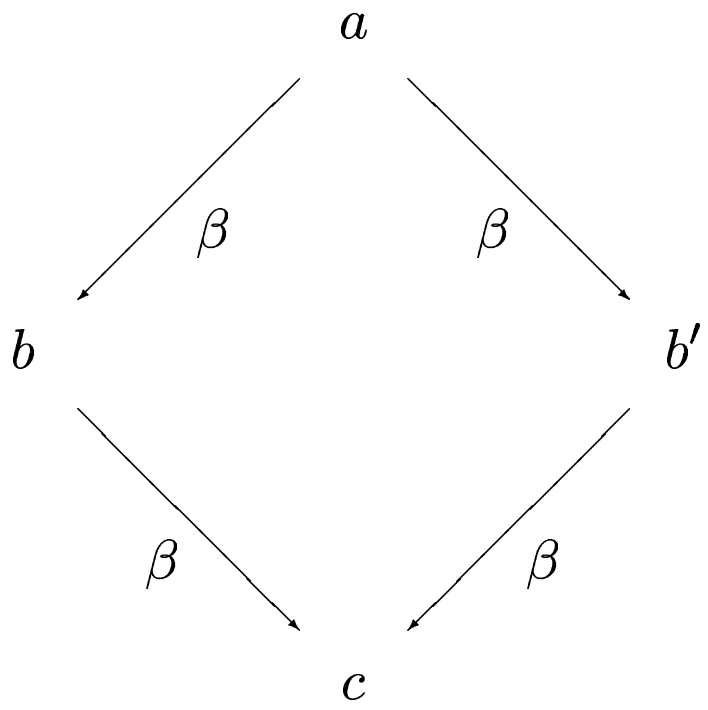
One-step β-reduction

The above is the definition of β-reduction in an arbitrary number of steps, including 0 steps.

If we remove the rules of reflexivity and transitivity we get a definition of one-step β-reduction.

Confluence

$\rightarrow_\beta$ is *Church-Rosser* iff $a \ \rightarrow_\beta \ b$ and $a \ \rightarrow_\beta \ b'$ implies that there is a c such that



Normal and canonical forms

b *is a normal form* iff b is irreducible: there is no one-step reduction $b \rightarrow_\beta b'$.

a *has normal form* b iff $a \rightarrow_\beta b$ and b *is a normal form*.

c *is a canonical form* iff c is a closed normal form. We write $a \Rightarrow_\beta c$ if the closed term a has canonical form c.

Church-Rosser implies uniqueness of normal (and canonical) forms: If a has normal forms b and b', then $b = b'$.

Subject reduction

Reduction preserves typing:

If $a \to_\beta a'$ and $\Gamma \vdash a : A$, then $\Gamma \vdash a' : A$.

Proof by induction on $a \to_\beta b$. Key case

$$(\lambda x.\, b)\, a \to_\beta b[x := a]$$

and $\Gamma \vdash (\lambda x.\, b)\, a : B$. Then there exists A such that $\Gamma \vdash \lambda x.\, b : A \to B$ and $\Gamma \vdash a : A$. The conclusion follows from the fact that substitution preserves typing: if $\Gamma, x : A \vdash b : B$ and $\Gamma \vdash a : A$, then $\Gamma \vdash b[x := a] : B$.

Weak and strong normalization

A term is *weakly normalizing* if it has a normal form.

A term a is *strongly normalizing* if there is no infinite reduction sequence of *one-step* reductions

$$a \ \rightarrow_\beta \ a_1 \ \rightarrow_\beta \ a_2 \ \rightarrow_\beta \ \cdots$$

A one-step β-reduction is a reduction where just one β-redex is contracted. $\rightarrow_\beta$ is the reflexive-transitive closure of one-step β-reduction.

Proof of strong normalization

What goes wrong if we try to prove strong normalization by induction on the structure of a type derivation for a term?

The problem is that two terms may be strongly normalizing but their application is not. Eg $\Delta = \lambda x.\, x\, x$ is strongly normalizing but $\Omega = \Delta\, \Delta$ is not.

Tait's method: Strengthen the induction hypothesis! Prove that for $f : A \to B$, not only is f strongly normalizing, but also maps sn to sn, that is, $f\, a$ is sn for all sn a. This need to be extended up the type hierarchy leading to the definition of "reducible term" (also called "computable term").

Reducible terms - definition

Base types. A term of base type is reducible if it is strongly normalizing.

Function types. A term $\Gamma \vdash f : A \to B$ is reducible if $\Gamma \vdash f\, a : B$ is reducible for all reducible terms $\Gamma \vdash a : A$.

All reducible terms are strongly normalizing

Lemma 1. All reducible terms are strongly normalizing.

Lemma 2. If $a_1, \ldots, a_n$ are strongly normalizing, then

$$\Gamma \vdash x\, a_1 \cdots a_n : A$$

is reducible.

Lemma 1 and 2 are proved simultaneously by induction on the type of the term.

All typeable terms are reducible

We first prove a stronger lemma by induction on the term:

Lemma 3. If $x_1 : A_1, \ldots, x_n : A_n \vdash a : A$ is reducible and $\Gamma \vdash a_i : A_i$ for $i = 1, \ldots, n$ are reducible, then $\Gamma \vdash a[x_1 := a_1, \ldots, x_n := a_n] : A$ is reducible.

