FOLLIA annual meeting, Torino 19-20/01/2006

Typing Problems in Second Order Light Logics

Paolo Tranquilli
Dipartimento di Matematica
Università di Roma Tre

Outline

- Why automatic typing?
- Typing problems: Type Checking (TC) and Typability (TYP)
- Simply typed λ-calculus
- Propositional EAL, LAL and DLAL
- Second order
  - From system F...
  - ...to light logics
- Automatic typing vs. expressiveness

Background

In general:

- Σ type assignment system:
  - terms M
  - types τ
  - rules for derivation of sequents
- A sequent is of the form $\Gamma \vdash M : \tau$ with Γ a function from variables to types with finite domain.
- If $\Gamma \vdash M : \tau$ is derivable in Σ I'll write $\Gamma \vdash_{\Sigma} M : \tau$.
- Typing is “good”.

Why automatic typing?

- Some form of automatic typing wuold be appreciated. Well, that is if we want to let people other than computer scientists use the system...
- The best would be for a programmer to just write the program, while a machine does the work needed to embed the program in the system (or reject it).

Typing problems

I'm going to concentrate on decision problems.

- TC (Type Checking):
  - Input: Γ, M, τ
  - Decide: $\Gamma \vdash_\Sigma M : \tau$
- TYP (Typability):
  - Input: M
  - Decide: $\exists \Gamma, \tau \ (\Gamma \vdash_\Sigma M : \tau)$

Type inference solves TYP, strong type inference (i.e. giving all possible types through principal typing) solves TC also.

Simply typed λ-calculus

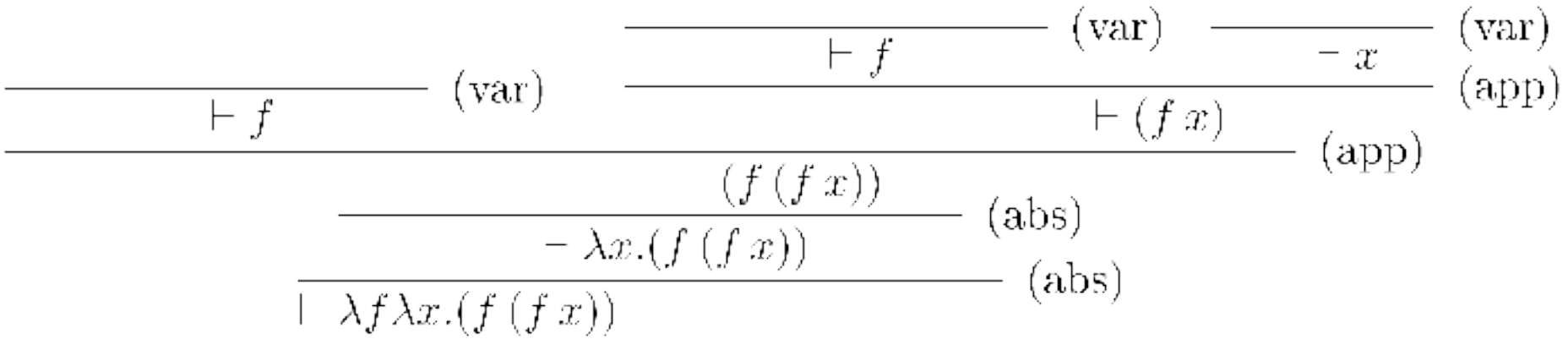
We have Hindley-Milner's strong type inference algorithm.

Its core is a procedure that solves *unification* for types, i.e. given two (quantifier-free) types σ and τ it finds a (most general) substitution S such that $S(\sigma)=S(\tau)$.

The rest is done by decorating the syntactic tree of the term.

