Revised Notes on Records, Variants, and Subtyping

April 30, 2000

There are some errors and inconsistencies in the typing rules presented in lectures, and they are spread out over several days. Here I've tried to summarize the various typing systems we've looked at, correcting some mistakes along the way.

1 Simply-typed λ-calculus with constants

Syntax of types:

$$\begin{array}{rcll} \tau & ::= & b & \text{Base types} \\ & | & \tau_1 \to \tau_2 & \text{Function types} \end{array}$$

Syntax of expressions:

$$\begin{array}{rcll} M & ::= & c & \text{Built-in constants} \\ & | & v & \text{Variable names} \\ & | & (M_1 M_2) & \text{Function applications} \\ & | & (\lambda v\!:\!\tau.M) & \text{Function abstractions} \end{array}$$

Typing rules (syntax-directed) :

$$\frac{\mathit{Typeof}(c) = \tau}{E \rhd c : \tau}\ \mathit{CONST}$$

$$\frac{E(x) = \tau}{E \rhd x : \tau}\ \mathit{VAR}$$

$$\frac{E + \{x : \tau_1\} \rhd M : \tau_2}{E \rhd (\lambda x{:}\tau_1 . M) : \tau_1 \to \tau_2} \; ABS$$

$$\frac{E \rhd M : \tau_1 \to \tau_2 \quad E \rhd N : \tau_1}{E \rhd (MN) : \tau_2} \; APP$$

2 Adding Record and Variant Types

I've changed "sum" to "variant" because sum is normally the dual of product, and our records are not just simple products.

Both the empty record type $\{\}$ and the empty variant type $<>$ are valid. It is possible to construct records of type $\{\}$, but nothing can be selected from them. Hence, all values of this type are indistinguishable, i.e., the type corresponds to the single-element type `unit` in ML. This type is very useful as the argument type to functions that ignore their argument. Conversely, variants of type $<>$ are not very useful, because it is impossible to construct a value of this type. So $<>$ is a type with no elements, which might sensible be called `void`. Although the typing rules below allow an expression like

λ `x : <>. switch x of` ...

for any set of arms (...), `x` can never be bound to a value, so the `switch` can never be executed.

Syntax of types:

$$\begin{array}{lcll} \tau & ::= & b & \text{Base types} \\ & | & \tau_1 \to \tau_2 & \text{Function types} \\ & | & \{l_1 : \tau_1, \ldots, l_n : \tau_n\} & \text{Record types} \\ & | & < l_1 : \tau_1, \ldots, l_n : \tau_n > & \text{Variant types} \end{array}$$

Syntax of expressions:

$$\begin{array}{lcll} M & ::= & c & \text{Built-in constants} \\ & | & v & \text{Variable names} \\ & | & (M_1 M_2) & \text{Function applications} \\ & | & (\lambda v{:}\tau . M) & \text{Function abstractions} \\ & | & \texttt{\{ } l_1 \texttt{ = } M_1 \texttt{, } \ldots \texttt{, } l_n \texttt{ = } M_n \texttt{ \}} & \text{Record construction} \\ & | & M.l & \text{Record selection} \\ & | & \texttt{< } l \texttt{ = } M \texttt{ >} & \text{Variant construction} \\ & | & \texttt{switch } M \texttt{ of } l_1 \texttt{ => } M_1 \texttt{ | } \ldots \texttt{ | } l_n \texttt{ => } M_n & \text{Variant switch} \end{array}$$

Typing rules (still syntax-directed):

CONST, *VAR*, *ABS*, *APP* as above, plus

$$\frac{E \rhd M_i : \tau_i \ (1 \leq i \leq n) \quad (l_i \ \mathit{distinct}) \quad (n \geq 0)}{E \rhd \{l_1 \texttt{ = } M_1 \texttt{,} \ldots \texttt{,} l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \ \mathit{RECORD}$$

$$\frac{E \rhd M : \{\ldots, l : \tau, \ldots\}}{E \rhd M.l : \tau} \ \mathit{SELECT}$$

$$\frac{E \rhd M : \tau}{E \rhd \texttt{<}l\texttt{=}M\texttt{>} \ : \ < l : \tau >} \ \mathit{VARIANT}$$

$$\frac{\begin{array}{c} E \rhd M :< l'_1 : \tau'_1, \ldots, l'_m : \tau'_m > \quad \{l'_1, \ldots, l'_m\} \subseteq \{l_1, \ldots, l_n\} \\ E \rhd M_i : \tau_i \to \tau \ (1 \leq i \leq n) \qquad l'_j = l_i \Rightarrow \tau'_j = \tau_i \\ (l_i \ \mathit{distinct}) \ \ (n \geq 1) \end{array}}{E \rhd \texttt{switch} \ M \ \texttt{of} \ l_1 \texttt{ => } M_1 \texttt{ | ... | } l_n \texttt{ => } M_n : \tau} \ \mathit{SWITCH}$$