From this and the fact that variables are reducible (lemma 2) it follows that

Lemma 4. All typeable terms are reducible.

Our theorem is a corollary of lemma 1 and 4:

Corollary. All typeable terms are strongly normalizing.

Proof of lemma 3

The proof of lemma 3 is straightforward, except in the case of $a = \lambda x.\, b$. To prove this we use the following lemma:

Lemma 5. If a is strongly normalizing and

$$\Gamma \vdash b[x := a]\, a_1 \cdots a_n : A$$

is reducible, then

$$\Gamma \vdash (\lambda x.\, b)\, a\, a_1 \cdots a_n : A$$

is reducible.

This is proved by induction on A. (Why is the condition a strongly normalizing needed?)

Standard model - types as sets

The standard meaning of a type A is a set $[\![A]\!]_R$ which depends on a type environment R (which assigns a set to each base type α). It is defined by induction on derivations as follows:

$$\begin{aligned} [\![\alpha]\!]_R &= R(\alpha) \\ [\![A \to B]\!]_R &= [\![A]\!]_R \to [\![B]\!]_R \end{aligned}$$

Remark: $\to$ on the RHS denotes the set-theoretic function space (of total functions).

Standard model - terms as elements

The standard meaning of a typeable term $\Gamma \vdash a : A$ is an element $[\![a]\!]^A_{R,\rho} \in [\![A]\!]_R$ which depends on a type environment R and a term environment ρ (which assigns to each variable $x_i : A_i$ in Γ an element $a_i \in [\![A_i]\!]_R$). It is defined as follows:

$$\begin{aligned} [\![x]\!]^A_{R,\rho} &= \rho(x) \\ [\![f\, a]\!]^B_{R,\rho} &= [\![f]\!]^{A\to B}_{R,\rho}([\![a]\!]^A_{R,\rho}) \\ [\![\lambda x.\, b]\!]^{A\to B}_{R,\rho}(a) &= [\![b]\!]^B_{R,(\rho,x\mapsto a)} \end{aligned}$$

Note the use of set-theoretic function application on the RHS.

β-reduction preserves meaning

If $\Gamma \vdash a : A$ and $a \to_\beta a'$, then $[\![a]\!]^A_{R,\rho} = [\![a']\!]^A_{R,\rho}$ for all R and ρ.

How to look at the standard model?

The real meaning? Which? Platonic vs intuitionistic?

A translation of an object language to a metalanguage?

A formal translation of type theory to set theory?

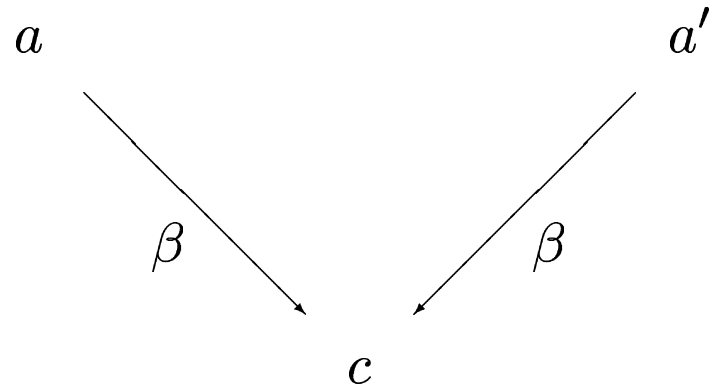
Standard model and meaning explanations. The explanation of the notion of a function.

Tarski.

β-conversion

β-convertibility $=_\beta$ is the least congruence (wrt application and λ-abstraction) containing $\rightarrow_\beta$.

Alternative characterization: $a =_\beta a'$ iff there is a term c such that



If $\Gamma \vdash a, a' : A$ then we can decide $a =_\beta a'$ by comparing β-normal forms.

Is β-conversion complete?

Does $\Gamma \vdash a, a' : A$ and $[\![a]\!]^A_{R,\rho} = [\![a']\!]^A_{R,\rho}$ imply that $a =_\beta a'$?

No! $\lambda f.\, f$ and $\lambda f\, x.\, f\, x$ both have type $(\alpha \to \beta) \to \alpha \to \beta$ and

$$[\![\lambda f.\, f]\!]^{(\alpha\to\beta)\to\alpha\to\beta}_{R,\rho} = [\![\lambda f\, x.\, f\, x]\!]^{(\alpha\to\beta)\to\alpha\to\beta}_{R,\rho}$$

but $\lambda f\, x.\, f\, x$ and $\lambda f.\, f$ are different β-normal forms.

η-conversion

Add the rule of η-conversion

$$\lambda x.\, f\, x \;\; =_{\beta\eta} \;\; f$$

where $x \notin \mathrm{FV}(f)$.

Friedman's completeness theorem: $\beta\eta$-convertibility is complete, that is, if $\Gamma \;\vdash\; a : A$ and $[\![a]\!]^A_{R,\rho} = [\![a']\!]^A_{R,\rho}$ for all R and ρ then $a \;\; =_{\beta\eta} \;\; a'$.

