Type Inference for Language Prototyping

Kent D. Lee
Luther College
kentdlee@luther.edu

Abstract

This paper provides an overview of a framework for prototyping programming languages while providing an in-depth description of type inference for a subset of Standard ML, implemented in Prolog. As part of this research, type inference rules for Standard ML were developed along with an implementation utilizing the full unification power of Prolog. The paper details the techniques used to implement type inference in Prolog and describes how the larger framework may be applied to other language prototyping.

1. Introduction

Prototyping a language often leads to discovery. The prototyping framework presented in this paper consists of a virtual machine, a compiler from a sample source language to the virtual machine, and a type inference system. The type inference system is covered in detail in this paper. Another paper (Lee 2015) provides a more detailed overview of the entire framework.

Prototyping a language is easier when compiling to a virtual machine. Python is a programming language that supports imperative, object-oriented, and functional programming. Internally, Python is compiled to a virtual machine before it is interpreted. The Python virtual machine is seen as an implementation detail of Python and is not guaranteed to remain stable like externalized virtual machines. The Python virtual machine is not an externalized interface to which other languages could be targeted.

While the Java Virtual Machine is an externalized virtual machine, it is not suitable as a target language in many cases. The Java Virtual Machine does not directly support nested functions and static scope. Nested inner classes are supported, but not in the same way that nested functions are supported in many functional languages. Supporting nested functions and static scope within the JVM would require support from a set of classes. In other words, a supporting run-time system would be required. An externalized Python virtual machine would be better suited as a target for languages with nested functions and scope since no run-time system is needed, thus simplifying the generated code.

```
1  let fun f(x,y) =
2    let fun g(x,y) =
3      if x < 1 then y else f(y-1,x)
4    in
5      g(x,y)
6    end
7  in
8    println(f(5,12))
9  end
```

Figure 1. A Standard ML Program

```
1  import disassembler
2  def f(x,y):
3    def g(x,y):
4      if x < 1:
5        return y
6      else:
7        return f(y-1,x)
8    return g(x,y)
9  def main():
10   print(f(5,12))
11 main()
12 #disassembler.disassemble(f)
13 #disassembler.disassemble(main)
```

Figure 2. A Python Program

CoCo is an externalized Python Virtual Machine. CoCo was built by reverse engineering the Python virtual machine. It is written in C++, using Object-Oriented design principles like inheritance, information hiding, and polymorphism. It is easily extensible for language prototyping. And, it has a stable interface that will remain backwards compatible. In addition, CoCo includes a disassembler of Python 3.2 programs that can be used to discover correct code for implementing various language features. For instance, consider the ML program in figure 1 (liberty was taken with the use of println

- so not quite ML). A similar program can be written in Python, as shown in figure 2, to discover how to implement the ML program using the CoCo virtual machine. By commenting out the call to the *main* function and uncommenting the calls to a CoCo disassembler module, the Python program can be disassembled to discover its CoCo implementation. The CoCo version of the programs presented in figures 1 and 2 is presented in appendix A.

```
1  let val x = 6
2  in
3    println(x+1)
4  end
```

Figure 3. A Static Binding Example

```
1  let val x = ref 6
2  in
3    println(!x+1)
4  end
```

Figure 4. A Reference Example

Of course, not every program is going to have a direct counter-part in Python. But the disassembler can be used to discover many code generation *secrets* of Python to quickly familiarize the researcher with the virtual machine. Consider the ML program presented in figure 3. Implementing this in CoCo requires the use of a variable to hold the value of x. There is no concept of static bindings in CoCo (like Python). This means the generated CoCo code for figures 3 and 4 are identical leaving room for a third version of this program in figure 5 which is not a valid ML program and yet compiles and runs successfully on the CoCo virtual machine. There are at least a couple of type errors in the program of figure 5 depending on how you go about fixing it. The variable x might be declared as a reference on line 1 which means that the dereferencing of x on lines 3 and 4 should be rewritten as $!x$.

```
1  let val x = 6
2  in
3    x := x + 1;
4    println(x+1)
5  end
```

Figure 5. An Invalid ML Program

Type inference or type checking is clearly an important part of any programming language, whether the type checking is done statically before run-time, or dynamically at run-time.

2. Mapping from Source to AST to Type

Implementing type inference within the prototyping framework presented here first entails mapping a program into its abstract syntax. Then the type inference system maps from the abstract syntax tree (i.e. AST) into its type as depicted in figure 6. Abstract syntax and types must both be defined. The abstract syntax of the *Small* language is given in appendix B.

