# PicoML Semantics

Max Kopinsky (modified from Elsa Gunter)

February 2021

## 1 Types

### 1.1 Expressions

$$\frac{}{\Gamma \vdash c : \mathit{instantiate}(\mathit{signature}(c)) \mid \varnothing}\ \text{T-Const}$$

$$\frac{}{\Gamma \vdash x : \mathit{instantiate}(\Gamma(x)) \mid \varnothing}\ \text{T-Var}$$

$$\frac{\Gamma \vdash e_1 : \beta \mid \phi_1 \qquad \Gamma \vdash e_2 : \tau_1 \mid \phi_2 \qquad \Gamma \vdash e_3 : \tau_2 \mid \phi_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau_1 \mid \phi_1 \cup \phi_2 \cup \phi_3 \cup \{\texttt{bool} \sim \beta, \tau_1 \sim \tau_2\}}\ \text{T-If}$$

$$\frac{\Gamma \vdash e : \tau_1 \mid \phi}{\Gamma \vdash \otimes e : \tau \mid \{\tau_1 \to \tau \sim \mathit{instantiate}(\mathit{signature}(\otimes))\} \cup \phi}\ \text{T-Monop}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \phi_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid \phi_2}{\Gamma \vdash e_1 \otimes e_2 : \tau \mid \{\tau_1 \to \tau_2 \to \tau \sim \mathit{instantiate}(\mathit{signature}(\otimes))\} \cup \phi_1 \cup \phi_2}\ \text{T-Binop}$$

$$\frac{\Gamma \cup \{x : \tau_1\} \vdash e : \tau_2 \mid \phi}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1 \to \tau_2 \mid \phi}\ \text{T-Fun}$$

$$\frac{\Gamma \vdash f : \tau_1 \mid \phi_1 \qquad \Gamma \vdash e : \tau_2 \mid \phi_2}{\Gamma \vdash f\ e : \tau_3 \mid \{\tau_1 \sim \tau_2 \to \tau_3\} \cup \phi_1 \cup \phi_2}\ \text{T-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash b : \tau_b \mid \phi_b \\ \sigma = \mathit{solve}(\phi_b) \qquad \Gamma \cup \{b : \mathit{generalize}(\sigma(\Gamma), \sigma(\tau_b))\} \vdash e : \tau \mid \phi\end{array}}{\Gamma \vdash \texttt{let}\ x = b\ \texttt{in}\ e : \tau \mid \phi_b \cup \phi}\ \text{T-Let}$$

$$\frac{\begin{array}{c}\Gamma \cup \{f : \tau_1 \to \tau_2, x : \tau_1\} \vdash e_f : \tau_3 \mid \phi_1 \qquad \sigma = \mathit{solve}(\{\tau_2 \sim \tau_3\} \cup \phi_1) \\ \Gamma \cup \{f : \mathit{generalize}(\sigma(\Gamma), \sigma(\tau_1 \to \tau_2))\} \vdash e : \tau \mid \phi_2\end{array}}{\Gamma \vdash \texttt{let rec}\ f\ x = e_f\ \texttt{in}\ e : \tau \mid \phi_1 \cup \phi_2}\ \text{T-Letrec}$$

(continued on next page)

Typing for `match` must be split into "one pattern" and "many pattern" cases, because we must inductively describe how the arms of the match are related to each other. Also, the rules are simplified by including an inference for $e$ in both cases, but as $e$ is always inferred in the same environment, in practice we would only infer its type once and then re-use that type for emitting the $\{\tau_e \sim \tau_p\}$ constraints. Notice that, correspondingly, we don't include the $\phi_e$ constraint set in the conclusion of the "many pattern" case.

We also need to make use of a judgement $\Gamma \Vdash p : \tau, \Gamma^p$ for typing patterns. Such judgements give a type to the pattern, *and also* to each of the pattern variables.

$$\frac{\Gamma \vdash e : \tau_e \mid \phi_e \qquad \Gamma \Vdash p : \tau_p, \Gamma^p \qquad \Gamma \cup \Gamma^p \vdash b : \tau_b \mid \phi_b}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \mid p\ \texttt{->}\ b : \tau_b \mid \phi_e \cup \phi_b \cup \{\tau_e \sim \tau_p\}}\ \text{T-matchOne}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \mid \phi_e \qquad \Gamma \Vdash p : \tau_p, \Gamma^p \\ \Gamma \cup \Gamma^p \vdash b : \tau_b \mid \phi_b \qquad \Gamma \vdash \texttt{match}\ e\ \texttt{with} \mid ...C... : \tau_c \mid \phi_c\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{match}\ e\ \texttt{with} \mid p\ \texttt{->}\ b \mid ...C... \\ : \tau_c \mid \phi_b \cup \phi_c \cup \{\tau_e \sim \tau_p, \tau_b \sim \tau_c\}\end{array}}\ \text{T-matchMany}$$

### 1.1.1 Patterns

In these rules, we use the fact that any constructors which could appear in a pattern must also appear in $\Gamma$, and then we use their types as *expressions* to guide our inference about their types as *patterns*. Recall the rule T-Var. In particular, T-Var never generates any constraints.

