Compiling functional languages

http://www.cse.chalmers.se/edu/year/2011/course/CompFun/

Lecture 5
Haskell-style overloading

Johan Nordlander

Some terminology

- Parametric polymorphism: a term is defined not to depend on (parts of) its type

```
length :: [a] -> Int          = ...
```

- Ad hoc polymorphism: a term is defined differently on basis of (parts of) its type

```
(+) :: Int->Int->Int          = ...
(+) :: Float->Float->Float    = ...
```

- The latter is also known as overloading

Going ad hoc

- Parametric polymorphism is neatly captured in the Hindley/Milner system
- Ad hoc polymorphism can be treated separately (Standard ML), but that leads to problems. E.g., consider the type of

```
f x y = x + y
```

- Haskell addresses the ad hoc problem by the introduction of type classes. For example:

```
f :: Num a => a -> a -> a
```

Type classes

- The basic idea behind "making ad hoc polymorphism less ad hoc" is to constrain polymorphism by means of class predicates like Num a, Eq a, etc
- The idea was suggested by Wadler&Blott in 1989, but its theory was primarily developed by Jones (A Theory of Qualified Types, 1992)

Qualified types

- Compared to the Hindley/Milner system, the basic difference is that judgements are extended with qualifying predicates:

  $P \mid A \vdash e : \sigma$

- Read: e has type σ in scope A, provided all predicates in P are true
- Qualifying predicates, a.k.a. the "context"

  $P ::= q_1, \ldots, q_n$ $\qquad$ $q ::= C\ t_1 \ldots t_m$

  $\sigma ::= t \mid \forall a\ .\ \sigma \mid q \Rightarrow \sigma$

Qualified types

$$\frac{P \mid A \vdash e : t' \to t \qquad P \mid A \vdash e' : t'}{P \mid A \vdash e\ e' : t}\ \text{App}$$

$$\frac{P \mid A, x{:}t' \vdash e : t}{P \mid A \vdash \backslash x \to e : t' \to t}\ \text{Abs}$$

$$\frac{x{:}\sigma \in A}{P \mid A \vdash x : \sigma}\ \text{Var}$$

$$\frac{P \mid A \vdash e : \sigma \qquad P \mid A, x{:}\sigma \vdash e' : t}{P \mid A \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : t}\ \text{Let}$$

merge

$$\frac{P \mid A \vdash e : \forall a.\sigma}{P \mid A \vdash e : [t/a]\sigma}\ \text{Inst}$$

$$\frac{P \mid A \vdash e : \sigma \qquad a \notin fv(A,P)}{P \mid A \vdash e : \forall a.\sigma}\ \text{Gen}$$

merge

$$\frac{P \mid A \vdash e : q \Rightarrow \sigma \qquad P \Vdash q}{P \mid A \vdash e : \sigma}\ \text{Entail}$$

$$\frac{P, q \mid A \vdash e : \sigma}{P \mid A \vdash e : q \Rightarrow \sigma}\ \text{Qual}$$

Qualified types

$$\frac{P \mid A \vdash e : t' \to t \qquad P \mid A \vdash e' : t'}{P \mid A \vdash e\ e' : t}\ \text{App}$$

$$\frac{P \mid A, x{:}t' \vdash e : t}{P \mid A \vdash \backslash x \to e : t' \to t}\ \text{Abs}$$

$$\frac{x{:}\forall as.qs \Rightarrow t \in A \qquad P \Vdash [ts/as]qs}{P \mid A \vdash x : [ts/as]t}\ \text{Var}$$

