Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

http://www.infsec.ethz.ch/education/permanent/csmr/

Motivation and Background

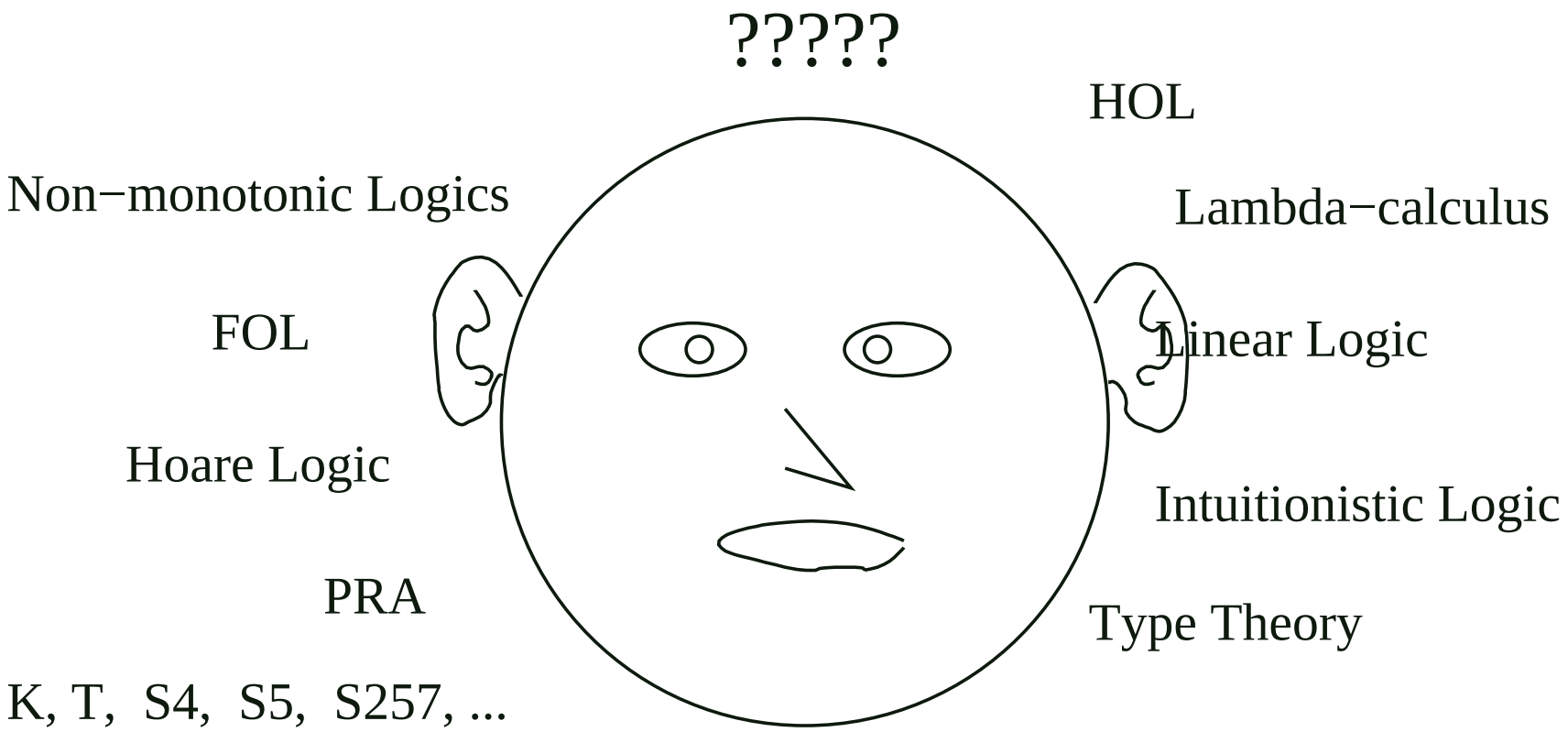
David Basin, Burkhart Wolff, and Jan-Georg Smaus

Overview

- We have studied reasoning in given theories

  Labs used predeveloped `.thy` files.
- How does one encode their own theories? Issues include:
  - Metalogic: formalism for formalizing theories
  - Pragmatics: how to use such a metalogic
- The next two lectures will examine:
  - Representing syntax using simple types
  - Representing proofs using dependent types
- We will be formal

What is the Problem?



Hilbert Presentations, Natural Deduction, Sequent Calculus, ...

Solutions?

- Implement individually

  +/− employment for thousands!

- Embed in a framework logic

  + Implement 'core' only once

  + Shared support for automation

  + Conceptual framework for exploring what a logic is

  +/− Meta-layer between user and logic

  − Makes assumptions about structure of logic

Overview — Encodings in Type Theory

- The λ-Calculus as programming language

$$f(x) = g(x, 3) \quad \rightsquigarrow \quad f = \lambda x.\, g\, x\, 3$$

- Simple types classify syntax (o = type of Propositions)

$$\begin{array}{lcll} \bot & \rightsquigarrow & False & \in & o \\ \wedge & \rightsquigarrow & And & \in & o \to o \to o \\ \forall & \rightsquigarrow & All & \in & (i \to o) \to o \end{array}$$

- Dependent types classify rules: pr:$o \to Type$

$$\frac{A \wedge B}{A} \quad \rightsquigarrow \quad andel \in \Pi x : o.\, \Pi y : o.\, pr(and\ x\ y) \to pr(x)$$

Overview (cont.)

- Judgments as Types (syntax in this lecture)

$$\begin{array}{l}\vdots\\ P\\ \vdots\\ \vdash \phi\end{array} \quad \rightsquigarrow \quad \ulcorner P \urcorner \in pr(\ulcorner \phi \urcorner)$$

  - Models syntax: $\phi \in Prop$ *iff* $\ulcorner \phi \urcorner \in o$
  - Models provability: $\vdash_L \phi$ *iff* $\vdash_{TT} pr(\ulcorner \phi \urcorner)$
  - Models proofs: P *iff* $\ulcorner P \urcorner$
- Correctness of encodings: faithfulness and adequacy

  Requires study of metatheory of metalogic

Simple Type Theory $\lambda^{\rightarrow}$

David Basin, Burkhart Wolff, and Jan-Georg Smaus

The Untyped λ-Calculus

Untyped λ-Calculus

	Conventional		λ-Calculus
Declaration:	$f(x) = g(x, 3)$	$\rightsquigarrow$	$f = \lambda x.\, g\, x\, 3$
Application:	$f(5)$		$(\lambda x.\, g\, x\, 3)(5)$
Reduction:	$g(x, 3)[x \leftarrow 5]$		$(g\, x\, 3)[x \leftarrow 5]$
Result:	$g(5, 3)$		$g\, 5\, 3$

We will use the (typed) λ-calculus to represent the syntax and proofs of deductive systems.

