Boxy types: Inference for higher-rank types and impredicativity

Dimitrios Vytiniotis[1]
Simon Peyton Jones[2]
Stephanie Weirich[1]

[1]Computer and Information Science Department
University of Pennsylvania

[2]Microsoft Research

Kalvi, October 2005

Future of FP

What will the type system of future functional programming languages look like?

- GADTs
- Poymorphic recursion
- Higher-rank
- Impredicativity
- Type-level lambdas
- Equi-recursive types
- Effects
- Dependent types

How can we reconcile HM-type inference with all of these?

Future of FP

What will the type system of future functional programming languages look like?

- GADTs
- Poymorphic recursion
- Higher-rank
- Impredicativity
- Type-level lambdas
- Equi-recursive types
- Effects
- Dependent types

How can we reconcile HM-type inference with all of these? And should we? (If not, this is the end of the talk.)

Programming in System F

There is a good chance that future programming languages will be based on System F.

Type inference for System F lacks principal types. For some terms, there is no "best" type

Programming in System F

There is a good chance that future programming languages will be based on System F.
Type inference for System F lacks principal types. For some terms, there is no "best" type
Two choices:

- Enrich type system
- Require user annotation to disambiguate

Our proposal

Boxy types:

- An extension of Haskell with higher-rank and impredicative polymorphism.
- Basic idea: propagate type annotations and contextual information using local type inference.
- Single pass, unlike Rémy's stratified type inference.

Goals

Design goal:

- Type check all Haskell code (use unification for monotypes)
- Not too fancy: use annotations for polytypes
- Reach all of System F
  - Use annotations to mark polymorphic instantiations and generalizations
- Compilation to System F (GHC core language)

Boxy Types

Idea: Make the type checker understand about "partially known and partially unknown types"

- Combine $\Gamma \vdash_{\uparrow} e : \rho$ and $\Gamma \vdash_{\downarrow} e : \rho$ into single judgment form: $\Gamma \vdash e : \rho'$.

$$
\begin{array}{lcl|lcl}
\sigma & ::= & \forall \overline{a}.\rho & \sigma' & ::= & \forall \overline{a}.\rho' \mid \boxed{\sigma} \\
\rho & ::= & \sigma \to \sigma \mid \tau & \rho' & ::= & \sigma' \to \sigma' \mid \boxed{\rho} \mid \tau \\
\tau & ::= & a \mid \tau \to \tau & & &
\end{array}
$$

- Constraints: No nested boxes, no quantified vars free inside boxes, no boxes in the type context.
- Reminiscent of coloured local type inference (Odersky, Zenger, and Zenger, 2001).

By-reference parameters

Typing judgment form: $\Gamma \vdash e : \rho'$.

- Boxes in ρ' are **filled in** by the algorithm during this call by the type checker. The rest of ρ' is **checkable** information.
- The specification includes the appropriate types that are the "output" of the algorithm.
  - If a box meets known information somewhere in the specification, then it may be filled in by a polytype.
  - If not, the box is filled in by a guessed monotype.

Examples:

- **Completely inference**: $\Gamma \vdash t : \boxed{\rho}$
- **Completely checking**: $\Gamma \vdash t : \rho$

Typing rules

- Typing rules are syntax-directed: instantiation occurs at variable occurrences, and generalization at let expressions.

$$\frac{\vdash \sigma \leq \rho' \quad x : \sigma \in \Gamma}{\Gamma \vdash x : \rho'}\ \text{VAR} \qquad \frac{\begin{array}{c}\Gamma \vdash u : \boxed{\rho} \\ \overline{a} = \mathit{ftv}(\rho) - \mathit{ftv}(\Gamma) \\ \Gamma, x : \forall \overline{a}.\rho \vdash t : \rho'\end{array}}{\Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ t : \rho'}\ \text{LET}$$

- A lot of trickyness in $\leq$, we'll get to that.
- Unbox ρ in let.

Application typing rules

$$\frac{\Gamma \vdash t : \boxed{\sigma} \to \rho' \qquad \Gamma \vdash^{poly} u : \sigma}{\Gamma \vdash t\ u : \rho'}\ \textsc{App} \qquad \frac{\Gamma \vdash t : \rho' \qquad \overline{a} \notin ftv(\Gamma)}{\Gamma \vdash^{poly} t : \forall \overline{a}.\rho'}\ \textsc{Gen1}$$

- Check the function argument type (possibly polymorphic).
- More to come for $\vdash^{poly}$.

