Higher-Order and Dependent Types

Sebastian Miele

Overview

- Until now: *Terms* have been abstracted by

terms	$\lambda x{:}T\,.\,t$	
types	$\lambda X\,.\,t$	(polymorphic values, system F)

- Natural question: Does it make sense to abstract *types* by terms and/or types as well?

Overview

- *Types* may be abstracted by

types	$\lambda \mathsf{X}\,.\,\mathsf{T}$	(type operators, system λ_ω)
terms	$\Pi \mathsf{x}{:}\mathsf{S}\,.\,\mathsf{T}$	(dependent types, system LF)

  as well.

- In this talk: Higher-order types (λ_ω) will be combined with universal types (F). The result is system F_ω (Girard, 1972).
- Then we will have a brief look into dependent types.
- Finally we will have a brief look into the λ-cube (Barendregt, 1991).

Intuition: Type Operators

- First-order type operators in SML:

```
type 'a stack = 'a list ref
type int_stack = int list ref
```

- In F_ω:

$$\begin{aligned} \mathsf{Stack} &= \lambda X\,.\,\mathsf{Ref\ List}\ X \\ \mathsf{IntStack} &= \mathsf{Stack\ Int} \end{aligned}$$

Intuition: Kinding

- Type-level abstraction and application allow writing meaningless types:

$$\mathsf{Nat}\ \mathsf{Nat}$$

- Solution: Typing of types, called *kinding*:

$$\Gamma \vdash T :: K$$

- Syntax of Kinds:

$$\begin{array}{rcll} K & ::= & \star & \text{(proper types)} \\ & | & K \Rightarrow K & \text{(type operators)} \end{array}$$

Intuition: Kinds

- Proper types:

$$\vdash\ \mathsf{Nat} :: \star$$
$$\vdash\ \mathsf{Nat} \to \mathsf{Nat} :: \star$$
$$\vdash\ (\lambda X{::}\star\,.\,X \to X)\ \mathsf{Nat} :: \star$$

- Type operators:

$$\vdash\ \lambda X\,.\,X :: \star \Rightarrow \star$$
$$\vdash\ \lambda X\,.\,\mathsf{Ref}\ \mathsf{List}\ X :: \star \Rightarrow \star$$
$$\vdash\ \lambda X\,.\,\lambda Y\,.\,\forall Z\,.\,(X \to Y \to Z) \to Z :: \star \Rightarrow \star$$
$$\vdash\ \lambda F{::}\star \Rightarrow \star\,.\,F\ \mathsf{Nat} :: (\star \Rightarrow \star) \Rightarrow \star$$

Intuition: Higher-Order Quantification

- Higher-order quantification in SML:

```
signature STACK_ADT
sig type 'a stack
    val new : unit -> 'a stack
    val push : 'a stack -> 'a -> unit
    val pop : 'a stack -> 'a
end
```

- In F_ω:

$$\begin{aligned} \mathsf{StackADT} = & \\ & \exists \mathsf{Stack}{::}\star \Rightarrow \star \,.\, \{ \begin{array}{rl} \mathsf{new}: & \forall X \,.\, \mathsf{Unit} \to \mathsf{Stack}\, X, \\ \mathsf{push}: & \forall X \,.\, \mathsf{Stack}\, X \to X \to \mathsf{Unit}, \\ \mathsf{pop}: & \forall X \,.\, \mathsf{Stack}\, X \to X \} \end{array} \end{aligned}$$

New Syntax in System F_ω

- System F_ω is *explicitly kinded*: Bound type variables have kinding annotations.

$$
\begin{array}{lcll}
\mathsf{t} & ::= & \ldots & \\
& | & \lambda \mathsf{X}::\mathsf{K}\,.\,\mathsf{t} & \text{(type abstraction)} \\
\\
\mathsf{T} & ::= & \ldots & \\
& | & \forall \mathsf{X}::\mathsf{K}\,.\,\mathsf{T} & \text{(universal type)} \\
& | & \lambda \mathsf{X}::\mathsf{K}\,.\,\mathsf{T} & \text{(operator abstraction)} \\
& | & \mathsf{T}\;\mathsf{T} & \text{(operator application)}
\end{array}
$$

- For $\star$ we will omit kinding annotations.