Syntax

$(x \in Var,\ c \in Const)$

$$e \ ::= \ x \ \mid \ c \ \mid \ (ee) \ \mid \ (\lambda x.\, e)$$

The objects generated by this grammar are called λ-terms or simply terms.

Conventions: iterated λ & left-associated application

$$\begin{aligned}(\lambda x.\,(\lambda y.\,(\lambda z.\,((xz)(yz))))) &\equiv (\lambda xyz.\,((xz)(yz))) \\ &\equiv \lambda xyz.\, xz(yz)\end{aligned}$$

We may take further syntactic liberties, e.g., $\lambda x.\, x + 5$

Substitution

- Reduction based on substitutions

$$(\lambda x.\, g\, x\, 3)(5) = (g\, x\, 3)[x \leftarrow 5] = g\, 5\, 3$$

- Must respect free and bound variables,

$$(\lambda x.\, x(\lambda x.\, xy))(e) = ((x(\lambda x.\, xy))[x \leftarrow e] = e(\lambda x.\, xy)$$

- Same problems as with quantifiers

$$\frac{\forall x.\, (P(x) \wedge \exists x.\, Q(x,y))}{P(e) \wedge \exists x.\, Q(x,y)}\ \forall\text{-}E \qquad \frac{\forall x.\, (P(x) \wedge \exists y.\, Q(x,y))}{P(y) \wedge \exists z.\, Q(y,z)}\ \forall\text{-}E$$

Bound, Free, Binding Occurrences

Recall the notions of bound, free, and binding occurrences of variables in a term. Same thing here:

λ-calculus	FOL
$FV(x) := \{x\}$	$= FV(x)$
$FV(c) := \emptyset$	$= FV(c)$
$FV(MN) := FV(M) \cup FV(N)$	$= FV(M \wedge N)$
$FV(\lambda x.\, M) := FV(M) \setminus \{x\}$	$= FV(\forall x.\, M)$

Example: $FV(xy(\lambda yz.\, xyz)) = \{x, y\}$

Definition of Substitution

$M[x \leftarrow N]$ means substitute N for x in M

1. $x[x \leftarrow N] = N$
2. $a[x \leftarrow N] = a$ if a is a constant or variable other than x
3. $(PQ)[x \leftarrow N] = (P[x \leftarrow N]Q[x \leftarrow N])$
4. $(\lambda x.\, P)[x \leftarrow N] = \lambda x.\, P$
5. $(\lambda y.\, P)[x \leftarrow N] = \lambda y.\, P[x \leftarrow N]$ if $y \neq x$ and $y \notin FV(N)$
6. $(\lambda y.\, P)[x \leftarrow N] = \lambda z.\, P[y \leftarrow z][x \leftarrow N]$ if $y \neq x$ and $y \in FV(N)$ where z is a variable such that $z \notin FV(NP)$

Cases similar to those for quantifiers: λ binding is 'generic'.

Substitution: Example

$$
\begin{aligned}
(x(\lambda x.\, xy))[x \leftarrow \lambda z.\, z] &\overset{3}{=} x[x \leftarrow \lambda z.\, z](\lambda x.\, xy)[x \leftarrow \lambda z.\, z] \\
&\overset{1,4}{=} (\lambda z.\, z)\lambda x.\, xy
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.\, xy)[y \leftarrow x] &\overset{6}{=} \lambda z.\, ((xy)[x \leftarrow z][y \leftarrow x]) \\
&\overset{3,1,2}{=} \lambda z.\, (zy[y \leftarrow x]) \\
&\overset{3,2,1}{=} \lambda z.\, zx
\end{aligned}
$$

In the last example, clause 6 avoids capture, i.e., $\lambda x.\, xx$.

Reduction: Intuition

Reduction is the notion of "computing", or "evaluation", in the λ-calculus.

$$f\ x = x + 5 \ \rightsquigarrow\ f = \lambda x.\, x + 5$$
$$f\ 3 = 3 + 5 \ \rightsquigarrow\ (\lambda x.\, x + 5)(3) \rightarrow_\beta (x + 5)[x \leftarrow 3] = 3 + 5$$

β-reduction replaces a parameter by an argument.

This should propagate into contexts, e.g.

$$\lambda x.((\underline{(\lambda x.\, x + 5)(3)}) \rightarrow_\beta \lambda x.(3 + 5).$$

Reduction: Definition

- β-reduction: $(\lambda x.M)N \to_\beta M[x \leftarrow N]$
- Rules for contraction (of redices) in contexts:

$$\frac{M \to_\beta M'}{NM \to_\beta NM'} \qquad \frac{M \to_\beta M'}{MN \to_\beta M'N} \qquad \frac{M \to_\beta M'}{\lambda z.M \to_\beta \lambda z.M'}$$

- Reduction is reflexive-transitive closure

$$\frac{M \to_\beta N}{M \to_\beta^* N} \qquad \frac{}{M \to_\beta^* M} \qquad \frac{M \to_\beta^* N \quad N \to_\beta^* P}{M \to_\beta^* P}$$

- A term without redices is in β-normal form.

Reduction: Examples

$$\underline{(\lambda x.\, \lambda y.\, g\, x\, y)a\, b} \to_\beta \underline{(\lambda y.\, (g\, a\, y))b} \to_\beta g\, a\, b$$

$$\text{So } (\lambda x.\, \lambda y.\, g\, x\, y)a\, b \to_\beta^* g\, a\, b$$

Shows Currying

$$\underline{(\lambda x.\, xx)(\lambda x.\, xx)} \to_\beta \underline{(\lambda x.\, xx)(\lambda x.\, xx)} \to_\beta \ldots$$

Shows divergence

$$\text{But } \underline{(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx))} \to_\beta \lambda y.\, y$$

Conversion

- β-conversion: "symmetric closure" of β-reduction

$$\frac{M \to_\beta^* N}{M =_\beta N} \qquad \frac{M =_\beta N}{N =_\beta M}$$

- α-conversion: bound variable renaming (usually implicitly)

$$\lambda x.M =_\alpha \lambda z.M[x \leftarrow z] \quad \textit{where } z \notin FV(M)$$

- η-conversion: for normal-form analysis

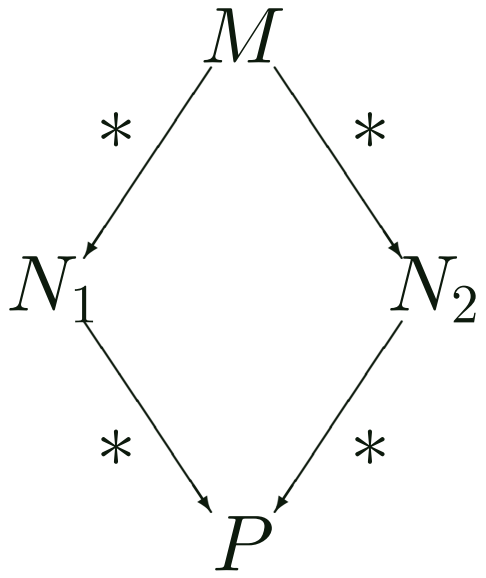
$$M =_\eta \lambda x.\,(Mx) \quad \textit{if } x \notin FV(M)$$

reflects an extensional equality on functions.