$\beta\eta$-normal form

Wrt reduction one considers both η-reduction to $\beta\eta$-normal form and η-expansion to $\beta\eta$ long normal form. The latter depends on the type of an expression: a $\beta\eta$ long normal form of type

$$A_1 \to \cdots \to A_n \to \alpha$$

has the form

$$\lambda x_1 \cdots x_n . b$$

where b is a normal form of base type α not starting with λ - it is a "neutral" term.

Typed conversion

Conversion

$$a \ =_{\beta\eta} \ a'$$

is a relation between untyped terms.

We may also consider typed conversion judgements

$$\Gamma \ \vdash \ a = a' : A$$

meaning that a and a' are convertible terms of type A, that is, $\Gamma \ \vdash \ a, a' : A$.

Martin-Löf's intuitionistic type theory is formalized using typed conversion judgements.

Normalization by evaluation

An efficient way to compute the η-long normal form by inverting the standard meaning function, where base types are interpreted as sets of terms: $R(\alpha) = \mathrm{Term}(\alpha)$. Here $\mathrm{Term}(A)$ is the set of terms of type A.

The normalization function for closed terms $\vdash a : A$:

$$\mathrm{nbe}_A(a) = \downarrow_A [\![a]\!]_{\mathrm{Term}}$$

where

$$\downarrow_A \quad : \quad [\![A]\!]_{\mathrm{Term}} \to \mathrm{Term}(A)$$

is called the reification of a term.

Reification and reflection

Reification is defined simultaneously with reflection $\uparrow_A$ of a denotation:

$$\begin{aligned} \downarrow_A &: [\![A]\!]_{\mathrm{Term}} \to \mathrm{Term}(A) \\ \uparrow_A &: \mathrm{Term}(A) \to [\![A]\!]_{\mathrm{Term}} \end{aligned}$$

$$\begin{aligned} \downarrow_\alpha a &= a \\ \uparrow_\alpha a &= a \\ \downarrow_{A\to B} f &= \lambda x.\ \downarrow_B f(\uparrow_A x) \\ (\uparrow_{A\to B} f)(a) &= \uparrow_B f\,(\downarrow_A a) \end{aligned}$$

Reduction in Gödel's T

Extend $\rightarrow_\beta$ with the following rules.

$$\begin{array}{rcl} \mathrm{R}\, d\, e\, 0 & \rightarrow_\beta & d \\ \mathrm{R}\, d\, e\, (\mathrm{Succ}\, n) & \rightarrow_\beta & e\, n\, (\mathrm{R}\, d\, e\, n) \end{array}$$

Church-Rosser, subject reduction, and strong normalization hold. It is straightforward to extend the standard semantics to system T: the base type N is interpreted as the set of natural numbers, etc.

PCF: reduction

Extend $\to_\beta$ with the following rules

$$\begin{array}{rcl}
\text{if True}\, b\, b' & \to_\beta & b \\
\text{if False}\, b\, b' & \to_\beta & b' \\
\text{pred}\, (\text{Succ}\, n) & \to_\beta & n \\
\text{isZero}\, 0 & \to_\beta & \text{True} \\
\text{isZero}\, (\text{Succ}\, n) & \to_\beta & \text{False} \\
\text{fix}\, f & \to_\beta & f\, (\text{fix}\, f)
\end{array}$$

Church-Rosser and subject reduction hold. Normalization does not hold. The standard semantics is in terms of Scott domains.

Evaluation in functional languages

Differs from β-reduction in lambda calculus in the following respects:

- uses deterministic strategy for selecting redexes (call-by value or call-by-name); β-reduction is indeterministic.
- reduce only closed terms; β-reduction applies to open terms too.
- really only reduce closed terms of base type; (reducing open terms or terms of function type is related to partial evaluation, that is, optimizing programs by performing "static" operations.)

Canonical terms

Closed normal terms are called canonical terms.

The canonical terms of type Bool are

$$a ::= \mathrm{True} \mid \mathrm{False}$$

The canoncial terms of type N are

$$a ::= 0 \mid \mathrm{Succ}\, a$$

Evaluation relation for System T and PCF

Let $\Rightarrow_\beta$ be the evaluation relation between a closed term and its normal form (a canonical term): $a \Rightarrow_\beta v$ iff $a \rightarrow_\beta v$ and v canonical. $\Rightarrow_\beta$ is sometimes called "big step" and $\rightarrow_\beta$ "small step".

$\Rightarrow_\beta$ is deterministic, ie a partial function, both for System T and PCF.

It is a total function on typeable terms for System T.

It is possible to give a direct inductive definition of $\Rightarrow_\beta$ without reference to $\rightarrow_\beta$. Kahn called this "natural semantics", see eg Martin-Löf (1979).

Fundamental property of typing in System T

If $\vdash\ a : \mathrm{Bool}$ then $a \Rightarrow_\beta \mathrm{True}$ or $a \Rightarrow_\beta \mathrm{False}$.

If $\vdash\ a : \mathrm{N}$ then $a \Rightarrow_\beta 0$ or $a \Rightarrow_\beta \mathrm{Succ}\, b$ and $\vdash\ b : \mathrm{N}$.

