CS103 Daily Doggie Bag 19

The most important topics and details of the day.

April 13, 2006

**Topic 1. Advantages of static type analysis.** As we have seen in the language D, untyped expressions can have a meaningful evaluation semantics. And many untyped languages do exist, such as Scheme and SmallTalk, and languages like C don’t really have a strong type discipline. However, many modern languages like OCaml have adapted static type disciplines, because types offer certain advantages, especially:

- **Safety.** By ruling out unsafe expressions, static type disciplines ensure that programs run safely and do not crash (or worse).

- **Efficiency.** Type information can be used by compilers for certain optimizations.

- **Specification benefits.** Types give programmers power to specify data structure and operations, and enforce the specification.

**Topic 2. Flavors of type analysis.** There are many different sorts of type systems. They have theoretical connections, but vary in form, expressiveness and implementation. Here are some major distinctions between the implementation of type analyses:

- **Dynamic vs. static type analysis.** In your implementations of ARITH and D, you always checked that the types of expressions were admissible, e.g.:

  ```
  | Plus(exp0, exp1) ->
  |   (match (eval exp0, eval exp1) with
  |     (Int(n1), Int(n2)) -> Int(n1 + n2)
  |     _ -> raise TypeMismatch)
  ```

  This amounts to a system of dynamic type analysis, a method used in other languages such as Lisp and Scheme. Any syntactically correct program is admitted by the compiler, type errors are caught and rejected at run-time. In contrast, languages like OCaml are statically typed, meaning that type analysis is performed before run-time, and ill-typed programs are rejected by the compiler, and not run at all.

- **Type checking vs. inference.** When performing type analysis, there is a spectrum of possibilities for the extent of type information programmers are required to provide. On one end, we have languages such as OCaml and ML, which require no type annotations. On the other end, we have languages where every expression must be assigned a type. However, no languages live on this end, because assigning types to every expression would be a mess, e.g.:

  ```
  (fun (x : int) -> (x:int + 1:int):int):(int -> int)
  ```

  Furthermore, this is redundant; it’s always easy to determine that an integer value has type int, and that x has type int, if we assert that it does in the function definition.

  Type analyses that require no type annotations are called type inference. Type analyses that require annotations are called type checking. There is a qualitative difference, in terms of theoretical and practical complexity, between type checking and inference. It turns out that there is a minimal requirement on annotations to yield type checking, as opposed to inference, which is nice because we can expect programmers to deal with these minimal annotations, as opposed to the mess described above. For example, C requires
that the type of arguments and return values of functions must be stated, as well as the type of variables at declaration; but no other type assertions are needed.

**Topic 3. D with static type checking: the language TD.** To begin our study of types, we extend D with a static type checking discipline. Given our formal develop, we will be able to very precisely specify a Type Safety property for the language, that ensures we can statically rule out unsafe expressions:

If $e$ is a typable expression in TD, then it is not the case that $e \Rightarrow \text{fail}$.

Not only will we be able to state it, but we can prove it. Note that this property relates static and dynamic program properties.

**Type syntax** First, we define a language of simple types:

$$\tau ::= \text{Int} \mid \text{Bool} \mid \tau \rightarrow \tau$$

These are very similar to OCaml types.

**Concrete syntax** Now, we define the syntax of TD expressions. These are the same as D expressions, except that function arguments are type-annotated. For simplicity, we set aside Fix for the moment, but it is easy to add.

- $b ::= \text{True} \mid \text{False}$ booleans
- $n ::= 0 \mid 1 \mid 2 \mid 3 \mid \cdots$ natural numbers
- $v ::= b \mid n \mid (\text{Function} (x : \tau) \rightarrow e)$ values
- $e ::= v \mid e \text{ And } e \mid e \text{ Or } e \mid \text{Not } e \mid e + e \mid e - e \mid e = e \mid e e$ expressions

**Operational semantics** The operational semantics of TD is identical to the semantics of D; we only need to slightly modify the APPL rule to ignore type information (it’s a static type discipline, remember?):

$$\frac{e_1 \Rightarrow (\text{Function} (x : \tau) \rightarrow e) \quad e_2 \Rightarrow v'}{e_1 e_2 \Rightarrow v'}$$

**Type judgements** We defined evaluation as a relation $e \Rightarrow v$ on a pair of expressions, and a derivation system for formally describing evaluation. We will do the same thing for type judgements, providing rules for deriving valid type descriptions, called type judgements.