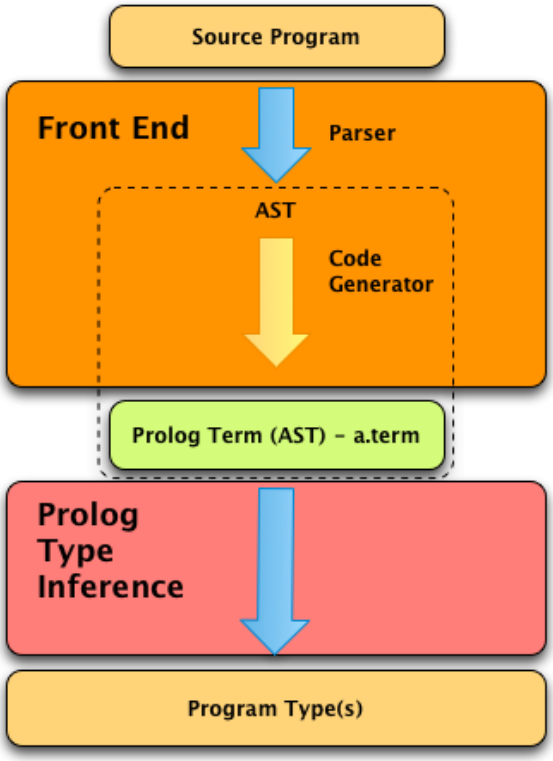

Figure 6. Overview of the Framework

Strings, integers, characters, and booleans are supported as primitive data. Small, like Standard ML, has a core language which the abstract syntax represents. Syntactic sugar is used to support language features like Curried functions, local bindings (i.e. let expressions) of more than one identifier, and anonymous functions.

```
1  datatype SmallType =
2         bool
3       | int
4       | str
5       | exn
6       | tuple of type list
7       | listOf of type
8       | fn of type * type
9       | ref of type
10      | typevar of string
11      | typeerror
```

Figure 7. Small Types

The back end of the compiler writes to a file called *a.term* as depicted in figure 6, which is the Prolog term read by the type inference system part of the framework. The allowed

syntax for the Prolog representation of the AST is given in appendix B.

The type inference system maps the AST of a program into its type. The allowable types are given in figure 7. As with many type systems, Small's type system is strict in *type errors* meaning that once a type error has been detected, the overall type of the expression is *typeerror*. Otherwise, we have the basic types, including exceptions, tuples, lists, functions, and references. The *typevar* is a Standard ML type inference variable used in representing polymorphic types as in Standard ML. Prolog does not require an explicit declaration of the allowable types. They are presented in figure 7 for reference only in the Standard ML datatype format.

3. The Type Environment

The type environment is represented as ε in the type inference rules of appendix C. The environment includes the types for built-in functions of the *Small* language. The infix operators of the language are also represented as functions in the abstract syntax and in the environment to increase uniformity and reduce the amount of code in the type checker. A complete description of the type environment can be found in (Lee 2014). Part of the environment is given in figure 8.

Type inference rules map from the AST definition of a program to its type. This mapping, from AST to its type dictates that several Prolog predicates be written, including *typeCheckExp*, *typeCheckMatch*, *typeCheckPat*, and *typecheckDec*, all corresponding to the various abstract syntax node types. There are also a number of supporting predicates that are written for these main predicates.

Type checking a program in the prototyping framework starts by setting up the environment and calling the *typeCheckExp* Prolog predicate to instantiate a type given a program (i.e. a *Small* expression). The implementation of *typeCheckExp* is dictated by the type inference rules presented in the next sections and appendix C.

4. Type Inference

Type inference rules are presented in the standard format of antecedents over conclusions. A type environment is represented as ε as described in the last section. The complete set of type inference rules for the *Small* language are presented in appendix C and can also be found in (Lee 2014).

Type checking a simple constant value does not require the type environment, but it is included for consistency. For instance, consider the *BoolCon* rule. The environment is not needed because the constant is already labeled with its type. The Prolog code that corresponds to this inference rule is given in figure 9.

BoolCon

$$\frac{}{\varepsilon \vdash bool(v) : bool}$$

```
typecheckProgram(Expression,Type) :-
  typecheckExp(
    [('Exception',fn(typevar(a),exn)),
     ('raise',fn(exn,typevar(a))),
     ('andalso',
      fn(tuple([bool,bool]),bool)),
     ('orelse',
      fn(tuple([bool,bool]),bool)),
     ('=',
      fn(tuple([typevar(a),typevar(a)]),
       bool)),
     ('<',
      fn(tuple([typevar(a),typevar(a)]),
       bool)),
     (@,
      fn(tuple([listOf(typevar(a)),
                listOf(typevar(a))]),
      listOf(typevar(a)))),
     ('input',fn(str,str)),
     ('explode',fn(str,listOf(str))),
     ('implode',fn(listOf(str),str)),
     ('println',
      fn(typevar(a),tuple([])),
     ('type',fn(typevar(a), str)),
     (+,fn(tuple([int,int]),int)),
     (-,fn(tuple([int,int]),int)),
     (*,fn(tuple([int,int]),int)),
     ...
     ('div',fn(tuple([int,int]),int))],
     Expression,Type).
```