Note that the *SWITCH* rule has been liberalized to permit the argument type to carry any subset of the variants for which switch arms are defined; this brings it in line with the *SELECT* rule, and allows us to type obviously sensible expressions like

```
switch <a=1> of a => ... | b => ...
```

In addition, I've added two restrictions that aren't necessary to ensure safety of execution, but which guarantee that we have a syntax-directed way to reconstruct the overall result type of any `switch` expression without adding an explicit type annotation to the syntax, even in degenerate cases. First, a `switch` must always have at least one arm. Second, all arms must share the same result type, even those that can never be executed because their labels don't appear in the argument variant type.

3 Adding subtyping

Next we add subtyping by means of a subsumption typing rule and an associated set of subtyping rules. If desired, we can also remove the flexibility from the *SELECT* and *SWITCH* rules, since subsumption does all the needed work – but it is also harmless to keep the old rules. In any case, the resulting typing system is *not* syntax-directed, because subsumption can be used most anywhere.

Syntax of types and expressions is unchanged.

Typing rules:

CONST, *VAR*, *ABS*, *APP*, *RECORD*, *VARIANT* as above, plus

$$\frac{E \rhd M : \tau' \quad \tau' <: \tau}{E \rhd M : \tau}\ \mathit{SUBSUME}$$

plus *either* *SELECT* and *SWITCH* as above, or

$$\frac{E \rhd M : \{l : \tau\}}{E \rhd M.l : \tau}\ \mathit{S\text{-}SELECT}$$

$$\frac{\begin{array}{c} E \rhd M :< l_1 : \tau_1 \ldots, l_n : \tau_n > \\ E \rhd M_i : \tau_i \to \tau \ (1 \leq i \leq n) \\ (l_i \ \mathit{distinct}) \ \ (n \geq 1) \end{array}}{E \rhd \texttt{switch}\ M\ \texttt{of}\ l_1\ \texttt{=>}\ M_1\ \texttt{|}\ \ldots\ \texttt{|}\ l_n\ \texttt{=>}\ M_n : \tau}\ \mathit{S\text{-}SWITCH}$$

Subtyping rules:

$$\tau <: \tau\ \mathit{SUB\text{-}REFL}$$

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ \mathit{SUB\text{-}TRANS}$$

$$\frac{\{l_1, \ldots, l_n\} \subseteq \{l'_1, \ldots, l'_m\} \quad l'_j = l_i \Rightarrow \tau'_j <: \tau_i}{\{l'_1 : \tau'_1, \ldots, l'_m : \tau'_m\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}\ \mathit{SUB\text{-}RECORD}$$

$$\frac{\{l'_1, \ldots, l'_m\} \subseteq \{l_1, \ldots, l_n\} \quad l'_j = l_i \Rightarrow \tau'_j <: \tau_i}{< l'_1 : \tau'_1, \ldots, l'_m : \tau'_m > \ \ <: \ \ < l_1 : \tau_1, \ldots, l_n : \tau_n >}\ \mathit{SUB\text{-}VARIANT}$$

$$\frac{\tau_2 <: \tau_1 \quad \tau'_1 <: \tau'_2}{\tau_1 \to \tau'_1 \ <: \ \tau_2 \to \tau'_2}\ \mathit{SUB\text{-}ARROW}$$

Plus possible additional rules relating base types.