In practice, we often want to allow `_` (a literal underscore) anywhere that a pattern variable is expected. This means "don't give the thing a name." We'll ignore that here.

$$\frac{\Gamma \vdash C : \tau}{\Gamma \Vdash C : \tau, \varnothing}\ \text{P-Nullary} \qquad \frac{\Gamma \vdash C : \tau_x \to \tau}{\Gamma \Vdash Cx : \tau, \{x : \tau_x\}}\ \text{P-Flat}$$

$$\frac{\Gamma \vdash C : (\tau_1 * \cdots * \tau_n) \to \tau}{\Gamma \Vdash C(x_1, \cdots, x_n) : \tau, \{x_1 : \tau_1, \cdots, x_n : \tau_n\}}\ \text{P-Tuple}$$

P-Tuple only applies if the number of arguments in the tuple given to $C$ matches the number of fields in the (tuple) type obtained from $\Gamma \vdash C : \ldots$ - otherwise, we have a type error, akin to trying to unify two tuple types with the wrong number of arguments (see Unification).

### 1.1.2 Statements

To type-check statements, we must take an initial context and produce a context when we are done. This captures the fact that a statement may introduce new names into the environment. A statement may also introduce new *types*, but we don't put information about types in $\Gamma$. Instead, we use yet another form

of judgement: $S \rhd \Gamma \to \Gamma'$. This means that the statement $S$ represents the shown environment transition.

$$\frac{\Gamma \vdash e : \tau}{e \rhd \Gamma \to \Gamma} \text{ S-Anon} \qquad \frac{\Gamma \vdash \texttt{let } x = e \texttt{ in } x : \tau \mid \phi \qquad \sigma = solve(\phi)}{\texttt{let } x = e \rhd \Gamma \to \Gamma \cup \{x : quantify(\sigma(\tau))\}} \text{ S-Let}$$

$$\frac{\Gamma \vdash \texttt{let rec } fx = e \texttt{ in } f : \tau \mid \phi \qquad \sigma = solve(\phi)}{\texttt{let rec } fx = e \rhd \Gamma \to \Gamma \cup \{f : quantify(\sigma(\tau))\}} \text{ S-LetRec}$$

$$\frac{}{\begin{array}{c}\texttt{type } \vec{a}\ T = C_1 \texttt{ of } \tau_1 \mid \cdots \mid C_n \texttt{ of } \tau_n \rhd \\ \Gamma \to \Gamma \cup \{C_1 : \forall \vec{a}.\tau_1 \to \vec{a}\ T, \cdots, C_n : \forall \vec{a}.\tau_n \to \vec{a}\ T\}\end{array}} \text{ S-Type}$$

If a constructor is nullary, we instead assign it the type $\forall \vec{a}.\vec{a}\ T$. If a constructor refers to type variables other than those in $\vec{a}$, it is an error.

## 1.2 Unification

To solve all the unification constraints we generate, we use a *solve* function. The solver operates on a set of constraints to solve and returns a substitution that solves them.

$\alpha$ refers to a type variable, $\tau$ refers to any monotype.

$$\frac{solve\{\dots\} \Rightarrow \sigma}{solve\{\tau \sim \tau, \dots\} \Rightarrow \sigma} \text{ Delete} \qquad \frac{solve\{\alpha \sim \tau, \dots\} \Rightarrow \sigma}{solve\{\tau \sim \alpha, \dots\} \Rightarrow \sigma} \text{ Orient}$$

$$\frac{solve\{\tau_1 \sim \tau_1', \cdots, \tau_n \sim \tau_n', \dots\} \Rightarrow \sigma}{solve\{\langle \tau_1, \cdots, \tau_n \rangle\ T \sim \langle \tau_1', \cdots, \tau_n' \rangle\ T, \dots\} \Rightarrow \sigma} \text{ Decompose}$$

$$\frac{solve\ [\tau/\alpha]\{\dots\} \Rightarrow \sigma}{solve\{\alpha \sim \tau, \dots\} \Rightarrow \sigma[\alpha \mapsto \sigma(\tau)]} \text{ Eliminate, } \alpha \notin freeVars(\tau)$$

Decompose applies also to the builtin `->` and tuple types. The arrow and * operators in types are simply syntax sugar for the builtin type constructors.

In the last rule, we replace $\alpha$ with $\tau$ in all of the remaining constraints and solve them, getting $\sigma$. The remaining constraints might refer to free variables of $\tau$, so we make sure to also apply the substitution $\sigma$ to $\tau$ in the final mapping.

