Semantic Subtyping

Alain Frisch (ENS Paris)

Giuseppe Castagna (ENS Paris)

Véronique Benzaken (LRI U Paris Sud)

`http://www.cduce.org/`

CDuce

- A functional language adapted to XML applications: `http://www.cduce.org/`
- Types are pervasives in ℂDuce:
  - Soundness of transformations
  - Informative error messages
  - Type-driven semantics, pattern matching
  - Type-driven compilation and optimization
- This talk: theoretical foundation of ℂDuce type system.
- Most of the technical difficulties in the subtyping relation, which is kept simple by using a semantic approach.

Semantic and syntactic subtyping

How to define a subtyping relation ?

- ▷ Syntactic approach: a formal system (axioms,rules)
- ▷ Semantic approach:
  - ▷ start with a denotational model of the language
  - ▷ interpret types as subsets of the model
  - ▷ define subtyping as inclusion of denotations
  - ▷ derive a subtyping algorithm

Advantages of semantic subtyping

- subtyping is complete w.r.t the intuitive interpretation
- when $t \leq s$ does not hold, it is possible to exhibit an element of the model in the interpretation of t and not of s ($\rightarrow$ error message)
- modularization of proofs: use the semantic interpretation, not the rules of the subtyping algorithm
- properties "for free": transitivity of $\leq$, ...
- Whenever possible, avoid *ad hoc* rules, formulas and algorithms: derive them from computations.

XDuce

(H. Hosoya, B. Pierce, J. Vouillon)

- XDuce: a typed programming language for XML applications
  - value = XML document (tree)
  - type = regular tree language
  - subtyping = inclusion of languages
- A powerful pattern-matching operation
  - Type-driven semantics
  - Recursive patterns to extract information in the middle of a document
  - *Exact* propagation of the type from the matched expression to binding variables

Semantic subtyping in XDuce

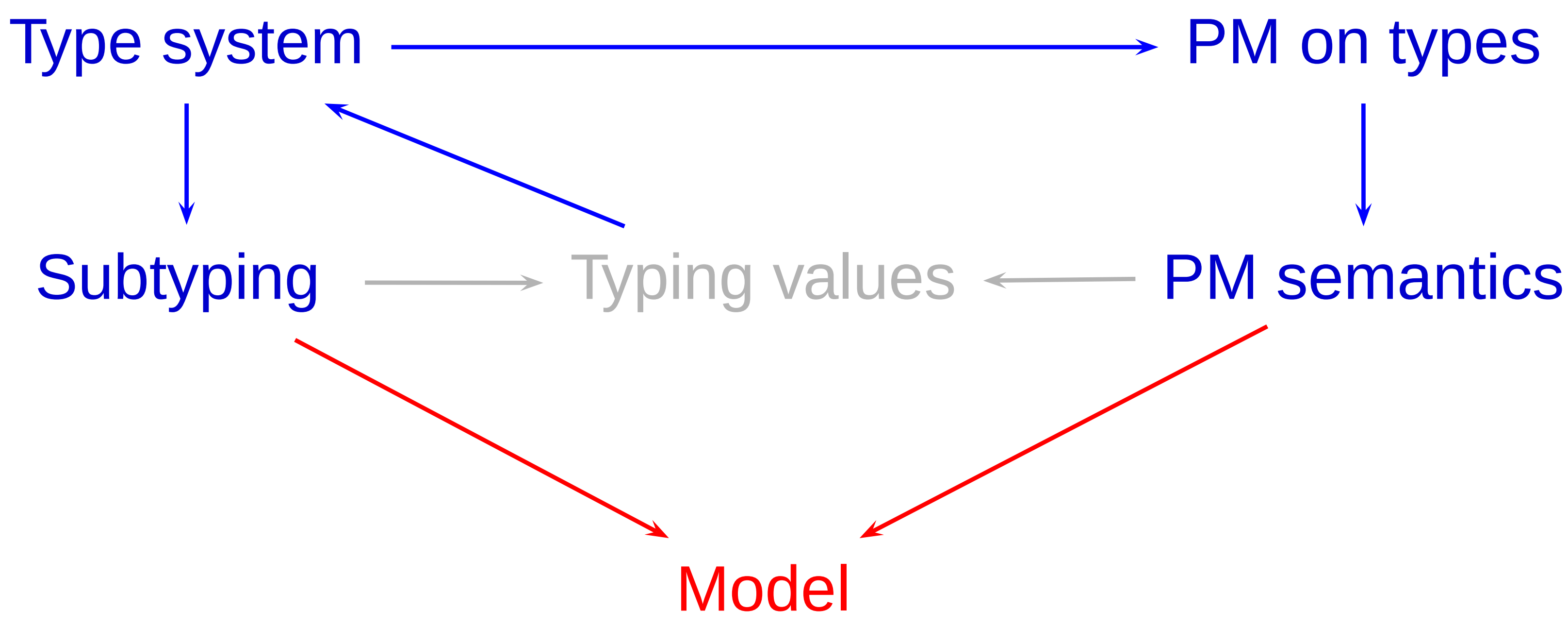
- Semantic subtyping: easy because no first class function, so typing of values does not depend on subtyping !
- Start from the typing judgment $\vdash v : t$
- Interpretation $[\![t]\!] = \{v \mid \vdash v : t\}$
- Subtyping $t \leq s \iff [\![t]\!] \subseteq [\![s]\!]$