Type annotations

Type annotations let us introduce unboxed polytypes.

$$\frac{\Gamma \vdash^{poly} u : \sigma \quad \Gamma, x : \sigma \vdash t : \rho'}{\texttt{let}\ \texttt{x} :: \sigma = \texttt{u}\ \texttt{in}\ \texttt{t} : \rho'}\ \text{SIGLET}$$

- Note: type annotations do not contain boxes
- This rule has been simplified, in the full system we support *lexically-scoped* type variables.

Abstraction rules

$$\frac{\Gamma \vdash (\lambda x.t) : \boxed{\sigma_1} \rightarrow \boxed{\sigma_2}}{\Gamma \vdash (\lambda x.t) : \boxed{\sigma_1 \rightarrow \sigma_2}} \text{ ABS1}$$

$$\frac{\vdash \sigma_1' \sim \boxed{\sigma_1} \quad \Gamma, x : \sigma_1 \vdash^{poly} t : \sigma_2'}{\Gamma \vdash (\lambda x.t) : \sigma_1' \rightarrow \sigma_2'} \text{ ABS2}$$

$$\frac{\Gamma \vdash t : \boxed{\rho}}{\Gamma \vdash^{poly} t : \boxed{\rho}} \text{ GEN2}$$

- Note higher rank
- The relation $\sim$ is *boxy-matching*.
- Don't generalize in inference mode.

Boxy matching

- The two types complement eachother.
- Symmetric, but not reflexive or transitive.
- For monotypes, an equivalence relation.
- Walk down structure of type, filling in holes on either side.

Examples:

$$\vdash \boxed{\forall a.a \to a} \sim \forall a.a \to a$$
$$\vdash \boxed{\forall a.a \to a} \to \forall a.a \to a \sim (\forall a.a \to a) \to \boxed{\forall a.a \to a}$$
$$\not\vdash \boxed{\forall a.a \to a} \sim \boxed{\forall a.a \to a}$$
$$\vdash \boxed{Int} \sim \boxed{Int}$$
$$\vdash \boxed{Int} \to \boxed{Int} \sim \boxed{Int \to Int}$$

Boxes for impredicativity

Recall the rule for variables.

$$\frac{\vdash \sigma \leq \rho' \quad x : \sigma \in \Gamma}{\Gamma \vdash x : \rho'} \text{VAR}$$

Suppose that $f : \forall a.a \to a$ in the context. Then **our goal** is:

$$\Gamma \vdash f : \boxed{\tau \to \tau} \quad \text{but not} \quad \Gamma \not\vdash f : \boxed{\sigma \to \sigma}$$

On, the other hand we should be able to **check arbitrary polytypes**:

$$\Gamma \vdash f : \sigma \to \sigma$$

So we want:

$$\forall a.a \to a \leq \boxed{\tau \to \tau} \quad \forall a.a \to a \not\leq \boxed{\sigma \to \sigma} \quad \forall a.a \to a \leq \sigma \to \sigma$$

More Examples of subsumption

- Guess monotype instantiations:

$$\vdash \forall a.a \to a \leq \boxed{Int} \to \boxed{Int}$$

$$\vdash \forall a.a \to a \leq \boxed{Int \to Int}$$

More Examples of subsumption

- Guess monotype instantiations:

$$\vdash \forall a.a \rightarrow a \leq \boxed{Int} \rightarrow \boxed{Int}$$

$$\vdash \forall a.a \rightarrow a \leq \boxed{Int \rightarrow Int}$$

- Even in result type of functions:

$$\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (\boxed{Int} \rightarrow \boxed{Int})$$

More Examples of subsumption

- Guess monotype instantiations:

$$\vdash \forall a.a \to a \leq \boxed{Int} \to \boxed{Int}$$

$$\vdash \forall a.a \to a \leq \boxed{Int \to Int}$$

- Even in result type of functions:

$$\vdash (\forall ab.a \to b) \to (\forall a.a \to a) \leq (\forall ab.a \to b) \to (\boxed{Int} \to \boxed{Int})$$

- Pull quantifiers out: $\vdash Int \to \forall a.a \to a \leq \forall a.Int \to a \to a$

More Examples of subsumption

- Guess monotype instantiations:

$$\vdash \forall a.a \rightarrow a \leq \boxed{Int} \rightarrow \boxed{Int}$$

$$\vdash \forall a.a \rightarrow a \leq \boxed{Int \rightarrow Int}$$

- Even in result type of functions:

$$\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (\boxed{Int} \rightarrow \boxed{Int})$$

