15–212: Fundamental Structures of Computer Science II

Some Notes on Interpreters

Frank Pfenning

Draft of November 18, 1997

These notes provide a brief introduction to the specification techniques used for type-checking and evaluation in the context of writing interpreters or compilers for programming languages.

1 Introduction

Specifications are an indispensible part of software development. They explain *what* must be implemented without necessarily saying *how*. Depending of the nature of the problem domain, specifications may range from incomplete, natural language descriptions to mathematically precise formulations of the functionality to be implemented.

Much of the task of software engineering is to decompose a large and vague system description into modules with clear specifications which can then be coded in a programming language. The ML language helps in this task by providing a language of signatures to express at least some aspects of the specification formally in a way that can be checked mechanically by a compiler. This includes the types of the module interfaces and the information about which representations are concrete and which abstract.

However, the need for deeper, mathematically rigorous specifications remains, especially for domains which are themselves precise and formal. Programming languages themselves present one such domain. We have already seen how *context-free grammars* provide precise means for specifying the concrete syntax of programming languages. Via parser generators, certain classes of context-free grammars can in fact be turned into implementations automatically.

The next phase in an interpreter or compiler consists of type-checking. So far, we have used semi-formal descriptions of the rules for typing ML expressions of the kind:

The application $e_1\, e_2$ has type t_1 if e_1 has type $t_2 \to t_1$ and e_2 has type t_2.

The understanding is that if an application does not follow this schema, it is not well-typed. Rules of this form originate in logic. For example, if we write $A \supset B$ for A *implies* B we might say:

If $A \supset B$ is true and A is true, then B must be true.

In fact, this can be seen as a *specification* of what "$A \supset B$" means.

Because of this connection to logic, there has been a well-developed specification formalism for some time, certainly well before the advent of programming languages. This formalism is centered on the notion of *inference rule*. Inference rules are used to define when *judgments* hold. Judgments in the examples above are "*Expression e has type t*" and "*Proposition A is true*". Judgments and inference rules together make up a *system of deduction* or *deductive system*. We now explain the basic concepts of deductive systems as they are needed for the purpose of this course.

Judgment. A *judgment* may be *evident*. In that case we must have evidence for it in the form of a derivation. We therefore also say that a judgment J is *derivable* or that a judgment J *holds*. For example, $\mathtt{0} : \mathtt{int}$, or $\mathtt{(fn\ x\ =>\ x)} : \mathtt{bool}$ are judgments.

Inference Rule. An *inference rule* is written as

$$\frac{J_1 \ldots J_n}{J}\,\mathit{name}$$

where $J_1, \ldots, J_n$ are the *premises* of the rule, J is its *conclusion*, and *name* is its name. A rule of this of this form specifies that if the premises $J_1, \ldots, J_n$ are derivable, then so is the conclusion J. In practice, most inference rules are *schematic*. This means that they contain variables and any *instance* represents a valid inference. For example,

$$\frac{e_1 : t_2 \to t_1 \qquad e_2 : t_2}{e_1\, e_2 : t_1}\,\text{tp\_app}$$

is an inference rule schematic in expressions e_1 and e_2 and types t_1 and t_2. Since most inference rules are schematic, we simply refer to them as inference rules, dropping the qualifier "schematic".

Axiom. An *axiom* is simply an inference rule with 0 premises. Therefore, the conclusion holds unconditionally, and the evidence for it is trivial. For example

$$\frac{}{\mathtt{true} : \mathtt{bool}}\,\text{tp\_true}$$

is an axiom.

Derivation. A *derivation* is complete evidence for a judgment given by a tree of valid inferences starting from axioms. For example,

$$\frac{\dfrac{}{\mathtt{not} : \mathtt{bool} \to \mathtt{bool}}\,\text{tp\_not} \qquad \dfrac{}{\mathtt{false} : \mathtt{bool}}\,\text{tp\_false}}{\mathtt{not\ false} : \mathtt{bool}}\,\text{tp\_app}$$

is a derivation of the judgment $\mathtt{not\ false} : \mathtt{bool}$.