Kinding ($\Gamma \vdash T :: K$)

- Contexts now may contain type variable bindings:

$$
\begin{array}{lcll}
\Gamma & ::= & \emptyset & \text{(empty context)} \\
 & \mid & \mathsf{x}{:}\mathsf{T} & \text{(term variable binding)} \\
 & \mid & \mathsf{X}{::}\mathsf{K} & \text{(type variable binding)}
\end{array}
$$

- Kinding rule for type variables:

$$
\frac{\mathsf{X}{::}\mathsf{K} \in \Gamma}{\Gamma \vdash \mathsf{X} :: \mathsf{K}} \quad \text{(K-TVar)}
$$

Kinding ($\Gamma \vdash T :: K$): Proper Types

$$\Gamma \vdash \mathsf{Nat} :: \star \qquad \text{(K-NAT)}$$

$$\Gamma \vdash \mathsf{Bool} :: \star \qquad \text{(K-BOOL)}$$

$$\vdots \qquad \vdots$$

$$\frac{\Gamma \vdash \mathsf{S} :: \star \qquad \Gamma \vdash \mathsf{T} :: \star}{\Gamma \vdash \mathsf{S} \to \mathsf{T} :: \star} \qquad \text{(K-ARROW)}$$

$$\frac{\Gamma, \mathsf{X}{::}\mathsf{K} \vdash \mathsf{T} :: \star}{\Gamma \vdash \forall \mathsf{X}{::}\mathsf{K} \,.\, \mathsf{T} :: \star} \qquad \text{(K-ALL)}$$

Kinding ($\Gamma \vdash T :: K$): Type Operators

$$\frac{\Gamma, \mathsf{X}{::}\mathsf{K} \vdash \mathsf{T} :: \mathsf{K}'}{\Gamma \vdash \lambda \mathsf{X}{::}\mathsf{K}\,.\,\mathsf{T} :: \mathsf{K} \Rightarrow \mathsf{K}'} \qquad \text{(K-ABS)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: \mathsf{K} \Rightarrow \mathsf{K}' \qquad \Gamma \vdash \mathsf{T} :: \mathsf{K}}{\Gamma \vdash \mathsf{S}\ \mathsf{T} :: \mathsf{K}'} \qquad \text{(K-APP)}$$

Type Equivalence ($\mathsf{S} \equiv \mathsf{T}$)

- Type operators give us the possibility to write a single type in several ways:

$$(\lambda \mathsf{X} \,.\, \mathsf{X})\ \mathsf{Nat} \quad \equiv \quad \mathsf{Nat}$$

- Key rule of type equivalence:

$$(\lambda \mathsf{X}{::}\mathsf{K} \,.\, \mathsf{T})\ \mathsf{S} \;\equiv\; [\mathsf{X} \mapsto \mathsf{S}]\mathsf{T} \qquad \text{(Q-BETA)}$$

Type Equivalence ($S \equiv T$): Structural Rules

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \qquad \text{(Q-APP)}$$

$$\frac{S \equiv T}{\lambda X{::}K\,.\,S \equiv \lambda X{::}K\,.\,T} \qquad \text{(Q-ABS)}$$

$$\frac{S \equiv T}{\forall X{::}K\,.\,S \equiv \forall X{::}K\,.\,T} \qquad \text{(Q-ALL)}$$

Type *Equivalence* ($S \equiv T$)

$$T \equiv T \qquad \text{(Q-Refl)}$$

$$\frac{T \equiv S}{S \equiv T} \qquad \text{(Q-Symm)}$$

$$\frac{S \equiv T \qquad T \equiv U}{S \equiv U} \qquad \text{(Q-Trans)}$$

Typing: Basic Property

- We want our type system to have the following basic property:

  Well-typed terms have well-kinded proper types.

