Statics

Alex Kavvos

Reading: PFPL, §4.1, 4.2

1 The phase distinction

The lifetime of a computer program is divided into two phases:

- the **static** phase — which comprises everything that occurs *before* running a program; and
- the **dynamic** phase — which comprises everything that happens when a program is actually run.

Thus, the **statics** of a program include things such as lexing, parsing, type-checking, static analysis, etc. In contrast, the **dynamics** of a program include its runtime behaviour: final value, side-effects, exceptions, etc.

In this course, both the statics and the dynamics of a PL will be specified in a fairly idealised, mathematical manner. We will use **abstract syntax** as the syntax of our program; this will absolve us from having to deal with lexing, parsing, grammars, and so on. Our only statics will be a **type system** for this abstract syntax.

Correspondingly, our dynamics will be given by specifying the **operational semantics** of our programs. These will also be presented in a mathematical style, by specifying a little **abstract machine** that evaluates a program.

2 Typing judgements

In this unit we will concern ourselves with **typing judgments** of the following form:

$$\underbrace{\Gamma}_{\text{context}} \vdash \underbrace{e}_{\text{term}} : \underbrace{\sigma}_{\text{type}}$$

A typing judgement is a ternary relation between three elements:

- the **context**—an unordered list Γ of variable-type bindings
- the **term**—the program e that is being typed
- the **type** of the term—which classifies what the program computes

We read $\Gamma \vdash e : \sigma$ as "the program e has type σ in context Γ."

The **context** Γ consists of (variable, type) pairs. E.g. the context $\Gamma = x : \sigma, y : \tau$ declares two **free variables**:

- x, which stands for a term of type σ
- y, which stands for a term of type τ

These are in no particular order: the context $x : \sigma, y : \tau$ is the same as the context $y : \tau, x : \sigma$.

Thus, we can read the judgement $x : \tau \vdash e : \sigma$ as follows: "assuming that the free variable x stands for a program that computes a value of type τ, the program e computes a value of type σ."

We will only say that "e is a **term**" if there exist Γ and σ such that the judgement $\Gamma \vdash e : \sigma$ is evident. However, we will identify a larger class of programs, which we will call **pre-terms**. These will have the same 'shape' as terms, but they will not necessary be well-typed. In short, the well-typed pre-terms will be called terms.

Finally, in this unit we will only consider so-called **simple types**, which will come from an inductively generated syntax (see next section).

3 A little language of numbers and strings

To illustrate the aforementioned concepts we will present the statics of a language of numbers and strings.

The abstract syntax, types, and pre-terms of the language are presented by the following **syntax chart**.

$$
\begin{array}{llllll}
\text{types} & \tau & ::= & \mathsf{Num} & & \text{numbers} \\
 & & & \mathsf{Str} & & \text{strings} \\
\text{pre-terms} & e & ::= & x & & \text{variables} \\
 & & & \mathsf{num}[n] & & \text{numeral} \\
 & & & \mathsf{str}[s] & & \text{string literals} \\
 & & & \mathsf{plus}(e_1; e_2) & e_1 + e_2 & \text{addition} \\
 & & & \mathsf{times}(e_1; e_2) & e_1 * e_2 & \text{multiplication} \\
 & & & \mathsf{cat}(e_1; e_2) & e_1 +\!\!+\, e_2 & \text{concatenation} \\
 & & & \mathsf{len}(e) & |e| & \text{length} \\
 & & & \mathsf{let}(e_1; x.\, e_2) & \mathsf{let}\, e_1 \Leftarrow x \,\mathsf{in}\, e_2 & \text{let-definition}
\end{array}
$$

This notation is sometimes called an extended **Backus-Naur** form. It generates **syntax trees**.

The first symbol represents the **syntactic category** (e.g. type τ, expression e, etc.).

The second column (immediately to the right of $::=$) is the **abstract syntax**: it corresponds closely to the way you would represent the expression in a high-level functional programming language as an abstract syntax tree. Subscripted occurrences (e.g. e_1, e_2) are recursive occurrences of the same syntactic element. For example, $\mathsf{cat}(e_1; e_2)$ is an expression, provided e_1 and e_2 are also expressions. We tacitly assume $n \in \mathbb{N}$ and $s \in \Sigma^*$ for some alphabet Σ. We also tacitly assume that variables x come from some predetermined, infinite supply.

The third column is the **concrete syntax**: it is a user-friendly abbreviation for the abstract syntax.

In this language a type τ is either a Num or a Str. A pre-term e is given by one of the many forms listed above.

The following rules generate the typing judgements, and hence the well-typed **terms** of the language.

$$
\textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma}
\qquad
\textsc{Num}\ \frac{n \in \mathbb{N}}{\Gamma \vdash \mathsf{num}[n] : \mathsf{Num}}
\qquad
\textsc{Str}\ \frac{s \in \Sigma^*}{\Gamma \vdash \mathsf{str}[s] : \mathsf{Str}}
$$

$$
\textsc{Plus}\ \frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \mathsf{plus}(e_1; e_2) : \mathsf{Num}}
\qquad
\textsc{Times}\ \frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \mathsf{times}(e_1; e_2) : \mathsf{Num}}
$$

$$
\textsc{Cat}\ \frac{\Gamma \vdash e_1 : \mathsf{Str} \qquad \Gamma \vdash e_2 : \mathsf{Str}}{\Gamma \vdash \mathsf{cat}(e_1; e_2) : \mathsf{Str}}
\qquad
\textsc{Len}\ \frac{\Gamma \vdash e : \mathsf{Str}}{\Gamma \vdash \mathsf{len}(e) : \mathsf{Num}}
\qquad
\textsc{Let}\ \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \mathsf{let}(e_1; x.\, e_2) : \sigma_2}
$$

Some points about variables and binding:

- Writing $\Gamma, x : \sigma$ assumes that x does not occur elsewhere in Γ.
- x is **bound** within e_2 in $\mathsf{let}(e_1; x.\, e_2)$. Thus, it is subject to α**-conversion**.

An example derivation; for any $s \in \Sigma^*$:

$$
\dfrac{
\dfrac{
\dfrac{}{x : \mathsf{Str} \vdash x : \mathsf{Str}}
\qquad
\dfrac{\dfrac{}{s \in \Sigma^*}}{x : \mathsf{Str} \vdash \mathsf{str}[s] : \mathsf{Str}}
}{x : \mathsf{Str} \vdash \mathsf{cat}(x; \mathsf{str}[s]) : \mathsf{Str}}
\qquad
\dfrac{
\dfrac{\dfrac{}{x : \mathsf{Str}, y : \mathsf{Str} \vdash y : \mathsf{Str}}}{x : \mathsf{Str}, y : \mathsf{Str} \vdash \mathsf{len}(y) : \mathsf{Num}}
\qquad
\dfrac{\dfrac{}{1 \in \mathbb{N}}}{x : \mathsf{Str}, y : \mathsf{Str} \vdash \mathsf{num}[1] : \mathsf{Num}}
}{x : \mathsf{Str}, y : \mathsf{Str} \vdash \mathsf{plus}(\mathsf{len}(y); \mathsf{num}[1]) : \mathsf{Num}}
}{x : \mathsf{Str} \vdash \mathsf{let}(\mathsf{cat}(x; \mathsf{str}[s]); y.\, \mathsf{plus}(\mathsf{len}(y); \mathsf{num}[1])) : \mathsf{Num}}
$$

In words: if we plug in a program that computes a string for $x : \mathsf{Str}$, this program will append the string $s \in \Sigma^*$ to it; it will then compute its length, and add 1 to it.