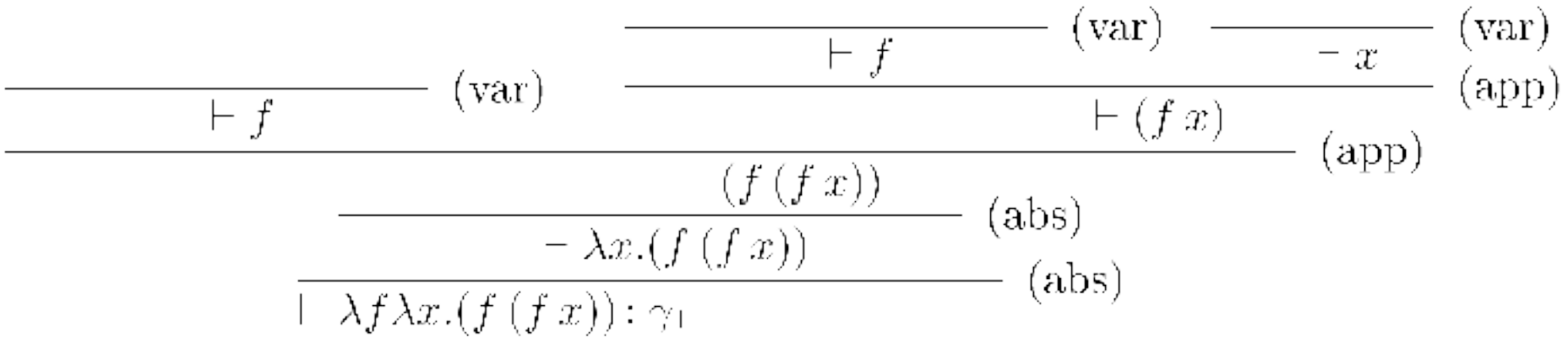
Simply typed λ-calculus

Example:



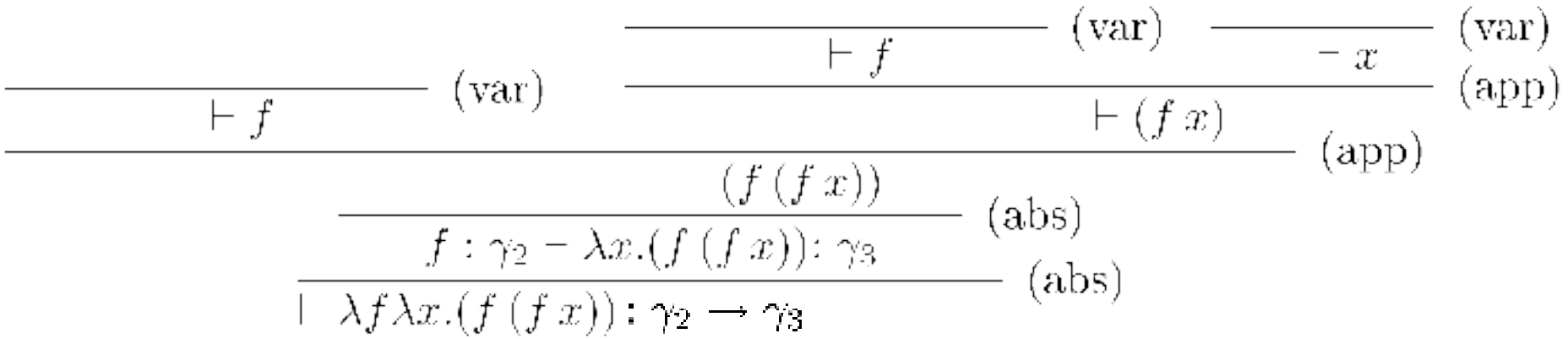
Simply typed λ-calculus

Example:



