PFPL Supplement: The Structure of an Interpreter[*]

Robert Harper

Fall, 2020

1 Introduction

To gain an understanding of a programming language concept it is often useful—even essential—to implement it. But what does an implementation look like? For the purposes of this class an implementation of a language is an *interactive interpreter* consisting of these components (see Figure 1):

1. A *parser* that translates *concrete* syntax—the representation of programs as strings—into *abstract* syntax—the representation of programs as abstract binding trees. If a string cannot be parsed, it is rejected as a *syntax error*, and otherwise translates the string into an abstract syntax tree.

2. An *elaborator* that validates a piece of abstract syntax according to the *statics* of the language, which is given by a collection of typing rules. Often the elaborator also translates the raw abstract syntax into an *elaborated* abstract syntax tree.

3. An *executor* that takes a piece of elaborated abstract syntax and executes it according to the *dynamics* of the language. In this class the implementation of the executor hews close to the rules of the dynamics, in particular by using substitution to implement binding. More efficient implementations *translate* the language into a lower-level form by a series of transformations, called *phases*, whose composition forms a *compiler*.

4. A *formatter* that translates a final state to a string suitable for output.

The interactive interpreter reads input from a keyboard or file, passes it to the parser, elaborator, and executor, then prints the results (erroneous or successful) before repeating.

Within this general framework there is considerable room for variation in both the elaboration and the execution phases. For very simple languages elaboration amounts to very little; the parser can generate elaborated abstract syntax directly, with no need of further work. But in most cases such a direct formulation is not possible. One issue is *scope resolution*, the determination of the binding and scope of variables, which is essential to representing programs as abt's. In simple cases the parser can make this determination itself, but often information about the types of identifiers is needed to determine their scope. A good example of such a situation is the SML `open` construct, which incorporates all of the variables declared in a structure `S`, which is determined by the signature

[*]

$$\mathit{string} \xrightarrow{\mathit{parse}} \mathit{ast} \xrightarrow{\mathit{elab}} \mathit{abt} \xrightarrow{\mathit{exec}} \mathit{state} \xrightarrow{\mathit{format}} \mathit{string}$$

Figure 1: Structure of an Interpreter

(type) of `S`, which is not known during parsing, but only during type checking. Another trouble spot is the definition of *derived forms*, conveniences for the programmer that are not built into the language as primitive constructs. Sometimes derived forms can be expanded by the parser, but more often type information is required, and the expansion process has to be deferred to elaboration. But the biggest reason for distinguishing raw from elaborated syntax is *type inference*, the process of determining type information that is implied from the context in which it occurs. That is a job for the elaborator, which knows about types, and not the parser, which does not.

Similarly, there is room for variation in the execution phase, with efficiency being the most important motivation. For proof-of-concept purposes it is usually possible to implement the dynamics more or less literally, using capture-avoiding substitution for variables, and by performing case analysis on the abstract syntax of the program. Such an implementation is often easy to get up-and-running, but can be unbearably slow to use for any but the smallest examples. For example, the dynamics of a language is usually given in terms of substitution, but to implement this directly introduces considerable overhead. One way to make it more efficient is to maintain the substitution itself as a lookup table that is accessed on demand. But doing so runs afoul of higher-order concepts, such as returning functions from within other functions. To implement this properly requires the use of *closures*, which associate an *environment* to a function that determines the bindings of its free variables, and that is achieved by a translation called *closure conversion*.

Peculiarly, it is commonly thought that interpreters and compilers are somehow in opposition to one another, with interpreters being interactive, and necessarily inefficient, and compilers being non-interactive, but efficient. *This is total nonsense!* There is *no* fundamental difference between an interpreter and a compiler. Rather, a compiler, which works by transformation, is just one strategy for implementing an interpreter. There is no bright-line distinction between the two concepts, and neither is inherently tied to being interactive or non-interactive.

The pipeline depicted in Figure 1 is implemented using *(lazy) streams* by composing *stream transducers*. At a high level the type of the pipeline is

$$\texttt{string stream} \to \texttt{string stream}$$

which, given an input source, produces the formatted output arising from that input. The nature of the input varies with the language, but is typically a series of programs that are executed, producing a series of results. Notice that the pipeline is not at all dependent on the nature of the input; it can be a file that has been prepared in advance, or may be an interaction loop such as the one provided by the ML implementation. An interpreter is activated by applying the pipeline to an input source to obtain a stream of outputs. *Nothing happens at the moment of application, other than to "open" the input in an appropriate sense, and connecting it to the pipeline.* To activate the interpreter, simply request a result from the output stream, and send it to the appropriate sink, a file or a screen. The demand for a result gives rise to a demand for an answer from execution, which gives rise to a demand for a well-typed program, which gives rise to a demand for an abstract binding tree, which gives rise to a demand for an abstract syntax tree, which gives rise to a demand for tokens, which gives rise to a demand for something to evaluate!