- Pull quantifiers out: $\vdash Int \rightarrow \forall a.a \rightarrow a \leq \forall a.Int \rightarrow a \rightarrow a$
- Require guessed polytypes to meet known information:

$$\not\vdash \boxed{\forall a.a \rightarrow a} \leq \boxed{\forall a.a \rightarrow a} \qquad \vdash \boxed{\forall a.a \rightarrow a} \leq \forall a.a \rightarrow a$$

More Examples of subsumption

- Guess monotype instantiations:

$$\vdash \forall a.a \to a \leq \boxed{Int} \to \boxed{Int}$$

$$\vdash \forall a.a \to a \leq \boxed{Int \to Int}$$

- Even in result type of functions:

$$\vdash (\forall ab.a \to b) \to (\forall a.a \to a) \leq (\forall ab.a \to b) \to (\boxed{Int} \to \boxed{Int})$$

- Pull quantifiers out: $\vdash Int \to \forall a.a \to a \leq \forall a.Int \to a \to a$
- Require guessed polytypes to meet known information:

$$\nvdash \boxed{\forall a.a \to a} \leq \boxed{\forall a.a \to a} \qquad \vdash \boxed{\forall a.a \to a} \leq \forall a.a \to a$$

- Monotypes may be boxed $\quad \vdash \boxed{\tau} \leq \boxed{\tau}$

More Examples of subsumption

- Guess monotype instantiations:

$$\vdash \forall a.a \to a \leq \boxed{Int} \to \boxed{Int}$$

$$\vdash \forall a.a \to a \leq \boxed{Int \to Int}$$

- Even in result type of functions:

$$\vdash (\forall ab.a \to b) \to (\forall a.a \to a) \leq (\forall ab.a \to b) \to (\boxed{Int} \to \boxed{Int})$$

- Pull quantifiers out: $\vdash Int \to \forall a.a \to a \leq \forall a.Int \to a \to a$
- Require guessed polytypes to meet known information:

$$\not\vdash \boxed{\forall a.a \to a} \leq \boxed{\forall a.a \to a} \qquad \vdash \boxed{\forall a.a \to a} \leq \forall a.a \to a$$

- Monotypes may be boxed $\quad \vdash \boxed{\tau} \leq \boxed{\tau}$
- All together:

$$\vdash (\forall ab.a \to b) \to \forall a.a \to a \leq \boxed{\forall ab.a \to b} \to (Int \to Int)$$

Subsumption relation

- Defines when a type is "at least as general" as another.
- Instantiate type variables with boxy polytypes.

$$\frac{}{\vdash \tau \leq \tau}\ \text{MONO} \qquad \frac{\vdash \forall \overline{a}.\rho_1' \leq \rho_2' \quad \overline{b} \notin ftv(\forall \overline{a}.\rho_1')}{\vdash \forall \overline{a}.\rho_1' \leq \forall \overline{b}.\rho_2'}\ \text{SKOL}$$

$$\frac{\vdash [\overline{a \mapsto \boxed{\sigma}}]\rho_1' \leq \rho_2'}{\vdash \forall \overline{a}.\rho_1' \leq \rho_2'}\ \text{SPEC}$$

- More rules to come, but note, with τ instead of $\boxed{\sigma}$ this is HM subsumption relation.

Copying into boxes

When box meets non-box, the algorithm copies the information into the box.

$$\frac{}{\vdash \boxed{\sigma} \leq \sigma}\ \textsc{sboxy-simpl}$$

Copying into boxes

When box meets non-box, the algorithm copies the information into the box.

$$\frac{}{\vdash \boxed{\sigma} \leq \sigma} \text{ SBOXY-SIMPL}$$

Generalize this rule to allow boxes on the right hand side.

$$\frac{\vdash \boxed{\sigma} \sim \sigma'}{\vdash \boxed{\sigma} \leq \sigma'} \text{ SBOXY}$$

A subtle point

- What if we add this (suggestively-named) rule:

$$\frac{\vdash \sigma' \sim \boxed{\sigma}}{\vdash \sigma' \leq \boxed{\sigma}}\ \text{SBOXY-WRONG}$$

A subtle point

- What if we add this (suggestively-named) rule:

$$\frac{\vdash \sigma' \sim \boxed{\sigma}}{\vdash \sigma' \leq \boxed{\sigma}}\ \textsc{sboxy-wrong}$$

- Overlap between SBOXY-WRONG and SPEC. If a polytype meets a box, what should we do?

$$\frac{\vdash [\overline{a \mapsto \boxed{\sigma}}]\rho_1' \leq \rho_2'}{\vdash \forall\overline{a}.\rho_1' \leq \rho_2'}\ \textsc{spec}$$