In these notes we will apply deductive systems to the specification of the typing and evaluation rules for a small functional language. Often, such specifications can be turned into implementations in a straightforward way, although we will see that there are some limitations.

2 Typing Simple Expressions and Declarations

We begin with a small language of expressions which encompass rationals and booleans. In addition, we allow local declarations using a `let` form. First, we specify the abstract syntax in the form of a BNF-grammar, ignoring certain aspects of the concrete syntax such as the precise form of identifiers, or the precedence of the arithmetic and boolean operators. These have been treated earlier in the course. We use x to range over variables and n to range over integers.

$$
\begin{array}{rcll}
\textit{Types} & t & ::= & \texttt{rat} \mid \texttt{bool} \\[1ex]
\textit{Expressions} & e & ::= & n \mid e_1\texttt{+}e_2 \mid e_1\texttt{-}e_2 \mid e_1\texttt{*}e_2 \mid e_1\texttt{/}e_2 \mid \texttt{-}e \\
 & & & \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
 & & & \mid e_1\texttt{=}e_2 \mid e_1\texttt{<}e_2 \\
 & & & \mid \texttt{let}\ d\ \texttt{in}\ e\ \texttt{end} \\
 & & & \mid x \\[1ex]
\textit{Declarations} & d & ::= & \cdot \mid d\ \texttt{val}\ x\texttt{=}e
\end{array}
$$

Note that expressions may contain variables. How do we specify the type of variables? In our language so far, variables are introduced by declarations of the form val x=e and we can determine the type of x from the type of e. When we analyze the scope of a declaration we must remember the type we infered for the variable. This is the purpose of a *type environment* or *context*. We use Γ (capital Gamma) as a letter ranging over type environments, writing $\cdot$ for the empty environment and $\Gamma, x{:}t$ for the environment which extends Γ by assigning type t to variable x.

$$\textit{Type Environments} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x{:}t$$

The typing judgment has the form $\Gamma \vdash e : t$ which we read as "*expression e has type t in type environment Γ*". The type environment assigns types to the (free) variables in e. Most inference rules for the typing judgment (also called *typing rules*) are rather obvious.

$$\frac{\Gamma \vdash e_1 : \texttt{rat} \qquad \Gamma \vdash e_2 : \texttt{rat}}{\Gamma \vdash e_1\texttt{+}e_2 : \texttt{rat}}\,\text{tp\_+} \qquad \frac{\Gamma \vdash e_1 : \texttt{rat} \qquad \Gamma \vdash e_2 : \texttt{rat}}{\Gamma \vdash e_1\texttt{-}e_2 : \texttt{rat}}\,\text{tp\_}-$$

$$\frac{\Gamma \vdash e_1 : \texttt{rat} \qquad \Gamma \vdash e_2 : \texttt{rat}}{\Gamma \vdash e_1\texttt{*}e_2 : \texttt{rat}}\,\text{tp\_}* \qquad \frac{\Gamma \vdash e_1 : \texttt{rat} \qquad \Gamma \vdash e_2 : \texttt{rat}}{\Gamma \vdash e_1\texttt{/}e_2 : \texttt{rat}}\,\text{tp\_/}$$

$$\frac{}{\Gamma \vdash n : \texttt{rat}}\,\text{tp\_int} \qquad \frac{\Gamma \vdash e : \texttt{rat}}{\Gamma \vdash \texttt{-}e : \texttt{rat}}\,\text{tp\_neg}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\,\text{tp\_true} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\,\text{tp\_false}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\,\text{tp\_if}$$

$$\frac{\Gamma \vdash e_1 : \texttt{rat} \qquad \Gamma \vdash e_2 : \texttt{rat}}{\Gamma \vdash e_1\texttt{=}e_2 : \texttt{bool}}\,\text{tp\_} = \qquad \frac{\Gamma \vdash e_1 : \texttt{rat} \qquad \Gamma \vdash e_2 : \texttt{rat}}{\Gamma \vdash e_1\texttt{<}e_2 : \texttt{bool}}\,\text{tp\_} <$$