Figure 8. The (partial) Type Environment

```
typecheckExp(_,bool(_),bool):-!.
```

Figure 9. BoolCon Prolog Definition

The cut operator (i.e. !) used in figure 9 is useful in this code. Without it, type checking a program that contained a boolean followed by another expression that could not be properly type checked due to some error in the coding of an inference rule or some unimplemented part of the type checking might undo the match of the boolean value in its attempt to backtrack and find another way to correctly type check the given expression.

For example, consider type checking this expression.

```
typecheckExp(Env,apply(id('='),
  tuple([bool('true'),E2])))
```

Assume that type checking expression *E2* contains an error. Prolog would fail to find a way to satisfy the predicate for *typeCheckExp(Env, E2, T)* and then would backtrack to the *BoolCon* rule implementation, undo that unification, and

then look for a further unifiable term, finding it at the end of the *typeCheckExp* code found in figure 10. Unifying with the code in figure 10 it would print that *bool* is an unknown expression.

```
typecheckExp(_,Exp,_) :-
  print('Error: Unknown expression '),
  print(Exp),
  throw(error('unknown expression')), !.
```

Figure 10. typeCheckExp Catch-All

This sort of problem does not routinely occur, but debugging these problems is easier if the cut operator is applied to limit any unwanted backtracking.

Verifying that all elements of a list constant are homogenous is one instance where Prolog's unification algorithm works exactly as desired. The *ListCon* rule requires all elements of a list constant have the same type.

ListCon

$$\frac{\forall i \;\; 1 \leq i \leq n, n \geq 0 \qquad \varepsilon \vdash e_i : \alpha}{\varepsilon \vdash [e_1, e_2, ..., e_n] : \alpha \;\; list}$$

Type checking a list constant requires a supporting predicate to call *typeCheckExp* on each of the elements of the list. Of course each element of the list must have the same type. The subtle code in this example is the first case of the *typecheckList* predicate. If the list is empty (i.e. `[]`), then the type of the list is $\alpha \;\; list$ where α is a Prolog variable that results from the use of the underscore in this definition.

```
typecheckList(_,[],_) :- !.
typecheckList(Env,[H|T],A) :-
  typecheckExp(Env,H,A),
  typecheckList(Env,T,A), !.
typecheckExp(Env,listcon(L),listOf(T)):-
  typecheckList(Env,L,T), !.
```

Figure 11. List Type Checking

There are two extensions to typical type inference rules that are used in the definition of the *Small* type inference rules. Inferring the type of a declaration produces a new type environment. This is indicated in rules like the *FunDecs* rule where the environment described to the right of the $\Rightarrow$ is yielded when successfully applying the rule to a program. These yielded environments are further applied to rules through the use of the $\oplus$ operator as used in the *Matches* or *FunDecs* rules, for instance.

ConsPat

$$\frac{pat_1 : \alpha \Rightarrow \varepsilon_{pat_1}, \;\; pat_2 : \alpha \; list \Rightarrow \varepsilon_{pat_2}}{pat_1 :: pat_2 : \alpha \; list \Rightarrow \varepsilon_{pat_1} + \varepsilon_{pat_2}}$$

TuplePat

$$\frac{\forall i \; 1 \leq i \leq n, n \geq 0 \qquad pat_i : \alpha_i \Rightarrow \varepsilon_{pat_i}}{(pat_1, pat_2, ..., pat_n) : \times_{i=1}^{n} \alpha_i \Rightarrow \sum_{i=1}^{n} \varepsilon_{pat_i}}$$

While the $\oplus$ operator indicates a possibly overlapping concatenation of environments where the left environment takes precedence over similarly named types in the right environment, the $+$ operator and the Σ operator are restricted to concatenating non-overlapping environments. When $+$ is used in an inference rule, the type environments on either side of the $+$ cannot overlap. For instance, consider the *TuplePat* rule. The pattern identifiers within a tuple pattern cannot have the same name. In the *ConsPat* rule the same identifiers cannot be used in the head or the tail pattern.

New environments are yielded while type checking declarations and patterns. These new environments are consulted while type checking expressions that contain identifiers. This propogation of type environment information is shown explicitly in the type inference rules for *Small* through the use of the $\Rightarrow$, $\oplus$, $+$, and Σ operators.