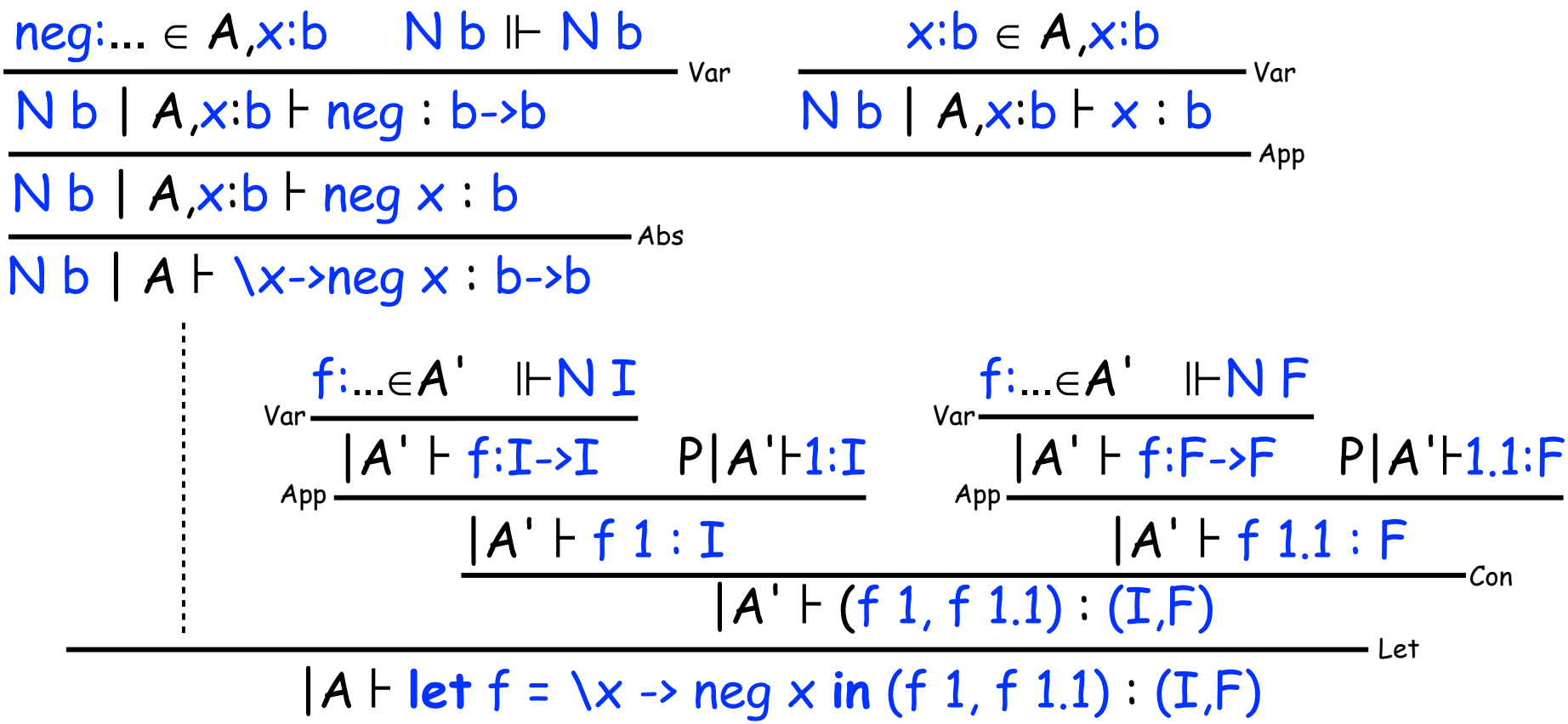
$$\frac{qs \mid A \vdash e : t \qquad P \mid A, x{:}\sigma \vdash e' : t'}{P \mid A \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : t'}\ \text{Let}$$

where $\sigma = gen(qs \Rightarrow t, A)$

$= \forall\ fv(qs \Rightarrow t) \backslash fv(A)\ .\ qs \Rightarrow t$

(Type schemes: $\sigma ::= \forall as\ .\ qs \Rightarrow t$)

Example



where A = neg:∀a. N a => a->a A' = A, f:∀b. N b => b->b

Entailment

- Read each q as a truth statement on types
  - Num a — a is a numeric type
  - Collection c a — c and a are in the collection relation
- Relation $P \Vdash qs$ reads "P entails qs" or

  "each $q \in qs$ is true if all $q' \in P$ are true"
- Simplistic interpretation for now:

  $P \Vdash qs$ if $qs \subseteq P, P_0$

  where P_0 are the top-level instance "facts"
- But what does it mean for a q to be true?
- How do we compile overloaded code?

