Constraint-based type inference for GADTs

Vincent Simonet, François Pottier

November 16, 2004

INRIA

Algebraic data types

The data constructors associated with an ordinary algebraic data type constructor ε receive type schemes of the form:

$$K :: \forall \bar{a}.\tau_1 \cdots \tau_n \longrightarrow \varepsilon(\bar{a})$$

For instance,

$$\begin{aligned} Leaf &:: \forall a.tree(a) \\ Node &:: \forall a.tree(a) \cdot a \cdot tree(a) \longrightarrow tree(a) \end{aligned}$$

Matching a value of type $tree(a)$ against the pattern $Node(l, v, r)$ binds l, v, and r to values of types $tree(a)$, a, and $tree(a)$.

Läufer-Odersky-style existential types

In Läufer and Odersky's extension of Hindley and Milner's type system with existential types, the data constructors receive type schemes of the form:

$$K :: \forall \bar{\alpha}\bar{\beta}.\tau_1 \cdots \tau_n \longrightarrow \varepsilon(\bar{\alpha})$$

For instance,

$$Key :: \forall \beta.\beta \cdot (\beta \longrightarrow int) \longrightarrow key$$

Matching a value of type *key* against the pattern $Key\,(v, f)$ binds v and f to values of type β and $\beta \longrightarrow int$, for an unknown β.

Guarded algebraic data types

Let us now assign *constrained* type schemes to data constructors:

$$K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \rightarrow \varepsilon(\bar{\alpha})$$

Matching a value of type $\varepsilon(\bar{\alpha})$ against the pattern $K\, x_1 \cdots x_n$ binds x_i to a value of type τ_i, *for some unknown types $\bar{\beta}$ that satisfy the constraint D*.

In general, constraints may be arbitrary first-order formulæ involving basic predicates on types such as type equality, subtyping, membership in a type class, etc.

Guarded algebraic data types (cont'd)

Let

$$K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \rightarrow \varepsilon(\bar{\alpha})$$

In the clause $(K\, x_1 \cdots x_n).e$, the expression e is typechecked under the assumption that $\bar{\beta}$ is unknown, but D holds.

Thus, two phenomena arise:

- D may mention $\bar{\beta}$, so the types $\bar{\beta}$ are *partially abstract*;
- D may mention $\bar{\alpha}$, so the success of a *dynamic* test yields extra *static* type information.

GADTs in the setting of equality constraints

In the simplest case, constraints are made of type equations:

$$\begin{array}{rcl} \tau & ::= & a \mid \tau \rightarrow \tau \mid \varepsilon(\tau, \ldots, \tau) \\ C, D & ::= & (\tau = \tau) \mid C \wedge C \mid \exists a.C \mid \neg C \end{array}$$

Without loss of expressiveness, data constructors may then receive *unconstrained* type schemes:

$$K :: \forall \bar{\beta}.\tau_1 \cdots \tau_n \rightarrow \varepsilon(\bar{\tau})$$

A typical example

For instance, following Crary, Weirich, and Morrisett, one might declare a *singleton* type of runtime type descriptors:

$$\begin{aligned} Int &:: ty(int) \\ Pair &:: \forall \beta_1 \beta_2 . ty(\beta_1) \cdot ty(\beta_2) \rightarrow ty(\beta_1 \times \beta_2) \end{aligned}$$

This may also be written

$$\begin{aligned} Int &:: \forall \alpha[\alpha = int].ty(\alpha) \\ Pair &:: \forall \alpha \beta_1 \beta_2 [\alpha = \beta_1 \times \beta_2].ty(\beta_1) \cdot ty(\beta_2) \rightarrow ty(\alpha) \end{aligned}$$

A typical example (cont'd)

Runtime type descriptors allow defining *generic* functions:

let rec print : $\forall a.ty(a) \rightarrow a \rightarrow unit$ **= fun t** $\rightarrow$
  match t with
  | **Int** $\rightarrow$
      (* a **is** int *)
      print_int
  | **Pair (t1, t2)** $\rightarrow$
      (* a **is** $\beta_1 \times \beta_2$ *)
      fun (x1, x2) $\rightarrow$
        print t1 x1; print_string " * "; print t2 x2

The two branches have *incompatible* types $int \rightarrow unit$ and $\beta_1 \times \beta_2 \rightarrow unit$, but *they also have a common type*, namely $a \rightarrow unit$.