λ-Calculus Meta-Properties

Confluence (equivalently, Church-Rosser): reduction is order-independent.

For all M, N_1, N_2, if $M \rightarrow_\beta^* N_1$ and $M \rightarrow_\beta^* N_2$, then exists a P where $N_1 \rightarrow_\beta^* P$ and $N_2 \rightarrow_\beta^* P$.



Uniqueness of Normal Forms

Corollary of the Church-Rosser property:

If $M \to_\beta^* N_1$ and $M \to_\beta^* N_2$ where N_1 and N_2 in normal form, then $N_1 =_\alpha N_2$.

$$(\lambda xy.\, y)(\underline{(\lambda x.\, xx)a}) \to_\beta \underline{(\lambda xy.\, y)(aa)} \to_\beta \lambda y.\, y$$

$$\underline{(\lambda xy.\, y)((\lambda x.\, xx)a)} \to_\beta \lambda y.\, y$$

N.B. As a computational formalism, the λ-calculus can represent all computable functions.

The Simply Typed λ-Calculus ($\lambda^{\rightarrow}$)

Simply Typed λ-Calculus — Syntax

- Syntax for types ($\mathcal{B}$ a set of base types, $T \in \mathcal{B}$)

$$\tau ::= T \mid \tau \rightarrow \tau$$

Examples: $\mathcal{N}$, $\mathcal{N} \rightarrow \mathcal{N}$, $(\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N}$, $\mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{N}$

- Syntax for (raw) terms: λ-calculus augmented with types

$$e ::= x \mid c \mid (ee) \mid (\lambda x^{\tau}.\, e)$$

($x \in Var$, $c \in Const$)

Signatures and Contexts

Generally (in various logic-related formalisms) a signature defines the "fixed" symbols of a language, and a context defines the "variable" symbols of a language. In $\lambda^{\rightarrow}$,

- a signature Σ is a sequence $(c \in Const)$

$$\Sigma ::= \langle\rangle \mid \Sigma, c : \tau$$

- a context Γ is a sequence $(x \in Var)$

$$\Gamma ::= \langle\rangle \mid \Gamma, x : \tau$$

What's the difference to signatures you have seen so far?

Type Assignment Calculus

We now define type judgements: "a term has a type" or "a term is of a type". Generally this depends on a signature Σ and a context Γ. For example

$$\Gamma \vdash_\Sigma c\ x : \sigma$$

where $\Sigma = x : \tau$ and $\Gamma = c : \tau \rightarrow \sigma$.

We usually leave Σ implicit and write $\vdash$ instead of $\vdash_\Sigma$.

If Γ is empty it is omitted.

Type Assignment Calculus: Rules

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau}\ \textit{assum} \qquad \Gamma, x : \tau, \Delta \vdash x : \tau \quad \textit{hyp}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash e e' : \tau}\ \textit{app} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^{\sigma}.\, e\ : \sigma \rightarrow \tau}\ \text{abs}$$

Note analogy to minimal logic over $\rightarrow$.

β-reduction defined as before, has subject reduction property and is strongly normalizing.

Example 1

$$\dfrac{\dfrac{\dfrac{}{x : \sigma,\ y : \tau \vdash x : \sigma}\,\mathit{hyp}}{x : \sigma \vdash \lambda y^{\tau}.\, x : \tau \longrightarrow \sigma}\,\text{abs}}{\vdash \lambda x^{\sigma}.\, \lambda y^{\tau}.\, x : \sigma \longrightarrow (\tau \longrightarrow \sigma)}\,\text{abs}$$

Note the use of schematic types!

Also note that applications of *hyp* are usually not explicitly marked in proof.

Example 2

$$\Gamma = f : \sigma \rightarrow \sigma \rightarrow \tau, x : \sigma$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f\, x : \sigma \rightarrow \tau}\, app \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\, x\, x : \tau}\, app}{f : \sigma \rightarrow \sigma \rightarrow \tau \vdash \lambda x^{\sigma}.\, f\, x\, x : \sigma \rightarrow \tau}\, abs}{\vdash \lambda f^{\sigma \rightarrow \sigma \rightarrow \tau}.\, \lambda x^{\sigma}.\, f\, x\, x : (\sigma \rightarrow \sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau}\, abs$$

Example 3

$$\begin{array}{rcl} \Sigma & = & f : \sigma \rightarrow \sigma \rightarrow \tau \\ \Gamma & = & x : \sigma \end{array}$$

$$\dfrac{\dfrac{\dfrac{f : \sigma \rightarrow \sigma \rightarrow \tau \in \Sigma}{\Gamma \vdash f : \sigma \rightarrow \sigma \rightarrow \tau}\,\mathit{assum} \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\, x : \sigma \rightarrow \tau}\,\mathit{app} \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f\, x\, x : \tau}\,\mathit{app}$$

Note that this time, f is a constant.

We will often suppress applications of *assum*.

Example 4: First-Order Syntax within $\lambda^{\rightarrow}$

- Propositional logic

$$P ::= x \mid \neg P \mid P \wedge P \mid P \rightarrow P \ldots$$

- Programming languages/algebraic specification

datatype *Prop* = *VarInject* of *Variable* | *not* of *Prop*
| *and* of *Prop**Prop | *imp* of *Prop**Prop

- $\lambda^{\rightarrow}$ approach
  - Type declarations for context $\mathcal{B} = \{o\}$
  - Signature types constants:

$$\Sigma = \{not : o \rightarrow o, and : o \rightarrow o \rightarrow o, imp : o \rightarrow o \rightarrow o\}$$

- Context types propositional variables

Example 4: First-Order Syntax within $\lambda^{\rightarrow}$(cont.)