5. Polymorphic Types

Standard ML supports polymorphic functions and values. So, figure 12 is a valid Standard ML and *Small* program. However, without a little work, the Prolog type variable bound to x will be instantiated to $int \; list$ by the first expression and therefore will not unify with $string \; list$ when type checking the second expression.

```
1 let val x = []
2 in
3   1::x;
4   "hi"::x
5 end
```

Figure 12. Polymorphic Values

```
typecheckDec(Env,bindval(Pat,E),NewEnv):-
  typecheckPat(Pat,ExpType,NewEnv),
  typecheckExp(Env,E,ExpType),
  closeExp(Pat,ExpType).
```

Figure 13. Closing Value Types

The example in figure 12 illustrates the need for closing and instantiating types within the Standard ML type inference system. A Prolog variable will unify with the first term applied to it. So, without further work, the type α unifies with int in the first expression. This demonstrates that type variables must be distinct from Prolog variables and provides an understanding of the need for *typevar* types in figure 7. The *ValDec* rule closes the type of *e*, resulting in

converting any Prolog variables in the type of α into a type containing polymorphic *typevar* types as needed. The code in figure 13 is the Prolog implementation of this rule.

ValDec

$$\frac{pat : \alpha \Rightarrow \varepsilon_{pat},\ \ \varepsilon \vdash e : close(\alpha)}{\varepsilon \vdash val\ pat = e \Rightarrow \varepsilon_{pat}}$$

FunApp

$$\frac{\varepsilon \vdash e_1 : \alpha \to \beta,\ \ \alpha' \to \beta' : inst(\alpha \to \beta),\ \ \varepsilon \vdash e_2 : \alpha'}{\varepsilon \vdash e_1 e_2 : \beta'}$$

```
typecheckExp(Env, apply(Exp1, Exp2), ITT):-
  typecheckExp(Env, Exp1, fn(FT,TT)),
  typecheckExp(Env, Exp2, Exp2Type),
  inst(Exp2Type, Exp2TypeInst),
  catch(inst(fn(FT,TT),
    fn(Exp2TypeInst, ITT)), _,
  printApplicationErrorMessage(
    Exp1, fn(FT,TT), Exp2, Exp2Type, ITT)), !.
```

Figure 14. Instantiating Types

When the polymorphic value is applied in an expression, its polymorphic type may be instantiated into a type containing Prolog variables that are suitable for unification once again. This is illustrated in the *FunApp* rule. The corresponding Prolog code is given in figure 14. The Prolog predicate creates an instance of the argument type as *Exp2TypeInst*. This instantiated argument type must unify with the argument type of the function. If it does not, a function application error message is printed. Otherwise, the type *ITT* is the resulting type of the function application. Closing and instantiating polymorphic types is covered in the next section.

6. Close and Inst

Two operations, *close* and *inst* are needed to go between polymorphic types containing ML type variables and those containing Prolog (i.e. unifiable) variables. The *inst* operation is pretty straightforward and is covered first. The *inst* predicate simply calls the *instanceOf* predicate.

Line 8 of figure 15 handles any simple types like *int* and *bool* where the type is a simple atom. No need to write code for each individual primitive type in this case.

The real purpose of the *instanceOf* predicate is to remember the instantiation of all ML type variables and their Prolog counterparts. When a *typevar(a)* is found within a type expression, any other occurrences of *typevar(a)* within the type expression should be instantiated to the same Prolog type variable. That is the purpose of lines 20 and 21 in the *instanceOf* definition.

If the Prolog interpreter attempts to satisfy the definition of *instanceOf* on lines 20-21 and fails, then it will backtrack

```
1  exists(Env, Id):-
2    member((Id, _), Env), !.
3  instanceOfList(Env, [], [], Env).
4  instanceOfList(Env, [H|T], [G|S], NEnv):-
5    instanceOf(Env, H, G, Env1),
6    instanceOfList(Env1, T, S, NEnv).
7  instanceOf(Env, A, A, Env):-var(A), !.
8  instanceOf(Env, A, A, Env):-simple(A), !.
9  instanceOf(Env, fn(A,B),
10       fn(AInst, BInst), Env2):-
11   instanceOf(Env, A, AInst, Env1),
12   instanceOf(Env1, B, BInst, Env2), !.
13 instanceOf(Env, listOf(A),
14       listOf(B), NEnv):-
15   instanceOf(Env, A, B, NEnv), !.
16 instanceOf(Env, ref(A), ref(B), NEnv):-
17   instanceOf(Env, A, B, NEnv), !.
18 instanceOf(Env, tuple(L), tuple(M), NEnv):-
19   instanceOfList(Env, L, M, NEnv), !.
20 instanceOf(Env, typevar(A), B, Env):-
21   exists(Env, A), find(Env, A, B), !.
22 instanceOf(Env,
23       typevar(A), B, [(A,B)|Env]):-!.
24 instanceOf(_, A, B, _):-
25   print('Type Error: Type '),
26   printType(B, _),
27   print(' is not an instance of '),
28   printType(A, _), nl,
29   throw(
30     typeerror('type mismatch')), !.
31 inst(X,Y):-instanceOf([], X, Y, _).
```

