Project 6

Due 11:59:59pm Thu, Dec 10, 2015

Updates

- None yet.

Introduction

In this project, you will add a static type checking system to the Rube programming language. Recall the formal syntax for Rube programs, shown in Figure 1. In this project, we will extend this grammar to include static types, and then build a type checker for the resulting language.

To keep everything simpler, this project does *not* build directly on your project 5 code. But, it does use the same lexer and parser, and it should be easy for you to see how you might incorporate type checking into your compiler.

Project Structure

The project skeleton code is divided up into the following files:

`Makefile`	Makefile
`lexer.mll`	Rube lexer
`parser.mly`	Rube parser
`ast.mli`	Abstract syntax tree type
`rubet.ml`	The main type checker logic
`main.ml`	Parse the specified input file and call type checker
`r1.ru`	Hello world example

You will only change `rubet.ml`, `lexer.mll`, and `parser.mly`, and write `part5.txt`; you should not edit any of the other files. The file `main.ml` includes code to run the parser and then dispatch to the type checker. Right now, the "type checker" implementation just unparses the input file to standard output, e.g., after running make you can run:

```
$ ./main.byte r1.ru
"Hello, world!".print()
yes
```

to parse `r1.ru` and print the parsed result to standard out. You'll change this code so that the parsed input is no longer printed to standard output. The `yes` indicates that this program type checks. In fact, the current implementation will always print `yes`, until you implement your type checker.

Part 1: Adding Type Annotations to Rube

Next, we are going to add type annotations to Rube. Figure 2 shows the necessary changes to the source language. In the revised language, classes begin with field definitions, which include types, followed by method definitions, which include type annotations on arguments and the method return (this type is to the left of the method name). We also introduce a new expression ($E : T$) that performs a dynamic type cast to check at run time that E has type T.

P	::=	$C^* \ E$	Rube program
C	::=	class $id < id$ begin M^* end	Class definition
M	::=	def id $(id, \ldots, id)$ begin E end	Method definition
E	::=	n	Integers
	|	`nil`	Nil
	|	"str"	String
	|	`self`	Self
	|	id	Local variable
	|	$@id$	Field
	|	`if` E `then` E `else` E `end`	Conditional
	|	`while` E `do` E `end`	While loop
	|	$E; \ E$	Sequencing
	|	$id = E$	Local variable write
	|	$@id = E$	Field write
	|	`new` id	Object creation
	|	$E.id(E, \ldots, E)$	Method invocation

Figure 1: Rube syntax (without types)

C	::=	class $id < id$ begin $F^* \ M^*$ end	Class definition
F	::=	$@id : T$	Field type
M	::=	def T id $(id : T, \ldots, id : T)$ L^* begin E end	Method definition
E	::=	... | $(E : T)$	Type cast
L	::=	$id : T$	Local variable type
T	::=	id	Types
MT	::=	$(T \times \cdots \times T) \rightarrow T$	Method type

Figure 2: Changes to Rube to add static types

Types T are simply class names. Our system will include distinguished classes `Bot`, the class of `nil` (which can masquerade as an instance of any class) and `Object`, the root of the class hierarchy. In other settings we might write $\perp$ ("bottom") rather than `Bot` and $\top$ ("top") rather than `Object`.

Finally, we give a definition of *method types* MT of the form $(T_1 \times \cdots \times T_n) \rightarrow T$, which is a method such that its ith argument has type T_i and it returns type T. Method types are *not* allowed to appear in the surface syntax, so you don't need to parse them; however, they are handy to have during type checking.

Abstract syntax trees Figure 3 shows the OCaml abstract syntax tree data types for programs with type annotations. It is quite similar to the grammar for the untyped variant of the language; we'll highlight the differences next. **Important:** Don't modify these constructors or types. Otherwise our grading scripts won't work.

`EInvoke(e,s,el)` corresponds to calling method `s` of object `e` with the arguments given in `el`. (The arguments are in the same order in the list as in the program text, and may be empty.) The argument list is a sequence of pairs containing an optional string and the expression passed as an arugment. The string is used to indicate a call to a named argument, and will only be used for part 5 of the project. Thus, you may safely assume it is `None` for parts 1–4. Finally, `ECast(e,t)` represents a type cast.

