A Framework for Teaching Programming Languages

Kent D. Lee
Luther College
700 College Drive
Decorah, IA 52101 USA
kentdlee@luther.edu

ABSTRACT

This paper provides a description of a framework for programming language implementation that is accessible to students in a one semester course focused on programming languages. Rather than concentrating solely on language features, this paper describes a framework where imperative, functional and logic programming languages are all used to develop a compiler for a non-trivial subset of a functional programming language. Provided by the framework are a virtual machine, a disassembler of Python programs, a partial implementation of a Standard ML compiler with suggested tests, and a partial implementation of a type inference system implemented using Prolog, again with suggested tests.

Classroom experience gained while using this framework is also shared. A suggested fourteen week sequence is provided. Real reactions from students and reflections on coverage of the ACM 2013 Curricula guildelines conclude the paper.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Languages,Theory

Keywords

Programming Languages, Compilers, Virtual Machines, Type Inference, Type Checking

1. INTRODUCTION

Doing is better than seeing. This has been said many times before by many Computer Science professors and professionals. The premise holds for learning about programming languages as well. Using a programming language to implement something of substance, more than just a few lines of code, helps students form a better understanding of a language and how it can best be used. Some texts on programming languages are primarily surveys of programming languages where important aspects of languages are described, mostly with short examples. For instance, type systems are very important, but students have a hard time getting invested in learning about type checking and type inference without a real goal in mind. Recently, several texts on programming languages have abandoned this survey approach to teaching programming languages in favor of using programming languages to implement programming languages[?, ?, ?, ?], giving students more purpose in their language exploration.

SIGCSE'15, March 4–7, 2015, Kansas City, MO, USA.

ACM 978-1-4503-2966-8/15/03 ...$15.00.
http://dx.doi.org/10.1145/2676723.2677245

Doing is more complicated than seeing. Of course, the tradeoff is added complexity versus time. There is only so much time in a semester and language implementation is a generally complex problem. It is critical that this approach be accessible while still having substance. This paper presents a successfully taught framework for the implementation of programming languages. In particular, a compiler is developed from the bottom-up for a substantial subset of Standard ML including portable code generation for a virtual machine, support for higher-order functions and closures, and type infererence.

The paper begins by introducing CoCo, a virtual machine based on the Python virtual machine. The Python virtual machine is an excellent choice since the language supports higher-order functions and closures. The framework includes a disassembler for Python programs. This makes it possible for students to write Python programs and discover the necessary virtual machine instructions. This disassembly and discovery makes learning the machine language and code generation relatively fast so other facets of programming languages and their implementation may be explored.

The paper goes on to demonstrate compiling a non-trivial Standard ML program and all that entails. Language implementation issues are explored including the CoCo virtual machine, currying, mutually recursive functions, and type inference. The paper concludes with a proposed fourteen week course outline based on past teaching experience. Finally, reflections on that experience from both teachers and students is provided. And perhaps best of all, the resources for this framework are freely available on-line.

2. THE FRAMEWORK

The language implementation framework can be divided into three pieces. From the bottom up they are a virtual ma-

chine and Python disassembler, a partial implementation of a Standard ML compiler written in Standard ML, and a partial implementation of a type inference system for Standard ML written in Prolog. The virtual machine, called CoCo, is written in C++ and is a faithful implementation of a subset of the Python virtual machine. The Python virtual machine was chosen because Python has some functional features and supports nested functions, higher-order functions, and static scope.