Figure 15. The Inst Predicate

and attempt to satisfy *instanceOf* using lines 22-23. It is here where the *typevar(a)* gets instantiated to the Prolog variable *B*. The *instanceOf* predicate also yields a new environment, adding the mapping of *typevar(a)* to *B* into the evironment so any further occurrences of *typevar(a)* may also be replaced with Prolog variable *B*. Notice that while *B* appears twice in lines 22-23, it is uninstantiated at this point and will unify with another type expression during type inference.

```
1  closeFunTypes([]) :- !.
2  closeFunTypes([(Id, Type)|Tail]) :-
3    closeFunTypes(Tail), print('val '),
4    print(Id), print(' = fn : '),
5    printType(Type, _), nl, !.
6  closeExp(Pat, Type) :-
7    print('val '),
8    printPat(Pat), print(' : '),
9    printType(Type, _), nl, !.
```

Figure 16. The Close Predicate

Closing a type is a bit more nuanced. A side-effect of closing a type is printing its type to the screen. Seeing the types of all named functions and values within an expression can be useful in some circumstances. So, closing a type and printing it to the screen happen at the same time. Because functions and values are printed slightly differently, there is a *closeFunTypes* predicate and a separate *closeExp* predicate. Both predicates call the *printType* predicate which does the actual closing of the type.

Recall that closing a type means instantiating any Prolog variables to an ML *typevar* instance. This happens in the first four lines of figure 17. Any Prolog variable will unify with the first term it encounters. When printing a type, a Prolog variable will unify with the first definition of *printTypeH*. Thus, the Prolog variable becomes instantiated to a *typevar*.

The *printTypeH* also remembers any uninstantiated *typevar* variables it encounters by building a list of all uninstantiated typevars found in an expression. If a Prolog variable is instantiated in lines 1-4, the value to which it is instantiated, *typevar(A)* also contains an uninstantiated Prolog variable *A*. So, *printTypeH* is able to distinguish between an uninstantied Prolog variable on lines 1-4, and a previously instantiated *typevar* variable on lines 5-6. If the final program contains uninstantiated Prolog type variables, a warning message is printed that the uninstantiated types were instantiated to dummy type variables. For example, consider the trivial *Small* program of `[]`. When this program is type checked, its resulting type is $\alpha\ list$. The type inference system reports that α was instantiated to a dummy variable.

The call to the *getNextVar* on lines 4-5 instantiates *A* to a unique *typevar* identifier after it was determined that *A* was uninstantiated previously. Notice that printType builds an environment of uninstantiated type variables, but does not ever use that environment. No environment is needed since the unclosed type contains Prolog variables and unification automatically converts all occurrences of an uninstantiated type to its instantiated counter-part at the same time. Prolog's unification mechanism makes it possible to close a type without the use of any extra environment. Effectively, the environment is maintained by Prolog in this case.

```
1  printTypeH(Env,typevar(A),NewEnv):-
2    var(A), getNextVar(Env,A),
3    printTypeH([A|Env],
4      typevar(A),NewEnv),!.
5  printTypeH(Env,typevar(A),[A|Env]):-
6    print(''''), print(A), !.
7  printTypeH(Env,tuple([]),Env):-
8    print(unit), !.
9  printTypeH(Env,tuple(L),NewEnv):-
10   printTupleTypeH(Env,L,NewEnv), !.
11 printTypeH(Env,listOf(A),NewEnv):-
12   printSubTypeH(Env,A,NewEnv),
13   print('␣list'),!.
14 printTypeH(Env,ref(A),NewEnv):-
15   printSubTypeH(Env,A,NewEnv),
16   print('␣ref'),!.
17 printTypeH(Env,fn(A,B),Env2):-
18   var(A), printSubTypeH(Env,A,Env1),
19   print('␣->␣'),
20   printTypeH(Env1,B,Env2),!.
21 printTypeH(Env,fn(tuple(L),B),NewEnv):-
22   printTypeH(Env,tuple(L),Env1),
23   print('␣->␣'),
24   printTypeH(Env1,B,NewEnv),!.
25 printTypeH(Env,fn(A,B),Env2):-
26   printSubTypeH(Env,A,Env1),
27   print('␣->␣'),
28   printTypeH(Env1,B,Env2), !.
29 printTypeH(Env,typeerror,Env):-
30   print(typeerror), !.
31 printTypeH(Env,T,Env):-
32   simple(T), print(T), !.
33 printTypeH(Env,T,Env):-
34   print('Error:␣unknown␣type␣'),
35   print(T),
36   throw(typeerror('unknown␣type')),!.
37 printType(T,TypeVars):-
38   printTypeH([],T,TypeVars).
```