- Example: $a : o \vdash imp(not\, a)a : o$

$$\dfrac{\dfrac{\dfrac{a : o \vdash imp : o \rightarrow o \rightarrow o \qquad \dfrac{a : o \vdash not : o \rightarrow o \quad a : o \vdash a : o}{a : o \vdash not\, a : o}}{a : o \vdash imp(not\, a) : o \rightarrow o} \qquad a : o \vdash a}{a : o \vdash imp(not\, a)a : o}$$

- Non example: $a : o \vdash not(imp\, a)a : o$

$$\dfrac{\dfrac{}{a : o \vdash not : o \rightarrow o} \qquad \dfrac{\dfrac{}{a : o \vdash imp : o \rightarrow o \rightarrow o} \quad \dfrac{}{a : o, \vdash a : o}}{a : o \vdash imp\, a : o \rightarrow o}}{???}$$

No proof possible! (requires analysis of normal forms)

Example 5: Encoding Binding in FOL

- This also works for languages with quantifiers !

$$\begin{array}{lll} \textit{Terms } T & ::= & x \mid 0 \mid sT \mid T+T \mid T \times T \\ \textit{Formulae } F & ::= & T = T \mid \neg F \mid F \wedge F \mid \ldots \\ & & \forall x.\, F \mid \exists x.\, F \end{array}$$

- Type declarations for context $\mathcal{B} = \{i, o\}$
- Signature $\Sigma = \Sigma_T \cup \Sigma_P \cup \Sigma_Q$:

$$\begin{array}{lcl} \Sigma_T & = & \{0 : i,\ s : i \to i,\ plus : i \to i \to i,\ times : i \to i \to i\} \\ \Sigma_P & = & \{eq : i \to i \to o,\ not : o \to o,\ and : o \to o \to o,\ \ldots\} \\ \Sigma_Q & = & \{all : (i \to o) \to o,\ exists : (i \to o) \to o\} \end{array}$$

Example 5: Encoding Binding in FOL (cont)

- Faithfulness/adequacy: terms and formulae represented by (canonical) members of i and o

$$
\begin{array}{lcl}
0 + s0 & \Leftrightarrow & plus\,0\,(s0) \\
\forall x.\, x = x & \Leftrightarrow & all(\lambda x^i.\, eq\, x\, x) \\
\forall x.\, \exists y.\, \neg(x + x = y) & \Leftrightarrow & all(\lambda x^i.\, exists(\lambda y^i.\, not\,(eq\,(plus\, x
\end{array}
$$

- Example derivation

$$\dfrac{\dfrac{\dfrac{x : i \vdash eq : i \to i \to o \quad x : i \vdash x : i}{x : i \vdash eq\, x : i \to o} \qquad x : i \vdash x :}{\dfrac{x : i \vdash eq\, x\, x : o}{\vdash \lambda x^i.\, eq\, x\, x : i \to o}}}{\vdash all : (i \to o) \to o \qquad \vdash \lambda x^i.\, eq\, x\, x : i \to o}$$

$$\vdash all(\lambda x^i.\, eq\, x\, x) : o$$

More Detailed Explanations

$3 + 5 = 8$?

As you might guess, the formalism of the λ-calculus is not directly related to usual arithmetic and so it is not built into this formalism that $3 + 5$ should evaluate to 8. However, it may be a reasonable choice, depending on the context, to extend the λ-calculus is this way, but this is not our concern at the moment.

Var and $Const$

Similarly as for first-order logic, a language of the untyped λ-calculus is characterized by giving a set of variables and a set of constants.

One can think of $Const$ as a signature.

Note that $Const$ could be empty.

Note also that the word constant has a different meaning in the λ-calculus from that of first-order logic. In both formalisms, constants are just symbols.

In first-order logic, a constant is a special case of a function symbol, namely a function symbol of arity 0.

In the λ-calculus, one does not speak of function symbols. In the untyped λ-calculus, any λ-term (including a constant) can be applied to another term, and so any λ-term can be called a "unary function". A constant being applied to a term is something which would contradict

the intuition about constants in first-order logic. So for the λ-calculus, think of constant as opposed to a variable, an application, or an abstraction.

How do We Call those Terms?

A λ-term can either be

- a variable (case x), or
- a constant (case c), or
- an application of a λ-term to another λ-term (case (ee)), or
- an abstraction over a variable x (case $(\lambda x.\, e)$).

Backus-Naur Form

A notation like

$$
\begin{aligned}
e &::= x \mid c \mid (ee) \mid (\lambda x.\, e) \\
\tau &::= T \mid \tau \to \tau \\
e &::= x \mid c \mid (ee) \mid (\lambda x^{\tau}.\, e) \\
P &::= x \mid \neg P \mid P \wedge P \mid P \to P \ldots
\end{aligned}
$$

for specifying syntax is called Backus-Naur form (BNF) for expressing grammars. For example, the first BNF-clause reads: a λ-term can be
a variable, or
a constant, or
a λ-term applied to a λ-term, or
a λ-abstraction, which is a λ-term of the form $\lambda x.\, e$, where e is a λ-term.

The BNF is a very common formalism for specifying syntax, e.g., of

programming languages. See here or here.

(λ-)Terms

So just like first-order logic, the λ-calculus has a syntactic category called terms. Bit the word "term" has a different meaning for the λ-calculus than for first-order logic, and so one can say λ-term for emphasis.

Note that at this stage, we have no syntactic category called "formula" for the λ-calculus.

λ-Calculus: Notational Conventions

We write $\lambda x_1 x_2 \ldots x_n.e$ instead of $\lambda x_1.(\lambda x_2.(\ldots e)\ldots)$.

$e_1\ e_2 \ldots e_n$ is equivalent to $(\ldots(e_1\ e_2)\ldots e_n)\ldots$, not $(e_1(e_2 \ldots e_n)\ldots)$.

Note that this is in contrast to the associativity of logical operators.

There are some good reasons for these conventions.

Infix Notation

Strictly speaking, $\lambda x.\, x + 5$ does not adhere to the definition of syntax of λ-terms, at least if we parse it in the usual way: $+$ is an infix constant applied to arguments x and 5.

If we parse $x + 5$ as $((x+)5)$, i.e., x applied to (the constant) $+$, and the resulting term applied to (the constant) 5, then $\lambda x.\, x + 5$ would indeed adhere to the definition of syntax of λ-terms, but of course, this is pathological and not intended here.

It is convenient to allow for extensions of the syntax of λ-terms, allowing for:

- application to several arguments rather than just one;
- infix notation.

Such an extension is inessential for the expressive power of the

λ-calculus. Instead of having a binary infix constant $+$ and writing $\lambda x.\, x + 5$, we could have a constant $plus$ according to the original syntax and write $\lambda x.\, ((plus\, x)\, 5)$ (i.e., write $+$ in a Curryed way).