For example, the rule *tp_if* expresses that both branches of the conditional must have the same type t, which is also the type of the conditional. Variables are simply looked up in the type environment, where we must be careful to obey the rules of shadowing: the rightmost occurrence of an identifier in a context declares its type.

$$\frac{}{\Gamma, x{:}t \vdash x : t}\,\text{tp\_var\_eq} \qquad\qquad \frac{\Gamma \vdash x : t \quad \text{where } x \neq x'}{\Gamma, x'{:}t' \vdash x : t}\,\text{tp\_var\_neq}$$

What will happen if a variable is not declared in the type environment? From the point of view of the deductive systems, we simply will not be able to derive any typing judgment for such an expression. For example, there is no type t such that the judgment $\cdot \vdash x\texttt{+3} : t$ is derivable. On the other hand, we have

$$\frac{\dfrac{}{\cdot, x{:}\texttt{rat} \vdash x : \texttt{rat}}\,\text{tp\_var\_eq} \qquad \dfrac{}{\cdot, x{:}\texttt{rat} \vdash \texttt{3} : \texttt{rat}}\,\text{tp\_int}}{\cdot, x{:}\texttt{rat} \vdash x \texttt{ + 3} : \texttt{rat}}\,\text{tp\_+}$$

In an implementation, the failure to establish a typing judgment will presumably lead to an error message, but this is not reflected in the formal specification.

Declarations occur in expressions and expressions occur in declarations. This means that we need another typing judgment for declarations which mutually depends on the typing judgment for expressions. We write $\Gamma \vdash d : \Gamma'$ which we read as "*declarations d extend the type environment Γ to Γ'*". The rule for `let`-expressions then refers to the rule for declarations.

$$\frac{\Gamma \vdash d : \Gamma' \qquad \Gamma' \vdash e : t}{\Gamma \vdash \texttt{let}\ d\ \texttt{in}\ e\ \texttt{end} : t}\,\text{tp\_let}$$

Note that the extended environment Γ' which results from checking d is used as the type environment for checking e.

Declarations d are processed in sequence. Therefore we must "thread" the typing environment through the derivation to make earlier declarations available for later ones (as in `let val x = 3 val y = x*x in y*y end`). The empty declaration does not extend the environment at all.

$$\frac{}{\Gamma \vdash \cdot : \Gamma}\,\text{tp\_empty} \qquad\qquad \frac{\Gamma \vdash d : \Gamma' \qquad \Gamma' \vdash e : t}{\Gamma \vdash (d\ \texttt{val}\ x\texttt{=}e) : (\Gamma', x{:}t)}\,\text{tp\_dec}$$

This concludes the set of rules for typing expressions and declarations in our language. In lecture and recitation we discussed informally how to turn a specification of this form into an implementation in ML. In Section 4 we add functions and recursion and appropriate typing rules.

3 Evaluating Expressions and Declarations

Next we specify the operational semantics for our small rational expression language. In the presentation of ML, we have used a style of presentation called a *small-step semantics* or *structural operational semantics*. In this style we think of an initial expression being rewritten step by step until we reached a final value. This was appropriate for our goal, namely to specify ML in such a way that it would allow us to easily prove properties of programs. As a basis for an interpreter, such a definition is quite complicated and horrendously inefficient, so we use a different style called *big-step semantics* or *natural semantics*.

