2nd-order Propositional Logic and $\lambda 2$

Julio Mariño

Universidad Politécnica de Madrid

1 de junio de 2005

references

- Girard, Lafont, Taylor: *Proofs and Types*
- Logical Verification at VU Amsterdam:
  `http://www.cs.vu.nl/~tcs/al/04-05/notes/`

motivation

- In (1st-order) propositional logic, the following are tautologies:

$$\vdash p \rightarrow p$$
$$\vdash (p \wedge q) \rightarrow (p \wedge q)$$
$$\vdash (p \vee q) \rightarrow (p \vee q)$$

  but there is no way to express (formally) that for every ϕ, $\vdash \phi \rightarrow \phi$ although we know this has to be true, considering how natural deduction works.
- Similarly, we would like to extend TA_λ with polymorphism, i.e. being able to express that a given term (e.g. $\lambda x.x$) may be assigned a whole (parametric) family of simple types (e.g. $a \rightarrow a$, $(a \times b) \rightarrow (a \times b)$, $(a \rightarrow b) \rightarrow (a \rightarrow b)$, etc.) with a common *shape* $(\alpha \rightarrow \alpha)$
- In this lecture we will see that there is an extension of propositional logic that allows for quantification over propositional variables and whose associated term calculus via the Curry-Howard isomorphism is actually a polymorphic version of the λ-calculus.

2nd-order propositional logic

syntax

$$\begin{array}{rcl} x ::= & a, b, c, \ldots, p, q, \ldots & \text{(vars)} \\ \phi ::= & x \mid \bot \mid \phi \rightarrow \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall x.\phi \mid \exists x.\phi & \text{(formulae)} \end{array}$$

Example:

$$\forall a.a \rightarrow a$$

every proposition implies itself

impredicativity

The domain of quantification is the whole set of propositional formulae, i.e. from the formula above we can deduce $p \rightarrow p$, $(p \wedge q) \rightarrow (p \wedge q)$, $(p \vee q) \rightarrow (p \vee q)$, etc.

We might even substitute $\forall a.a \rightarrow a$ or a *bigger* formula for a...

This kind of quantification is referred to as *impredicative* — compare with FOL.

natural deduction

- we will take a natural deduction calculus for propositional logic and will just add introduction and elimination rules for the two quantifiers
- universal quantifier introduction

$$\frac{A}{\forall a.A}\ \forall\mathrm{i}$$

variable condition: a not free in any open assumption

- universal quantifier elimination

$$\frac{\forall a.A}{A[B/a]}\ \forall\mathrm{e}$$

- existential quantifier introduction

$$\frac{A[B/a]}{\exists a.A}\ \exists\mathrm{i}$$

- existential quantifier elimination

$$\frac{\exists a.A}{A[B/a]}\ \exists\mathrm{e}$$

variable condition: a not free in B

exercises

1. $\vdash (\forall b.b) \rightarrow a$
2. $\vdash a \rightarrow \forall b.((a \rightarrow b) \rightarrow b)$
3. $\vdash (\exists b.a) \rightarrow a$
4. $\vdash \exists b.(a \rightarrow b) \vee (b \rightarrow a)$

solution.1

$$\dfrac{\dfrac{[\forall b.b]^0}{a}\ \forall\mathrm{e}}{(\forall b.b) \to a}\ {\to}\mathrm{i}^0$$

solution.2

$$\dfrac{\dfrac{\dfrac{\dfrac{[a]^0 \qquad [a \to b]^1}{b}\,{\to}\mathrm{e}}{(a \to b) \to b}\,{\to}\mathrm{i}^1}{\forall b.(a \to b) \to b}\,\forall\mathrm{i}}{a \to \forall b.(a \to b) \to b}\,{\to}\mathrm{i}^0$$

solution.3

$$\dfrac{\dfrac{[\exists b.a]^0}{a}\,\exists\text{e}}{(\exists b.a) \to a}\,{\to}\text{i}^0$$

solution.4

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\forall c.c \longrightarrow c}}{a \longrightarrow a}\;\forall\mathrm{e}}{(a \longrightarrow a) \vee (a \longrightarrow a)}\;\vee\mathrm{i}}{\exists b.(a \longrightarrow b) \vee (b \longrightarrow a)}\;\exists\mathrm{i}$$

normalization

- in addition to those detours already present in propositional logic, now we have those introduced by the new quantifiers
- universal quantification an application of the (∀i) rule followed immediately by an application of the (∀e) rule:

$$\dfrac{\dfrac{B}{\forall a.B}\,\forall\mathrm{i}}{B[A/a]}\,\forall\mathrm{e} \qquad \rightsquigarrow \qquad B[A/a]$$