4 Syntax-directed rules for subtyping

To obtain a syntax-directed algorithm, we drop the *SUBSUME* rule, substitute a more liberal version of the *APP* rule, reinstate the liberal *SELECT* rule, and use a still more liberal *SWITCH* rule. This results in a set of rules that supports syntax-directed type-checking. I claim that the rules of this section are *minimial* and sound with respect to those of the previous section. That is, if $E \triangleright M : \tau$ under the rules of the previous section, then $E \triangleright M : \tau'$ for some $\tau' <: \tau$ under the rules of this section. Conversely, if $E \triangleright M : \tau$ under the rules of this section, then $E \triangleright M : \tau$ under the rules of the previous section. (Exercise: Prove or disprove this claim.)

Complete summary of typing rules:

$$\frac{\mathit{Typeof}(c) = \tau}{E \triangleright c : \tau} \; \mathit{CONST}$$

$$\frac{E(x) = \tau}{E \triangleright x : \tau} \; \mathit{VAR}$$

$$\frac{E + \{x : \tau_1\} \triangleright M : \tau_2}{E \triangleright (\lambda x{:}\tau_1 . M) : \tau_1 \to \tau_2} \; \mathit{ABS}$$

$$\frac{E \triangleright M : \tau_1 \to \tau_2 \quad E \triangleright N : \tau_1' \quad \tau_1' <: \tau_1}{E \triangleright (MN) : \tau_2} \; \mathit{SDS\text{-}APP}$$

$$\frac{E \triangleright M_i : \tau_i \; (1 \le i \le n) \quad (l_i \; \mathit{distinct}) \quad (n \ge 0)}{E \triangleright \{l_1 \texttt{ = } M_1 \texttt{,} \ldots \texttt{,} l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \; \mathit{RECORD}$$

$$\frac{E \triangleright M : \{\ldots, l : \tau, \ldots\}}{E \triangleright M.l : \tau} \; \mathit{SELECT}$$

$$\frac{E \triangleright M : \tau}{E \triangleright \texttt{<}l\texttt{=}M\texttt{>} \; : \; < l : \tau >} \; \mathit{VARIANT}$$

$$\frac{\begin{array}{cc} E \triangleright M :< l_1' : \tau_1', \ldots, l_m' : \tau_m' > & \{l_1', \ldots, l_m'\} \subseteq \{l_1, \ldots, l_n\} \\ E \triangleright M_i : \tau_i \to \tau_i'' \; (1 \le i \le n) & l_j' = l_i \Rightarrow \tau_j' <: \tau_i \\ \tau = \tau_1'' \sqcup \ldots \sqcup \tau_n'' & (l_i \; \mathit{distinct}) \; (n \ge 1) \end{array}}{E \triangleright \texttt{switch} \; M \; \texttt{of} \; l_1 \; \texttt{=>} \; M_1 \; \texttt{|} \; \ldots \; \texttt{|} \; l_n \; \texttt{=>} \; M_n : \tau} \; \mathit{SDS\text{-}SWITCH}$$

SDS-SWITCH differs from *SWITCH* in two ways. First, the type carried by a particular label in the argument variant type may be a *subtype* of the type expected by the corresponding arm, rather than matching it exactly; this is essentially similar to the subtyping in *SDS-APP*, because there is an implicit function application inside the execution of `switch`. Second, the overall result type is calculated as the *join* ($\sqcup$) of the results types of the arms; if this join doesn't exist, the `switch` is not well-typed.

The join operator calculates the minimum common supertype of two types. That is,

$\tau = \tau_1 \sqcup \tau_2$

if and only if

$\tau_1 <: \tau$
$\tau_2 <: \tau$
$(\tau_1 <: \tau' \mathit{and}\ \tau_2 <: \tau') \Rightarrow \tau <: \tau'$

Two given types do not always have any common supertype; if they do, however, I claim that they have a (unique) join. (Exercise: Prove or disprove this claim.)

There is a dual *meet* ($\sqcap$) operator, which calculates the maximum common subtype of two types. That is,

$\tau = \tau_1 \sqcap \tau_2$

if and only if

$\tau <: \tau_1$
$\tau <: \tau_2$
$(\tau' <: \tau_1 \mathit{and}\ \tau' <: \tau_2) \Rightarrow \tau' <: \tau$

Two given types do not always have any common subtype; if they do, however, I claim that they have a (unique) meet. (Exercise: Prove or disprove this claim.)

The rules for calculating joins and meets are based directly on the definition of the subtype relation. They are tedious to express declaratively; a (revised) algorithm is given in the (revised) homework. (Exercise: Prove or disprove the correctness of the algorithm.)