In a big-step semantics, we have one main judgment which relates an expression to its value. We write $e \hookrightarrow v$ and read it as "*expression e evaluates to value v*". Since expressions contain variables, this is not quite sufficient: we also need to keep track of the values that variables are bound to in a *value environment*. Thus we define values and value environment. We write r for a rational number.

$$\begin{array}{rrcl} \textit{Values} & v & ::= & r \mid \textit{true} \mid \textit{false} \\ \textit{Value Environment} & \eta & ::= & \cdot \mid \eta, x = v \end{array}$$

The evaluation judgment then is $\eta \vdash e \hookrightarrow v$, which reads "*expression e evaluates to value v in environment η*". The usual arithmetic operations are simply defined by reference to their mathematical counterpart. The rule *ev_int* states that each integer evaluates to the corresponding rational which we write as $n/1$.

$$\frac{\eta \vdash e_1 \hookrightarrow r_1 \qquad \eta \vdash e_2 \hookrightarrow r_2}{\eta \vdash e_1\texttt{+}e_2 \hookrightarrow r_1 + r_2}\,\text{ev\_+} \qquad \frac{\eta \vdash e_1 \hookrightarrow r_1 \qquad \eta \vdash e_2 \hookrightarrow r_2}{\eta \vdash e_1\texttt{-}e_2 \hookrightarrow r_1 - r_2}\,\text{ev\_}-$$

$$\frac{\eta \vdash e_1 \hookrightarrow r_1 \qquad \eta \vdash e_2 \hookrightarrow r_2}{\eta \vdash e_1\texttt{*}e_2 \hookrightarrow r_1 * r_2}\,\text{ev\_}* \qquad \frac{\eta \vdash e_1 \hookrightarrow r_1 \qquad \eta \vdash e_2 \hookrightarrow r_2}{\eta \vdash e_1\texttt{/}e_2 \hookrightarrow r_1/r_2}\,\text{ev\_/}$$

$$\frac{}{n \hookrightarrow n/1}\,\text{ev\_int} \qquad \frac{\eta \vdash e \hookrightarrow r}{\eta \vdash \texttt{-}e \hookrightarrow -r}\,\text{ev\_neg}$$

$$\frac{}{\eta \vdash \texttt{true} \hookrightarrow \textit{true}}\,\text{ev\_true} \qquad \frac{}{\eta \vdash \texttt{false} \hookrightarrow \textit{false}}\,\text{ev\_false}$$

$$\frac{\eta \vdash e_1 \hookrightarrow \textit{true} \qquad \eta \vdash e_2 \hookrightarrow v}{\eta \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_2 \hookrightarrow v}\,\text{ev\_if\_true} \qquad \frac{\eta \vdash e_1 \hookrightarrow \textit{false} \qquad \eta \vdash e_3 \hookrightarrow v}{\eta \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_2 \hookrightarrow v}\,\text{ev\_if\_false}$$

We omit the obvious four rules for evaluating e_1 = e_2 and e_1 < e_2. Note that there are two rules for evaluating conditionals. This is because the condition may evaluate to either *true* or *false*, and we account for that in two separate rules. With these rules we can, for example, conclude that $\cdot \vdash \texttt{2+3} \hookrightarrow 5/1$.

$$\frac{\dfrac{}{\cdot \vdash \texttt{2} \hookrightarrow 2/1}\,\text{ev\_int} \qquad \dfrac{}{\cdot \vdash \texttt{3} \hookrightarrow 3/1}\,\text{ev\_int}}{\cdot \vdash \texttt{2+3} \hookrightarrow 5/1}\,\text{ev\_plus}$$

Variables are looked up in the environment, retrieving their value.

$$\frac{}{\eta, x = v \vdash x \hookrightarrow v}\,\text{ev\_var\_eq} \qquad \frac{\eta \vdash x \hookrightarrow v \quad \text{where } x \neq x'}{\eta, x' = v' \vdash x \hookrightarrow v}\,\text{ev\_var\_neq}$$