Meaning explanations for System T

Martin-Löf: the meaning of a typing judgement

$$\vdash \; a : \mathrm{Bool}$$

is that $a \Rightarrow_\beta \mathrm{True}$ or $a \Rightarrow_\beta \mathrm{False}$. The meaning of

$$\vdash \; a : \mathrm{N}$$

is that $a \Rightarrow_\beta 0$ or $a \Rightarrow_\beta \mathrm{Succ}\, b$ and $\vdash \; b : \mathrm{N}$.

The typing rules for System T are correct wrt these meaning explanations. They can be extended to function types and general judgements $\Gamma \vdash a : A$ for arbitary Γ. How?

Fundamental property of typing in PCF

Well-typed programs cannot go wrong:

If $\vdash\ a : \mathrm{Bool}$ and $a \Rightarrow_\beta v$ then $v = \mathrm{True}$ or $v = \mathrm{False}$.

If $\vdash\ a : \mathrm{N}$ and $a \Rightarrow_\beta v$ then $v = 0$ or $v = \mathrm{Succ}\, b$ and $\vdash\ b : \mathrm{N}$.

Meaning explanations for PCF

We can imagine meaning explanations also for PCF: "partial types".

The meaning of

$$\vdash \ a : \mathrm{Bool}$$

is that if $a \Rightarrow_\beta v$ then $v = \mathrm{True}$ or $v = \mathrm{False}$. The meaning of

$$\vdash \ a : \mathrm{N}$$

is that if $a \Rightarrow_\beta v$ then $v = 0$ or $v = \mathrm{Succ}\, b$ and $\vdash \ b : \mathrm{N}$.

The typing rules for PCF are correct wrt this interpretation.

Such meaning explanations can be given for general typing judgements in PCF. How?

Dependent types

What is a dependent type? Synonyms "family of types", "indexed types", "indexed family of types".

Examples of dependent types:

- $\mathrm{Vect}\, A\, n$ – vectors (arrays) of length n with elements in A
- $\mathrm{Matrix}\, A\, m\, n$ – $m \times n$-matrices with elements in A
- $\mathrm{BBT}\, A\, n$ – balanced binary trees of height n with nodes in A
- $\mathrm{BST}\, A\, lb\, ub$ – binary search trees with elements in the interval (lb, ub) with nodes in A (an ordered type)

Dependent types - terminology

By dependent types we mean types indexed by elements of another type. (For example, in the three first examples on the previous slide, we index by natural numbers, and in the fourth example, we index on an arbitrary ordered type.)

However, polymorphic types such as the type $[A]$ of lists indexed by the type A of elements is usually not called a dependent type. It's a type-indexed family of types.

The `zip`-function

Haskell library function

```
zip :: [a] -> [b] -> [(a,b)]

zip []     []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []
```

exceptional cases when lists are of unequal length.

With dependent types we avoid the exceptional cases:

$$\mathrm{zip} : \mathrm{Vect}\, A\, n \to \mathrm{Vect}\, B\, n \to \mathrm{Vect}\, (A \times B)\, n$$

More examples

Matrix multiplication (elements are natural numbers)

$$\mathrm{matrixMult} : \mathrm{Matrix}\,\mathrm{N}\,m\,n \to \mathrm{Matrix}\,\mathrm{N}\,n\,p \to \mathrm{MatrixN}\,m\,p$$

An AVL-tree is a balanced binary search tree. By dependent types we can capture the fact that AVL-insertion preserves this property.

Propositions as types

Curry 1957 observed the similarity between the types of the K and S-combinators

$$\begin{aligned} \mathrm{K} &: A \to B \to A \\ \mathrm{S} &: (A \to B \to C) \to (A \to B) \to A \to C \end{aligned}$$

and two Hilbert-style axioms for implication

$$A \supset B \supset A$$

$$(A \supset B \supset C) \supset (A \supset B) \supset A \supset C$$

Which axioms do the types of the I and B-combinators correspond to?

Modus ponens as application

Modus ponens

$$\frac{\Gamma \vdash A \supset B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

The typing rule for application

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B}$$

Propositions as types - more connectives

Correspondences between intuitionistic connectives and type formers

$$
\begin{aligned}
A \supset B &= A \to B \\
A \wedge B &= A \times B \\
A \vee B &= A + B
\end{aligned}
$$

The Brouwer-Heyting-Kolmogorov interpretation of the intuitionistic connectives

A proof of $A \supset B$ is a method which transforms a proof of type A to a proof of type B.

A proof of $A \wedge B$ is a pair consisting of a proof of A and a proof of B.

A proof of $A \vee B$ is either a proof of A or a proof of B.

Martin-Löf's meaning explanations for the connectives

$c : A \supset B$ iff $c \Rightarrow_\beta \lambda x.\, b$ and $b[x := a] : B$ for all $a : A$.

$c : A \wedge B$ iff $c \Rightarrow_\beta (a, b)$ and $a : A$ and $b : B$.

$c : A \vee B$ iff either $c \Rightarrow_\beta \mathrm{Inl}\, a$ and $a : A$, or $c \Rightarrow_\beta \mathrm{Inr}\, b$ and $b : B$.