XDuce + higher-order $\Rightarrow$ $\mathbb{C}$*Duce*

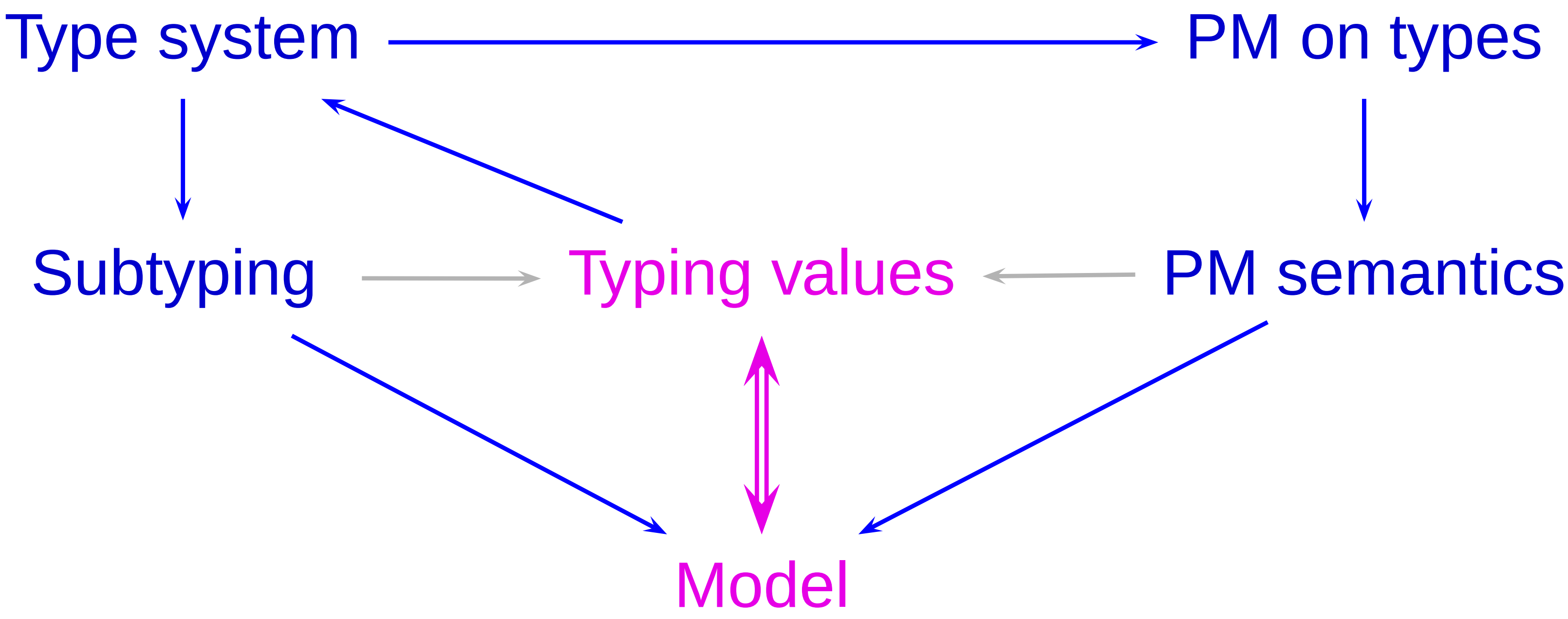
- Integrate XDuce features to a general higher order functional language
- Extend XDuce semantic approach ?
  - $[\![t \to s]\!] = \{\lambda x.M \mid \vdash \lambda x.M : t \to s\}$
  - But: typing of values now depends on subtyping !
- Classical semantic approach ?
  - Define an untyped denotational model
  - But: semantics depends on typing !