Other applications in the setting of equality

Applications of GADTs include:

- *Generic programming* (Xi, Cheney and Hinze)
- *Tagless interpreters* (Xi, Sheard)
- *Tagless automata* (Pottier and Régis-Gianas)
- Type-preserving *defunctionalization* (Pottier and Gauthier)
- and more...

GADTs allow inductive definitions of predicates on types, that is, they allow embedding *proofs* (about types) into values.

Beyond equality

Constraints may involve

- *Presburger arithmetic* (Xi's Dependent ML)
- *complex polynomials* (Zenger's indexed types)
- *subtyping* (runtime security levels à la Tse and Zdancewic)
- and more: what about *type class membership* assertions?

Xi and Zenger *refine* Hindley and Milner's type system. Instead, we *extend* it.

Introduction

HM(*X*)

HMG(*X*)

Some design choices

Why constraints?

Constraints are useful for two reasons:

- they help specify type inference in a *modular, declarative* way.
- constraints need not be equations; they are *more general*.

In this talk, I assume that constraints are built on top of equations, so as to remain in the spirit of Hindley and Milner's type system. The second motive vanishes; the first one remains.

The type system HM(X)

We choose HM(X) as a starting point because it is the most elegant constraint-based presentation of Hindley and Milner's type system.

HM(X) assigns *constrained type schemes* to expressions:

$$\sigma ::= \forall \bar{a}[C].\tau$$

The two facets of HM(X)

HM(X) comes with a *logic* specification, that is, a set of deduction rules for typing judgments of the form

$$C, \Gamma \vdash e : \sigma$$

HM(X) also comes with a *functional* specification, that is, an inductively defined mapping that takes every pre-judgement $\Gamma \vdash e : \sigma$ to a constraint $(\!|\Gamma \vdash e : \sigma|\!)$.

This mapping is also known as a *constraint generator*.

The two facets of HM(X) (cont'd)

The two specifications are related by the following

Theorem

$C, \Gamma \vdash e : \sigma$ is equivalent to $C \Vdash (\!|\Gamma \vdash e : \sigma|\!)$.

This is the analogue of the *principal types* theorem in Hindley and Milner's type system.

Deciding whether a (closed) program *e is well-typed* reduces to deciding whether the (closed) constraint *$\exists a.(\!|\varnothing \vdash e : a|\!)$ is true.*

The logic facet of HM(X)

The syntax-directed rules are as follows:

$$\text{Var}\quad \frac{\Gamma(x) = \sigma \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma}$$

$$\text{Abs}\quad \frac{C, \Gamma[x \mapsto \tau'] \vdash e : \tau}{C, \Gamma \vdash \lambda x.e : \tau' \longrightarrow \tau}$$

$$\text{App}\quad \frac{C, \Gamma \vdash e_1 : \tau' \longrightarrow \tau \qquad C, \Gamma \vdash e_2 : \tau'}{C, \Gamma \vdash e_1\, e_2 : \tau}$$

$$\text{Fix}\quad \frac{C, \Gamma[x \mapsto \sigma] \vdash v : \sigma}{C, \Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).v : \sigma}$$

$$\text{Let}\quad \frac{C, \Gamma \vdash e_1 : \sigma' \qquad C, \Gamma[x \mapsto \sigma'] \vdash e_2 : \sigma}{C, \Gamma \vdash \mathbf{let}\; x = e_1 \;\mathbf{in}\; e_2 : \sigma}$$

In this talk, in Fix, we require σ to be of the form $\forall\bar{\alpha}.\tau$. Users do not have access to constraints.