Figure 17. The PrintType Predicate

7. Mutually Recursive Functions

Type checking mutually recursive functions creates some special challenges. When functions are mutually recursive their types may also be mutually dependent on each other. This is another instance where the unification algorithm is useful. The *FunDecs* rule is fairly complicated to read. It specifies that to type check a series of mutually recursive functions, each function is first assigned a type $\alpha_i \rightarrow \beta_i$. The types α_i and β_i are Prolog variables initially.

FunDecs

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

To implement this rule, an auxiliary predicate called *gatherFuns* is used to collect the list of mutually recursive functions, assign each of them an $\alpha_i \rightarrow \beta_i$ type, and bind each of the function identifiers to their types in the type environment. Notice that the *typecheckFun* predicate does not yield

a type. This is because the types of each of the potentially mutually recursive functions is further specified in the environment as function applications are type checked. Each time a function is applied, either its α_i or β_i type is further specified, or both. However, the *FunApp* rule creates a new instance of a type each time the function is applied. This would hinder the further narrowing of the types in the mutually recursive functions except for line 7 of figure 15 which specifies that creating an instance of a type which is already a Prolog variable leaves the Prolog variable *as-is*, leaving the original Prolog variable in place. The *inst* operation only creates instances of ML type variables. Prolog type variables are left *as-is*.

```
1  gatherFuns([],[]):-!.
2  gatherFuns([funmatch(Id,_)|Tail],
3     [(Id,fn(_,_))|FEnv]):-
4        gatherFuns(Tail,FEnv),!.
5  typecheckFun(Env,funmatch(Id,Matches)):-
6     typecheckMatches(Env,Id,Matches).
7  typecheckFuns(_,[]):-!.
8  typecheckFuns(Env,[FunMatch|Tail]):-
9     typecheckFun(Env,FunMatch),
10    typecheckFuns(Env,Tail),!.
11 typecheckDec(Env,funmatches(L),NewEnv):-
12    gatherFuns(L,FunsEnv),
13    append(FunsEnv,Env,NewEnv),
14    typecheckFuns(NewEnv,L),
15    closeFunTypes(FunsEnv).
```

Figure 18. The FunDecs Implementation

8. Conclusions and Future Work

The framework presented here provides a structure that can be applied to prototyping languages. Mapping a language from its concrete syntax to abstract syntax is relatively easy once both the grammar and abstract syntax are formally specified. Type checking a language can be accomplished by writing the abstract syntax to an intermediate file, which in this case was called *a.term*. The abstract syntax tree can then be read by Prolog along with a set of type inference rules written for type checking programs from the prototyped language. Finally, code generation to the CoCo virtual machine is relatively easy to write with the use of a disassembler for Python programs that can be used to discover *secrets* of code generation for similar Python programs.

This framework is flexible in that it is modular. Any part of the framework could be replaced with a similarly designed component. CoCo could be replaced by another architecture like the JVM. Type inference can be accomplished via a specific type inference implementation employing its own unification algorithm if desired. However, each of the provided modular pieces is also easily extensible. Prolog is obviously very flexible. The CoCo virtual machine is written using OO design techniques and could easily be extended with new types and built-in functions. Functionality provided by some other library can easily be added to CoCo through new instructions or calls to new built-in functions.

```
1  let val r = ref (fn x => x)
2  in
3     r:= (fn x => x+1);
4     !r true
5  end
```

Figure 19. A Value Restriction Problem

There is additional work that can be done with the Small language and its implementation. It could be extended to support more of the types of Standard ML including real numbers and *option*al values. More of the Standard ML library could be implemented along with the module system and support for datatypes, structures, and signatures.

The value restriction of Standard ML '97 was not considered in this work (MacQueen 2000). The type inference implementation could be extended to implement the value restriction making the type inference system sound in the presence of side effects. Consider the example from (MacQueen 2000) in figure 19. Currently, this program passes the type checker described here because each expression is type checked independently. However, the reference *r* is used to *communicate* between the two expressions, causing a runtime error in the CoCo virtual machine.