Circularities

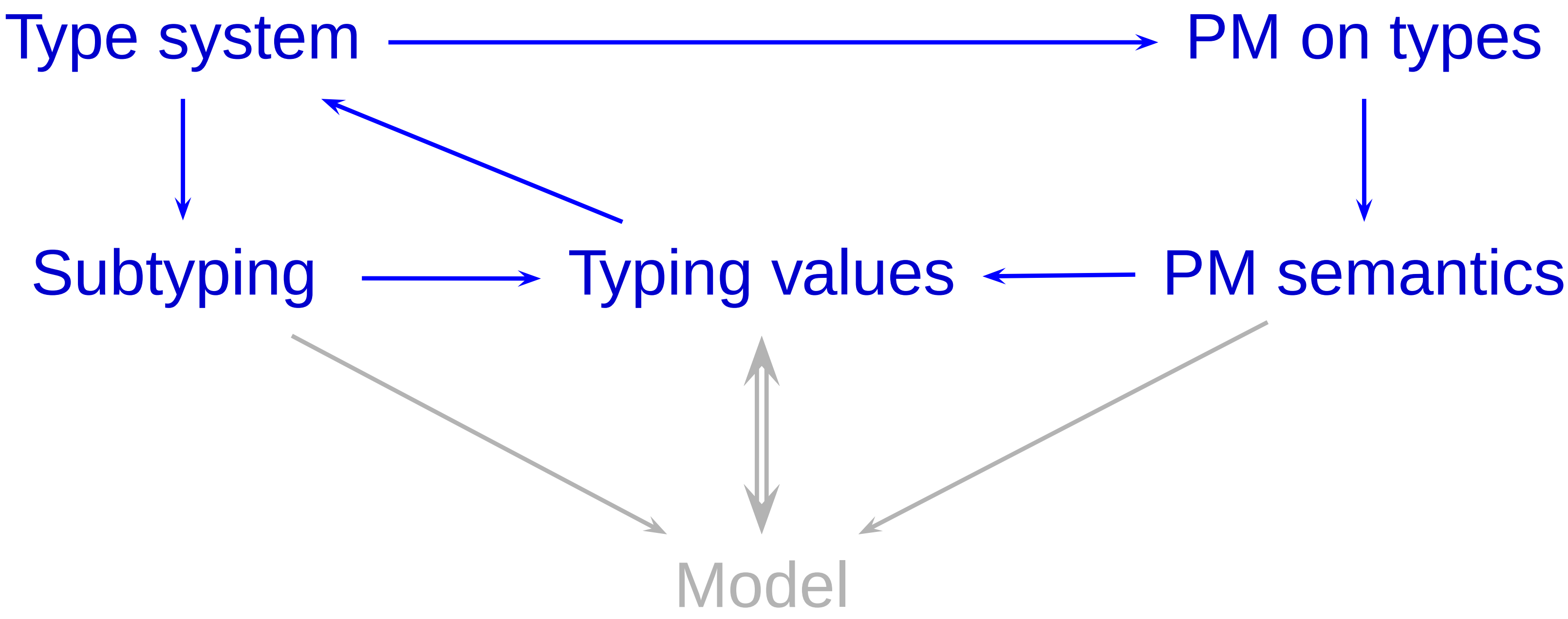
Circularities



Circularities



Circularities



Types

Type algebra

$$t \quad ::= \quad b \mid t \rightarrow t \mid t \times t$$

Types:

- ▷ Constructors: basic, product, arrow types.

Type algebra

$$
\begin{array}{rcl}
t & ::= & b \mid t \to t \mid t \times t \\
& \mid & \neg t \mid t \vee t \mid t \wedge t \mid \mathbf{0} \mid \mathbf{1}
\end{array}
$$

Types:

- ▷ Constructors: basic, product, arrow types.
- ▷ Arbitrary finite boolean combinations.

Type algebra

$$
\begin{array}{rcl}
t & ::= & b \mid t \to t \mid t \times t \\
& \mid & \neg t \mid t \vee t \mid t \wedge t \mid \mathbf{0} \mid \mathbf{1} \\
& \mid & \alpha \mid \mu\alpha.t
\end{array}
$$

Types:

- ▷ Constructors: basic, product, arrow types.
- ▷ Arbitrary finite boolean combinations.
- ▷ Guarded recursive types.