The logic facet of HM(X) (cont'd)

There are also a few non-syntax-directed rules:

Gen
$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{a} \# \mathrm{ftv}(\Gamma, C)}{C \wedge \exists \bar{a}.D, \Gamma \vdash e : \forall \bar{a}[D].\tau}$$

Inst
$$\frac{C, \Gamma \vdash e : \forall \bar{a}[D].\tau \qquad C \Vdash D}{C, \Gamma \vdash e : \tau}$$

Sub
$$\frac{C, \Gamma \vdash e : \tau' \qquad C \Vdash \tau' \leq \tau}{C, \Gamma \vdash e : \tau}$$

Hide
$$\frac{C, \Gamma \vdash e : \sigma \qquad \bar{a} \# \mathrm{ftv}(\Gamma, \sigma)}{\exists \bar{a}.C, \Gamma \vdash e : \sigma}$$

In this talk, $\leq$ is interpreted as equality.

The functional facet of HM(X)

The constraint generator is defined as follows:

$$
\begin{aligned}
(\!|\Gamma \vdash x : \tau|\!) &= \Gamma(x) \leq \tau \\
(\!|\Gamma \vdash \lambda x.e : \tau|\!) &= \exists a_1 a_2.(\tau = a_1 \to a_2 \wedge (\!|\Gamma[x \mapsto a_1] \vdash e : a_2|\!)) \\
(\!|\Gamma \vdash e_1\, e_2 : \tau|\!) &= \exists a.((\!|\Gamma \vdash e_1 : a \to \tau|\!) \wedge (\!|\Gamma \vdash e_2 : a|\!)) \\
(\!|\Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).v : \tau|\!) &= \exists\bar{\beta}.((\!|\Gamma[x \mapsto \sigma] \vdash v : \sigma|\!) \wedge \sigma \leq \tau) \\
(\!|\Gamma \vdash \mathbf{let}\; x = e_1 \;\mathbf{in}\; e_2 : \tau|\!) &= (\!|\Gamma[x \mapsto \forall a[(\!|\Gamma \vdash e_1 : a|\!)].a] \vdash e_2 : \tau|\!)
\end{aligned}
$$

Constraints of the form $\sigma \leq \tau$ are interpreted as follows:

$$(\forall \bar{a}[C].\tau) \leq \tau' = \exists \bar{a}.(C \wedge \tau \leq \tau')$$

The treatment of fixpoints relies on the following notation:

$$(\!|\Gamma \vdash e : \forall \bar{a}.\tau|\!) = \forall \bar{a}.(\!|\Gamma \vdash e : \tau|\!)$$

The functional facet of HM(X) (cont'd)

The constraint $(\!|\Gamma \vdash e : \tau|\!)$ is in the following grammar:

$$C ::= (\tau = \tau) \mid C \wedge C \mid \exists a.C \mid \forall a.C$$

We have not used implication or negation. Constraint solving amounts to *first-order unification under a mixed prefix*.

Patterns

The calculus is extended with data constructors K, patterns p, and functions defined by cases.

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda \bar{c} \mid K\,\bar{e} \mid e\,e \mid \mu x.v \mid \mathsf{let}\, x = e \,\mathsf{in}\, e \\
c & ::= & p.e \\
p & ::= & 0 \mid 1 \mid x \mid p \wedge p \mid p \vee p \mid K\,\bar{p}
\end{array}
$$

The operational semantics is extended by defining an extended substitution $[p \mapsto v]$ which is either undefined or a mapping of the variables bound by p to values.

There is a (classic) *catch!* This semantics states that matching, say, an integer value against a pair pattern is legal—it just doesn't match. Yet, most compilers implement a semantics where dereferencing an integer is illegal.

Typechecking expressions

The specification is extended with new rules for data constructors and function definitions by cases.