- More precisely:

  If $\Gamma \vdash \mathsf{t} : \mathsf{T}$ ***and*** Γ *is well-formed*
  then $\Gamma \vdash \mathsf{T} :: \star$*.*

- Γ is well-formed if: $\Gamma = \Gamma_1, \mathsf{x}{:}\mathsf{T}, \Gamma_2 \ \Rightarrow\ \Gamma_1 \vdash \mathsf{T} :: \star$

- Without Γ well-formed, we could derive ill-kinded types:

$$\mathsf{x}{:}\mathsf{Nat}\ \mathsf{Nat} \vdash \mathsf{x} : \mathsf{Nat}\ \mathsf{Nat}$$

Typing: New Rule T-EQ

- From
  - $\Gamma \vdash t_1 : (\lambda X . X \to X)\mathsf{Nat}$
  - $\Gamma \vdash t_2 : \mathsf{Nat}$

  we want to conclude that $\Gamma \vdash t_1\ t_2 : \mathsf{Nat}$.

- Solution: New typing rule using type equivalence and well-kindedness.

$$\frac{\Gamma \vdash t : S \qquad S \equiv T \qquad \Gamma \vdash T :: \star}{\Gamma \vdash t : T} \quad \text{(T-EQ)}$$

- Why premise $\Gamma \vdash T :: \star$? $\qquad \mathsf{Nat} \equiv (\lambda X{::}\star \Rightarrow \star . X)\mathsf{Nat}$

Typing: λ-Abstraction

$$\frac{\Gamma \vdash \mathsf{S} :: \star \qquad \Gamma, \mathsf{x}{:}\mathsf{S} \vdash \mathsf{t} : \mathsf{T}}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{S}\,.\,\mathsf{t} : \mathsf{S} \to \mathsf{T}} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{S} \to \mathsf{T} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{S}}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}} \quad \text{(T-App)}$$

Typing: Polymorphic Values

$$\frac{\Gamma, X{::}K \vdash t : T}{\Gamma \vdash \lambda X{::}K\,.\,t : \forall X{::}K\,.\,T} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t : \forall X{::}K\,.\,T \qquad \Gamma \vdash S :: K}{\Gamma \vdash t\,[S] : [X \mapsto S]T} \qquad \text{(T-TApp)}$$

Typechecking Algorithm

- The kinding rules are syntax directed

$\Rightarrow$ The kinding relation is decidable.

- All typing rules except T-Eq are syntax-directed.

$\Rightarrow$ The typing relation is decidable if we can build T-Eq into the premises of some other typing rules.

- Where do we need T-Eq for what?

Typechecker: Purpose of T-Eq

- Matching of argument types:

$$\frac{\Gamma \vdash t_1 : S \to T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : T} \quad \text{(T-APP)}$$

- Transforming a type to a specific form:

$$\frac{\Gamma \vdash t_1 : S \to T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : T} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : \forall X{::}K\,.\,T \qquad \Gamma \vdash S :: K}{\Gamma \vdash t\ [S] : [X \mapsto S]T} \quad \text{(T-TAPP)}$$

Types Are Normalizing

- We can define a function ϕ mapping types of F_ω to terms in system F.
- Every β-reduction of $\phi(\mathrm{T})$ corresponds to a β-reduction step of T.
- System F is strongly normalizing (Matthias).

$\Rightarrow$ Beta-reduction on F_ω-types is stronly normalizing.

Typechecking

- It can be shown:

  $\mathsf{S} \equiv \mathsf{T}$ if and only if the normal forms of S and T are equal (modulo α-conversion).

- The normal form of a universal type has the form $\forall \mathsf{X}::\mathsf{K}\,.\,\mathsf{T}$.

- The normal form of an arrow type has the form $\mathsf{S} \rightarrow \mathsf{T}$.

$\Rightarrow$ Using β-reduction, we can

  - transform types to the needed forms or
  - detect that it is not possible with $\equiv$.