Can't restrict spec

Could restrict SPEC so that the RHS cannot be a box:

$$\frac{\vdash [\overline{a \mapsto \boxed{\sigma}}]\rho_1' \leq \rho_2' \quad \rho_2' \neq \boxed{\rho}}{\vdash \forall \overline{a}.\rho_1' \leq \rho_2'} \text{ SPEC-NOBOX}$$

but then we would lose some Haskell programs:

$$id : \forall a.a \to a \vdash id : \boxed{Int \to Int}$$

requires $\vdash \forall a.a \to a \leq \boxed{Int \to Int}$

Tension between higher-rank and impredicativity

Standard subsumption rule for higher-rank types:

$$\frac{\vdash \sigma_3' \geq \sigma_1' \quad \vdash \sigma_2' \leq \sigma_4'}{\vdash \sigma_1' \to \sigma_2' \leq \sigma_3' \to \sigma_4'}\ \text{F2}$$

But we aren't going to use this rule.

Subsumption and function types

Want to encode all of System F type instantiations using type annotations.

- Need $\vdash \forall \overline{a}.\rho \leq \rho[\overline{\sigma}/a]$
- $\textsc{spec}$ introduces boxes on the left. If we are to fill them, they better stay on the left.
- Invariance for the argument of a function type.

$$\frac{\vdash \sigma_3' \sim \sigma_1' \qquad \vdash \sigma_2' \leq \sigma_4'}{\vdash \sigma_1' \to \sigma_2' \leq \sigma_3' \to \sigma_4'}\ \textsc{f2}$$

- Essential to show:

$$\forall a.a \to a \leq (\forall a.a \to a) \to \forall a.a \to a$$

Properties of the type system

- Type-safety through translation to System F.
- Algorithm computes principal types.
- Type system extends Hindley-Milner.
- Monotypes can be unboxed/boxed arbitrarily. Unification takes care of that.
- Can embed System F.

Expressiveness

There are several programs that *don't* typecheck, that we really would like to.

For example:

$$id : \forall a.a \rightarrow a$$

$$sing : \forall a.a \rightarrow [a]$$

Even if we know the result type:

$$\Gamma \not\vdash sing\ \ id : [\forall a.a \rightarrow a]$$

This requires that:

$$\vdash \forall a.a \rightarrow [a] \leq \boxed{\forall a.a \rightarrow a} \rightarrow [\forall a.a \rightarrow a]$$

Smart application

We have been exploring alternative rules for application.

$$\frac{\begin{array}{c} x : \forall \overline{a}.\overline{\sigma} \rightarrow \sigma \in \Gamma \\ \overline{a}_c = \overline{a} \cap ftv(\sigma) \qquad \overline{a}_e = \overline{a} - \overline{a}_c \\ \vdash [\overline{a_c \mapsto \boxed{\sigma_c}}]\sigma \leq \rho' \\ \Gamma \vdash^{poly} u_i : [\overline{a_e \mapsto \boxed{\sigma_e}}, \overline{a_c \mapsto \sigma_c}]\sigma_i \end{array}}{\Gamma \vdash x\ \overline{u} : \rho'}$$

Smart application

We have been exploring alternative rules for application.

$$\frac{\begin{array}{c} x : \forall \overline{a}.\overline{\sigma} \to \sigma \in \Gamma \\ \overline{a}_c = \overline{a} \cap ftv(\sigma) \qquad \overline{a}_e = \overline{a} - \overline{a}_c \\ \vdash [\overline{a_c \mapsto \boxed{\sigma_c}}]\sigma \leq \rho' \\ \Gamma \vdash^{poly} u_i : [\overline{a_e \mapsto \boxed{\sigma_e}}, \overline{a_c \mapsto \sigma_c}]\sigma_i \end{array}}{\Gamma \vdash x\ \overline{u} : \rho'}$$

Not quite satisfactory:

- Completeness problem
- Can't typecheck $\Gamma \vdash hd\ ids : a \to a$

Questions

- Is this the right tradeoff between expressiveness and simplicity?
- Stratified vs. monolithic type inference?
- Is there a different strategy altogether?

More questions

- Is System F the right "core" language?
- Can the user understand when the program type checks? "Simple" specification vs. powerful inference vs. good error messages?
- Is it easy to modify programs if there are a lot of type annotations all over the place?
- Why is thinking about type inference addictive?

More information

Draft paper available at:

`www.cis.upenn.edu/~dimitriv/boxy`

Revision appearing soon.