Rules for conjunction

Introduction rule (generating canonical elements/proofs):

$$\mathrm{Pair} \quad : \quad A \to B \to A \times B$$

$(a, b) = \mathrm{Pair}\, a\, b.$

Elimination rules (methods for transforming canonical elements to canonical elements):

$$\begin{aligned} \mathrm{fst} &: A \times B \to A \\ \mathrm{snd} &: A \times B \to B \end{aligned}$$

Rules for disjunction

Introduction rules (generating canonical elements/proofs):

$$\begin{aligned} \mathrm{Inl} &: A \to A + B \\ \mathrm{Inr} &: B \to A + B \end{aligned}$$

Elimination rule (method for transforming canonical elements to canonical elements):

$$\mathrm{case} : (A \to C) \to (B \to C) \to (A + B) \to C$$

Truth, falsity, negation

Truth:

$$\top = \mathbf{1}$$

There is one canonical proof of $\top$.

Falsity:

$$\bot = \emptyset$$

There is no canonical proof of $\bot$.

Negation:

$$\neg A = A \supset \bot = A \to \emptyset$$

A proof of a negation is an empty function.

Howard: quantifiers and dependent types

Correspondence between

$$\begin{aligned} \forall x : A.B &= \Pi x : A.B \\ \exists x : A.B &= \Sigma x : A.B \end{aligned}$$

Here we shall write $(x : A) \to B$ for $\Pi x : A.\, B$.

The propositional function B depends on x. Its corresponding type of proofs B depends on x.

For example, $\mathrm{Even}\, n$ is a family of types depending on $n : \mathrm{N}$.

Brouwer-Heyting-Kolmogorov interpretation of the intuitionistic quantifiers

A proof of $\forall x : A. B$ is a method which for an arbitrary element a of A returns a proof of $B[x := a]$.

A proof of $\exists x : A. B$ is a pair consisting of a element a of A (the witness) and a proof of $B[x := a]$.

Martin-Löf's meaning explanations for the quantifiers

$c : \forall x : A.\, B$ iff $c \Rightarrow_\beta \lambda x.\, b$ and $b[x := a] : B[x := a]$ for all $a : A$.

$c : \exists x : A.\, B$ iff $c \Rightarrow_\beta (a, b)$ and $a : A$ and $b : B[x := a]$.

The correctness of all rules of intuitionistic typed predicate logic can be justified by Martin-Löf's meaning explanations.

A proposition as a type

The proposition (formula)

$$\forall m : \mathrm{N}.\forall n : \mathrm{N}.\mathrm{Even}\, m \supset \mathrm{Even}\, n \supset \mathrm{Even}\, (m + n)$$

becomes the type

$$(m : \mathrm{N}) \to (n : \mathrm{N}) \to \mathrm{Even}\, m \to \mathrm{Even}\, n \to \mathrm{Even}\, (m + n)$$

Proving the theorem constructively is the same as writing a function (a program!) of the corresponding type! Proving by induction is the same as writing a function by primitive recursion.

Curry-Howard for programming

a	:	A
element	belongs to	type
proof	proves	proposition
program	satisfies	specification

A proposition as a specification

The sorting proposition

$$\forall xs : [\mathrm{N}].\, \exists ys : [\mathrm{N}].\, \mathrm{Sorted}\, ys \wedge \mathrm{Perm}\, xs\, ys$$

The corresponding type

$$(xs :: [\mathrm{N}]) \to \Sigma ys : [\mathrm{N}].\, \mathrm{Sorted}\, ys \times \mathrm{Perm}\, xs\, ys$$

To prove the proposition constructively is the same as constructing a function $\mathrm{sortProof}$ of the corresponding type.

$\mathrm{sortProof}\, xs$ returns a triple $(ys, (p, q))$, where ys is the sorted version of xs, p is the proof that ys is sorted and q is the proof that xs and ys are permuations of each other.

Program extraction

From $\mathrm{sortProof}$ we can extract a program

$$\mathrm{sortProgram} :: [\mathrm{N}] \to [\mathrm{N}]$$

which only returns ys but not the proofs p and q. We do "dead code elimination" to remove the parts of $\mathrm{sortProof}$ which do not contribute to computing ys.

Intuitionistic type theory

The idea: have a foundational theory for intuitionistic mathematics, which is based on the Curry-Howard isomorphism.

- a functional language with dependent types where all programs terminate;
- a specification language including predicate logic;
- a full-scale constructive set theory – like "Zermelo-Fraenkel set theory" or "Church simple theory of types" but for constructive mathematics!

Functional programming and Martin-Löf type theory

simply typed lambda calculus	dependently typed lambda calculus
recursive datatypes	inductive sets and families
general recursive functions	primitive recursive functions

Note: recursive datatypes in functional programming can be "non strictly positive". We can have a reflexive type $\mathrm{Reflexive}$ with a constructor

$$\mathrm{C} : (\mathrm{Reflexive} \to \mathrm{Reflexive}) \to \mathrm{Reflexive}$$

This is not allowed in Martin-Löf type theory.

Primitive recursive means structurally recursive, including primitive listrecursion, etc.