Witness translation

- Each **class** C as **where** $x_i :: t_i$ translates into

  data C as = $C\ t_1 \ldots t_n$ — a witness type

  $x_1\ (C\ y_1 \ldots y_n) = y_1$ — projection functions

  ...

  $x_n\ (C\ y_1 \ldots y_n) = y_n$

- Each **instance** C ts **where** $x_i = e_i$ translates into

  wit :: C ts — a witness term (fact)

  wit = $C\ e_1 \ldots e_n$ (assuming ordered x_i)

  where wit is a new witness name

Witness translation

- Contexts are extended with witness names:

  $P ::= y_1 : q_1, \ldots, y_n : q_n$

  (note similarity with assumptions A!)

- Overloaded terms are translated according to their type derivation:

  $P \mid A \vdash e : t \rightsquigarrow e'$

- Read: e has type t in scope A and context P, and has a non-overloaded equivalent e'

Witness translation

$$\frac{P|A \vdash e_1 : t'\text{->}t \rightsquigarrow e_1' \quad P|A \vdash e_2 : t' \rightsquigarrow e_2'}{P|A \vdash e_1\ e_2 : t \rightsquigarrow e_1'\ e_2'}\ \text{App}$$

$$\frac{P|A,x{:}t' \vdash e : t \rightsquigarrow e'}{P|A \vdash \backslash x\text{->}e : t'\text{->}t \rightsquigarrow \backslash x\text{->}e'}\ \text{Abs}$$

$$\frac{x{:}\forall as.qs\text{=>}t \in A \qquad P \Vdash es : [ts/as]qs}{P|A \vdash x : [ts/as]t \rightsquigarrow x\ es}\ \text{Var}$$

$$\frac{ys{:}qs|A \vdash e_1 : t \rightsquigarrow e_1' \qquad P|A,x{:}\sigma \vdash e_2 : t' \rightsquigarrow e_2'}{P|A \vdash \textbf{let}\ x{=}e_1\ \textbf{in}\ e_2 : t' \rightsquigarrow \textbf{let}\ x{=}\backslash ys\text{->}e_1'\ \textbf{in}\ e_2'}\ \text{Let}$$

where σ = gen(qs=>t,A)

Witness translation

- Translate qualified type schemes as

$$[\![\ \forall as\ .\ q_1, \ldots, q_n \Rightarrow t\ ]\!] = \forall as\ .\ [\![q_1]\!] \to \ldots \to [\![q_n]\!] \to t$$

$$[\![\ C\ ts\ ]\!] = C\ ts$$

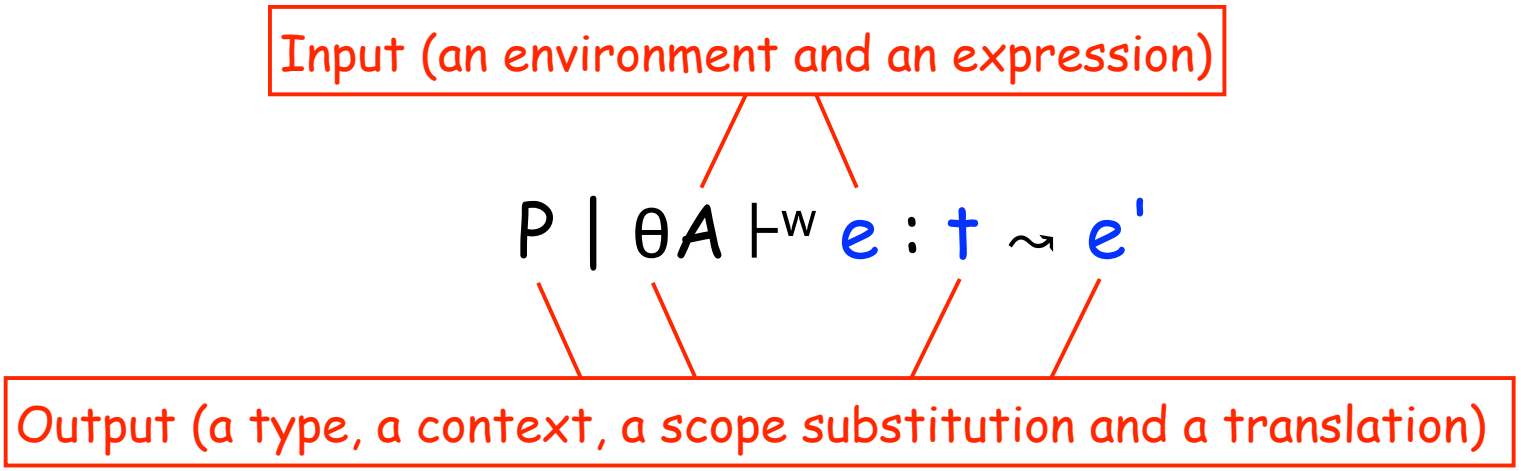
$$[\![\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ ]\!] = x_1 : [\![\sigma_1]\!], \ldots, x_n : [\![\sigma_n]\!]$$