Evaluating declarations will evaluate the embedded expressions in sequence and construct an extended value environment. We write $\eta \vdash d \hookrightarrow \eta'$ which we read as "*declarations d evaluate* *to extended environment* η' *in environment* η". We appeal to this judgment when evaluating a `let`-expression.

$$\frac{\eta \vdash d \hookrightarrow \eta' \qquad \eta' \vdash e \hookrightarrow v}{\eta \vdash \texttt{let}\ d\ \texttt{in}\ e\ \texttt{end} \hookrightarrow v}\ \text{ev\_let}$$

Declarations are evaluated in sequence, accumulating an extended value environment.

$$\frac{}{\eta \vdash \cdot \hookrightarrow \eta}\ \text{ev\_empty} \qquad \frac{\eta \vdash d \hookrightarrow \eta' \qquad \eta' \vdash e \hookrightarrow v}{\eta \vdash (d\ \texttt{val}\ x\texttt{=}e) \hookrightarrow (\eta', x = v)}\ \text{ev\_dec}$$

This concludes the evaluation rules for this simple language. In order to formulate our main theorem, we add a third judgment which gives the typing of values: $\vdash v : t$ which reads "value v has type t". It has only three rules.

$$\frac{}{\vdash r : \texttt{rat}}\ \text{tpv\_rat} \qquad \frac{}{\vdash \mathit{true} : \texttt{bool}}\ \text{tpv\_true} \qquad \frac{}{\vdash \mathit{false} : \texttt{bool}}\ \text{tpv\_false}$$

We can see a strong correspondence between the typing and evaluation rules, which is no accident. Both follow the structure of syntax closely, and they are tied together by the theorem of *Type Preservation*: if $\cdot \vdash e : t$ and $\cdot \vdash e \hookrightarrow v$ then $\vdash v : t$. We will not attempt to prove this here, but it is an important in the design of the rules for functions and recursion.

4 Functions and Closures

We now extend our language to include functions. For the moment we still postpone the discussion of recursive function, which are discussed in the next section. First, we extend our language of types and expressions.

$$\begin{array}{lcll} \mathit{Types} & t & ::= & \ldots \mid t_1 \to t_2 \\ \mathit{Expressions} & e & ::= & \ldots \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e_1\, e_2 \end{array}$$

Declarations and environments do not need to change in this generalization step. In the typing rule for functions, we need to extend the type environment by the formal parameter of the function.

$$\frac{\Gamma, x_1{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{fn}\ x_1\ \texttt{=>}\ e_2 : t_1 \to t_2}\ \text{tp\_fn} \qquad \frac{\Gamma \vdash e_1 : t_2 \to t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\, e_2 : t_1}\ \text{tp\_app}$$

The operational semantics requires a new kind of value for functions. In the small-step semantics for ML, we simply used the function itself as a value. Here, however, we have a problem, because variables in the body of a function may be bound in the environment. For example, if we assume the judgment $\cdot, \texttt{y} = 3 \vdash \texttt{fn x => x+y} \hookrightarrow \texttt{fn x => x+y}$ holds, then the judgment

$$\cdot \vdash \texttt{let y = 3 in fn x => x+y end} \hookrightarrow \texttt{fn x => x+y}$$

also holds. But this makes no sense, since the variable `y` on the right-hand side is not declared the empty environment. Therefore, type preservation is violated, since

$$\cdot \vdash \texttt{let y = 3 in fn x => x+y end} : \texttt{rat} \to \texttt{rat}$$

is derivable, but

$$\cdot \vdash \texttt{fn x => x+y} : \texttt{rat} \rightarrow \texttt{rat}$$

does **not** hold. The way out of this dilemma is to pair up the function with its environment to form a so-called *closure*. This way, the value of a function is always a closure and carries with it the bindings for all variables occurring in its body. A simple optimization (done in all functional compilers) is to only carry bindings for the variables which actually occur, but as a specification the rule below is certainly sufficient and the two can be easily seen to be equivalent. First, we extend the language of values to include closures.