Dependent function types

Abstract syntax: instead of $A \to B$ we now have $(x : A) \to B$ where the type B may contain free occurrences of x. $A \to B$ is an abbreviation of $(x : A) \to B$ when B does not contain free occurrences of x.

Other notation for dependent function space (cartesian product of a family of types): $\Pi x : A.\, B$, $\prod_{x:A} B$, and $(x : A)B$.

Informal meaning explanation: $f : (x : A) \to B$ iff $f\, a : B[x := a]$ for all $a : A$.

The lambda calculus with dependent types

Typing judgement:

$$\Gamma \vdash a : A$$

where the type A may depend on the variables in the context Γ.

Contexts:

$$\Gamma ::= x_1 : A_1, \ldots, x_n : A_n$$

Note: A_2 may depend on $x_1 : A_1$, ..., A_n may depend on $x_1 : A_1, \ldots, x_{n-1} : A_{n-1}$.

We here consider terms à la Curry (can also consider terms à la Church).

Typing rules à la Curry

$$\frac{}{\Gamma \vdash x : A}\ (x : A \in \Gamma)$$

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B[x := A]} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.\, b : (x : A) \to B}$$

Note the difference in the rule for application wrt the simply typed case. Is this enough? No, there is also a type equality rule. Typing depends on reduction. Other complication: well-formedness of types is not a simple matter of parsing (in the simply typed case we just had a context free grammar). Now me must prove that a type is well-formed.

The four judgement forms

Categorical judgements (no assumptions)

$$\vdash A \ \mathbf{type} \qquad\qquad \vdash A = A'$$

$$\vdash a : A \qquad\qquad \vdash a = a' : A$$

Note the typed equality (conversion) judgement for terms.

(We often suppress $\vdash$.)

Hypothetical judgements

$$\Gamma \vdash A \ \mathbf{type} \qquad\qquad \Gamma \vdash A = A'$$

$$\Gamma \vdash a : A \qquad\qquad \Gamma \vdash a = a' : A$$

Γ is a context of the form $x_1 : A_1, \ldots, x_n : A_n$ as above. We also need a judgement form for well-formed contexts:

$$\Gamma \ \mathbf{context}$$

The type equality rule

A crucial rule for dependent types:

$$\frac{\Gamma \vdash A = A' \qquad \Gamma \vdash a : A}{\Gamma \vdash a : A'}$$

The type of sets

We want to express that Vect is an N-indexed family of types. First try

$$\mathrm{Vect} : \mathrm{N} \to \mathbf{type}$$

But what is the type of $\mathrm{N} \to \mathbf{type}$? Is it a type? Unfortunately, $\mathbf{type}\ \mathbf{type}$ leads to a paradox - Girard's paradox.

We therefore need to distinguish between "small" and "large" types, and call the small types "sets". So we have a type of sets $\mathbf{Set}$. The distinction set vs type in type theory is a bit like the distinction set vs class in set theory.

The type of elements of a set

We have the rule

$$\frac{\Gamma \vdash A : \mathbf{Set}}{\Gamma \vdash \mathbf{El}\, A\ \mathbf{type}}$$

Usually, we suppress $\mathbf{El}$ and write A instead of $\mathbf{El}\, A$.

Sets and propositions are identified, so $\mathbf{Set}$ also plays the role of the type o in Church's higher order logic.

The type of Vect

For example, if we have

$$\mathrm{N} : \mathbf{Set}$$

then

$$\mathbf{El}\,\mathrm{N}\ \mathbf{type}$$

and

$$\mathbf{El}\,\mathrm{N} \to \mathbf{Set}\ \mathbf{type}$$

and we can give Vect the following type:

$$\mathrm{Vect} : \mathbf{El}\,\mathrm{N} \to \mathbf{Set}$$

The logical framework type of zip

Before we gave a very polymorphic type of zip

$$\mathrm{zip} : \mathrm{Vect}\, A\, n \to \mathrm{Vect}\, B\, n \to \mathrm{Vect}\, (A \times B)\, n$$

Now we can explicitly quantify over A, B and n:

$$\mathrm{zip} : (A, B : \mathbf{Set}) \to (n : \mathrm{N}) \to \mathrm{Vect}\, A\, n \to \mathrm{Vect}\, B\, n \to \mathrm{Vect}\, (A \times B)\, n$$

Similarly, for other operations, eg

$$\mathrm{Pair} : (A, B : \mathbf{Set}) \to A \to B \to A \times B$$

Remark: in both examples we have suppressed $\mathbf{El}$.

Martin-Löf's logical framework

Consists of rules for generating valid judgements concerning $(x : A) \to B$, $\mathbf{Set}$, and $\mathbf{El}\, A$. There are several such rules in addition to the typing rules and the type equality rule. For example, the rules of β- and η-conversion are now rules generating typed equality judgements.

Normal forms in the logical framework

There is a strong normalization theorem for Martin-Löf's logical framework. It is proved by an extension of Tait's method.

The set of normal terms t can be defined simultaneously with the set of neutral terms u:

$$\begin{array}{lcl} t & ::= & u \mid \lambda x.\, t \\ u & ::= & x \mid u\, t \end{array}$$

The set of normal types is defined as follows

$$T \quad ::= \quad \mathbf{Set} \mid \mathbf{El}\, t \mid (x : T) \to T$$

Type checking in the logical framework

One approach to type-checking in Martin-Löf's logical framework is to check terms and types à la Curry in normal form.