Cstr

$$\frac{\forall i \quad C, \Gamma \vdash e_i : \tau_i \qquad K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \longrightarrow \varepsilon(\bar{\alpha}) \qquad C \Vdash D}{C, \Gamma \vdash K\, e_1 \cdots e_n : \varepsilon(\bar{\alpha})}$$

Abs

$$\frac{\forall i \quad C, \Gamma \vdash c_i : \tau}{C, \Gamma \vdash \lambda(c_1 \cdots c_n) : \tau}$$

Clause

$$\frac{C \vdash p : \tau' \rightsquigarrow \exists \bar{\beta}[D]\Gamma' \qquad C \wedge D, \Gamma\Gamma' \vdash e : \tau \qquad \bar{\beta} \# \mathrm{ftv}(C, \Gamma, \tau)}{C, \Gamma \vdash p.e : \tau' \longrightarrow \tau}$$

Typechecking patterns

Typing judgments about patterns are written

$$C \vdash p : \tau \rightsquigarrow \Delta$$

where *environment fragments* are defined by

$$\Delta ::= \exists\bar{\beta}[D]\Gamma$$

For instance, here are two valid judgments:

$$\mathsf{true} \vdash \mathit{Int} : \mathit{ty}(a) \rightsquigarrow \exists\varnothing[a = \mathit{int}]\varnothing$$

$$\mathsf{true} \vdash \mathit{Pair}\,(t_1, t_2) : \mathit{ty}(a) \rightsquigarrow \exists\beta_1\beta_2[a = \beta_1 \times \beta_2](t_1 : \mathit{ty}(\beta_1); t_2 : \mathit{ty}(\beta_2))$$

Operations on environment fragments

These will be used in the following slides...

Qualification: $\exists\bar{\alpha}[C]\Delta$
$\exists\bar{\alpha}[C](\exists\bar{\beta}[D]\Gamma) = \exists\bar{\alpha}\bar{\beta}[C \wedge D]\Gamma$

Conjunction: $\Delta_1 \times \Delta_2$ (where Δ_1 and Δ_2 have disjoint domains)
$(\exists\bar{\beta}_1[D_1]\Gamma_1) \times (\exists\bar{\beta}_2[D_2]\Gamma_2) = \exists\bar{\beta}_1\bar{\beta}_2[D_1 \wedge D_2](\Gamma_1 \times \Gamma_2)$

Disjunction: $\Delta_1 + \Delta_2$ (where Δ_1 and Δ_2 have a common domain)
$(\exists\bar{\beta}_1[D_1]\Gamma_1) + (\exists\bar{\beta}_2[D_2]\Gamma_2) = \exists\bar{\beta}_1\bar{\beta}_2\bar{\alpha}[(D_1 \wedge \Gamma \leq \Gamma_1) \vee (D_2 \wedge \Gamma \leq \Gamma_2)]\Gamma$

Side conditions omitted.

Typechecking patterns (cont'd)

p-Empty

$$C \vdash 0 : \tau \leadsto \exists\varnothing[\textsf{false}]\varnothing$$

p-Wild

$$C \vdash 1 : \tau \leadsto \exists\varnothing[\textsf{true}]\varnothing$$

p-Var

$$C \vdash x : \tau \leadsto \exists\varnothing[\textsf{true}](x \mapsto \tau)$$

p-And

$$\frac{\forall i \quad C \vdash p_i : \tau \leadsto \Delta_i}{C \vdash p_1 \wedge p_2 : \tau \leadsto \Delta_1 \times \Delta_2}$$

p-Or

$$\frac{\forall i \quad C \vdash p_i : \tau \leadsto \Delta}{C \vdash p_1 \vee p_2 : \tau \leadsto \Delta}$$

Typechecking patterns (cont'd)

The key typechecking rule for patterns is

p-Cstr

$$\frac{\forall i \quad C \wedge D \vdash p_i : \tau_i \rightsquigarrow \Delta_i \qquad K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \rightarrow \varepsilon(\bar{\alpha}) \qquad \bar{\beta} \# \mathrm{ftv}(C)}{C \vdash K\, p_1 \cdots p_n : \varepsilon(\bar{\alpha}) \rightsquigarrow \exists \bar{\beta}[D](\Delta_1 \times \cdots \times \Delta_n)}$$

Typechecking patterns (cont'd)