While much work remains that could still be done, much was also learned through prototyping the *Small* language using this framework. This paper has highlighted the use of Prolog in implementing type checking of a program. Type inference must be implemented via unification in some form or other. This paper has shown how this can be accomplished in Prolog and how the framework can be applied in a systematic way for other languages as well.

The system covered in this paper is available from the public repository *http://github.com/kentdlee/MLComp*. Since this code is also used for teaching purposes not all of the supporting functions are implemented in this publicly available repository. A private repository is also available to researchers and educators that contains the full implemention of the *Small* language and its type inference system. Access to the private repository can be granted by contacting the author and providing proof of status as a researcher or educator.

A. CoCo Code for Figures 1 and 2

```
Function: f/2
    Function: g/2
    Constants: None, 1
    Locals: x, y
    Globals: f
    BEGIN
                LOAD_FAST                   0
                LOAD_CONST                  1
                COMPARE_OP                  0
                POP_JUMP_IF_FALSE label00
                LOAD_FAST                   1
                RETURN_VALUE
    label00:    LOAD_GLOBAL                 0
                LOAD_FAST                   1
                LOAD_CONST                  1
                BINARY_SUBTRACT
                LOAD_FAST                   0
                CALL_FUNCTION               2
                RETURN_VALUE
                LOAD_CONST                  0
                RETURN_VALUE
    END
Constants: None, code(g)
Locals: x, y, g
BEGIN
            LOAD_CONST                      1
            MAKE_FUNCTION                   0
            STORE_FAST                      2
            LOAD_FAST                       2
            LOAD_FAST                       0
            LOAD_FAST                       1
            CALL_FUNCTION                   2
            RETURN_VALUE
END
Function: main/0
Constants: None, 5, 12
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
```

B. Small AST Definition

```
datatype
    exp = int of string
        | ch of string
        | str of string
        | bool of string
        | id of string
        | listcon of exp list
        | tuplecon of exp list
        | apply of exp * exp
        | expsequence of exp list
        | letdec of dec * (exp list)
        | handlexp of exp * match list
        | ifthen of exp * exp * exp
        | whiledo of exp * exp
        | func of int * match list
  and
    match = match of pat * exp
  and
    pat = intpat of string
        | chpat of string
        | strpat of string
        | boolpat of string
        | idpat of string
        | wildcardpat
        | infixpat of string * pat * pat
        | tuplepat of pat list
        | listpat of pat list
        | aspat of string * pat
  and
    dec = bindval of pat * exp
        | bindvalrec of pat * exp
        | funmatch of string * match list
        | funmatches of
              (string * match list) list
```

C. The Small Type Inference Rules

BoolCon

$$\frac{}{\varepsilon \vdash bool(v) : bool}$$

IntCon

$$\frac{}{\varepsilon \vdash int(v) : int}$$

StringCon

$$\frac{}{\varepsilon \vdash str(v) : str}$$

ListCon

$$\frac{\forall i \ 1 \leq i \leq n, n \geq 0 \qquad \varepsilon \vdash e_i : \alpha}{\varepsilon \vdash [e_1, e_2, ..., e_n] : \alpha \ list}$$

TupleCon

$$\frac{\forall \ 1 \leq i \leq n, n \geq 0 \qquad \varepsilon \vdash e_i : \alpha_i}{\varepsilon \vdash (e_1, e_2, ..., e_n) : \times_{i=1}^{n} \alpha_i}$$

Identifier

$$\frac{}{\varepsilon[id \mapsto \alpha] \vdash id : \alpha}$$

FunApp

$$\frac{\varepsilon \vdash e_1 : \alpha \to \beta, \ \ \alpha' \to \beta' : inst(\alpha \to \beta), \ \ \varepsilon \vdash e_2 : \alpha'}{\varepsilon \vdash e_1 e_2 : \beta'}$$

Let

$$\frac{\varepsilon \vdash dec \Rightarrow \varepsilon_{dec},\ \ \varepsilon_{dec} \oplus \varepsilon \vdash e_{sequence} : \beta}{\varepsilon \vdash let\ dec\ in\ e_{sequence}\ end : \beta}$$

ValDec

$$\frac{pat : \alpha \Rightarrow \varepsilon_{pat},\ \ \varepsilon \vdash e : close(\alpha)}{\varepsilon \vdash val\ pat = e \Rightarrow \varepsilon_{pat}}$$

ValRecDec

$$\frac{[id : \alpha] \oplus \varepsilon \vdash e : \alpha}{\varepsilon \vdash val\ rec\ id = e \Rightarrow [id : close(\alpha)]}$$