The type-checking problem for closed terms is the following: given a normal term t and a normal type expression T, decide whether $\vdash t : T$.

More generally, we need to do type-checking of open terms: if we also have a normal context expression $\Gamma ::= x_1 : T_1, \ldots x_n : T_n$, where all T_i are normal type expressions with $\mathrm{FV}(T_i) \subseteq \{x_1, \ldots, x_{i-1}\}$, check whether $\Gamma \vdash t : T$.

Type checking algorithm - sketch

- To check whether a neutral term $x\,t_1 \cdots t_n$ has type T in the context Γ, look for $x : (y_1 : T_1) \to \cdots \to (y_n : T_n) \to T'$ in Γ. If so, check whether $t_1 : T_1, \ldots, t_n : T_n[y_1 := t_1, \ldots, y_{n-1} := t_{n-1}]$ in Γ and whether $T = T'[y_1 := t_1, \ldots, y_n := t_n]$ (Note that the latter needs normalization.)

- To check whether a normal term $\lambda x.\, b$ has type $(x : T) \to T'$ in Γ, check whether b has type T' in the context $\Gamma, x : T$.

Checking types

Given a normal context Γ and a normal type expression T, check whether $\Gamma \vdash T\ \mathbf{type}$. This algorithm is immediate and calls the algorithm for type-checking terms in the case of checking $\Gamma \vdash \mathbf{El}\, t\ \mathbf{type}$.

There is also an algorithm for checking normal contexts. To check whether $\Gamma, x : T$ is a correct context, check whether Γ is a correct context, and whether T is a correct type in context Γ.

The typed quantifiers

$$\begin{aligned} \forall &: (A : \mathbf{Set}) \to (\mathbf{El}\, A \to \mathbf{Set}) \to \mathbf{Set} \\ \exists &: (A : \mathbf{Set}) \to (\mathbf{El}\, A \to \mathbf{Set}) \to \mathbf{Set} \end{aligned}$$

Compare the (polymorphic) higher-order quantifiers in HOL:

$$\begin{aligned} \forall &: (A \to o) \to o \\ \exists &: (A \to o) \to o \end{aligned}$$

Natural numbers

Formation rule:

$$\mathrm{N} : \mathbf{Set}$$

Introduction rules:

$$\begin{array}{rcl} 0 & : & \mathrm{N} \\ \mathrm{Succ} & : & \mathrm{N} \to \mathrm{N} \end{array}$$

Mathematical induction

Elimination rule:

$$\begin{aligned} \mathrm{R} \quad : \quad & (C : \mathrm{N} \to \mathbf{Set}) \to C\,0 \to ((x : \mathrm{N}) \to C\,x \to C\,(\mathrm{Succ}\,x)) \\ & \to (n : \mathrm{N}) \to C\,n \end{aligned}$$

Dependent elimination rule = rule for building proofs by mathematical induction = rule for typing functions from natural numbers where the target is a dependent type.

Equality rules for natural numbers

$$\begin{aligned} \Gamma \vdash \mathrm{R}\,C\,d\,e\,0 &= d : C\,0 \\ \Gamma \vdash \mathrm{R}\,C\,d\,e\,(\mathrm{Succ}\,n) &= e\,n\,(\mathrm{R}\,C\,d\,e\,n) : C\,(\mathrm{Succ}\,n) \end{aligned}$$

under appropriate assumptions on C, d, e, and n.

Sum-elimination

$$
\begin{aligned}
\mathrm{case} \quad : \quad & (A, B : \mathbf{Set}) \to (C : A + B \to \mathbf{Set}) \\
& \to ((x : A) \to C\,(\mathrm{Inl}\,x)) \\
& \to ((y : B) \to C\,(\mathrm{Inr}\,y)) \\
& \to (z : A + B) \to C\,z
\end{aligned}
$$

This rule has four uses, depending on whether A, B are thought of as sets or propositions, and whether C is thought of as a family of sets or a family of propositions (a predicate).

Equality in intuitionistic type theory

an unfinished story ...

Recall that we already have equality *judgements*

$$\Gamma \vdash a = a' : A$$

But we would also like to have equality *propositions*

$$\mathrm{Eq} : (A : \mathbf{Set}) \to A \to A \to \mathbf{Set}$$

How to define Eq?

Fundamental principles of intuitionistic type theory

There are three fundamental parts:

- the logical framework - basic rules for dependently typed lambda calculus
- inductive definitions of sets and families
- structural recursive definitions of functions (including families of sets)

Where does the definition of equality belong?

A recursive definition - Martin-Löf 1972

Computable equality on N

$$\mathrm{eqN} : \mathrm{N} \to \mathrm{N} \to \mathrm{Bool}$$

can be defined by the following equations:

$$\begin{aligned}
\mathrm{eqN}\,0\,0 &= \mathrm{True} \\
\mathrm{eqN}\,0\,(\mathrm{Succ}\,n) &= \mathrm{False} \\
\mathrm{eqN}\,(\mathrm{Succ}\,m)\,0 &= \mathrm{False} \\
\mathrm{eqN}\,(\mathrm{Succ}\,m)\,(\mathrm{Succ}\,n) &= \mathrm{eqN}\,m\,n
\end{aligned}$$

Define it in terms of R! (Use primitive recursion of higher type.)