The next sections cover each of these three parts of the framework while compiling a simple, but non-trivial Standard ML program. Consider the Standard ML program in figure **??**. *println* is a polymorphic function that prints its value to the screen and returns *unit*, or the unit equivalent of *None* from the Python virtual machine.

```
let fun f 0 y = y
      | f x y = g x (x*y)
    and g x y = f (x-1) y
in
  println (f 10 1)
end
```

Figure 1: A Standard ML Program

The program features two function definitions using pattern matching. The two functions are mutually recursive, essentially counting down to zero while multiplying the values together, in the end computing and printing 10!. The next sections use this program as an example, taking it through the same process, much abbreviated for this paper, that students work through using this framework in a semester long course.

```
import disassembler

def f(x,y):
    if x==0:
        return y
    return g(x,x*y)
def g(x,y):
     return f(x-1,y)
def main():
    print(f(10,1))

disassembler.disassemble(f)
disassembler.disassemble(g)
disassembler.disassemble(main)
```

Figure 2: A Python Program

2.1 The Disassembler and VM

The CoCo VM (Virtual Machine) is based on Python 3.2. The framework includes a disassembler that outputs CoCo VM programs. The CoCo disassembler uses the Python disassembler, which is supplied with the Python distribution. Python does not guarantee backwards compatibility of its virtual machine, so Python 3.2 is required to run the disassembler. Consider the Standard ML program in figure **??**. The Python program in figure **??** is in many ways similar to the Standard ML program. Calling the *main* function of this program has the same result. In this case the disassembler module was imported and the three functions were disassembled. The output from the disassembler is CoCo VM assembly language as shown in figure **??**. The instructions are taken from the Python 3.2 VM specification with just a few minor differences to support the CoCo VM.

```
Function: f/2
Constants: None, 0
Locals: x, y
Globals: g
BEGIN
            LOAD_FAST                0
            LOAD_CONST               1
            COMPARE_OP               2
            POP_JUMP_IF_FALSE  label00
            LOAD_FAST                1
            RETURN_VALUE
label00:    LOAD_GLOBAL              0
            LOAD_FAST                0
            LOAD_FAST                0
            LOAD_FAST                1
            BINARY_MULTIPLY
            CALL_FUNCTION            2
            RETURN_VALUE
END
Function: g/2
Constants: None, 1
Locals: x, y
Globals: f
BEGIN
            LOAD_GLOBAL              0
            LOAD_FAST                0
            LOAD_CONST               1
            BINARY_SUBTRACT
            LOAD_FAST                1
            CALL_FUNCTION            2
            RETURN_VALUE
END
Function: main/0
Constants: None, 10, 1
Globals: print, f
BEGIN
            LOAD_GLOBAL              0
            LOAD_GLOBAL              1
            LOAD_CONST               1
            LOAD_CONST               2
            CALL_FUNCTION            2
            CALL_FUNCTION            1
            POP_TOP
            LOAD_CONST               0
            RETURN_VALUE
END
```

Figure 3: A CoCo VM Program

Each function is defined separately in the CoCo language. All constant values used in a function are defined in the *Constants* list. The *Locals* includes all locally initialized variables including function arguments. *Globals* include identifiers defined outside the current function definition, including built-in functions. Labels are used in the common assembly language format as the targets of jumps and branches. The */2* that appears in the *Function: f/2* definition indicates that *f* is a function of two arguments.

CoCo is a stack-based VM, like the Python and Java virtual machines. Operands are pushed onto the stack before the instruction that operates on them. Calling a function is accomplished by pushing the function and the arguments onto the stack and then executing the *CALL_FUNCTION* instruction as shown on line 37-42 of figure **??**. The 1 or 2 that appears as the operand for the call function instructions indicates the number of arguments. The arguments appear on top of the function to be called on the operand stack of the VM.

Much can be gleaned from exercises like this. The correlation between VM instructions and Python code helps to inform the student of each instruction's meaning. That, together with available documentation means a shorter learning curve to writing CoCo assembly directly from similar examples.