A type `typ` is just a string, wrapped in the constructor `TClass`, e.g., `TClass "Bot"` is the type of `nil`. We added a constructor here to make it slightly harder to mix up strings corresponding to types with other strings. We also define `mtyp` for method types; we'll use this as the return type of a key function below.

A method `meth` is a record containing the method name, return type, arguments (with types), local variable definitions, and method body. Method arguments also have an optional expression and an optional

```
type expr =
    EInt of int
  | ENil
  | ESelf
  | EString of string
  | ELocRd of string      (* Read a local variable *)
  | ELocWr of string * expr
(* Write a local variable *)
  | EFldRd of string      (* Read a field *)
  | EFldWr of string * expr  (* Write a field *)
  | EIf of expr * expr * expr
  | EWhile of expr * expr
  | ESeq of expr * expr
  | ENew of string
  | EInvoke of expr * string *
      ((string option * expr) list )
  | ECast of expr * typ

and typ = TClass of string
and mtyp = (typ list ) * typ
```

```
(*  meth name * arg name list * method body *)
type meth = { meth_name : string;
              meth_ret: typ;
              meth_args : (string * typ *
                expr option * string option) list ;
              meth_locals : (string * typ) list ;
              meth_body : expr }

(* class name * superclass name * methods *)
type cls = { cls_name : string ;
             cls_super : string ;
             cls_fields : (string * typ) list ;
             cls_meths : meth list }

(* classes * top-level expression *)
type prog = { prog_clss : cls list ;
              prog_main : expr }
```

Figure 3: Abstract syntax tree for Rube with type annotations

string, denoting a default value and a name. You can assume these last two fields are always `None` until you get to part 5 of the project. A class `cls` is a record containing the class name, superclass, fields, and methods. Finally, a program `prog` is a record containing the list of classes and the top-level expression.

What to Do For this part of the project, you must extend the Rube lexer and parser to support the grammar extensions just described: fields with types; method type annotations; locals with types; and type casts. (As mentioned earlier, method types (with arrows) will never appear in the surface syntax.) Right now the parser does produce ASTs of the form just described, but it puts type `Bot` wherever a type is needed, and uses empty lists for the locals and fields.

Part 2: Typing Utilities

To implement type checking for Rube with type annotations, we'll need the following utility functions. You should write these functions in `rubet.ml`. **Important:** We will test your code by calling these functions directly, so it's important you put them in the right file.

In the following functions, you should assume the existence of five built-in classes: `Object`, `Integer`, `String`, `Map`, and `Bot`, where `Object` is the root of the inheritance hierarchy, `Integer`, `String`, and `Map` extend `Object`. The class `Bot` is the bottom type, and it also has all the methods of `Object`. Figure 4 gives type signatures for built-in methods of these classes; you should assume these built-in methods exist.

1. Write a function `defined_class p c : prog -> string -> bool` that returns true if and only if class `c` is defined in program `p`. This function should return true if asked whether `Object`, `Integer`, `String`, `Map`, or `Bot` are defined.

2. Write a function `lca_class p c1 c2 : prog -> string -> string -> string` that, given classes `c1` and `c2`, returns the lowest common ancestor of classes `c1` and `c2` in the inheritance hierarchy. For example, `lca_class p "String" "Integer"` would return `"Object"`. Note that if `A` is a subclass of `B`, then `lca_class p "A" "B"` would return `"B"`. Also, `lca_class p "A" "A"` should return `"A"`. Finally, for purposes of this function, `Bot` should be a descendent of every other class, e.g., `lca_class p "String" "Bot"` would return `"String"`.