However, unlike evaluation, we cannot define type judgements on closed expressions only. Consider: type derivation, like evaluation, will be defined as an inductive process. In particular, the types of compound terms will be derived from the types of their subterms. For example, the type of $(\text{Function} (x : \text{Int}) \rightarrow x + 1)$ is based on the type of $x + 1$, which has a free variable. However, if we assume that $x$ has type Int, as the type annotation asserts we should, then it is no trick to determine that $x + 1$ has type Int, so $(\text{Function} (x : \text{Int}) \rightarrow x + 1)$ has type $\text{Int} \rightarrow \text{Int}$. To keep track of these assumptions in type derivations, we define type environments:

$$\Gamma ::= \emptyset \mid \Gamma ; x : \tau \quad \text{type environments}$$

For brevity, we write $(\emptyset; x_1 : \tau_1; \ldots; x_n : \tau_n)$ as $(x_1 : \tau_1; \ldots; x_n : \tau_n)$. Note that type environments are ordered structures, not sets, so e.g. $(x : \text{Int}; x : \text{Bool}) \neq (x : \text{Bool}; x : \text{Int})$. We can also look up bindings in environments:

**Definition 1.1** Lookup in a type environment, denoted $\Gamma (x)$, is inductively defined as follows:

$$\Gamma (x : \tau) (x) = \tau$$

$$\Gamma (x' : \tau) (x) = \Gamma (x) \quad x \neq x'$$
These specifications impose a LIFO behavior on environments, e.g. \((x : \text{Bool}; x : \text{Int})(x) = \text{Int}\). This is so we can keep track of variable scoping:

\[
\text{(Function } (x : \text{Bool}) \rightarrow (\text{Function } (x : \text{Int}) \rightarrow x + 1))
\]

Thus, type judgements are defined in terms of a type environment, an expression, and a type:

**Definition 1.2** Type judgements are denoted \(\Gamma \vdash e : \tau\).

In words, what these judgements say is that “assuming given the type bindings in \(\Gamma\), \(e\) has type \(\tau\).” The rules for deriving type judgements are then as follows:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type Environment</th>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VAR</strong></td>
<td>(\Gamma(x) = \tau)</td>
<td>(\Gamma \vdash x : \tau)</td>
<td></td>
</tr>
<tr>
<td><strong>BOOL</strong></td>
<td>(\Gamma \vdash b : \text{Bool})</td>
<td>(\Gamma \vdash n : \text{Int})</td>
<td></td>
</tr>
<tr>
<td><strong>INT</strong></td>
<td>(\Gamma \vdash e_1 : \text{Bool})</td>
<td>(\Gamma \vdash e_2 : \text{Bool})</td>
<td>(\Gamma \vdash e_1 \text{ And } e_2 : \text{Bool})</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>(\Gamma \vdash e_1 : \text{Int})</td>
<td>(\Gamma \vdash e_2 : \text{Int})</td>
<td>(\Gamma \vdash e_1 + e_2 : \text{Int})</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>(\Gamma \vdash e_1 \text{ Or } e_2 : \text{Bool})</td>
<td>(\Gamma \vdash \text{Not } e : \text{Bool})</td>
<td>(\Gamma \vdash e_1 + e_2 : \text{Int})</td>
</tr>
<tr>
<td><strong>MINUS</strong></td>
<td>(\Gamma \vdash e_1 : \text{Int})</td>
<td>(\Gamma \vdash e_2 : \text{Int})</td>
<td>(\Gamma \vdash e_1 - e_2 : \text{Int})</td>
</tr>
<tr>
<td><strong>EQUAL</strong></td>
<td>(\Gamma \vdash e_1 : \text{Int})</td>
<td>(\Gamma \vdash e_2 : \text{Int})</td>
<td>(\Gamma \vdash e_1 = e_2 : \text{Bool})</td>
</tr>
<tr>
<td><strong>IF</strong></td>
<td>(\Gamma \vdash e_1 : \text{Bool})</td>
<td>(\Gamma \vdash e_2 : \tau)</td>
<td>(\Gamma \vdash e_3 : \tau)</td>
</tr>
<tr>
<td><strong>ABS</strong></td>
<td>(\Gamma ; x : \tau_1 \vdash e : \tau_2)</td>
<td>(\Gamma \vdash (\text{Function } (x : \tau_1) \rightarrow e) : \tau_1 \rightarrow \tau_2)</td>
<td>(\Gamma \vdash e_1 e_2 : \tau)</td>
</tr>
</tbody>
</table>

**Definition 1.3** A type judgement \(\Gamma \vdash e : \tau\) is valid iff it occurs as the root of some derivation tree. If \(\emptyset \vdash e : \tau\) is valid, then we write \(e : \tau\) for brevity.

Example (omitting labels on instances of VAR,BOOL,INT for brevity):

\[
\emptyset \vdash 1 : \text{Int} \quad \emptyset \vdash x : \text{Int} \quad \emptyset \vdash 3 : \text{Int} \quad (\text{EQUAL}) \quad \emptyset \vdash \text{True} : \text{Bool} \quad (\text{AND})
\]

\[
\emptyset \vdash (x : \text{Int}) \vdash x : \text{Int} \quad \emptyset \vdash 1 : \text{Int} \quad (\text{PLUS}) \quad \emptyset \vdash (\text{Function } (x : \text{Int}) \rightarrow x + 1) : \text{Int} \rightarrow \text{Int}
\]

\[
\quad \emptyset \vdash 2 : \text{Int} \quad \emptyset \vdash 3 : \text{Int} \quad (\text{PLUS}) \quad \emptyset \vdash 2 + 3 : \text{Int} \quad (\text{APP})
\]

\[
\emptyset \vdash (\text{Function } (x : \text{Int}) \rightarrow x + 1)(2 + 3) : \text{Int}
\]