$$[\![\ y_1 : q_1, \ldots, y_n : q_n\ ]\!] = y_1 : [\![q_1]\!], \ldots, y_n : [\![q_n]\!]$$

- We now have

$$\text{If } P \mid A \vdash e : t \rightsquigarrow e' \text{ then } [\![P_0, P]\!], [\![A]\!] \vdash e' : t$$

Qualified type inference algorithm W



- Main ideas:
  - Generate fresh witness context at Var leafs
  - Accumulate contexts from all subexpressions
  - Qualify with accumulated context at generalization

Qualified type inference algorithm W

$$\frac{P_1 | \theta_1 A \vdash^w e_1 : t \rightsquigarrow e_1' \quad P_2 | \theta_2(\theta_1 A) \vdash^w e_2 : t' \rightsquigarrow e_2' \quad \theta_2 t \overset{\theta_3}{\sim} t' \to a}{\theta_3\theta_2 P_1, \theta_3 P_2 | (\theta_3\theta_2\theta_1) A \vdash^w e_1\ e_2 : \theta_3\, a \rightsquigarrow e_1'\ e_2'}\ \text{App}$$

where a is new

$$\frac{x : \forall as.qs \Rightarrow t \in A}{ys : \theta qs \mid [] A \vdash^w x : \theta t \rightsquigarrow x\ ys}\ \text{Var}$$

where $\theta = [bs/as]$ and ys, bs are new

$$\frac{P | \theta(A, x:a) \vdash^w e : t \rightsquigarrow e'}{P | \theta A \vdash^w \backslash x \to e : \theta a \to t \rightsquigarrow \backslash x \to e'}\ \text{Abs}$$

where a is new

$$\frac{ys : qs | \theta_1 A \vdash^w e_1 : t \rightsquigarrow e_1' \qquad P | \theta_2(\theta_1 A, x:\sigma) \vdash^w e_2 : t' \rightsquigarrow e_2'}{P | (\theta_2\theta_1) A \vdash^w \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t' \rightsquigarrow \textbf{let}\ x = \backslash ys \to e_1'\ \textbf{in}\ e_2'}\ \text{Let}$$

where $\sigma = gen(qs \Rightarrow t, \theta_1 A) = \forall\ fv(qs \Rightarrow t) \backslash fv(\theta_1 A)\ .\ qs \Rightarrow t$

The ususal properties

- (Soundness)
  - If $P \mid \theta A \vdash^{w} e : t \rightsquigarrow e'$ succeeds then $P \mid \theta A \vdash e : t \rightsquigarrow e'$
- (Completeness)
  - If $P \mid \theta A \vdash e : t \rightsquigarrow e'$ then $P' \mid \theta' A \vdash^{w} e : t' \rightsquigarrow e''$ succeeds such that for some θ'':
    - $\theta = \theta''\theta'$
    - $t = \theta''t'$
    - $P \Vdash \theta''P'$