Reduction

Reduction is the notion of "computing", or "evaluation", in the λ-calculus.

Notations for Substitutions

Here we use the notation $e[x \leftarrow t]$ for the term obtained from e by replacing x with t. There is also the notation $e[t/x]$, and confusingly, also $e[x/t]$. We will attempt to be consistent within this course, but be aware that you may find such different notations in the literature.

λ Binding Is 'Generic'

Recall the definition of substitution for first-order logic.

We observe that binding and substitution are some very general concepts. So far, we have seen four binding operators: $\exists$, $\forall$ and λ, and set comprehensions. The λ operator is the most generic of those operators, in that it does not have a fixed meaning hard-wired into it in the way that the quantifiers do. In fact, it is possible to have it as the only operator on the level of the metalogic. We will see this later.

Avoiding Capture

If it wasn't for clause 6, i.e., if we applied clause 5 ignoring the requirement on freeness, then $(\lambda x.\, xy)[y \leftarrow x]$ would be $\lambda x.\, xx$.

Parameters and Arguments

In the λ-term $(\lambda x.M)N$, we say that N is an argument (and the function $\lambda x.M$ is applied to this argument), and every occurrence of x in M is a parameter (we say this because x is bound by the λ).

This terminology may be familiar to you if you have experience in functional programming, but actually, it is also used in the context of function and procedure declarations in imperative programming.

Propagation into Contexts

In

$$\lambda x.(\underline{(\lambda x.\, x + 5)(3)}),$$

the underlined part is a subterm occurring in a context. β-reduction should be applicable to this subterm.

Like a Proof System

As you see, β-reduction is defined using rules (two of them being axioms, the rest proper rules) in the same way that we have defined proof systems for logic before. Note that we wrote the first axiom defining β-reduction without a horizontal bar.

Redex

In a λ-term, a subterm of the form $(\lambda x.\, M)N$ is called a redex (plural redices). It is a subterm to which β-reduction can be applied.

Currying

You may be familiar with functions taking several arguments, or equivalently, a tuple of arguments, rather than just one argument.

In the λ-calculus, but also in functional programming, it is common not to have tuples and instead use a technique called Currying (Schönfinkeln in German). So instead of writing $g(a, b)$, we write $g\,a\,b$, which is read as follows: g is a function which takes an argument a and returns a function which then takes an argument b.

Recall that application associates to the left, so $g\,a\,b$ is read $(g\,a)\,b$.

Currying will become even clearer once we introduce the typed λ-calculus.

Divergence

We say that a β-reduction sequence diverges if it is infinite.

Note that for $(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx))$, there is a finite β-reduction sequence

$$(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta \lambda y.\, y$$

but there is also a diverging sequence

$$(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta (\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \to_\beta \ldots$$

α-Conversion

α-conversion is usually applied implicitly, i.e., without making it an explicit step. So for example, one would simply write:

$$\lambda z.\, z =_\beta (\lambda x.\, xx)\lambda r.\, r$$

η-Conversion

η-conversion is defined as

$$M =_\eta \lambda x.\,(M\,x) \quad \textit{if } x \notin FV(M)$$

It is needed for reasoning about normal forms.

$$g\,x =_\eta \lambda y.\,g\,x\,y \quad \text{reflects} \quad g\,x\,b =_\beta (\lambda y.\,g\,x\,y)b$$

More specifically: if we did not have the η-conversion rule, then $g\,x$ and $\lambda y.\,g\,x\,y$ would not be "equivalent" up to conversion. But that seems unreasonable, because they behave the same way when applied to b. Applied to b, both terms can be converted to $g\,x\,b$. This is why it is reasonable to introduce a rule such that $g\,x$ and $\lambda y.\,g\,x\,y$ are "equivalent" up to conversion.

Confluence and Church-Rosser

A reduction $\rightarrow$ is called confluent if

for all M, N_1, N_2, if $M \rightarrow^* N_1$ and $M \rightarrow^* N_2$, then there exists a P where $N_1 \rightarrow^* P$ and $N_2 \rightarrow^* P$.

A reduction is called Church-Rosser if

for all N_1, N_2, if $N_1 \overset{*}{\leftrightarrow} N_2$, then there exists a P where $N_1 \rightarrow^* P$ and $N_2 \rightarrow^* P$.

Here, $\leftarrow := (\rightarrow)^{-1}$ is the inverse of $\rightarrow$, and $\leftrightarrow := \leftarrow \cup \rightarrow$ is the symmetric closure of $\rightarrow$, and $\overset{*}{\leftrightarrow} := (\leftrightarrow)^*$ is the reflexive transitive symmetric closure of $\rightarrow$.

So for example, if we have

$$M_1 \to M_2 \to M_3 \to M_4 \leftarrow M_5 \leftarrow M_6 \to M_7 \leftarrow M_8 \leftarrow M_9$$

then we would write $M_1 \overset{*}{\leftrightarrow} M_9$.

Confluence is equivalent to the Church-Rosser property [BN98, page 10].

λ-Calculus Metaproperties

By metaproperties, we mean properties about reduction and conversion sequences in general.

Turing Completeness

The untyped λ-calculus is Turing complete. This is usually shown not by mimicking a Turing machine in the λ-calculus, but rather by exploiting the fact that the Turing computable functions are the same class as the μ-recursive functions. In a lecture on theory of computation, you have probably learned that the μ-recursive functions are obtained from the primitive recursive functions by so-called unbounded minimalization, while the primitive recursive functions are built from the 0-place zero function, projection functions and the successor function using composition and primitive recursion [LP81].

The proof that the untyped λ-calculus can compute all μ-recursive functions is thus based on showing that each of the mentioned ingredients can be encoded in the untyped λ-calculus. While we are not going to study this, one crucial point is that it should be possible to

encode the natural numbers and the arithmetic operations in the untyped λ-calculus.

Term Language

We also say that we have defined a term language. A particular language is given by a signature, although for the untyped λ-calculus this is simply the set of constants $Const$.

Type Language

We can say that we define a type language, i.e., a language consisting of types. A particular type language is characterized by a giving a set of base types $\mathcal{B}$. One might also call $\mathcal{B}$ a type signature.

A typical example of a set of base types would be $\{\mathcal{N}, bool\}$, where $\mathcal{N}$ represents the natural numbers and $bool$ the Boolean values $\bot$ and $\top$.

All that matters is that $\mathcal{B}$ is some fixed set "defined by the user".