We also need a few non-syntax-directed rules:

p-EqIn
$$\frac{C \vdash p : \tau' \leadsto \Delta \qquad C \Vdash \tau = \tau'}{C \vdash p : \tau \leadsto \Delta}$$

p-SubOut
$$\frac{C \vdash p : \tau \leadsto \Delta' \qquad C \Vdash \Delta' \leq \Delta}{C \vdash p : \tau \leadsto \Delta}$$

p-Hide
$$\frac{C \vdash p : \tau \leadsto \Delta \qquad \bar{a} \# \mathrm{ftv}(\tau, \Delta)}{\exists \bar{a}.C \vdash p : \tau \leadsto \Delta}$$

This completes the logic specification of HMG(X).

Typechecking patterns: examples

The following are valid derivations:

$$\dfrac{}{\textbf{true} \vdash \mathit{Int} : ty(a) \leadsto \exists\varnothing[a = int]\varnothing}\ \text{p-Cstr}$$

$$\dfrac{\forall i \in \{1,2\} \qquad \dfrac{}{a = \beta_1 \times \beta_2 \vdash t_i : ty(\beta_i) \leadsto (t_i : ty(\beta_i))}\ \text{p-Var}}{\begin{array}{c}\textbf{true} \vdash \mathit{Pair}\,(t_1, t_2) : ty(a) \leadsto \\ \exists\beta_1\beta_2[a = \beta_1 \times \beta_2](t_1 : ty(\beta_1); t_2 : ty(\beta_2))\end{array}}\ \text{p-Cstr}$$

Recall that

$$\mathit{Int} :: \forall a[a = int].ty(a)$$

$$\mathit{Pair} :: \forall a\beta_1\beta_2[a = \beta_1 \times \beta_2].ty(\beta_1) \cdot ty(\beta_2) \longrightarrow ty(a)$$

Typechecking clauses: examples

Here is a derivation for the first clause of *print*:

$$
\dfrac{(\ldots) \qquad \dfrac{\dfrac{(\ldots)}{a = int, \Gamma \vdash print\_int : int \longrightarrow unit} \quad a = int \Vdash int \longrightarrow unit \leq a \longrightarrow unit}{a = int, \Gamma \vdash print\_int : a \longrightarrow unit}\,\text{Sub}}{\mathbf{true}, \Gamma \vdash Int.print\_int : ty(a) \longrightarrow a \longrightarrow unit}\,\text{Clause}
$$

Typechecking clauses: examples (cont'd)

Here is a derivation for the second clause:

$$
\dfrac{(\ldots) \qquad \dfrac{\dfrac{(\ldots)}{\begin{array}{c}\ldots, \Gamma; t_1 : ty(\beta_1); t_2 : ty(\beta_2) \vdash \\ \lambda \ldots : \beta_1 \times \beta_2 \longrightarrow unit\end{array}} \quad a = \beta_1 \times \beta_2 \Vdash \beta_1 \times \beta_2 \longrightarrow unit \leq a \longrightarrow unit}{\begin{array}{c}a = \beta_1 \times \beta_2, \Gamma; t_1 : ty(\beta_1); t_2 : ty(\beta_2) \vdash \\ \lambda \ldots : a \longrightarrow unit\end{array}}\;\text{Sub}}{\begin{array}{c}\mathbf{true}, \Gamma \vdash Pair\,(t_1, t_2).\lambda(x_1, x_2).(print\, t_1\, x_1; print\, t_2\, x_2) : \\ ty(a) \longrightarrow a \longrightarrow unit\end{array}}\;\text{Clause}
$$

Type soundness

Theorem

If e is well-typed and contains exhaustive case analyses only, then it does not go wrong.

Nonexhaustive case analyses are accepted, provided the typechecker can prove that all omitted branches are dead—details in the paper.