Subtyping and models

- We want to define $\leq$ by: $\boxed{t \leq s \Leftrightarrow [\![t]\!] \subseteq [\![s]\!]}$
- For each type, $[\![t]\!]$ is subset of a structure $\mathscr{D}$:

$$\mathscr{D} = \mathscr{D}_{\textbf{basic}} + \mathscr{D}_{\textbf{prod}} + \mathscr{D}_{\textbf{fun}} \qquad \text{with:} \qquad \mathscr{D}_{\textbf{prod}} \simeq \mathscr{D}^2$$

- The interpretation function $[\![\_]\!]$ must satisfy natural conditions:

$$\begin{array}{lcl} [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \subseteq \mathscr{D}_{\textbf{prod}} \\ [\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\ [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\ [\![\neg t]\!] & = & \mathscr{D} \backslash [\![t]\!] \\ [\![\mathbf{0}]\!] & = & \varnothing \end{array}$$

Model condition for arrows

$$\bigwedge_{i\in I} t_i \to s_i \leq \bigvee_{j\in J} t'_j \to s'_j$$

$$\iff$$

$$(\exists j \in J)(t'_j \leq \bigvee t_i) \wedge (\forall I' \subseteq I)(t'_j \leq \bigvee_{i\in I\setminus I'} t_i) \vee (\bigwedge_{i\in I'} s_i \leq s'_j)$$

A condition on $[\![\to]\!]$ motivated by:

- ▷ Intuition: extensive view of functions (=binary relations)
- ▷ Derived from simple set-theoretic computations
- ▷ Makes the safety proof work
- ▷ The type system induces a model of values
- ▷ (Turns out to hold in the revelance logic B+; M. Dezani)

Is there at least one model !?

- ▷ Yes, this one:

$$
\begin{array}{lcll}
d & ::= & c & c \in \mathscr{D}_{\textbf{basic}} \\
 & | & (d_1, d_2) & \\
 & | & \{(d_1, d_1'), \ldots, (d_n, d_n')\} & d_i' \in \mathscr{D} \cup \{\Omega\}
\end{array}
$$

- ▷ It is universal (largest subtyping relation).
- ▷ For this model, we derive an algorithm to compute $\leq$.

The language

The language: syntax

e	$::=$	x	variable
	$\mid$	$\mu f^{(t_1 \to s_1; \dots; t_n \to s_n)}(x).e$	abstraction
	$\mid$	$e_1 e_2$	application
	$\mid$	c	constant
	$\mid$	(e_1, e_2)	pair
	$\mid$	$\texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2$	pattern matching

Type system

$$\frac{}{\Gamma \vdash c : t_c} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$\frac{\Gamma \vdash e_1 : t \to s \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : s} \quad \frac{\Gamma \vdash e : s \leq t}{\Gamma \vdash e : t}$$

$$\frac{t \equiv \bigwedge t_i \to s_i \qquad \Gamma, (x : t_i), (f : t) \vdash e : s_i}{\Gamma \vdash \mu f^{(t_1 \to s_1; \dots ; t_n \to s_n)}(x).e : t}$$

Type system

$$\frac{}{\Gamma \vdash c : t_c} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$\frac{\Gamma \vdash e_1 : t \to s \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : s} \qquad \frac{\Gamma \vdash e : s \leq t}{\Gamma \vdash e : t}$$

$$\frac{t \equiv \bigwedge t_i \to s_i \quad t_\neg \equiv \bigwedge \neg(t'_j \to s'_j) \text{ with } \forall j.\, t \not\leq t'_j \to s'_j \qquad \Gamma, (x : t_i), (f : t) \vdash e : s_i}{\Gamma \vdash \mu f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e : t \wedge t_\neg}$$

- Need it for subject reduction ...
- ... but can discard it for typechecking.

Results

- Classical syntactical results, such as admissibility of conjunction rule and subsumption elimination.
- Subject reduction for a small step semantics.
- A new interpretation of types as sets of values:

$$\llbracket t \rrbracket_{\mathscr{V}} = \{ v \mid \vdash v : t \}$$

  - This is indeed a model.
  - It induces the same subtyping relation as the bootstrap model.
  - This holds because of overloaded functions !