The function in figure **??** is a curried function. The Python program in figure **??** is not written in curried form. Python does not support writing functions in curried form. But, currying can be thought of as syntactic sugar. Any curried function can be rewritten as a series of one argument functions. The Python program in figure **??** provides one possible uncurry transformation applied to the program in figure **??**. The transformed program now contains two higher-order functions as shown. The function *f* takes one argument and returns another function, *f3* that also takes one argument. In this sense, the function f is now curried, taking its arguments one at a time. The same transformation applies to *g*.

```
1  def f(v0):
2    def f3(v1):
3      def f2(x,y):
4        if x==0:
5          return y
6        return g(x)(x*y)
7      return f2(v0,v1)
8    return f3
9  def g(v4):
10   def g7(v5):
11     def g6(x,y):
12       return f(x-1)(y)
13     return g6(v4,v5)
14   return g7
15 def main():
16     print(f(10)(1))
17 if __name__=="__main__":
18   main()
```

Figure 4: A Curried Python Program

Again, the disassembler can be used to see how the two higher-order functions in figure **??** are implemented in the VM. Disassembling the function *f* automatically disassembles any nested functions, in this case *f3* and *f2*. The disassembled code for function *f* appears in figure **??**.

The CoCo VM supports nested functions. In figure **??** the second line begins the nested function *f3* inside the *f* function. The variable *v0* is used in the function *f3* and is the argument to the function *f*. Since *v0* is referenced from the enclosing scope of *f3* it is a free variable in *f3* as shown on line 25 of the CoCo VM code. The usefulness of the disassembler is evident in understanding how to properly generate code for non-trivial programs like the code in figure **??**.

```
1  Function: f/1
2    Function: f3/1
3      Function: f2/2
4      Constants: None, 0
5      Locals: x, y
6      Globals: g
7      BEGIN
8            LOAD_FAST                0
9            LOAD_CONST               1
10           COMPARE_OP               2
11           POP_JUMP_IF_FALSE label00
12           LOAD_FAST                1
13           RETURN_VALUE
14 label00: LOAD_GLOBAL               0
15           LOAD_FAST                0
16           CALL_FUNCTION            1
17           LOAD_FAST                0
18           LOAD_FAST                1
19           BINARY_MULTIPLY
20           CALL_FUNCTION            1
21           RETURN_VALUE
22     END
23   Constants: None, code(f2)
24   Locals: v1, f2
25   FreeVars: v0
26   BEGIN
27         LOAD_CONST                 1
28         MAKE_FUNCTION              0
29         STORE_FAST                 1
30         LOAD_FAST                  1
31         LOAD_DEREF                 0
32         LOAD_FAST                  0
33         CALL_FUNCTION              2
34         RETURN_VALUE
35   END
36 Constants: None, code(f3)
37 Locals: v0, f3
38 CellVars: v0
39 BEGIN
40       LOAD_CLOSURE                 0
41       BUILD_TUPLE                  1
42       LOAD_CONST                   1
43       MAKE_CLOSURE                 0
44       STORE_FAST                   1
45       LOAD_FAST                    1
46       RETURN_VALUE
47 END
```

Figure 5: A Higher-order Function Definition

Encountering the *MAKE_CLOSURE* instruction on line 43, students are naturally curious as to what the instruction does. As mentioned earlier in the paper, the CoCo VM is implemented in C++, making it possible to investigate the meaning of an instruction by looking at its implementation. The implementation of the make closure instruction is provided in figure **??**. Consulting both the code in figure **??** and the example CoCo program in figure **??** helps in deter-

mining that a closure is made up of code and environment. Lines 40-41 of figure **??** build a tuple containing the variable *v0* from the enclosing environment. Line 42 loads the code for *f3* onto the operand stack. Line 43 builds the closure of the code and environment. Line 44 stores that closure in the local variable *f3* so it can be returned by *f*.

```
1  case MAKE_CLOSURE:
2    u = safetyPop();
3    v = safetyPop();
4    w = new PyFunction(*((PyCode*) u),
5          globals, v);
6    opStack->push(w);
7    break;
```

Figure 6: The MAKE_CLOSURE Implementation

From the C++ code in figure **??** and the example of its use in figure **??** the closure of a function is found to be the environment (i.e. the variable from the enclosing scope) along with the code. A closure is environment and function together. The closure provides the static scoping of the language. Rather than just tell students about closures, how nice it is to show them its significance in a program by examining the CoCo VM and a sample program that needs the closure.

Writing and disassembling Python programs, consulting the disassembled code, and examining the C++ implementation of CoCo all work together toward providing a means for the understanding of code generation for a non-trivial language. A next step towards further understanding is to implement a compiler for a completely different language, like Standard ML.