F_ω-Kinds $\longrightarrow$ F-Types

- Basetype in system F:

  $\mathsf{T}_\star$

- Translation of F_ω-kinds to F-types:

$$\begin{array}{lcl} \Phi(\star) & = & \mathsf{T}_\star \\ \Phi(\mathsf{K}_1 \Rightarrow \mathsf{K}_2) & = & \Phi(\mathsf{K}_1) \rightarrow \Phi(\mathsf{K}_2) \end{array}$$

F_ω-Types $\longrightarrow$ F-Terms

- Term-level constants in system F:

$$\begin{array}{lcl} \mathsf{c}_{\mathsf{Nat}} & : & \mathsf{T}_\star \\ & \vdots & \\ \mathsf{c}_{\rightarrow} & : & \mathsf{T}_\star \rightarrow \mathsf{T}_\star \rightarrow \mathsf{T}_\star \end{array}$$

- Translation of F_ω-types to F-terms:

$$\begin{array}{lcl} \phi(\mathsf{Nat}) & = & \mathsf{c}_{\mathsf{Nat}} \\ \phi(\mathsf{X}) & = & \mathsf{x} \\ \phi(\lambda \mathsf{X}{::}\mathsf{K}\,.\,\mathsf{T}) & = & \lambda \mathsf{x}{:}\Phi(\mathsf{K})\,.\,\phi(\mathsf{T}) \end{array}$$

F_ω-Types $\longrightarrow$ F-Terms

- Term-level constants in system F (continued):

$$\mathsf{c}_\forall \quad : \quad \forall \mathsf{T}_\mathsf{K}\,.\,(\mathsf{T}_\mathsf{K} \to \mathsf{T}_\star) \to \mathsf{T}_\star$$

- Translation of F_ω-types to F-terms (continued):

$$\phi(\forall \mathsf{X}{::}\mathsf{K}\,.\,\mathsf{T}) \quad = \quad \mathsf{c}_\forall\ [\Phi(\mathsf{K})]\ (\lambda \mathsf{x}{:}\Phi(\mathsf{K})\,.\,\phi(\mathsf{T}))$$

- The λ-abstraction introduced by $\phi(\forall \mathsf{X}{::}\mathsf{K}\,.\,\mathsf{T})$ is not part of a redex.

$\Rightarrow$ Only λ-abstractions corresponding to type-operators are reduced.

Fragments of F_ω

- We can define levels of kinds:

$$
\begin{array}{lcl}
\mathcal{K}_2 & = & \{\star\} \\
\mathcal{K}_{i+i} & = & \mathcal{K}_i \cup \{\mathsf{K}_1 \Rightarrow \ldots \Rightarrow \mathsf{K}_n \mid \mathsf{K}_j \in \mathcal{K}_i\} \\
\mathcal{K}_\omega & = & \bigcup_{i \in \mathbb{N}^+} \mathcal{K}_i
\end{array}
$$

- Fragmens fo F_ω:

System	Quantification over Types from	Types from
$F_1 = \lambda_\to$	$\emptyset$	$\{\star\}$
$F_2 = F$	$\{\star\}$	$\{\star\}$
$F_{i \geq 3}$	$\mathcal{K}_i$	$\mathcal{K}_i$
F_ω	$\mathcal{K}_\omega$	$\mathcal{K}_\omega$

Fragments of F_ω

- Higher-order type operators (system F_4 and higher) are of rare use in programming languages (if any use at all).

- But restriction to F_3 does not make things easier, because we already need the equivalence relation.

Dependent Types

- *Dependent function types* $\Pi x{:}S\,.\,T$ generalize function types $S \to T$.

- Now the result type T may depend on x:

$$\Pi n{:}\mathsf{Nat}\,.\,\mathsf{if}\ n > 3\ \mathsf{then}\ \mathsf{Nat}\ \mathsf{else}\ \mathsf{Bool}$$

- Function types correspond to the degenerate case where T does not mention x free:

$$S \to T \quad = \quad \Pi\_{:}S\,.\,T$$

Dependent Types

- In a setting with dependent types we could have:
  - $\mathsf{NatList} :: \star$ (nothing new)
  - $\mathsf{isSortedVersion} : \mathsf{NatList} \to \mathsf{NatList} \to \mathsf{Bool}$ (nothing new)
  - $\mathsf{DNatList} :: (\mathsf{NatList} \to \mathsf{Bool}) \to \star$

- Checking whether a term sort has type

  $$\mathsf{sort} : \Pi l{:}\mathsf{NatList} \,.\, \mathsf{DNatList}\ (\mathsf{isSortedVersion}\ l)$$

  now involves proving the correctnes of sort.

