CS403: Programming Languages

Notes on Static Typing

Static typing

There are two basic approaches to typing found in modern languages, static typing, which is generally used in languages for mission critical applications, and dynamic typing, which is generally used in languages for rapid prototyping and simple task automation (such as shell scripts). In a statically typed language, type mismatches are discovered at compile time (or parse time for interpreters). In a dynamically typed language, such errors are discovered at run time.

Building an interpreter for a dynamically typed language is quite simple - one simply ignores type consideration until the last possible moment. Thus, if a statement in the language tries to combine two objects in some illegal way, as in:

```
a = x % 4.3;  // modulus valid only for integral types
```

the parser for the interpreter will gladly build a parse tree for performing modulus with a non-integral divisor. Only when the statement is executed by the evaluator will the error be uncovered.

A Grammar for Types

The first step to adding static typing is to modify your grammar so that all variable definitions have a type. This includes function definitions as well. For example, a rule like:

```
varDef : VAR ID ASSIGN expr SEMI
```

would become:

```
varDef : type ID ASSIGN expr SEMI
```

In addition, a rule for defining functions, like:

```
funcDef : function ID OPAREN optParamList CPAREN block
```

would become:

```
funcDef : type ID OPAREN optTypedParamList CPAREN block
```

Note that both *varDef* and *funcDef* both start the same, so your grammar will have to deal with that, but for illustrative purposes, we will discuss them as separate rules. We are left with defining the *type* rule. At first, this seems to be a simple task:

```
type : INT
     | STR
```

where `INT` denotes the keyword for an integer type specifier and `STR` denotes the keyword for a string type specifier. The problem here is that your functions must be able to take functions as arguments and return them. These functions need to be typed as well. Here is one possible typing scheme which extends the previous rule to allow for function types:

```
type : INT
     | STR
     | OPAREN type COLON optTypeList CPAREN //function type

optTypeList : *empty*
            | typeList

typeList : type
         | type COMMA typeList
```

Consider the *square* function, which takes an integer and returns an integer. In a typeless language, it might be defined as

```
function square(x)
    {
    x * x;
    }
```

In our typing scheme, it would be

```
int square(int x)
    {
    x * x;
    }
```

Now consider a function that returns the *square* function! Typelessly, it would be:

```
function f()
    {
    square;
    }
```

But using our extended type rule above, it would be:

```
(int:int) f()
    {
    square;
    }
```

Suppose we wish to return a function that composes two given functions (like the *compose* function in the book). Here is one possible definition:

```
(int:int) compose((int:int) f,(int:int) g)
    {
    int both(int x)
        {
        f(g(x));
        }
    both;
    }
```

Suppose the *zorp* function returned the *compose* function. Its definition would be:

```
((int:int):(int:int),(int:int)) zorp()
    {
    compose;
    }
```

As you can see, keeping up with types when doing functional-style programming can be rather tedious.

Keeping track of types

After adding the syntax, actually checking to make sure types match boils down to keeping track of variables and their types with an *environment*. We did something very similar when *detecting undefined variables* at parse time. For static typing, every variable (including variables bound to functions) are inserted into an environment with its *type* as its value.

To begin, nearly every parsing routine is now passed an environment. Consider a parsing function for a simple variable definition:

```
// varDef : type ID ASSIGN expr SEMI
function varDef(env)
    {
    var varType;
    var name,value;

    varType = type();
    name = match(ID);
    match(ASSIGN);
    value = expr(env);
```

```
    match(SEMI);

    insert(name,varType,env);

    return cons(VARDEF,name,value);
    }
```

The environmental *insert* function places the variable name and its type in the hash table at the head of the chain, since we have just defined a local variable.

Note that we are using a typed *cons* function. It returns a lexeme whose left pointer is the name and whose right pointer is the initialization expression and whose type is `VARDEF`.

The modification for a function definition is similar, except the environment is extended prior to parsing the body:

```
// funcDef : type ID OPAREN optTypedParamList CPAREN block
function funcDef(env)
    {
    var name,params,body;
    var returnType;
    var xenv;

    returnType = type();
    name = match(ID);
    match(OPAREN);
    params = optTypedParamList();
    match(CPAREN);

    insert(name,makeFuncType(returnType,extractTypes(params)),env);

    xenv = extend(extractNames(params),extractTypes(params),env);

    body = block(xenv);

    return cons(FUNCDEF,name,cons(NO_TYPE,params,body));
    }
```

Note that any time a block is encountered, the environment needs to be extended, since local definitions may occur in the block. The environmental *extend* function takes a list of variables and a parallel list of types, builds a new frame from the two lists, and returns an environment with the new frame at the head.

For example, suppose we are parsing the *urp* function:

```
int urp(int x,int y)
    {
    x * y;
    }
```

At the point where we call the *insert* function, *name* is bound to *urp* and *returnType* is bound to `INT`. The variable *params* is bound to the list containing `int x,int y` (in some form). The function *extractTypes* should return the list containing `int,int` (in some form). The function *makeFuncType* should return `(int:int,int)` (in some form). Moreover, and this is very important: the structure returned by *makeFuncType* should match exactly the structure returned by the *type* parsing function if it encountered the type:

```
(int:int,int)
```

in the source code.
After inserting the type of the function into the environment, the environment is extended. This extension models the static scoping of the language. As you recall for evaluating function calls, the extended environment is populated with the formal parameters and their values. In this case, the value of a formal parameter is its type. Under this extended environment, the body of the function definition is parsed.