Context reduction

- Qualifying with all accumulated predicates in the Let-rule is semantically correct but may include
  - Duplicates ∀a . (Num a, Num a) => a -> a
    No point taking two identical arguments
  - Tautologies ∀a . (Num Int, Num a) => a -> a
    No point taking an argument that exists as a global
  - Local constants ∀a . (Num b, Num a) => a -> a
    No point taking an argument that will always be the same value
  - Absurdities ∀a . (Num Bool, Num a) => a -> a
    No point expecting an argument that will never be provided
- Better: reduce the ys:qs before calling gen(qs=>t,A)

Context reduction

$$\frac{P_1 | \theta_1 A \vdash^w e_1 : t \rightsquigarrow e_1' \qquad P_2 | \theta_2(\theta_1 A, x{:}\sigma) \vdash^w e_2 : t' \rightsquigarrow e_2'}{ys_1{:}\theta_2 qs_1, P_2 | (\theta_2\theta_1) A \vdash^w \textbf{let } x = e_1 \textbf{ in } e_2 : t'} \text{ Let}$$

$$\rightsquigarrow \textbf{let } x = \backslash ys_2 \rightarrow (\textbf{let } ys_3 = es \textbf{ in } e_1') \textbf{ in } e_2'$$

constant residue reducible

where $P_1 = ys_1{:}qs_1, \quad ys_2{:}qs_2, \quad ys_3{:}qs_3$

maximize

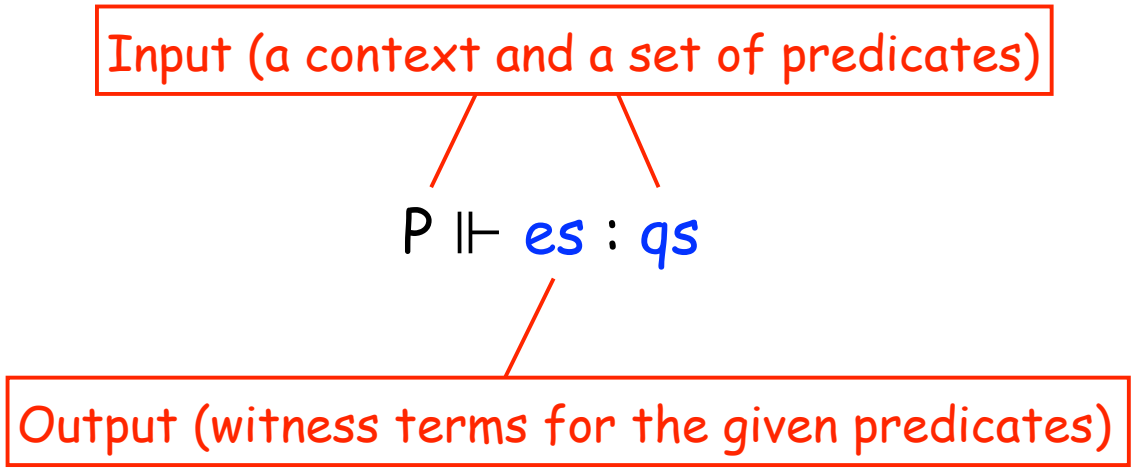
$fv(qs_1) \subseteq fv(\theta_1 A)$

$ys_2{:}qs_2 \Vdash es : qs_3$

$\sigma = gen(qs_2 \Rightarrow t, \theta_1 A)$

Entailment revisited

Both a logical system and a algorithm (no substitution threading!)