The circle is now complete

$$
\begin{aligned}
\vdash v : t_1 \wedge t_2 &\iff (\vdash v : t_1) \wedge (\vdash v : t_2) \\
\vdash v : t_1 \vee t_2 &\iff (\vdash v : t_1) \vee (\vdash v : t_2) \\
\vdash v : \neg t &\iff \neg(\vdash v : t) \\
\not\vdash v : \mathbf{0} &
\end{aligned}
$$

$$
\begin{aligned}
t \leq s &\iff \forall v.(\vdash v : t) \Rightarrow (\vdash v : s) \\
&\iff \forall \Gamma.\forall e.(\Gamma \vdash e : t) \Rightarrow (\Gamma \vdash e : s)
\end{aligned}
$$

$$
\vdash v : t_1 \bullet t_2 \iff \exists v_1, v_2.(\vdash v_1 : t_1) \wedge (\vdash v_2 : t_2) \wedge (v_1 v_2 \stackrel{*}{\rightarrow} v)
$$

$$
(t_1 \bullet t_2 = \min\{s \mid t_1 \leq t_2 \rightarrow s\})
$$

Subtyping algorithm

- As any coinductive relation, the subtyping algorithm can be expressed in an abstract way through a notion of simulation.
- A simulation is a subset R of all types closed under rules like:

$$(\bigwedge_{i\in I} t_i \to s_i \backslash \bigvee_{j\in J} t'_j \to s'_j) \in R$$
$$\Rightarrow$$
$$(\exists j \in J)(t'_j \backslash \bigvee t_i) \in R \wedge (\forall I' \subseteq I)(t'_j \backslash \bigvee_{i\in I\backslash I'} t_i) \in R \vee (\bigwedge_{i\in I'} s_i \backslash s'_j) \in R$$

- The largest simulation is exactly the set of empty types.
- This abstract presentation separates implementation issues (such as caching) and the theoretical study of the algorithm.

Using set-theoretic semantics

Simple set-theoretic facts yield a number of optimizations for the subtyping algorithm, avoiding exponential explosion in many cases. E.g.:

$$(t \times s) \backslash (t_1 \times s_1) \backslash \ldots \backslash (t_n \times s_n) \simeq$$

$$(t \wedge t_1 \times s \backslash s_1) \vee \ldots \vee (t \wedge t_n \times s \backslash s_n) \vee (t \backslash t_1 \backslash \ldots \backslash t_n \times s)$$

whenever

$$\forall i \neq j.\ t_i \wedge t_j \simeq \mathbf{0}$$

Patterns

Patterns

$$
\begin{array}{lcll}
p & ::= & x & \text{capture} \\
 & \mid & t & \text{type constraint} \\
 & \mid & p_1 \wedge p_2 & \text{conjunction} \\
 & \mid & p_1 | p_2 & \text{alternative} \\
 & \mid & (p_1,p_2) & \text{pair} \\
 & \mid & (x := c) & \text{constant} \\
 & \mid & \mu\rho.p & \text{recursive pattern} \\
 & \mid & \rho &
\end{array}
$$

Result of matching d/p: Ω (failure) or a substitution *Var*$(p) \to \mathscr{D}$.

Pattern-matching: static semantics

- The pattern p accepts a type characterized by:

$$[\![\llparenthesis p \rrparenthesis]\!] = \{ d \mid d/p \neq \Omega \}$$

  - <u>Thm</u>: it exists + algorithm

- If $t \leq \llparenthesis p \rrparenthesis$ and $x \in \mathit{Var}(p)$, type of the result for x characterized by:

$$[\![(t/p)(x)]\!] = \{ (d/p)(x) \mid d \in [\![t]\!] \}$$

  - <u>Thm</u>: it exists + algorithm

Type system: pattern matching

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 :}$$

Type system: pattern matching

$$\frac{\Gamma \vdash e : s \leq \wr p_1 \int \vee \wr p_2 \int}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2 :}$$

▷ Exhaustivity checking

- $\wr p \int$: set of values matched by p

Type system: pattern matching

$$\frac{\begin{array}{c}(s_1 \equiv s \wedge \wr p_1 \int,\ s_2 \equiv s \wedge \neg \wr p_1 \int) \\ \Gamma \vdash e : s \leq \wr p_1 \int \vee \wr p_2 \int\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2 :}$$

- ▷ Exhaustivity checking
  - $\wr p \int$: set of values matched by p
- ▷ Dispatching

Type system: pattern matching

$$(s_1 \equiv s \wedge \{\!\!\{ p_1 \}\!\!\},\; s_2 \equiv s \wedge \neg \{\!\!\{ p_1 \}\!\!\})$$

$$\frac{\Gamma \vdash e : s \leq \{\!\!\{ p_1 \}\!\!\} \vee \{\!\!\{ p_2 \}\!\!\} \qquad \Gamma, (s_i/p_i) \vdash e_i : t_i}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2 :}$$