We take special care to disallow Eliminate if $\alpha$ is itself a free variable of $\tau$. What would the constraint $a \sim [a]$ mean?

# 2 Evaluation

## 2.1 Expressions

$$\frac{}{< c, \sigma >\Downarrow_v c}\ \text{E-Const} \qquad \frac{}{< u, \sigma >\Downarrow_v v}\ \text{E-Var, if } u := v \in \sigma$$

$$\frac{< e_1, \sigma >\Downarrow_v \texttt{true} \qquad < e_2, \sigma >\Downarrow_v v}{< \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \sigma >\Downarrow_v v}\ \text{E-If1}$$

$$\frac{< e_1, \sigma >\Downarrow_v \texttt{false} \qquad < e_3, \sigma >\Downarrow_v v}{< \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \sigma >\Downarrow_v v}\ \text{E-If2}$$

$$\frac{< e, \sigma >\Downarrow_v v}{< \otimes e, \sigma >\Downarrow_v \otimes v}\ \text{E-Monop} \qquad \frac{< e_1, \sigma >\Downarrow_v v_1 \qquad < e_2, \sigma >\Downarrow_v v_2}{< e_1 \otimes e_2, \sigma >\Downarrow_v v_1 \otimes v_2}\ \text{E-Binop}$$

$$\frac{}{< \texttt{fun}\ x\ \texttt{->}\ e, \sigma >\Downarrow_v \langle x, e, \sigma \rangle}\ \text{E-Fun}$$

$$\frac{< f, \sigma >\Downarrow_v \langle x, b, \sigma_f \rangle \qquad < e, \sigma >\Downarrow_v v_1 \qquad < b, \sigma_f[x := v_1] >\Downarrow_v v_2}{< f\ e, \sigma >\Downarrow_v v_2}\ \text{E-App}$$

$$\frac{< b, \sigma >\Downarrow_v v_1 \qquad < e, \sigma[x := v_1] >\Downarrow_v v_2}{< \texttt{let}\ x = b\ \texttt{in}\ e, \sigma >\Downarrow_v v_2}\ \text{E-Let}$$

$$\frac{C = \langle x, b, \sigma[f := C] \rangle \qquad < e, \sigma[f := C] >\Downarrow_v v}{< \texttt{let rec}\ f\ x = b\ \texttt{in}\ e, \sigma >\Downarrow_v v}\ \text{E-Letrec}$$

$$\frac{< e, \sigma >\Downarrow_v v_1 \qquad \texttt{Just}\ \sigma' = match(p, v_1) \qquad < b, \sigma \cup \sigma' >\Downarrow_v v_2}{< \texttt{match}\ e\ \texttt{with} \mid p\ \texttt{->}\ b \mid \ldots C \ldots, \sigma >\Downarrow_v v_2}\ \text{E-Match1}$$

$$\frac{\begin{array}{c} < e, \sigma >\Downarrow_v v_1 \\ \texttt{Nothing} = match(p, v_1) \qquad < \texttt{match}\ e\ \texttt{with} \mid \ldots C \ldots, \sigma >\Downarrow_v v_2 \end{array}}{\texttt{match}\ e\ \texttt{with} \mid p\ \texttt{->}\ b \mid \ldots C \ldots, \sigma >\Downarrow_v v_2}\ \text{E-Match2}$$

## 2.2 Statements

$$\frac{\Gamma \vdash e : \tau \qquad < e, \sigma >\Downarrow_v v}{< e; ;, \sigma, \Gamma >\Downarrow (v, \sigma, \Gamma)}\ \text{S-Anon}$$

$$\frac{\Gamma \vdash e : \tau \qquad < e, \sigma >\Downarrow_v v}{< \texttt{let}\ x = e; ;, \sigma, \Gamma >\Downarrow (v, \sigma[x := v], \Gamma \cup \{x : \tau\})}\ \text{S-Let}$$

$$\frac{\Gamma \vdash \texttt{let rec}\ f\ x = e\ \texttt{in}\ f : \tau \qquad C = \langle x, e, \sigma[f := C] \rangle}{< \texttt{let rec}\ f\ x = e; ;, \sigma, \Gamma >\Downarrow (C, \sigma[f := C], \Gamma \cup \{f : \tau\})}\ \text{S-Letrec}$$

$$\frac{\texttt{type}\ \ldots \rhd \Gamma \to \Gamma'}{< \texttt{type}\ \vec{a}\ T = C_1\ \texttt{of}\ \tau_1 \mid \cdots \mid C_n\ \texttt{of}\ \tau_n; ; \sigma, \Gamma >\Downarrow ((), \sigma \cup \{C_1, \ldots, C_n\}, \Gamma')}\ \text{S-Type}$$