2.2 A Standard ML Compiler

The investigative techniques discussed in the previous section are put to use to develop a compiler for a subset of SML (Standard ML) that includes datatypes like int, real, bool, lists, and tuples. Pattern-matching is supported. Function definition and invocation is implemented including higher-order and mutually recursive functions like the example that appears in figure **??**.

As we saw earlier, curried functions can be transformed into higher-order functions of one argument. A transformation like that employed in figure **??** is provided to students and is used during the parsing of curried functions. The consequence of this syntactic transformation appears in figure **??** where the abstract syntax of SML programs only includes functions of one argument. Any curried function will have already been transformed into a series of one argument functions before it is represented in SML abstract syntax. Any uncurried function of more than one argument is actually a function of one argument, a tuple.

Compiling the program from figure **??** builds the AST shown in figure **??**. The currying transformation generated the extra functions named **anon@***x*. The Python program in figure **??** is a very literal translation of the AST in figure **??** into Python code. In figure **??** the *anon@3* is called *f3* and *anon@2* is called *f2*. In the function *g* the *anon@7* is named *g7* in figure **??** and *anon@6* is named *g6*.

```
1  structure MLAS = struct
2  datatype
3  exp = int of string
4      | ch of string
5      | str of string
6      | boolval of string
7      | id of string
8      | listcon of exp list
9      | tuplecon of exp list
10     | apply of exp * exp
11     | infixexp of string * exp * exp
12     | expsequence of exp list
13     | letdec of dec * (exp list)
14     | raisexp of exp
15     | handlexp of exp * match list
16     | ifthen of exp * exp * exp
17     | whiledo of exp * exp
18     | func of int * match list
19 and
20 match = match of pat * exp
21 and
22 pat = intpat of string
23     | chpat of string
24     | strpat of string
25     | boolpat of string
26     | idpat of string
27     | wildcardpat
28     | infixpat of string * pat * pat
29     | tuplepat of pat list
30     | listpat of pat list
31     | aspat of string * pat
32 and
33 dec = bindval of pat * exp
34     | bindvalrec of pat * exp
35     | funmatch of string * match list
36     | funmatches of
37         (string * match list) list
38 end;
```

Figure 7: SML Abstract Syntax

The SML compiler's responsibility is to translate an AST like the one that appears in figure **??** into a CoCo VM program, like the partial program that appears in figure **??**. The target language has some requirements. First, function definitions must appear before they are called in the CoCo language. However, anonymous functions denoted by a *func* descriptor in the SML abstract syntax, may appear anywhere an expression can appear. In addition, every function definition must contain lists of the constants, locals, globals, free variables, and cell variables used by the function. These requirements dictate how code is generated from SML ASTs like the one that appears in figure **??**. Several passes over the AST are required to generate the target program with the correct ordering of parts.

Consider generating the list of constants used by a function. Building this list entails traversing the patterns used by a function definition. Consider figure **??**. In the patterns used in the definition of *f* a 0 appears. Constants may also appear in the body of a function. The body is called *exp* in the code below. Constants in the body of a function include 1 and 10 in figure **??**. The code below traverses the definition of patterns and expressions in a function definition looking for all constants that were used in its definition. The constants *None*, *'Match Not Found'* and 0 are added for all function definitions. *removeDups* removes duplicate values from the list.

```
val consts = (removeDups ("None"::
    "'Match Not Found'"::"0"::
     (List.foldr (fn (match(pat,exp),y) =>
        (patConsts pat)@(constants exp)@y)
        [] expList)))
```

Building the list of constants is one of the simpler tasks. Teaching compiler generation for such a language would be a daunting task to undertake in a semester course without the proper approach. Extending the compiler to compile a set of test programs provides the proper organization to make learning compiler generation in a semester manageable.