- ▷ Exhaustivity checking
  - $\{\!\!\{ p \}\!\!\}$: set of values matched by p
- ▷ Dispatching
- ▷ Matching
  - (t/p) : typing environment for variables bound in p

Type system: pattern matching

$$\frac{\begin{array}{c}(s_1 \equiv s \wedge \lbag p_1 \rbag,\ s_2 \equiv s \wedge \neg \lbag p_1 \rbag) \\ \Gamma \vdash e : s \leq \lbag p_1 \rbag \vee \lbag p_2 \rbag \qquad \Gamma, (s_i/p_i) \vdash e_i : t_i\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 : \bigvee_{\{i | s_i \not\simeq \mathbf{0}\}} t_i}$$

- ▷ Exhaustivity checking
  - $\lbag p \rbag$: set of values matched by p
- ▷ Dispatching
- ▷ Matching
  - (t/p) : typing environment for variables bound in p
- ▷ Result
  - Discard useless branches

Typing patterns: an example

- Lists *à la* Lisp (head,tail) + terminator. Consider the following pattern and types:

$$
\begin{array}{rcl}
p & = & \mu\rho.\mathbf{(}x \wedge t_0\mathbf{,1)|(1,}\rho\mathbf{)} \\
t & = & \mu\alpha.(s_1 \times (s_2 \times \alpha)) \vee \texttt{nil} \\
t' & = & t \wedge \{\!\!| p |\!\!\}
\end{array}
$$

- Then:

If $s_1 \leq t_0$: $(t'/p)(x) = s_1$

If $s_1 \wedge t_0 \simeq \mathbf{0}$: $(t'/p)(x) = s_2 \wedge t_0$

Otherwise: $(t'/p)(x) = (s_1 \vee s_2) \wedge t_0$

Pattern algorithms (1)

$$
\begin{array}{lcl}
\{\!\!\{ x \}\!\!\} & \equiv & \mathbf{1} \\
\{\!\!\{ t \}\!\!\} & \equiv & t \\
\{\!\!\{ (x := c) \}\!\!\} & \equiv & \mathbf{1} \\
\{\!\!\{ p_1 | p_2 \}\!\!\} & \equiv & \{\!\!\{ p_1 \}\!\!\} \vee \{\!\!\{ p_2 \}\!\!\} \\
\{\!\!\{ p_1 \wedge p_2 \}\!\!\} & \equiv & \{\!\!\{ p_1 \}\!\!\} \wedge \{\!\!\{ p_2 \}\!\!\} \\
\{\!\!\{ (p_1, p_2) \}\!\!\} & \equiv & \{\!\!\{ p_1 \}\!\!\} \times \{\!\!\{ p_2 \}\!\!\}
\end{array}
$$

Pattern algorithms (2)

$$
\begin{array}{lll}
(t'/x)(x) & \equiv t' & \\
(t'/p_1|p_2)(x) & \equiv ((t' \wedge \{\!\{p_1\}\!\})/p_1)(x) \vee ((t' \wedge \neg \{\!\{p_1\}\!\})/p_2)(x) & \\
(t'/p_1 \wedge p_2)(x) & \equiv (t'/p_i)(x) & \text{if } x \in \textit{Var}(p_i) \\
(t'/(p_1,p_2))(x) & \equiv \bigvee_{(t_1,t_2)\in\pi(t')} (t_1/p_1)(x) \times (t_2/p_2)(x) & \\
 & & \text{if } x \in \textit{Var}(p_1) \cap \textit{Var}(p_2) \\
(t'/(p_1,p_2))(x) & \equiv (\pi_1(t')/p_1)(x) & \text{if } x \in \textit{Var}(p_1) \setminus \textit{Var}(p_2) \\
(t'/(p_1,p_2))(x) & \equiv (\pi_2(t')/p_2)(x) & \text{if } x \in \textit{Var}(p_2) \setminus \textit{Var}(p_1) \\
(t'/(x := c))(x) & \equiv t_c & \text{if } t' \not\simeq \mathbf{0} \\
(t'/(x := c))(x) & \equiv \mathbf{0} & \text{if } t' \simeq \mathbf{0}
\end{array}
$$

Compiling pattern matching

- ▷ Naive solution: backtracking.
- ▷ Less naive, but untractable: bottom-up tree automata.
- ▷ Adopted solution: hybrid top-down/bottom-up automata, which avoid backtracking, and take profit of static type information.
- ▷ Complex algorithm: set-theoretic semantics of types and patterns helps *a lot* here !