Types: Intuition

The type $\mathcal{N} \to \mathcal{N}$ is the type of a function that takes a natural number and returns a natural number.

The type $(\mathcal{N} \to \mathcal{N}) \to \mathcal{N}$ is the type of a function that takes a function, which takes a natural number and returns a natural number, and returns a natural number.

Types Are Right-Associative

To save parentheses, we use the following convention: types associate to the right, so $\mathcal{N} \to \mathcal{N} \to \mathcal{N}$ stands for $\mathcal{N} \to (\mathcal{N} \to \mathcal{N})$.

Recall that application associates to the left. This may seem confusing at first, but actually, it turns out that the two conventions concerning associativity fit together very neatly.

Raw Terms

In the context of typed versions of the λ-calculus, raw terms are terms built ignoring any typing conditions. So raw terms are simply terms as defined for the untyped λ-calculus, possibly augmented with type superscripts.

Augmenting with Types

So far, this is just syntax!

The notation $(\lambda x^{\tau}.\, e)$ simply specifies that binding occurrences of variables in simple type theory are tagged with a superscript, where the use of the letter τ makes it clear (in this particular context) that the superscript must be some type, defined by the grammar we just gave.

Var and $Const$

Var and $Const$ are the sets of variables and constants, respectively, as for the untyped λ-calculus.

Sequences

A sequence is a collection of objects which differs from sets in that a sequence contains the objects in a certain order, and there can be multiple occurrences of an object.

We write a sequence containing the objects $o_1, \ldots, o_n$ as $\langle o_1, \ldots, o_n \rangle$, or sometimes simply $o_1, \ldots, o_n$.

If Ω is the sequence $o_1, \ldots, o_n$, then we write Ω, o for the sequence $\langle o_1, \ldots, o_n, o \rangle$ and o, Ω for the sequence $\langle o, o_1, \ldots, o_n \rangle$.

A empty sequence is denoted by $\langle\,\rangle$.

Type Binding

We call an expression of the form $x : \tau$ or $c : \tau$ a type binding.

The use of the letter τ makes it clear (in this particular context) that the superscript must be some type, defined by the grammar we just gave.

Signatures in Various Formalisms

For propositional logic, we did not use the notion of signature, although we mentioned that strictly speaking, there is not just the language of propositional logic, but rather a language of propositional logic which depends on the choice of the variables.

In first-order logic, a signature was a pair $(\mathcal{F}, \mathcal{P})$ defining the function and predicate symbols, although strictly speaking, the signature should also specify the arities of the symbols in some way. Recall that we did not bother to fix a precise technical way of specifying those arities. We were content with saying that they are specified in "some unambiguous way".

In sorted logic, the signature must also specify the sorts of all symbols. But we did not study sorted logic in any detail.

In the untyped λ-calculus, the signature is simply the set of constants.

Summarizing, we have not been very precise about the notion of a

signature so far, since technically speaking, it was not strictly necessary to have this notion.

For $\lambda^{\rightarrow}$, the rules for "legal" terms become more tricky, and it is important to be formal about signatures.

In $\lambda^{\rightarrow}$, a signature associates a type with each constant symbol by writing $c : \tau$.

Usually, we will assume that $Const$ is clear from the context, and that Σ contains an expression of the form $c : \tau$ for each $c \in Const$, and in fact, that Σ is clear from the context as well. Since Σ contains an expression of the form $c : \tau$ for each $c \in Const$, it is redundant to give $Const$ explicitly. It is sufficient to give Σ.

Type Judgement

The expression

$$\Gamma \vdash_{\Sigma} c\, x : \sigma$$

is called a type judgement. It says that given the signature $\Sigma = x : \tau$ and the context $\Gamma = c : \tau \to \sigma$, the term

$c\, x$ has type σ or

$c\, x$ is of type σ or

$c\, x$ is assigned type σ.

Recall that you have seen other judgements before.

$\in$ for Sequences?

Recall that Σ is a sequence. By abuse of notation, we sometimes identify this sequence with a set and allow ourselves to write $c : \tau \in \Sigma$.

We may also write $\Sigma \subseteq \Sigma'$ meaning that $c : \tau \in \Sigma$ implies $c : \tau \in \Sigma'$.

System of Rules

Type assignment is defined as a system of rules for deriving type judgements, in the same way that we have defined derivability judgements for logics, and β-reduction for the untyped λ-calculus.

Minimal Logic over $\longrightarrow$

Recall the sequent rules of the $\longrightarrow / \wedge$ fragment of propositional logic. Consider now only the $\longrightarrow$ fragment. We call this fragment minimal logic over $\longrightarrow$.

If you take the rule

$$\Gamma, x : \tau, \Delta \vdash x : \tau \quad \textit{hyp}$$

of $\lambda^{\rightarrow}$ and throw away the terms (so you keep only the types), you obtain essentially the rule for assumptions

$$\Gamma \vdash A \quad (\text{where } A \in \Gamma)$$

of propositional logic.

Likewise, if you do the same with the rule

$$\frac{\Gamma \vdash e : \sigma \to \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau}\ \textit{app}$$

of $\lambda^{\to}$, you obtain essentially the rule

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \to\text{-}E$$

of propositional logic.

Finally, if you do the same with the rule

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x^{\sigma}.\, e\ : \sigma \to \tau}\ \textit{abs}$$

of $\lambda^{\to}$, you obtain essentially the rule

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \to B} \to\text{-}I$$

of propositional logic.

Note that in this setting, there is no analogous propositional logic rule for

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \textit{ assum}$$

So for the moment, we can observe a close analogy between $\lambda^{\to}$, for Σ being empty, and the $\to$ fragment of propositional logic, which is also called minimal logic over $\to$.

Such an analogy between a type theory (of which $\lambda^{\to}$ is an example) and a logic is referred to in the literature as Curry-Howard isomorphism. One

also speaks of propositions as types [GLT89]. The isomorphism is so fundamental that it is common to characterize type theories by the logic they represent, so for example, one might say:

$\lambda^{\rightarrow}$ is the type theory of minimal logic over $\rightarrow$.

Subject Reduction

Subject reduction is the following property: reduction does not change the type of a term, so if $\vdash_\Sigma M : \tau$ and $M \rightarrow_\beta N$, then $\vdash_\Sigma N : \tau$.

(Strongly) Normalizing β-Reduction

The simply-typed λ-calculus, unlike the untyped λ-calculus, is normalizing, that is to say, every term has a normal form. Even more, it is strongly normalizing, that is, this normal form is reached regardless of the reduction order.

An Alternative for *hyp*

One could also formulate *hyp* as follows:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathit{hyp}$$

That would be in close analogy to LF, a system not tretaed here.