Class	Method type	
`Object`	`equal?` : (`Object`) $\rightarrow$ `Object`	equality check
	`to_s` : () $\rightarrow$ `String`	convert to string
	`print` : () $\rightarrow$ `Bot`	print to standard out
`String`	`+` : (`String`) $\rightarrow$ `String`	string concatenation
	`length` : () $\rightarrow$ `Integer`	string length
`Integer`	`+` : (`Integer`) $\rightarrow$ `Integer`	addition
	`-` : (`Integer`) $\rightarrow$ `Integer`	subtraction
	`*` : (`Integer`) $\rightarrow$ `Integer`	multiplication
	`/` : (`Integer`) $\rightarrow$ `Integer`	division
`Map`	`find : (Object) -> Object`	return value mapped to key
	`insert : (Object, Object) -> nil`	add key-value mapping
	`has : (Object) -> Object`	check whether key is in map
	`iter : (Object) -> nil`	iterate through mapping
`Bot`		*No additional methods*

Figure 4: Built-in objects and methods

3. Write a function `castable p t1 t2 : prog -> typ -> typ -> bool` that returns true if and only if a cast from type `t1` to type `t2` could potentially succeed at runtime, meaning that either `t1` is an ancestor of `t2` in the inheritance hierarchy (downcast), `t2` is an ancestor of `t1` (upcast), or `t1` = `t2`. For example, `castable p (TClass "Object") (TClass "String")` would return true, but `castable p (TClass "String") (TClass "Integer")` would return false. Because of the odd nature of `nil`, `castable p (TClass x) (TClass "Bot")` should always succeed, for any `x`.

4. Write a function `no_builtin_redef : prog -> bool` that returns true if the program does not try to define classes `Object`, `String`, `Integer`, `Map`, or `Bot`. (Note that we have not specified what to do for programs that redefine user classes; our test cases will never do so, and so it's up to you how to handle such cases.)

5. Write a function `lookup_meth p c m : prog -> string -> string -> mtyp` that returns the type (an `mtyp`) of method `m` in class `c` or, if that method is not defined, it should return the type from `c`'s superclass (and so on, recursively up the class hierarchy). This function should `raise Not_found` if no such method exists in `c` or in any superclass. **Note**: Don't forget about the types of built-in and inherited methods in `Object`, `String`, `Integer`, `Map`, and `Bot`; they should be returned by this function. (Note that we have not specified any particular behavior if the same method is defined twice within one class; our test cases will never do so.)

6. Write a function `lookup_field p c f : prog -> string -> string -> typ` that returns the type of field `f` in class `c`. As with `lookup_meth`, it should recursively explore superclasses to find field definitions if necessary. This function should `raise Not_found` if `f` is not defined in `c` or in any of its superclasses. (Note that we have not specified any particular behavior if the same field is defined twice within one class; our test cases will never do so.)

Part 3: Subtyping

Since our language includes subclassing, we will need to define a subtyping relationship as part of type checking. Figure 5 shows the subtyping rules for this language. In words, the rules are as follows:

- Bot states that the bottom type `Bot` is a subtype of any other type.
- Obj states that any type is a subtype of `Object`.

$$\textsc{Bot}\quad \frac{}{P \vdash \texttt{Bot} \leq T} \qquad \textsc{Obj}\quad \frac{}{P \vdash T \leq \texttt{Object}} \qquad \textsc{Class}\quad \frac{(\text{class } id_1 < id_2 \text{ begin } \ldots \text{ end}) \in P}{P \vdash id_1 \leq id_2}$$

$$\textsc{Trans}\quad \frac{P \vdash id_1 \leq id_2 \qquad P \vdash id_2 \leq id_3}{P \vdash id_1 \leq id_3} \qquad \textsc{Refl}\quad \frac{}{P \vdash id \leq id}$$

$$\textsc{Meth}\quad \frac{T_1' \leq T_1 \quad \cdots \quad T_n' \leq T_n \qquad T \leq T'}{P \vdash (T_1 \times \cdots \times T_n) \to T \leq (T_1' \times \cdots \times T_n') \to T'}$$

Figure 5: Subtyping

- Class says that id_1 is a subtype of id_2 if id_1 is a subclass of id_2 in the program text.
- Trans says that subtyping of classes is transitive, and Refl says that subtyping is reflexive.
- For convenience, we also define method subtyping in this figure. Meth says that for one method type to be a subtype of another, the arguments and return types must have the correct subtyping relationships. Refer to the class notes for an explanation.