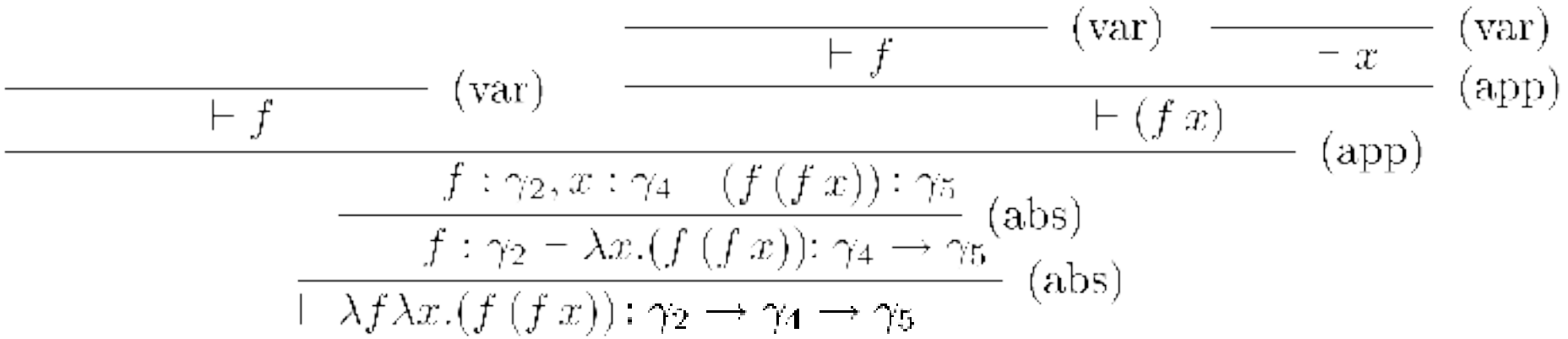
Simply typed λ-calculus

Example:



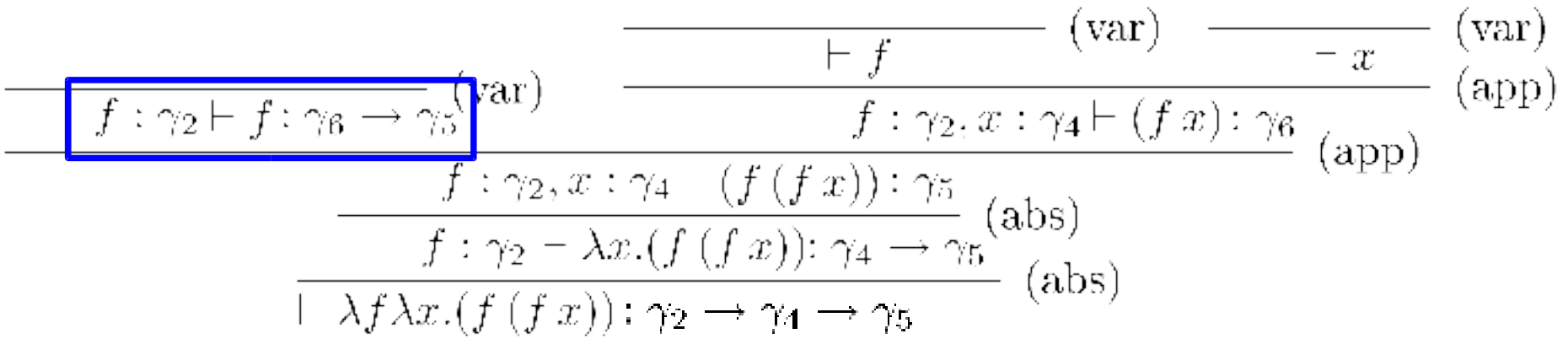
Simply typed λ-calculus

Example:



Simply typed λ-calculus

Example:



Simply typed λ-calculus

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \gamma_6 \to \gamma_5 \vdash f : \gamma_6 \to \gamma_5}\,(\text{var}) \quad \dfrac{\dfrac{}{\vdash f}\,(\text{var}) \quad \dfrac{}{- x}\,(\text{var})}{f : \gamma_6 \to \gamma_5, x : \gamma_4 \vdash (f\ x) : \gamma_6}\,(\text{app})}{f : \gamma_6 \to \gamma_5, x : \gamma_4 \quad (f\ (f\ x)) : \gamma_5}\,(\text{app})}{f : \gamma_6 \to \gamma_5 - \lambda x.(f\ (f\ x)) : \gamma_4 \to \gamma_5}\,(\text{abs})}{\vdash \lambda f \lambda x.(f\ (f\ x)) : (\gamma_6 \to \gamma_5) \to \gamma_4 \to \gamma_5}\,(\text{abs})$$