2 Parsing

The *concrete syntax* of a language is a set of finite sequences, or *strings*, of *characters* drawn from some fixed alphabet, such as ASCII or Unicode. A *grammar* for a language determines which strings of characters are sufficiently well-formed as to be considered by the later phases of processing. The grammar is given in two parts, specifying first the *lexical* structure of the language, and then the *hierarchical* structure of its lexical form.

The lexical structure of a language consists of *tokens* that define the atomic components of a piece of syntax. These components consist of identifiers, consisting of a sequence of letters and other characters; numerals, in some notational system; and symbols, including "reserved words" and arithmetic and logical operators. The process of translating character strings into token strings is called *lexing*. The lexer consolidates individual characters into tokens, and rejects characters that are not part of the language. Its behavior is governed by a *regular grammar* that defines the tokens of the language in terms of the characters you type.

The hierarchical structure of a language is given by an *abstract syntax tree* whose nodes are *operators* and whose children are abstract syntax trees regarded as arguments to these operators.[1] The process of translating from a sequence of tokens into an abstract syntax tree is called *parsing*. The parser is governed by a *context-free grammar*, which specifies the valid phrases of the language using a recursive (self-referential) notation to allow for nesting.

It is of course possible to write lexers and parsers by hand, but in practice it is more convenient to use a *lexer generator* and a *parser generator* which create lexers and parsers from grammars. In this class all lexers and parsers are generated by the `cmlex` and `cmyacc` tools, which were designed and implemented by Prof. Karl Crary here at Carnegie Mellon.[2] In general the lexers and parsers generated by these tools will be given to you, so familiarity with their use is not essential. However, it is essential that you learn to read the specifications that are used to generate them, because these define what are the valid inputs to the interpreter.

3 Elaboration

One of the trickiest things to get right in a programming language is the deceptively simple, yet subtle and powerful, concept of *binding and scope*, and the associated notions of *α-equivalence* and *(capture-avoiding) substitution*. These concepts are discussed in detail in Harper (2016), but it is helpful to situate those ideas in the context of an interpreter pipeline.

Abstract syntax trees expose the hierarchical structure of programs. For example, the addition operator combines two expressions to form a third, and is naturally represented as an addition node with its two children being its arguments. Using trees eliminates the ambiguities of concrete syntax, such as the associativity of binary operators. These matters are all resolved by the parser, eliminating them from consideration in the rest of an implementation.

Ast's do not, however, account for the *scope* (range of significance) of an identifier. For example, in the `let` expression `let` x `be` e_1 `in` e_2, the variable x is *bound* by the `let` to the expression e_1 for use within e_2. The variable name has no significance in any context in which the `let` occurs; it is, rather, a private matter for the `let` expression itself. But an ast representation of the `let` would

[1] See Part I of Harper (2016) for a general account of abstract syntax; in ML code abstract syntax trees are given by `datatype` declarations.

[2] These tools, and their documentation, are available at `http://www.cs.cmu.edu/~crary/cmtool`.

```
signature VAR = sig
  type var
  val var : string → var
  val name : var → string
  val eqvar : var × var → bool
end

signature EXP = sig
  structure Var : VAR

  type exp

  datatype view =
    Var of var
  | Num of int
  | Plus of exp × exp
  | Times of exp × exp
  | Let of exp × Var.var × exp

  val hide : view → exp
  val expose : exp → view

  val aequiv : exp × exp → bool
  val freevars : exp → Var.var Set.set
  val subst : exp × Var.var → exp → exp
end
```

Figure 2: A Wizard Signature for Arithmetic Expressions

have the form let(e_1,x,e_2), with three children, the second of which is the bound variable name. But nothing about the ast representation determines the scope of x, or explains how variables may be renamed so as to avoid capture during substitution. For this one needs abstract binding trees, or abt's, which enrich ast's with exactly this missing information. Thus, the abt representation of the let given above would be let(e_1,x.e_2), with two children, the second of which is an abstractor that binds x for use within e_2. The concept of free and bound identifiers, the renaming of bound identifiers, and capture-avoiding substitution are all defined for abt's so as to respect the binding and scope of identifiers.[3]

In programming terms these concepts are codified as an abstract type defined by the *wizard signature*,[4] which provides operations for creating abt's from abstractors, for exposing the root structure of an abt using pattern-matching, for checking two abt's for α-equivalence, and for performing capture-avoiding substitution. A simplified formulation of the wizard signature for a language of arithmetic expressions is given in Figure 2.