Write a function `is_subtype p t1 t2 : prog -> typ -> typ -> bool` that returns true if and only if type `t1` is a subtype of `t2` according to this definition. Also write a function `is_msubtype p mt1 mt2 : prog -> mtyp -> mtyp -> bool` that returns true if and only if method type `mt1` is a subtype of method type `mt2` according to this definition.

Next, notice that the subtyping functions you wrote assume that subclasses are actually subtypes (rule Class); but a programmer could violate this assumption. For example, a programmer could create a class that overrides a method from a parent class and changes the type of that method in an unsafe way:

```
class A < Object begin def int m() 42 end end
class B < A begin def string m() "oops!" end end
```

Thus, the next step is to write a function `check_class p c : prog -> string -> bool` that checks whether the type annotations in `c` make that class a valid subtype of its superclass, according to the following rules:

- For every method `m` defined in `c`, the type of `m` in `c` must be a subtype of `m` in its superclass, or `m` must not be defined (or inherited) by its superclass. (Use `lookup_meth` to help you here, so that you handle the case that the superclass inherits its type for `m`, and to handle the built-in methods of `Object`, `String`, `Integer`, `Map`, and `Bot`.)
- For every field `f` defined in `c`, there is no definition of `f` in any superclass (including transitively). In other words, shadowing fields in subclasses is not allowed in this language.
- Except for `Object`, class `c` may not be an ancestor of itself in the inheritance hierarhcy.
- No class may extend `Bot`.
- Any type mentioned in a method or field signature must exist in the program, or be a built-in class.

Put `is_subtype`, `is_msubtype`, and `check_class` in `rubet.ml`.

$$\textsc{Int}\ \frac{}{P; A \vdash n : \texttt{Integer}} \qquad \textsc{Nil}\ \frac{}{P; A \vdash \texttt{nil} : \texttt{Bot}} \qquad \textsc{Str}\ \frac{}{P; A \vdash \texttt{"}\mathit{str}\texttt{"} : \texttt{String}} \qquad \textsc{Self}\ \frac{}{P; A \vdash \texttt{self} : A(\texttt{self})}$$

$$\textsc{Id}\ \frac{id \in dom(A)}{P; A \vdash id : A(id)} \qquad \textsc{Field-R}\ \frac{A(\texttt{self}) = id_{\texttt{self}}}{P; A \vdash @id : (\texttt{lookup\_field}\ P\ id_{\texttt{self}}\ @id)} \qquad \textsc{If}\ \frac{P; A \vdash E_1 : T_1 \qquad P; A \vdash E_2 : T_2 \qquad P; A \vdash E_3 : T_3}{P; A \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ \texttt{end} : \texttt{lca}\ T_2\ T_3}$$

$$\textsc{While}\ \frac{\begin{array}{c} P; A \vdash E_1 : T_1 \\ P; A \vdash E_2 : T_2 \end{array}}{P; A \vdash \texttt{while}\ E_1\ \texttt{do}\ E_2\ \texttt{end} : \texttt{Bot}} \qquad \textsc{Seq}\ \frac{\begin{array}{c} P; A \vdash E_1 : T_1 \\ P; A \vdash E_2 : T_2 \end{array}}{P; A \vdash (E_1;\ E_2) : T_2} \qquad \textsc{Id-W}\ \frac{P; A \vdash id : T \qquad P; A \vdash E : T' \qquad P \vdash T' \leq T}{P; A \vdash id = E : T}$$

$$\textsc{Field-W}\ \frac{P; A \vdash @id : T \qquad P; A \vdash E : T' \qquad P \vdash T' \leq T}{P; A \vdash @id = E : T} \qquad \textsc{New}\ \frac{\texttt{defined\_class}\ P\ id}{P; A \vdash \texttt{new}\ id : id} \qquad \textsc{Cast}\ \frac{P; A \vdash E : T' \qquad \texttt{castable}\ P\ T'\ T}{P; A \vdash (E : T) : T}$$