Finally, the function object is created and returned. Since these are the only places variables are given types, we are done with the 'keeping track' part. What's left is the 'checking that types match' part.

Checking that all types match

The last stage of the implementation of static typing is to make sure that all expressions are typed and that when the parse trees of expressions are combined, type violations are reported.

This presents a bit of a quandary. The recursive descent parsing functions must return parse trees, but now they must return the types of those parse trees as well. A simple solution is to add a new field to the lexemes that make up the parse trees. We will call that field *resultType*, to distinguish it from the *type* field of the lexeme. Now, every parse tree will have *resultType* set to the type of the overall expression that the parse tree represents. The field *resultType* should point to a lexeme in order for the type matching routine (see below) to function properly.

Typing primaries

Since all expression parse trees start with *unary*, we modify the *unary* parse function to set *returnType*.

```
function unary(env)
    {
    var result;

    if (check(INTEGER))
        {
        result = match(INTEGER);
        result . resultType = new Lexeme(INT);
        }
    else if (check(STRING))
        {
        result = match(STRING);
        result . resultType = new Lexeme(STR);
        }
    else if (check(ID))
        {
        result = match(ID);
        result . resultType = lookup(result,env);
        if (check(OPAREN))
            {
            //process function call
            ...
            }
        }
    //other primaries
    ...

    return result;
    }
```

Recall that `INTEGER` is the type of an integer lexeme, while `INT` is the type of the integer type specifier in the target language.

Note that the code for identifiers is where we see the type environment come into play. The lookup routine serves two purposes. The first purpose is to detect undefined variables. If the variable just parsed is not in the type environment, then it is considered undefined. The second purpose is to retrieve the type of the variable that was stored in the environment when the variable was declared. In our example language, the type would be an `INT` or a `STR` or a function type.

The next unary we tackle is function call. Here, we have to ensure the identifier is actually bound to a function and that the argument types match the defined types of the formal parameters.

```
if (check(ID))
    {
    var idtype;
    result = match(ID);
    idtype = lookup(result,env);
    if (check(OPAREN))
        {
        var returnType;
        ensureFunction(result);
        returnType = extractReturnType(idtype);
        match(OPAREN);
```

```
        args = optArgList(env);
        match(CPAREN);
        matchTypes(extractParamTypes(idtype),extractArgTypes(args));
        result = cons(CALL,result,args);
        result . resultType = returnType;
        }
    else
        result . resultType = idtype;
    }
```

Once the `check(OPAREN)` confirms that this is truly a function call, we use `ensureFunction` to ensure that the identifier will be bound to a function. The *ensureFunction* routine, need only check that the given cons cell has type FUNCTYPE.

Next we extract and save the return type of the function as that will be the type of the overall function call parse tree.

After parsing the arguments, the *matchTypes* routine is called to compare the lists of types that *extractParamTypes* and *extractArgTypes* return. They must match exactly. If they do not, a fatal error can be generated at this point.

Finally, the call parse tree is constructed and typed with the return type of the function.

Other primaries, such as parenthesized expressions, are left to the student.

Typing expressions

Primaries (and expressions) are joined together by the expression rule. Here, the compatibility of the pieces being joined together is examined and the resulting parse tree is given a type. A data-directed approach is ideal for this application:

```
function expr(env)
    {
    var root,left,right;

    left = unary(env);
    if (opPending())
        {
        root = op();
        right = expr(env);
        root . resultType =
            get(root . type,left . resultType,right . resultType);
        root . left = left;
        root . right = right;
        }
    else
        root = left;

    return root;
    }
```

The *get* function examines a table loaded with triples `(operator,type,type)`, with each triple associated with a resulting type. The resulting type is returned by *get*. If get can't find a matching triple, it reports a type error and ends execution.

For example, if the table is loaded with:

```
(+,int,int) -> int
```

this means that an integer added to an integer yields an integer. If this is the only entry in the table, the source code fragment:

```
"hello" + ", world"
```

would generate a type error.

Checking function definitions

Another important type check is to make sure the value returned by a function matches the return value specified by the function definition. Here is a modified version of *funcDef* that does just that:

```
// funcDef : type ID OPAREN optTypedParamList CPAREN block
function funcDef(env)
    {
    var name,params,body;
    var returnType;
    var xenv;

    returnType = type();
    name = match(ID);
    match(OPAREN);
    params = optTypedParamList();
    match(CPAREN);

    insert(name,makeFuncType(returnType,extractTypes(params)),env);

    xenv = extend(extractNames(params),extractTypes(params),env);

    body = block(xenv);

    matchType(block.resultType,returnType);

    return cons(FUNCDEF,name,cons(NO_TYPE,params,body));
    }
```

For this to work, the type of the last statement in the block has to percolate up to be the result type of the block's root note:

```
function block(env)
    {
    var root,item;

    item = statement(env);
    if (statementPending())
        {
        root = cons(BLOCK,item,block(env));
        root.resultType = cdr(root).resultType;
        }
    else
        {
        root = cons(BLOCK,item,null);
        root.resultType = car(root).resultType;
        }
    return root;
    }
```

There are other places where type checks and tags need to be performed:

- the *test* expression for `if` statements
- if an `if` statement has both a true clause and a false clause, the result types of those clauses must match
- the `if` expression itself needs to be tagged
- the *test* expression for `while` loops
- the `while` expression itself needs to be tagged (if the grammar promotes `while`s to first class status
- any other unary operations