$\Rightarrow$ Unrestricted dependent types make typechecking a quite hard problem.

Dependent Types

- (Restricted) dependent types in practical programming languages have been studied:
  - Russel (Donahue and Demers, 1985; Hook, 1984)
  - Cayenne (Augustsson, 1998)
  - Dependent ML (Xi and Pfenning, 1998, 1999)
  - Dependent Assembly Language (Xi and Harper, 2001)
  - shape types (Jay and Sekanina, 1997)
- Dependent types are used in the design of module systems (Burstall and Lampson, 1984; MacQueen, 1986; Mitchell and Harper, 1988; Harper et al.,1990; Harper and Stone, 2000).

Dependent Types

- System LF, a *logical framework*, is the simply typed λ-calculus with dependent types (Harper, Honsell, Plotkin, 1992).
- The *calculus of constructions* (CC) is F_ω with dependent types (Coquand, Huet, 1988; Luo, 1994).
- Both have applications in theorem provers:
  - AutoMath (de Buijn, 1980)
  - NuPRL (Constable et al., 1986)
  - LEGA (Luo, Pollack, 1992; Pollack, 1994)
  - Coq (Barras et al., 1997)
  - ALF (Magnusson, Nordström, 1994)
  - ELF (Pfenning, 1994)

The λ-cube (Barendregt, 1991)

$$
\begin{array}{rcll}
\mathsf{E} & ::= & \mathsf{K} & \text{(constants)} \\
 & \mid & \mathsf{x} & \text{(variables)} \\
 & \mid & \lambda \mathsf{x}{:}\mathsf{E}\,.\,\mathsf{E} & \text{(abstraction)} \\
 & \mid & \mathsf{E}\;\mathsf{E} & \text{(application)} \\
 & \mid & \Pi \mathsf{x}{:}\mathsf{E}\,.\,\mathsf{E} & \text{(dependent function type)}
\end{array}
$$

The λ-cube (Barendregt, 1991)

- One typing relation for all sorts of expressions:
  - $\Gamma \vdash 5 : \mathsf{Nat}$
  - $\Gamma \vdash \mathsf{Nat} : \star$
  - $\Gamma \vdash \star : \Box$

The λ-cube (Barendregt, 1991)

- $\Pi \mathsf{x}{:}\mathsf{E}_1 \,.\, \mathsf{E}_2$

  Type of a function which, given an x of type E_1, returns a value of type E_2, where E_2 may depend on x.

- $\mathsf{E}_1 \to \mathsf{E}_2 \quad \stackrel{\text{def}}{=} \quad \Pi\_{:}\mathsf{E}_1 \,.\, \mathsf{E}_2$

- $\forall \alpha{:}\mathsf{E}_1 \,.\, \mathsf{E}_2 \quad \stackrel{\text{def}}{=} \quad \Pi \alpha{:}\mathsf{E}_1 \,.\, \mathsf{E}_2$
  - $\Gamma \vdash \mathsf{E}_1 : \square$
  - $\Gamma \vdash \mathsf{E}_2 : \star$

The λ-cube (Barendregt, 1991)

- Unrestricted λ-cube is equivalent to CC.
- It is possible to very easily restrinct the λ-cube.
- Judgement relation $\leadsto \subseteq \{\star, \square\} \times \{\star, \square\}$ used by typing rules:

$\star \leadsto \star$	terms can be abstracted by terms
$\square \leadsto \star$	terms can be abstracted by types
$\square \leadsto \square$	types can be abstracted by types
$\star \leadsto \square$	types can be abstracted by terms

The λ-cube (Barendregt, 1991)