For example, attempting to compile the program in figure **??** results in an error. The compiler code initially given to students will not compile this code. Attempting to run the compiler produces the following output.

```
Attempt to get constants for expression not
currently supported!
Expression was: ifthen
An error occurred while compiling!
```

The *if-then-else* expression was not supported by the *constants* function. Students look for this error message within the compiler, look at other surrounding code, and fix the problem once they understand how the problem occurred. Then, they try to compile again, potentially finding the next problem. Eventually, they get to the point where code must be generated for the *if-then-else* expression. A little bit of guidance may be needed here, but that can be provided by the Python disassembler!

Code generation is the last step to getting executable code for correct SML programs. Once the *if-then-else* code generation is added, code is generated for programs like the one in figure **??** which can be run on the CoCo VM. But, one important aspect of the compiler is still missing. Type inference is one of the unique features of the Standard ML

```
1  letdec(
2   funmatches([
3   funmatch('f',[
4    match(idpat('v0'),func('anon@3',[
5        match(idpat('v1'),
6            apply(func('anon@2',[
7            match(tuplepat([intpat(0),
8                idpat('y')]),id('y')),
9            match(tuplepat([idpat('x'),
10               idpat('y')]),
11                   apply(apply(id('g'),
12               id('x')),apply(id('*'),
13           tuple([id('x'),id('y')])))))
14       ]),tuple([id('v0'),id('v1')])))
15   ]))])
16  ,funmatch('g',[
17    match(idpat('v4'),func('anon@7',[
18        match(idpat('v5'),
19            apply(func('anon@6',[
20            match(tuplepat([idpat('x'),
21            idpat('y')]),apply(apply(id('f'),
22            apply(id('-'),tuple([id('x'),
23            int('1')])))),id('y')))
24        ]),tuple([id('v4'),id('v5')])))
25    ]))])
26  ])
27  ,
28  [
29  apply(id('println'),
30    apply(apply(id('f'),int('10')),int('1')))
31  ])
```

Figure 8: SML Program AST

```
1  let val x = Int.fromString(
2          input("Please␣enter␣an␣integer:␣"))
3      val y = Int.fromString(
4          input("Please␣enter␣an␣integer:␣"))
5  in
6    print "The␣maximum␣is␣";
7    println (if x > y then x else y)
8  end
```

Figure 9: SML Compiler test4.sml

language. Teaching type inference is covered in the next section.

2.3 Standard ML Type Inference

Type inference is the perfect kind of program to implement in Prolog and the framework presented in this paper includes a partially implemented type inference system. Like the code generation part of the compiler, the type inference system needs to be extended to correctly type check some of the test programs.

Implementing type inference is much easier when type inference rules are supplied. The framework provides a complete set of type inference rules for the supported subset of Standard ML. Consider the program in figure **??** which contains two functions that are mutually recursive. In a language other than Prolog handling mutual recursion would be difficult since it requires unification to properly infer the type of the mutually recursive functions. The type inference rule for mutually recursive functions appears in figure **??**.

$$\forall i\ 1 \leq i \leq n, \forall j\ 1 < j \leq n, \quad n \geq 1,$$
$$\frac{[id_1 \mapsto \alpha_1 \to \beta_1 \quad \{,\ id_j \mapsto \alpha_j \to \beta_j\}] \oplus \varepsilon \vdash id_i\ matches_i : \alpha_i \to \beta_i}{\varepsilon \vdash fun\ id_1\ matches_1\ \{and\ id_j\ matches_j\} \Rightarrow [id_1 \mapsto close(\alpha_1 \to \beta_1)\ \{,\ id_j \mapsto close(\alpha_j \to \beta_j)\}]}$$

Figure 10: The FunDecs Type Inference Rule

The type inference rule reads as follows: A set of functions separated by *and* keywords are correctly typed if each function has a function type that is consistent across all the matches. A match is a pattern and expression used in a function definition. For a function type to be consistent in a mutually recursive definition requires both its definition and application be consistently typed. The use of *close* freezes the definition of these function types so the types may be used in subsequent applications of these functions. Closing types is necessary when types may be polymorphic which Standard ML supports.

```
1 gatherFuns([],[]).
2 gatherFuns([funmatch(FName,_)|Tail],
3   [(FName,fn(_,_))|FEnv]) :-
4   gatherFuns(Tail,FEnv).
5 typecheckDec(Env,funmatches(L),NewEnv):-
6   gatherFuns(L,NewEnv),
7   append(NewEnv,Env,FunEnv),
8   typecheckFuns(FunEnv,L),
9   closeFunTypes(NewEnv).
```