Schematic Types

In this example, you may regard σ and τ as base types (this would require that $\sigma, \tau \in \mathcal{B}$), but in fact, it is more natural to regard them as metavariables standing for arbitrary types. Whatever types you substitute for σ and τ, you obtain a derivation of a type judgement.

This is in analogy to schematic derivations in a logic.

Note also that Σ is irrelevant for the example and hence arbitrary.

Constants vs. Variables

In Example 3, we have $f : \sigma \to \sigma \to \tau \in \Sigma$, and so f is a constant.

In Example 2, we have $f : \sigma \to \sigma \to \tau \in \Gamma$, and so f is a variable.

Looking at the different derivations of the type judgement $\Gamma \vdash f\, x\, x : \tau$ in Examples 2 and 3, you may find that they are very similar, and you may wonder: What is the point? Why do we distinguish between constants and variables?

In fact, one could simulate constants by variables. When setting up a type theory or programming language, there are choices to be made about whether there should be a distinction between variables and constants, and what it should look like. There is a famous epigram by Alan Perlis:

> One man's constant is another man's variable.

For our purposes, it is much clearer conceptually to make the distinction. For example, if we want to introduce the natural numbers in our $\lambda^{\rightarrow}$ language, then it is natural that there should be constants $1, 2, \ldots$ denoting the numbers. If $1, 2, \ldots$ were variables, then we could write strange expressions like $\lambda 2^{\mathcal{N}\rightarrow\mathcal{N}}. y$, so we could use 2 as a variable of type $\mathcal{N} \rightarrow \mathcal{N}$.

(Parametric) Polymorphism

In functional programming, you will come across functions that operate uniformly on many different types. For example, a function $append$ for concatenating two lists works the same way on integer lists and on character lists. Such functions are called polymorphic.

More precisely, this kind of polymorphism, where a function does exactly the same thing regardless of the type instance, is called parametric polymorphism, as opposed to ad-hoc polymorphism.

In a type system with polymorphism, the notion of base type (which is just a type constant, i.e., one symbol) is generalized to a type constructor with an arity ≥ 0. A type constructor of arity n applied to n types is then a type. For example, there might be a type constructor $list$ of arity 1, and int of arity 0. Then, $int\ list$ is a type.

Note that application of a type constructor to a type is written in postfix

notation, unlike any notation for function application we have seen. However, other conventions exist, even within Isabelle.

See [Pau96, Tho95, Tho99] for details on the polymorphic type systems of functional programming languages.

Ad-hoc Polymorphism

Ad-hoc polymorphism, also called overloading, refers to functions that do different (although usually similar) things on different types. For example, a function $\leq$ may be defined as $'a' \leq' b' \ldots$ on characters and $1 \leq 2 \ldots$ on integers. In this case, the symbol $\leq$ must be declared and defined separately for each type.

This is in contrast to parametric pomorphism, but also somewhat different from type classes

Type classes are a way of "making ad-hoc polymorphism less ad-hoc" [HHPW96, WB89].

Type Classes

Type classes are a way of "making ad-hoc polymorphism less ad-hoc" [HHPW96, WB89].

Type classes are used to group together types with certain properties, in particular, types for which certain symbols are defined.

For example, for some types, a symbol $\leq$ (which is a binary infix predicate) may exist and for some not, and we could have a type class ord containing all types for which it exists.

Suppose you want to sort a list of elements (smaller elements should come before bigger elements). This is only defined for elements of a type for which the symbol $\leq$ exists.

Note that while a symbol such as $\leq$ may have a similar meaning for different types (for example, integers and reals), one cannot say that it means exactly the same thing regardless of the type of the argument to

which it is applied. In fact, $\leq$ has to be defined separately for each type in ord.

This is in contrast to parametric poymorphism, but also somewhat different from ad-hoc polymorphism: The types of the symbols must not be declared separately. E.g., one has to declare only once that $\leq$ is of type $(a :: ord, \alpha)$.

Polymorphic Type Language

As before, we define a type language, i.e., a language consisting of types, and a particular type language is characterized by a giving a certain set of symbols $\mathcal{B}$. But unlike before, $\mathcal{B}$ is now a set of type constructors. Each type constructor has an arity associated with it just like a function in first-order logic. The intention is that a type constructor may be applied to types.

Following the conventions of ML [Pau96], we write types in postfix notation, something we have not seen before. I.e., the type constructor comes after the arguments it is applied to.

It makes perfect sense to view the function construction arrow $\rightarrow$ as type constructor, however written infix rather than postfix.

So the $\mathcal{B}$ is some fixed set "defined by the user", but it should definitely always include $\rightarrow$.

Type Substitutions

A type substitution replaces a type variable by a type, just like in first-order logic, a substitution replaces a variable by a term.

Syntactic Classes

A syntactic class is a class of types for which certain symbols are declared to exist. Isabelle has a syntax for such declarations. E.g., the declaration

```
sort ord < term
const <= : ['a::ord, 'a] => bool
```

may form part of an Isabelle theory file. It declares a type class ord which is subclass (that's what the $<$ means; in mathematical notation it will be written $\prec$) of a class $term$, meaning that any type in ord is also in $term$. the class $term$ must be defined elsewhere.

The second line declares a symbol <=. Such a declaration is preceded by the keyword `const`. The notation $\alpha :: ord$ stands for a type variable constrained to be in class ord. So <= is declared to be of type $[\alpha :: ord, \alpha] \Rightarrow bool$, meaning that it takes two arguments of a type in

the class ord and returns a term of type $bool$. The symbol $\Rightarrow(=>)$ is the function type arrow in Isabelle. Note that the second occurrence of α is written without :: ord. This is because it is enough to state the class constraint once.

Note also that $[\alpha :: ord, \alpha] => bool$ is in fact just another way of writing $\alpha :: ord => \alpha => bool$, similarly as for goals.

Haskell [HHPW96] has type classes but ML [Pau96] hasn't.

Axiomatic Classes

In addition to declaring the syntax of a type class, one can axiomatize the semantics of the symbols. Again, Isabelle has a syntax for such declarations. E.g., the declaration

```
axclass order < ord
  order_refl: ''x <= x ''
  order_trans: ''[| x <= y; y <= z |] ==> x <= z''
  ...
```

may form part of an Isabelle theory file. It declares an axiomatic type class $order$ which is a subclass of ord defined above.

The next two lines are the axioms. Here, `order_refl` and `order_trans` are the names of the axioms. Recall that $\Longrightarrow$ is the implication symbol in Isabelle (that is to say, the metalevel implication).