The functional facet of HMG(X)

Two new rules govern applications of data constructors and function definitions by cases (clauses):

$$(\!|\Gamma \vdash K\, e_1 \cdots e_n : \tau|\!) \;=\; \exists \bar{\alpha}\bar{\beta}.(\wedge_i (\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar{\alpha}) \leq \tau)$$
$$\text{where } K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \rightarrow \varepsilon(\bar{\alpha})$$

$$(\!|\Gamma \vdash p.e : \tau_1 \rightarrow \tau_2|\!) \;=\; (\!|p \downarrow \tau_1|\!) \wedge \forall \bar{\beta}.D \Rightarrow (\!|\Gamma\Gamma' \vdash e : \tau_2|\!)$$
$$\text{where } \exists \bar{\beta}[D]\Gamma' \text{ is } (\!|p \uparrow \tau_1|\!)$$

GADTs demand universal quantification (already required for Läufer-Odersky-style existential types) and *implication*.

Preconditions for patterns

The constraint $(\!|p \downarrow \tau|\!)$ asserts that it is *legal* to match a value of type τ against p.

$$
\begin{aligned}
(\!|0 \downarrow \tau|\!) &= \mathsf{true} \\
(\!|1 \downarrow \tau|\!) &= \mathsf{true} \\
(\!|x \downarrow \tau|\!) &= \mathsf{true} \\
(\!|p_1 \wedge p_2 \downarrow \tau|\!) &= (\!|p_1 \downarrow \tau|\!) \wedge (\!|p_2 \downarrow \tau|\!) \\
(\!|p_1 \vee p_2 \downarrow \tau|\!) &= (\!|p_1 \downarrow \tau|\!) \wedge (\!|p_2 \downarrow \tau|\!) \\
(\!|K\, p_1 \cdots p_n \downarrow \tau|\!) &= \exists \bar{\alpha}.(\varepsilon(\bar{\alpha}) = \tau \wedge \forall \bar{\beta}.D \Rightarrow \wedge_i (\!|p_i \downarrow \tau_i|\!)) \\
&\quad \textit{where } K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \rightarrow \varepsilon(\bar{\alpha})
\end{aligned}
$$

Postconditions for patterns

The environment fragment $(\!| p \uparrow \tau |\!)$ represents the *extra knowledge* obtained by *successfully* matching a value of type τ against p.

$$
\begin{aligned}
(\!| 0 \uparrow \tau |\!) &= \exists \varnothing [\mathsf{false}] \varnothing \\
(\!| 1 \uparrow \tau |\!) &= \exists \varnothing [\mathsf{true}] \varnothing \\
(\!| x \uparrow \tau |\!) &= \exists \varnothing [\mathsf{true}] (x \mapsto \tau) \\
(\!| p_1 \wedge p_2 \uparrow \tau |\!) &= (\!| p_1 \uparrow \tau |\!) \times (\!| p_2 \uparrow \tau |\!) \\
(\!| p_1 \vee p_2 \uparrow \tau |\!) &= (\!| p_1 \uparrow \tau |\!) + (\!| p_2 \uparrow \tau |\!) \\
(\!| K\, p_1 \cdots p_n \uparrow \tau |\!) &= \exists \bar{\alpha} \bar{\beta} [\varepsilon(\bar{\alpha}) = \tau \wedge D] (\times_i (\!| p_i \uparrow \tau_i |\!)) \\
&\quad \textit{where } K :: \forall \bar{\alpha} \bar{\beta} [D] . \tau_1 \cdots \tau_n \rightarrow \varepsilon(\bar{\alpha})
\end{aligned}
$$

The two facets of HMG(X)

The two specifications are related by the same theorem as in HM(X):

Theorem

$C, \Gamma \vdash e : \sigma$ is equivalent to $C \Vdash (\!|\Gamma \vdash e : \sigma|\!)$.

The *print* example

The constraint associated with *print* is as follows:

$$(\!|\Gamma_0 \vdash \mu print.\ldots : \forall a.ty(a) \to a \to unit|\!)$$

$$\equiv$$

$$\forall a.\left(\begin{array}{rcl} a = int & \Rightarrow & (\!|\Gamma \vdash print\_int : a \to unit|\!) \\ \wedge\, \forall \beta_1\beta_2.a = \beta_1 \times \beta_2 & \Rightarrow & (\!|\Gamma \vdash \lambda \ldots : a \to unit|\!) \end{array}\right)$$

Have we got carried away?