FunDecs

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

IntPat

$$\frac{}{integer\_constant : int \Rightarrow [\,]}$$

BoolPat

$$\frac{}{true : bool \Rightarrow [\,]}$$

$$\frac{}{false : bool \Rightarrow [\,]}$$

StrPat

$$\frac{}{string\_constant : str \Rightarrow [\,]}$$

NilPat

$$\frac{}{nil : \alpha\ list \Rightarrow [\,]}$$

ConsPat

$$\frac{pat_1 : \alpha \Rightarrow \varepsilon_{pat_1},\ \ pat_2 : \alpha\ list \Rightarrow \varepsilon_{pat_2}}{pat_1 :: pat_2 : \alpha\ list \Rightarrow \varepsilon_{pat_1} + \varepsilon_{pat_2}}$$

TuplePat

$$\frac{\begin{array}{l}\forall i\ 1 \leq i \leq n, n \geq 0 \\ pat_i : \alpha_i \Rightarrow \varepsilon_{pat_i}\end{array}}{(pat_1, pat_2, ..., pat_n) : \times_{i=1}^{n} \alpha_i \Rightarrow \sum_{i=1}^{n} \varepsilon_{pat_i}}$$

ListPat

$$\frac{\begin{array}{l}\forall i\ 1 \leq i \leq n, n \geq 0 \\ pat_i : \alpha \Rightarrow \varepsilon_{pat_i}\end{array}}{[pat_1, pat_2, ..., pat_n] : \alpha\ list \Rightarrow \sum_{i=1}^{n} \varepsilon_{pat_i}}$$

IdPat

$$\frac{}{id : \alpha \Rightarrow [id \mapsto \alpha]}$$

Matches There are two alternatives to the *Matches* rule differing only in the syntax of the match.

$$\frac{\begin{array}{l}\forall i\ 1 \leq i \leq n, \forall j\ 1 < j \leq n,\ n \geq 1 \\ \varepsilon \vdash id : \alpha \rightarrow \beta,\ \ pat_i : \alpha \Rightarrow \varepsilon_{pat_i},\ \ \varepsilon_{pat_i} \oplus \varepsilon \vdash e_i : \beta\end{array}}{\varepsilon \vdash id\ pat_1 = e_1 \{|\ id\ pat_j = e_j\} : \alpha \rightarrow \beta}$$

or

$$\frac{\begin{array}{l}\forall i\ 1 \leq i \leq n, \forall j\ 1 < j \leq n,\ n \geq 1 \\ \varepsilon \vdash id : \alpha \rightarrow \beta,\ \ pat_i : \alpha \Rightarrow \varepsilon_{pat_i},\ \ \varepsilon_{pat_i} \oplus \varepsilon \vdash e_i : \beta\end{array}}{\varepsilon \vdash id\ pat_1 => e_1 \{|\ pat_j => e_j\} : \alpha \rightarrow \beta}$$

AnonFun

$$\frac{[id \mapsto \alpha \rightarrow \beta] \oplus \varepsilon \vdash id\ matches : \alpha \rightarrow \beta}{\varepsilon \vdash fn\ id\ matches : \alpha \rightarrow \beta}$$

IfThen

$$\frac{\varepsilon \vdash e_1 : bool,\ \ \varepsilon \vdash e_2 : \alpha,\ \ \varepsilon \vdash e_3 : \alpha}{\varepsilon \vdash if\ e_1\ then\ e_2\ else\ e_3 : \alpha}$$

WhileDo

$$\frac{\varepsilon \vdash e_1 : bool,\ \ \varepsilon \vdash e_2 : \alpha}{\varepsilon \vdash while\ e_1\ do\ e_2 : \alpha}$$

Handler

$$\frac{\begin{array}{l}\varepsilon \vdash e : \alpha, \\ [handle@ \mapsto\ exn \rightarrow \alpha] \oplus \varepsilon \vdash \\ \quad handle@\ matches : exn \rightarrow \alpha\end{array}}{\varepsilon \vdash e\ handle\ matches : \alpha}$$

Acknowledgments

I would like to thank John Reppy who was consulted during this research for feedback on a couple of the type inference rules. His feedback was valuable to me and appreciated.

References

K. D. Lee. *Foundations of Programming Languages*. Springer, 2014. ISBN 978-3-319-13313-3.

K. D. Lee. A framework for teaching programming languages. In *SIGCSE '15: Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2966-8.

D. MacQueen. Types and type checking: A discussion of the value restriction of sml '97 on the sml/nj website. `http://www.smlnj.org/doc/Conversion/types.html`, 2000.