Whenever an Isabelle theory declares that a type is a member of such a class, it must prove those axioms.

The rationale of having axiomatic classes is that it allows for proofs that hold in different but similar mathematical structures to be done only once. So for example, all theorems that hold for dense orders can be proven for all dense orders with one single proof.

Renaming

Whenever a rule is applied, the metavariables occurring in it must be renamed to fresh variables to ensure that no metavariable in the rule has been used in the proof before.

The notion fresh is often casually used in logic, and it means: this variable has never been used before. To be more precise, one should say: never been used before in the relevant context.

Unification

The mechanism to instantiate metavariables as needed is called (higher-order) unification. Unification is the process of finding a substitution that makes two terms equal.

We will later see more formally what it is and also where it is used.

Type Class Syntax

$\kappa \quad ::= \quad ord \mid order \mid lattice \mid \ldots$

is a grammar defining what type classes are (syntactically). κ is the non-terminal we use for "type class". However, the grammar given here is incomplete (there are "...") and just exemplary.

So the set of type classes involved in an Isabelle theory is a finite set of names (written lower-case), typically including ord, $order$, and $lattice$.

The grammar does not tell us what syntax is used to declare the type classes. We have seen an example of that previously.

Type Constructor Syntax

$$\chi \quad ::= \quad bool \mid \_ \to \_ \mid ind \mid \_\ list \mid \_\ set \ldots$$

is a grammar defining what type constructors are (syntactically). χ is the non-terminal we use for "type constructor". As before, the grammar given here is incomplete (there are "...") and just exemplary.

Note also that an _ is used to denote the arity of a type constructor.

- _ $list$ means that $list$ is unary type constructor;
- _ $\to$ _ means that $\to$ is a binary infix type constructor.

The notation using _ is slightly abusive since the _ is not actually part of the type constructor (and the grammar is supposed to define type constructors). _ $list$ is not a type constructor; $list$ is a type constructor.

So the set of type constructors involved in an Isabelle theory is a finite set of names (written lower-case) with each having an arity associated,

typically including $bool$, $\rightarrow$, and $list$. Note however that $bool$ is fundamental (since object level predicates are modeled functions taking terms to a Boolean), and so is $\rightarrow$, the constructor of the function space between two types.

The grammar does not tell us what syntax is used to declare the type constructors.

$\longrightarrow$ as Type Constructor

In $\lambda^{\rightarrow}$, types were built from base types using a "special symbol" $\longrightarrow$.

When we generalize $\lambda^{\rightarrow}$ to a λ-calculus with polymorphism, this "special symbol" becomes a type constructor. However, the syntax is still special, and it is interpreted in a particular way.

Polymorphic Types Syntax

$\tau \quad ::= \quad \alpha \mid \alpha :: \kappa \mid (\tau, \ldots, \tau)\,\chi \qquad (\alpha \text{ is type variable})$

is a grammar defining what polymorphic types are (syntactically). As before, τ is the non-terminal we use for (now: polymorphic) types.

This grammar is not exemplary but generic, and it deserves a closer look.

A type variable is a variable that stands for a type, as opposed to a term. We have not given a grammar for type variables, but assume that there is a countable set of type variables disjoint from the set of term variables. We use α as the non-terminal for a type variable (abusing notation, we often also use α to denote an actual type variable).

First, note that a type variable may be followed by a class constraint $:: \kappa$ (recall that κ is the non-terminal for type classes). However, a type variable is not necessarily followed by such a constraint, for example if

the type variable already occurs elsewhere and is constrained in that place. We have already seen this.

Moreover, a polymorphic type is obtained by preceding a type constructor with a tuple of types. The arity of the tuple must be equal to the declared arity of the type constructor.

It is not shown here that for some special type constructors, such as $\rightarrow$, the argument may also be written infix.

Type Instantiation

The assumption and hypothesis rules have an assumption of the form $\tau \prec \sigma$.

The symbol $\prec$ is an ordering on types, induced by the subclass ordering $\prec$ on type classes. $\tau \prec \sigma$ means that τ is an instance of σ, and τ is in a type class c, and σ is in a type class d, such that $c \prec d$.

One can also write $\tau :: c$ and $\sigma :: d$. We have previously seen the notation $\alpha :: c$ for a type variable constrained to be in class c. We regarded the whole expression $\alpha :: c$ as a type, but we have also seen that a type variable is not necessarily followed by such a constraint.

Here, τ, σ are arbitrary types, not necessarily a type variables. For a type τ other than a type variable, the expression $\tau :: c$ must be read as an assertion that τ is in type class c.

One can formalize precisely when the type class declarations of an

Isabelle theory entail the assertion that a type is in a certain type class, but we do not go into these details here.

Note that $\prec$ is reflexive.

Consult [HHPW96, Nip93] for details on type classes.

Type Construction

Type construction is the problem of given a Σ, Γ and e, finding a t such that $\Sigma\,, \Gamma \vdash e : \tau$.

Sometimes one also considers the problem where Γ is unknown and must also be constructed.

Term Congruence

$\alpha\beta\eta$-conversion is defined as for $\lambda^{\rightarrow}$. Given two (extended) λ-terms e and e', it is decidable whether $e =_{\alpha\beta\eta} e'$.

Solutions for Unification Problems

A solution for $?X + ?Y =_{\alpha\beta\eta} x + x$ is $[x/?X, x/?Y]$.

A solution for $?P(x) =_{\alpha\beta\eta} x + x$ is $[(\lambda y.y + y)/?P]$.

A solution for $f(?X x) =_{\alpha\beta\eta} ?Y\ x$ is $[(\lambda z.z)/?X, (\lambda z.f\ z)/?Y]$.

Unification Modulo

Unification of terms e, e' modulo $\alpha\beta$ means finding a substitution θ for metavariables such that $\theta(e) =_{\alpha\beta} \theta(e')$.

Likewise, unification of terms e, e' modulo $\alpha\beta\gamma$ means finding a substitution σ for metavariables such that $\sigma(e) =_{\alpha\beta\gamma} \sigma(e')$.

References

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proofs*. Academic Press, 1986.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

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

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[GM93] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philipp

Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[Klo93] Jan Willem Klop. *Handbook of Logic in Computer Science*, chapter "Term Rewriting Systems". Oxford: Clarendon Press, 1993.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[Nip93] Tobias Nipkow. *Logical Environments*, chapter Order-Sorted Polymorphism in Isabelle, pages 164–188. Cambridge University Press, 1993.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[Tho95] Simon Thompson. *Miranda: The Craft of Functional Programming*. Addison-Wesley, 1995.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. Second Edition.

[WB89] Phillip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.