The constraint $(\!|\Gamma \vdash e : \tau|\!)$ is now in the *first-order theory of equality*, whose satisfiability problem is decidable, but of nonelementary complexity.

A restriction

By requiring functions that analyze GADTs to be explicitly annotated with *closed type schemes*, we are able to generate *tractable* constraints, where all implications are of the form

$$\forall \bar{\beta}.C_1 \Rightarrow C_2 \qquad \text{where } \mathrm{ftv}(C_1) \subseteq \bar{\beta}$$

These are (very) easy to solve and have most general unifiers.

This restriction is *stronger than we'd like*. Also, the details are not particularly elegant.

Introduction

HM(X)

HMG(X)

Some design choices

Are patterns evaluated left-to-right?

This uncurried version of *print* is *rejected*:

```
let rec print : ∀a.ty(a) × a → unit = fun tx →
  match tx with
  | (Int, x) →
      print_int x
  | (Pair (t1, t2), (x1, x2)) →
      print t1 x1; print_string " * "; print t2 x2
```

The pattern **(x1, x2)** is not legal until the second component of **tx** is known to be a pair, that is, until the pattern **Pair (t1, t2)** is *deemed* successful.

Are patterns evaluated left-to-right? (cont'd)

The uncurried version of *print* is *accepted if modified* as follows:

```
let rec print : ∀a.ty(a) × a → unit = fun tx →
  match tx with
  | (Int, x) →
      print_int x
  | (Pair (t1, t2), x) →
      let (x1, x2) = x in
      print t1 x1; print_string " * "; print t2 x2
```

One could modify HMG(X) to accept both versions, provided left-to-right evaluation of patterns is explicitly specified.

Precise treatment of disjunction

In HMG(X), the clause

$$(K_1\ x) \vee (K_2\ x).e$$

is well-typed if and only if both $(K_1\ x).e$ and $(K_2\ x).e$ are.

This is *not true in ocaml*, where both occurrences of x in $(K_1\ x) \vee (K_2\ x)$ must have the same type.

HMG(X) is more expressive and more expensive.

A pathological case

```
type T : * → * where
  K1 : T bool | K2 : T int

let f (x : T a) y =
  match x with K1 → y + 1 | K2 → not y
```

The (inferred) principal type of **f** is

$$\forall \alpha\beta[(\alpha = bool \Rightarrow \beta = int) \wedge (\alpha = int \Rightarrow \beta = bool)].T\ \alpha \rightarrow \beta \rightarrow \beta$$

Probably overwhelming! Also, the programmer has *no way* of specifying the type of **y**.

A pathological case (cont'd)

Imagine we instead have

```
type T : * → * where
  K1 : T int | K2 : T bool
```

The principal type of **f** is then

$$\forall \alpha\beta[(\alpha = int \Rightarrow \beta = int) \wedge (\alpha = bool \Rightarrow \beta = bool)].T\ \alpha \rightarrow \beta \rightarrow \beta$$

which, *in a syntactic interpretation of constraints*, is equivalent to

$$\forall \alpha\beta[\beta = \alpha].T\ \alpha \rightarrow \beta \rightarrow \beta$$

$$\forall \alpha.T\ \alpha \rightarrow \alpha \rightarrow \alpha$$

Conclusion

- HMG(X) is a *sound* and expressive type system.
- It enjoys a reduction from type inference to *constraint solving*.
- The system must be *restricted* for tractability and simplicity.

A prototype version of HMG(=) has been implemented by Yann Régis-Gianas.

Selected References

- Vincent Simonet, François Pottier.
  Constraint-Based Type Inference with Guarded Algebraic Data Types.
  Submitted, 2004.

- Hongwei Xi.
  Applied Type System.
  TYPES 2003.

- Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich.
  Wobbly types: type inference for generalised algebraic data types.
  Draft, 2004.