Figure 11: FunDecs in Prolog

The type inference rule in figure **??** is implemented by the Prolog predicates appearing in figure **??**. The *gatherFuns* predicate creates function types for each mutually recursive function. These types, through unification, instantiate to the (possibly polymorphic) types of the functions. The *closeFunTypes* predicate closes the function types for polymorphic types. The program in figure **??**, using this type inference rule among others, passes the typechecker and prints the following output.

```
val g = fn : int -> int -> int
val f = fn : int -> int -> int
val it : unit
The program passed the typechecker.
```

Students build on this type inference system in the same manner as the compiler. Test programs that currently are not supported are introduced and students work to enhance the type inference system to support these test programs.

3. CONCLUSIONS

This paper has presented a very abbreviated look at a framework for teaching programming languages. The framework builds from the bottom-up covering assembly language on a stack-based virtual machine, virtual machine implementation in C++, compiler implementation for a subset of Standard ML written in Standard ML, and type inference written in Prolog. Students are guided in their investigation of programming languages by extending the framework to compile and infer the types of test programs.

The CoCo VM, Standard ML compiler, and type inference system are all freely available at http://github.com/kentdlee. Two versions exist. A publicly available version is available to students to download. A private, but still free, version containing the full source code for CoCo, the Standard ML Compiler, and for inferring types is available for instructors upon request. A textbook is also available[**?**] that covers the material presented in this paper in much more detail.

The framework has been used in practice for two years with excellent results. A fourteen week semester is divided up as follows. About one week is devoted to learning about context free grammars, derivations, and abstract syntax trees. One week is spent learning the CoCo VM and writing a few CoCo programs. C++ and the implementation of the CoCo VM is studied for approximately two weeks. Two weeks is spent learning functional programming and in particular Standard ML. This is separate from the compiler project. Two weeks is spent compiling the subset of Standard ML presented in this framework. Two weeks is spent learning how to program in the Prolog programming language. Finally, two weeks is spent studying and implementing type inference. This leaves a couple of weeks to include administering exams and some flexibility in the topics being covered.

The framework's coverage of the ACM Computer Science Curricula 2013 guidelines is thorough, covering all but one area of the tier 1 and 2 guidelines and several elective guidelines as well. Functional and OOP programming paradigms are covered. These two programming paradigms, type systems, program representation, language translation and execution are all tier 1 and 2 (i.e. mandatory) topics of the guidelines. Elective guidelines covered by this framework include syntax analysis, compiler semantics, code generation, runtime systems, static analysis, advanced programming constructs, and logic programming. Of the tier 1 and 2 requirements, absent is only event-driven and reactive programming which is often covered in other courses.

Students feel a sense of real accomplishment in getting their test cases to compile, typecheck, and run correctly. With a clearly defined purpose, programming languages can be an exciting and interesting course. Class evaluations were

anonymous and students were allowed to comment on the strengths of the course. Here are a couple anonymous comments about the course and using this framework.

> *This course is designed to make you learn about programming languages and their implementation. I think the goal was achieved very well.*

> *Programming Languages has been a fantastic course. I feel as though I have gained quite an understanding of how many languages work and behave. It has been really fun and interesting figuring this out in a classroom setting. Definitely the most fun computer science class I have taken so far.*

While the language implementation is non-trivial there is much that can still be done with it including garbage collection in the VM, extending the VM for threaded computation, support for object-oriented languages within the VM, implementation of a richer subset of Standard ML, or compiler development for a completely different language. These projects can serve well as undergraduate research topics. In practice, after students have gotten familiar with the framework and what can be achieved using it, several have expressed interest in further extending the framework through independent study.