The variable signature, VAR, defines an abstract type of variables that are created from strings, and thus have a name, and that may be compared for equality.[5] The EXP signature defines an

[3] As ever, see Harper (2016) for a full account of ast's and abt's.

[4] The origin of the name is obscure, but remains the local vernacular.

[5] In practice they may also be linearly ordered, to facilitate search.

abstract type exp of expressions in terms of a structure Var implementing variables. This signature also defines a data type view that, please note, is *not* recursive. The types view and exp are related by the operations hide and expose. The operation aequiv is the test for α-equivalence of two expressions, the operation freevars computes the set of free variables in an expression, and the operation subst defines capture-avoiding substitution of an expression for a free variable in another expression.

Assume that Exp is a structure implementing EXP.[6] An expression is created by a layered series of hides. For example, the exp corresponding to $1 + 1$ is created as follows:

```
val one : exp = Exp.hide (Exp.Num 1)
val p11 : exp = Exp.hide (Exp.Plus (one, one))
```

Continuing, a let expression, which binds a variable, is created as follows:

```
val x : exp = Exp.hide (Exp.Var.var "x")
val pxx : exp = Exp.hide (Exp.Plus (x, x))
val letx : exp = Exp.hide (Exp.Let (p11, (x, pxx)))
```

The purpose of hiding is to ensure that the test for α-equivalence may be implemented in constant time, and allows substitution to enforce capture-avoidance, at the expense of renaming of bound variables.

The operation expose reveals the top-level structure of a given exp as a view. This allows programming using one-level pattern matching, the overwhelmingly common case.[7] For example, continuing from above,

```
val (e as Exp.Let(e1,y,e2)) = Exp.expose letx
val true = Exp.eq p11 e1
val false = Exp.eq y x
val false = Exp.eq pxx e2
val letx' = Exp.hide e
val true = Exp.eq letx letx'
```

None of the matches will fail when executed, even though they are not exhaustive! First, it is safe to decompose letx as indicated because it is defined to have that form. The expression e1 is indeed p11 as defined earlier. But it is assured that the expression y, which is a hidden variable, is *not* the expression x, the variable used to create letx in the first place! This is the price of renaming bound variables implicitly: whenever a bound variable is expose'd, it is renamed systematically to ensure that it is different from *any* other free variable, which is to say that it is *fresh.* Thus, the comparison of e2 with pxx fails, because the latter has a free variable, and the freshly exposed y is guaranteed to be different from it. However, if e is once again hidden, the result *is* α-equivalent to letx! This, in an nutshell, is the essence of the wizard signature.

The second major role for elaboration is to enforce the *context-sensitive* constraints on the formation of programs specified by the statics of a language. The statics is invariably given *declaratively* as an inductive definition of typing judgments of the form $\Gamma \vdash e : \tau$, where τ is a type, e is an expression, and Γ is a context assigning types to the free variables that may occur within e. The

[6] The implementation of the wizard signature is an important homework problem for this class.

[7] One can augment with interface with deeper views if needed in a particular circumstance.

rules are said to be declarative because they simply state what is the case, that is, under what circumstances the typing judgment can be correctly asserted. They do not, in themselves, define how to *implement* them.

First off, what would it even mean to implement a statics? Let us assume for the moment that we are given the types `typ`, `exp`, and `ctx` representing the components of the judgment in some manner. An implementation of the statics would be a function

```
val wf : ctx × exp × typ → bool
```

that checks whether or not the judgment comprised of its arguments is derivable in the statics. For very simple languages, and when the only purpose is to check well-formedness, the function `wf` may be implemented by simply checking which, if any, rule applies to the given expression, and recursively checking the premises of that rule. When `exp` is a variable, it is necessary to compare the type assigned to it by the context to the type given to `wf` as its third argument. Doing so requires two things. First, a context must permit lookup of the type assigned to a free variable, if any:

```
val find : ctx × Var.var → typ option
```

This is an important point of contact with the wizard, which defines the type `Var.var` of free variables, and determines how to compare them for equality.[8] Correspondingly, the implementation of `wf` for `Let` requires that contexts support an extension operation:

```
val extend : ctx × Var.var × typ → ctx
```

This is another point of contact with the wizard: because free variables are always distinct from one another, extending the context can never result in a conflict in which the same variable is assigned two different types!