$$\textit{Values} \quad v \quad ::= \quad \ldots \mid \{\eta; \texttt{fn}\ x\ \texttt{=>}\ e\}$$

The typing rule for closures is simple, since it just refers to the typing rule for the expression and environment contained in it.

$$\frac{\cdot \vdash \eta : \Gamma \qquad \Gamma \vdash \texttt{fn}\ x\ \texttt{=>}\ e : t_1 \rightarrow t_2}{\vdash \{\eta; \texttt{fn}\ x\ \texttt{=>}\ e\} : t_1 \rightarrow t_2}\ \text{tpv\_closure}$$

Evaluation for functions immediately creates a closure. Applying a function unpacks the closure and extends the environment η' contained in it by binding the formal parameter x to the argument value v_2.

$$\frac{}{\eta \vdash \texttt{fn}\ x\ \texttt{=>}\ e \hookrightarrow \{\eta; \texttt{fn}\ x\ \texttt{=>}\ e\}}\ \text{ev\_fn}$$

$$\frac{\eta \vdash e_1 \hookrightarrow \{\eta'; \texttt{fn}\ x\ \texttt{=>}\ e_1'\} \qquad \eta \vdash e_2 \hookrightarrow v_2 \qquad \eta', x = v_2 \vdash e_1' \hookrightarrow v}{\eta \vdash e_1\, e_2 \hookrightarrow v}\ \text{ev\_app}$$

5 Recursion

Adding recursion can be accomplished either be adding a recursive expression or a recursive declaration to our language. Here, we will explore adding a recursive declaration and corresponding recursive environments. This new declaration corresponds to the `val rec` declaration of ML, except that, for the sake of simplicity, we do not restrict the expression to be a function.

$$\textit{Declarations} \quad d \quad ::= \quad \ldots \mid \texttt{rec}\ x\texttt{=}e$$

For example the function $p(n) = 2^n$ for natural numbers n could be declared and then used to calculate 2^{10} as follows.

```
let
  rec p = fn n => if n = 0 then 1 else 2 * p (n-1)
in
  p 10
end
```

Declarations evaluate to value environments, so we must have a form of recursive binding which we write as $\eta, \mathit{rec}\, x = e$. Values themselves do not change (since expressions do not change).

$$\textit{Value Environments} \quad \eta \quad ::= \quad \ldots \mid \eta, \mathit{rec}\, x = e$$

The change the the typing rules is straightforward. We just have to remember that the variable that is declared recursively may occur in the expression and must therefore be added to its type environment.

$$\frac{\Gamma \vdash d : \Gamma' \qquad \Gamma', x{:}t \vdash e : t}{\Gamma \vdash d \; \texttt{rec} \; x\texttt{=}e : \Gamma', x{:}t} \text{tp\_rec}$$

When a recursive declaration is evaluated, we return immediately, simply extending the value environment with a recursive binding. When a variable declared in this way is encountered during evaluation, we need to "unroll" the recursion to obtain a value. This is achieved by evaluating the expression the identifier is bound to recursively.

$$\frac{\eta \vdash d \hookrightarrow \eta'}{\eta \vdash (d \; \texttt{rec} \; x\texttt{=}e) \hookrightarrow (\eta', \mathit{rec}\; x = e)} \text{ev\_rec}$$

$$\frac{\eta, \mathit{rec}\; x = e \vdash e \hookrightarrow v}{\eta, \mathit{rec}\; x = e \vdash x \hookrightarrow v} \text{ev\_recvar\_eq} \qquad \frac{\eta \vdash x \hookrightarrow v \quad \text{where } x \neq x'}{\eta, \mathit{rec}\; x' = e' \vdash x \hookrightarrow v} \text{ev\_recvar\_neg}$$

You should work through the example function p above to make sure you understand how functions, closures, and recursion work together to produce the correct answer.