$\lambda 2$

- $\lambda 2$, also known as the *polymorphic λ-calculus* was discovered independently by J-Y. Girard, who was looking for extensions of the basic Curry-Howard isomorphism (1971), and
- John C. Reynolds (1974), who wanted to study formalisations of polymorphism in programming languages
- the basic idea behind parametric polymorphism is that *terms may take types as arguments*
- we will see that $\lambda 2$ is the term calculus associated with 2nd-order intuitionistic propositional logic

terms

- term formation for the rules already present in intuitionistic propositional logic will stay as before
- universal quantification intro – type abstraction

$$\frac{t : A}{\Lambda a.t : \forall a.A}\ \forall\text{i}$$

variable condition: a not free in any open assumption

- universal quantification elim – type application

$$\frac{t : \forall a.A}{t[B] : A[B/a]}\ \forall\text{e}$$

conversion

- existing conversion rules for the traditional λ-calculus are adapted in the obvious way
- universal quantification normalization results in:

$$\dfrac{\dfrac{t : A}{\Lambda a.t : \forall a.A}\,\forall\mathrm{i}}{(\Lambda a.t)[B] : A[B/a]}\,\forall\mathrm{e} \quad \rightsquigarrow \quad t[B/a] : A[B/a]$$

examples

- naturals

$$
\begin{aligned}
nat &= \forall t.(t \rightarrow t) \rightarrow (t \rightarrow t) \\
0 &: \Lambda t.\lambda f : t \rightarrow t.\lambda x : t.x \\
1 &: \Lambda t.\lambda f : t \rightarrow t.\lambda x : t.f\,x \\
2 &: \Lambda t.\lambda f : t \rightarrow t.\lambda x : t.f(f\,x) \\
&\vdots \\
succ &= \lambda n : nat.\Lambda t.\lambda f : t \rightarrow t.\lambda x : t.f(n[t]\,f\,x) \\
add &= \lambda m : nat.\lambda n : nat.\Lambda t.\lambda f : t \rightarrow t.\lambda x : t.m[t]\,f\,(n[t]\,f\,x)
\end{aligned}
$$

- booleans
- lists

main results

- type checking The type checking problem is to decide whether, given Γ, τ and T, $\Gamma \vdash T : \tau$. Type checking is decidable for $\lambda 2$.
- type synthesis The type synthesis problem is to decide whether, given Γ and T, there is some τ s.t. $\Gamma \vdash T : \tau$. TSP for $\lambda 2$ is equivalent to TCP and thus decidable.
- type inhabitation The type inhabitation problem is to decide whether, given Γ and τ, there is some T s.t. $\Gamma \vdash T : \tau$. TIP for $\lambda 2$ is undecidable.
- type uniqueness In $\lambda 2$, terms have only one type (up to α conversion).
- subject reduction Types are preserved through β reduction, i.e. if $\Gamma \vdash T : \tau$ and $T \rightsquigarrow_\beta T'$ then $\Gamma \vdash T' : \tau$.
- strong normalization Every sequence of β conversions of a term is terminating.

type synthesis is not type inference

The fact that the TSP for $\lambda 2$ is decidable does not mean that these typings could be inferred from a program written in a modern functional programming language — remember that terms in $\lambda 2$ are fully annotated with types

differences with HM-style polymorphism

- The familiar type systems in modern functional programming languages (Haskell, ML and the like, etc) are, in fact, polymorphic, and many useful functions can be used with different types in different contexts.
- Polymorphic type schemes in the Hindley-Milner system correspond to those types of $\lambda 2$ in prenex form. i.e. with all the quantifiers in an outermost position.
- This may be too restrictive for type inference of certain definitions:

  ```
  f g = (g 1, g True)
  ```

  cannot be assigned a type in Haskell... although it is possible to write, for instance,

  ```
  (id 1, id True)
  ```

  so it would be great to be able to require polymorphism to an argument, e.g. by extra annotations:

  ```
  f :: (all a. a -> a) -> (Int, Bool)
  f (g::all a. a -> a) = (g 1, g True)

  f :: (all a. a -> [a]) -> ([Int], [Bool])
  f (g::all a. a -> [a]) = (g 1, g True)
  ```

  etc.

existential types

- $\lambda 2$ only considers universal quantification, but the isomorphism can be extended to interpret existential types.
- In a naïve set interpretation, types denote sets of terms. The polytype $\forall a.t$ denotes the intersection $\bigcap_\tau t[\tau/a]$
  For example, the intersection of the sets denoted by $\tau \to \tau$ only contains the identity function (the inhabitant of the polytype $\forall a.a \to a$).
- The interpretation of the existential type $\exists a.t$ is the union of the sets denoted by the instances, i.e. $\bigcup_\tau t[\tau/a]$
- The intuition is to capture *abstract data types* i.e. the fact that the implementation behind a certain interface is hidden:

  ```
  push :: [Int] -> Int -> [Int]

  push :: Tree Int -> Int -> Tree Int
  ```

  may be alternative implementations for an operation on stacks of integers and, in fact, both can be assigned the existential type

  $$\exists StackInt.StackInt \to Int \to StackInt$$

  which abstracts away the concrete implementation.