Simply typed λ-calculus

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f:\gamma_6\to\gamma_5\vdash f:\gamma_6\to\gamma_5}\,(\text{var}) \quad \dfrac{\dfrac{}{f:\gamma_6\to\gamma_5\vdash f:\gamma_7\to\gamma_6}\,(\text{var}) \quad \dfrac{}{x:\gamma_4\vdash x:\gamma_7}\,(\text{var})}{f:\gamma_6\to\gamma_5,x:\gamma_4\vdash (f\ x):\gamma_6}\,(\text{app})}{f:\gamma_6\to\gamma_5,x:\gamma_4 \quad (f\ (f\ x)):\gamma_5}\,(\text{app})}{f:\gamma_6\to\gamma_5\vdash \lambda x.(f\ (f\ x)):\gamma_4\to\gamma_5}\,(\text{abs})}{\vdash \lambda f\lambda x.(f\ (f\ x)):(\gamma_6\to\gamma_5)\to\gamma_4\to\gamma_5}\,(\text{abs})$$

Simply typed λ-calculus

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f:\gamma_5\to\gamma_5\vdash f:\gamma_5\to\gamma_5}(\text{var}) \quad \dfrac{\dfrac{}{f:\gamma_5\to\gamma_5\vdash f:\gamma_5\to\gamma_5}(\text{var}) \quad \dfrac{}{x:\gamma_4\vdash x:\gamma_5}(\text{var})}{f:\gamma_5\to\gamma_5, x:\gamma_4\vdash (f\ x):\gamma_5}(\text{app})}{f:\gamma_5\to\gamma_5, x:\gamma_4 \quad (f\ (f\ x)):\gamma_5}(\text{app})}{f:\gamma_5\to\gamma_5 \vdash \lambda x.(f\ (f\ x)):\gamma_4\to\gamma_5}(\text{abs})}{\vdash \lambda f\lambda x.(f\ (f\ x)):(\gamma_5\to\gamma_5)\to\gamma_4\to\gamma_5}(\text{abs})$$

Simply typed λ-calculus

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \gamma_5 \to \gamma_5 \vdash f : \gamma_5 \to \gamma_5}(\text{var}) \quad \dfrac{\dfrac{}{f : \gamma_5 \to \gamma_5 \vdash f : \gamma_5 \to \gamma_5}(\text{var}) \quad \dfrac{}{x : \gamma_5 \vdash x : \gamma_5}(\text{var})}{f : \gamma_5 \to \gamma_5, x : \gamma_5 \vdash (f\ x) : \gamma_5}(\text{app})}{f : \gamma_5 \to \gamma_5, x : \gamma_5 \quad (f\ (f\ x)) : \gamma_5}(\text{app})}{f : \gamma_5 \to \gamma_5 \vdash \lambda x.(f\ (f\ x)) : \gamma_5 \to \gamma_5}(\text{abs})}{\vdash \lambda f \lambda x.(f\ (f\ x)) : (\gamma_5 \to \gamma_5) \to \gamma_5 \to \gamma_5}(\text{abs})$$

EAL

Strong type inference for propositional elementary affine logic has been shown (Coppola and Ronchi della Rocca 2003).

This involves a syntax driven variant of the system, for which a refinement of the unification procedure is designed, handling variable numbers of bangs by introducing linear constraints on integers. Decoration of the syntax tree therefore involves solving a system of linear constraints.

Typing a pure term involves finding all the finitely many possible syntax driven versions of the term.

LAL

Strong type inference for propositional light affine logic has been shown (Baillot 2004).

As for EAL, the unification procedure is redesigned to give also constraints concerning modalities, which in this system are handled as linear inequalities on strings.

Also the variant of this system, DLAL (Baillot and Terui 2004), has a strong type inference algorithm, which tries to convert an EAL typing into a DLAL one.