Turning a boolean into a proposition

$$\mathrm{EqN}\, m\, n = \mathrm{T}\, (\mathrm{eqN}\, m\, n)$$

where

$$\mathrm{T} : \mathrm{Bool} \to \mathbf{Set}$$

is defined by case analysis

$$\begin{aligned} \mathrm{T}\, \mathrm{False} &= \emptyset \\ \mathrm{T}\, \mathrm{True} &= \mathbf{1} \end{aligned}$$

This is not covered by ordinary Bool-elimination, since the target is not a set but $\mathbf{Set}$ itself; it is a large type, not a small. We call this large Bool-elimination.

Inductively defined equality - Martin-Löf 1973

Equality is inductively generated by the reflexivity rule (Eq-introduction):

$$\mathrm{refl} : (A : \mathbf{Set}) \to (a : A) \to \mathrm{Eq}\, A\, a\, a$$

Eq-elimination is the rule of substitutivity of equality:

$$\mathrm{subst} : (A : \mathbf{Set}) \to (C : A \to \mathbf{Set}) \to (a, b : A) \to \mathrm{Eq}\, A\, a\, b \to C\, a \to C\, b$$

Equality reflection - Martin-Löf 1979

Turning a propositional equality into a judgemental equality:

$$\frac{\Gamma \vdash c : \mathrm{Eq}\, A\, a\, b}{\Gamma \vdash a = b : A}$$

This rule leads to *extensional* type theory with undecidable judgements and without the normalization property.

Without this rule we have *intensional* type theory with decidable judgements and the strong normalization property.

Inductive definitions in intuitionistic type theory

We have shown the rules for the natural numbers N.

There are similar rules for lists constructed by Nil and Cons and where the elimination rule is a typing rule for primitive list recursion and structural induction on lists.

There are also similar rules for binary trees of various kinds, and more generally of algebraic datatypes in general and even more generally for so called generalized inductive definitions. The precise general rules are outside the scope of these lectures.

We may also inductively define families of sets like Eq. We will soon see another example.

The four kinds of rules for inductive definitions

Formation rule gives the typing of the set former

Introduction rules give the types of the constructors

Elimination rule gives the type of a constant for defining functions by structural recursion or defining proofs by structural induction

Equality rules are recursion equations defining the constant in the elimination rule

Remark. In these slides we have only given a few examples. For N we gave all the rules however.

Lambda expressions as a set in type theory

As an example we consider the lambda expressions introduced by the following introduction rules given earlier:

$$\begin{aligned} \text{Apply} &: \text{Exp} \to \text{Exp} \to \text{Exp} \\ \text{Lambda} &: \text{Var} \to \text{Exp} \to \text{Exp} \\ \text{Var} &: \text{Var} \to \text{Exp} \end{aligned}$$

Exp-**elimination**

The elimination rule is definition (proof) by structural recursion (induction) on Exp:

$$\begin{aligned}
\mathrm{exprec} \quad : \quad & ((f : \mathrm{Exp}) \to (a : \mathrm{Exp}) \to C\, f \to C\, a \to C\, (\mathrm{Apply}\, f\, a)) \\
& \to ((x : \mathrm{Var}) \to (b : \mathrm{Exp}) \to C\, b \to C\, (\mathrm{Lambda}\, x\, b)) \\
& \to ((x : \mathrm{Var}) \to C\, (\mathrm{Var}\, x)) \\
& \to (z : \mathrm{Exp}) \to C\, z
\end{aligned}$$

The well-founded part of a relation

Starting with Exp we can formalize the metatheory of simply typed lambda calculus within intuitionistic type theory. This metatheory uses many concepts directly formalizable as inductively defined sets and families.

An example is the concept of strongly normalizing term wrt β-reduction. This can be formalized by using the more general concept of the *well-founded* or *accessible* part of a relation: a term is strongly normalizing iff it is in the accessible part of one-step β-reduction.

The well-founded part of a relation

This is an example of a *generalized* inductive definition of a *family* of sets. The formation rule is:

$$\mathrm{Acc} : (A : \mathbf{Set}) \to (R : A \to A \to \mathbf{Set}) \to A \to \mathbf{Set}$$

where $c : \mathrm{Acc}\, A\, R\, a$ means that c is a proof that a is in the well-founded part of the relation R on the set A.

Classically, an element is in the well-founded part of a relation iff there are no infinite descending R-chains from it.

Acc-introduction

Constructively, we give an inductive definition with the introduction rule:

$$\begin{aligned}\mathrm{AccIntro} \quad : \quad & (A : \mathbf{Set}) \to (R : A \to A \to \mathbf{Set}) \\ & \to (a : A) \\ & \to ((x : A) \to R\,a\,x \to \mathrm{Acc}\,A\,R\,x) \\ & \to \mathrm{Acc}\,A\,R\,a\end{aligned}$$

This is a *generalized* inductive definition since we have a (possibly infinite) indexed family of inductive premisses.