4 Execution

In Harper (2016) the dynamics of a language is defined by a transition system consisting of *states*, s, among which are the *initial* and *final* states, and a *transition relation*, $s \longmapsto s'$, between states defining the steps of execution. A transition system may be seen as an *abstract machine* whose states are those of the transition system, and whose transitions correspond to the instruction steps of the machine.

For any state s there may be zero, one, or many states s' such that $s \longmapsto s'$. By convention if a state is final, then it will admit no transitions to another state. A transition system is *deterministic* iff for every state s there is at most one state s' such that $s \longmapsto s'$; otherwise it is *non-deterministic*. A given initial state may or may not transition in any finite number of steps to a final state. For example, states representing divergent computations need never reach a final state. If a transition is deterministic, then each initial state may reach at most one final state, the outcome of that computation. It is important to understand that this property *may* or *may not* hold true for non-deterministic transition systems. The mere possiblity of there being many "next states" from a given state does not imply that that state leads to multiple outcomes. That is, it can be that there is at most one outcome, even though there are many transition sequences leading to it.

[8] There is where a linear ordering of variables would come into play, to allow for more efficient search.

```
signature TS = sig
  type state
  val initial : state → bool
  val final : state →  bool
  val step : state → state option
end
```

Figure 3: State Transition System

In ML a transition system is a structure with the signature `TS` given in Figure 3. Importantly, the signature `TS` does *not* define an abstraction; rather, describes a class of structures that provide the indicated components. What constitutes a state, and therefore how to create one and how to examine one, is left unspecified. For example, for the expression language defined above, a state is a *closed* expression (that is, one with no free variables). All states are initial, final states are numerals, and transition defines to a left-to-right evaluation strategy for closed expressions; it is deterministic.

Implementing a deterministic language consists of defining a function

```
val run : state → state option
```

that, given an initial state s, returns the final state s', if it exists, such that $s \longmapsto^* s'$. If $s \longmapsto^* s' \not\longmapsto$ with s' not final, then the execution fails.[9]

For non-deterministic languages, it is first of all necessary to ask, what does it even means to execute a state that can transition to many distinct states? Such programs may, in general, have multiple outcomes—there is no definite final state with a definite answer, rather there can be many. The solution to this problem is to maintain a set of *all possible* states that can be reached from a single initial state, and to consider the computation to be complete when *some* state in that set is final, it then representing an *an outcome* of the overall computation. Notice that more than one state can be final, and that some states may never reach a final state.

The languages that are considered in this course (and more generally) are *finitely non-deterministic* in that each state can transition to only finitely many next states.[10] A state transition system with finite non-determinism may be implemented by considering a *derived* transition system whose states are *finite sets of states* of the given transition system. In this context the component states are called *threads*, or *tasks*, and the finite sets of threads are called *thread pools*. Each step of the derived transition system consists of choosing a state on which to make progress, updating the thread pool accordingly, bearing in mind that the chosen state may transition to finitely many states that are then added to the thread pool.

Thus, to be more specific, a thread pool, S, is a finite set $\{ s_0, \ldots, s_{n-1} \}$, where $n \geq 0$, of states of the given transition system. A thread pool is initial when it consists of a single initial state of the underlying transition system, and is final when (at least) one of its constituent states is final.

[9] For simplicity the result is an option, but in practice the no-result case would provide an indication of the reason for the failure so as to permit meaningful error messages.

[10] There are perfectly sensible languages that would violate this requirement: consider, for example, a primitive operation that transitions to an arbitrary natural number.

Transition between thread pools is defined as follows. Suppose that $s \in S$ and s is not a final state. Then by definition

$$S \longmapsto S \setminus \{\, s \,\} \cup \{\, s' \mid s \longmapsto s' \,\}.$$

That is, the state s is replaced by all states s' to which s may transition. Notice that the thread pool S may transition to finitely many different thread pools according to how s is chosen. A *scheduler* is a function that, for a non-empty thread pool S, chooses, if possible, a non-final state $s \in S$. The purpose of a scheduler is to render the derived transition system deterministic. Although a scheduler is, by this definition, required to choose a non-final state if one is available, this does *not* imply that every non-final state will be scheduled to execute! (As an exercise, define a scheduler that does not have this property. One that does is said to be *fair*; fair schedulers are harder to come by than one might at first expect.)

References

Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, Cambridge, England, Second edition, 2016.