Second order

- In these system polymorphism of types is added to gain decent expressiveness.
- Results of completeness (all functions of a certain class are representable in the given system) depend on it.
- However this comes at a cost: automatic type inference is no longer granted.
- As easiness of unification is at the base of propositional type inference, hardness of the corresponding problem in second order, SUP (semiunification), is at the base of results of undecidability.
- SUP is undecidable (Kfoury, Tiuryn and Uzyczyn 1993).

Semiunification

- SUP (semiunification problem) with two pairs:
  - Input: two pairs of quantifier-free types (σ_1, τ_1), (σ_2, τ_2)
  - Decide:

$$\exists S, S_1, S_2 \left(S_1(S(\sigma_1)) = S(\tau_1) \& S_2(S(\sigma_2)) = S(\tau_2) \right)$$

The intended meaning is that a solution to the problem is a substitution that, rather than equalling all the types, brings them to a form where in both the pairs σ_i can be obtained from τ_i by instantiating separately some further variables.

Undecidability is proved by showing a reduction to of an undecidable problem over Turing machines (immortality).

Reduction of SUP to TC

Sketch (for system F, as shown by Wells):

- Given an instance $(\sigma_1, \tau_1), (\sigma_2, \tau_2)$ of SUP we build the instance of TC given by $\Gamma \vdash_{\mathbf{F}} b\,(\lambda x.cxx) : \beta$ where

$$\Gamma := \{\, b : \forall\gamma.(\gamma \to \gamma) \to \beta,\ c : \forall.(\tau_1 \to \delta_1) \to (\delta_2 \to \tau_2) \to (\sigma_1 \to \sigma_2) \,\}$$

$\exists$ a solution for $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \iff \Gamma \vdash_{\mathbf{F}} M : \tau$

Reduction of TC to TYP

Sketchier still (Wells 1999):

- Given an instance of TC $\Gamma \vdash_{\mathbf{F}} M : \tau$ a context $C[\,]$ is built such that it forces variables binding the hole to be typed with specific types in case $C[N]$ is typed, whatever be *N*. In particular among those variables are the free ones of *M*, forced to be typed as in Γ, plus a new variable forced to be typed with τ. Then

$$\Gamma \vdash_{\mathbf{F}} M : \tau \iff C[z\,M] \text{ typable}$$

Reduction of SUP to TC

For DLAL:

- Given an instance $(\sigma_1, \tau_1), (\sigma_2, \tau_2)$ of SUP we build the instance of TC given by $\Gamma \vdash_{\mathbf{DLAL}} b\,(\lambda x.cxx) : \beta$ where

$$\Gamma := ; b : \forall\gamma.(\gamma \to \S\gamma) \multimap \beta,\ c : \S\forall.(\tau_1 \multimap \delta_1) \multimap (\delta_2 \multimap \tau_2) \multimap (\sigma_1 \multimap \sigma_2)$$

$\exists$ a solution for $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \Rightarrow \Gamma \vdash_{\mathbf{DLAL}} b\,(\lambda x.cxx) : \beta$

The other direction is done by injecting DLAL all the way to system F, proving that the reduction holds for all the systems caught in between (such as LAL and EAL).

Open problem

So TC is undecidable for all relevant polymorphic systems.

Undecidability of TYP instead remains still up to now an open question, although we can conjecture for the negative result.

Adaptation of Wells' proof seems out of reach though.

Automatic typing vs. expressiveness

- Bringing in full polymorphism gives maximum expressiveness but gives away automatic typing. A priori we can't even say whether the function we've just programmed is from integers to integers as we intended, for example.
- Wells has shown type inference for rank 2 system F, where quantifiers are prohibited from having left depth more than 2 (i.e. proceeding from the root to the quantifier in the construction tree we do not encounter more than 2 left turns in implications). So we get again automatic typing, but we loose much expressiveness: for example in

$$\texttt{int} = \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$$

we can instantiate only simple types, diminishing the power of iteration.