... or failure if no rules match

$$\frac{P \Vdash e_1 : q_1 \quad \ldots \quad P \Vdash e_n : q_n}{P \Vdash e_1, \ldots, e_n : q_1, \ldots, q_n}\ \text{Multi}$$

$$\frac{y{:}q \in P}{P \Vdash y : q}\ \text{Equiv}$$

Qualified instances

- Example: **instance** Eq a => Eq [a] **where** $x_i = e_i$
- Translate into

  eqList :: Eq a -> Eq [a]
  eqList = \y -> Eq e_1 ... e_n

- Witness y stands for Eq a when checking the e_i
- Now eqList eqInt is a witness of Eq [Int]
- Note inherent polymorphism: Eq a => Eq [a] <u>for all</u> a

Sub/superclasses

- Translate **class** $(q_1, \ldots, q_m) \Rightarrow C\ as$ **where** $x_i :: t_i$ into

  data $C\ as = C\ q_1 \ldots q_m\ t_1 \ldots t_n$ — a witness type

  $sup_1\ (C\ z_1 \ldots z_m\ y_1 \ldots y_n) = z_1$ — (sup_i, z_i new)

  ...

  $sup_m\ (C\ z_1 \ldots z_m\ y_1 \ldots y_n) = z_m$

  $x_1\ (C\ z_1 \ldots z_m\ y_1 \ldots y_n) = y_1$

  ...

  $x_n\ (C\ z_1 \ldots z_m\ y_1 \ldots y_n) = y_n$

- If q_1 is a superclass of q_2 as projected by sup_k, and y_2 a witness of q_2, then $sup_k\ y_2$ is a witness of q_1
- Note inherent polymorphism: Eq a => Ord a <u>for all</u> a
  (superclass-of)

Entailment extended

$$\frac{P \Vdash e : C\ ts}{P \Vdash sup_j\ e : [ts/as]q_j}\ \text{Sup}$$

if **class** (..., q_j, ...) => C as **where** ... $\in$ top-decls

and sup_j is the projection from C as to q_j

$$\frac{P \Vdash es : [ts/as]qs}{P \Vdash wit\ es : [ts/as]q}\ \text{Inst}$$

if **instance** $\forall as\ .\ qs \Rightarrow q$ **where** ... $\in$ top-decls

and wit is its generated witness name

Ambiguity

- Intuitively, $P \mid A \vdash e : t \rightsquigarrow e'$ assigns term e' as the meaning of overloaded term e
- Problem: $P \mid A \vdash e : t \rightsquigarrow e_1$ and $P \mid A \vdash e : t \rightsquigarrow e_2$ does not imply equivalence between e_1 and e_2 in general
- But it can be shown that if the principal type scheme for e under A (as computed by W) is unambiguous, the meaning of overloading is well-defined
- For an unambiguous type scheme $\forall as \,.\, qs \Rightarrow t$:

  $fv(qs) \cap as \subseteq fv(t)$

check at each let-expression!

Qualified matching

- Definition: $\forall as.\ qs_a \Rightarrow t_a \preccurlyeq \forall bs.\ qs_b \Rightarrow t_b$ iff

$$\forall bs\ .\ \exists as\ .\ t_a = t_b\ \&\ qs_b \Vdash qs_a$$

- Matching algorithm:

$$\frac{t_a \overset{\theta}{\sim} \phi\, t_b \qquad \theta\,\phi\, qs_b \Vdash es : \theta qs_a}{\forall as.\ qs_a \Rightarrow t_a \overset{\theta}{\preccurlyeq} \forall bs.\ qs_b \Rightarrow t_b}$$

where ϕ is a substitution skolemizing bs with respect to

$$fv(\forall as.\ qs_a \Rightarrow t_a) \cup fv(\forall bs.\ qs_b \Rightarrow t_b)$$

(Caveat: this algorithm is not complete for multi-parameter type-classes, and in fact it cannot be because of the inability to qualify over "entailment constraints". On the other hand, no known Haskell implementation solves this problem either...)

Summary

- The essence of Haskell-style overloading: qualifying predicates extend judgements and type schemes
- Straightforward qualified variants of type correctness rules and type inference algorithm
- Witness translation removes overloading, defined as an additional attribute to type derivations
- Context reduction amounts to remove predicates entailed by other predicates or instance facts
- Challenge: split context to enable minimal residue
- Challenge: integrate reduction with signature matching