$$\textsc{Invoke}\ \frac{\begin{array}{c} P; A \vdash E_0 : T_0' \quad \cdots \quad P; A \vdash E_n : T_n' \\ (\texttt{lookup\_meth}\ P\ T_0'\ id_m) = (T_1 \times \cdots T_k) \to T \qquad k = n \qquad T_1' \leq T_1 \quad \cdots \quad T_n' \leq T_n \end{array}}{P; A \vdash E_0.id_m(E_1, \ldots, E_n) : T}$$

$$\textsc{Method}\ \frac{P; (L^*, id_1 : T_1, \ldots, id_n : T_n, \texttt{self} : id_c) \vdash E : T' \qquad P \vdash T' \leq T}{P; id_c \vdash \text{def}\ T\ \ id\ (id_1 : T_1, \ldots, id_n : T_n)\ L^*\ \text{begin}\ E\ \text{end}}$$

$$\textsc{Class}\ \frac{\begin{array}{c} P; id \vdash M_1 \quad \cdots \quad P; id \vdash M_n \qquad \texttt{check\_class}\ P\ id \\ id \notin \{\texttt{Object}, \texttt{Integer}, \texttt{String}, \texttt{Bot}\} \end{array}}{P \vdash \text{class}\ id < id'\ \text{begin}\ F^*\ M_1\ \cdots\ M_n\ \text{end}} \qquad \textsc{Program}\ \frac{P \vdash C_1 \quad \cdots \quad P \vdash C_n \qquad P; \cdot \vdash E : T}{\vdash C_1 \cdots C_n\ E}$$

Figure 6: Static Type Rules

Part 4: Type Checking

Finally, you must write a function `tc_prog p : prog -> unit` that returns successfully if and only if `p` type checks, and otherwise it raises an exception. Put this function in `rubet.ml`. Figure 6 gives the static type checking rules that you will translate into OCaml. Here A is a *type environment*, which as discussed in class is an associative list mapping local variables to their types. Most of the rules have the form $P; A \vdash E : T$, meaning that in program P with environment A, expression E has type T. We'll explain the other kinds of rules as we encounter them. We've labeled the rules so we can refer to them in the discussion:

- The rules Int, Nil, and Str all say that an integer, nil, or string have the obvious types.
- The *local variables* of a method include the parameters of the current method, locals defined at the top of the method, and `self`, which refers to the object whose method is being invoked. The rules Self and Id say that `self` or the identifier *id* has whatever type is assigned to it in the environment A. If `self` or *id* is not bound in the environment, then this rule doesn't apply—and hence your type checker would signal an error.
- The rule Field-R says that when a field is accessed, it has whatever type we get by looking it up in the program according to the `lookup_field` function you already wrote, finding the current class by looking up the type of `self`. Notice that unlike in untyped Rube, it is an error to refer to fields that have not been pre-defined. Also notice that like Ruby, only fields of `self` are accessible, and it is impossible to access a field of another object.

- The rule IF says that to type a conditional, the three sub-expressions must all be well-typed, and the type of the whole `if` is the least-common ancestor of the types of the then and else branches.
- The rule WHILE says that to type a while look, the two sub-expressions must be while-typed, and the type of the whole `While` is `Bot`.
- The rule SEQ says that the type of $E_1;\ E_2$ is the type of E_2. Notice that this rule requires E_1 to be well typed, but it doesn't matter what that type is.
- The rule ID-W says that a write to a local variable is well-typed if the type of the right-hand side of the assignment is a subtype of the variable type. Notice that unlike untyped Rube, it is an error to write to a variable that hasn't been defined as either a parameter or local. Notice also that it is an error to write to the local variable `self` (which is implicitly syntactically distinct from the non-terminal *id*), and your implementation should signal an error in this case.
- Similarly, the rule FIELD-W says that a field write is well-typed if the type of the right-hand side is a subtype of the field type. Again, unlike untyped Rube, fields must be defined with types prior to writing to them.
- The rule NEW says that a `new` expression is well-typed if the class being constructed exists in the program according to the `defined_class` function you wrote earlier. Notice that, as in Java, the class can appear *anywhere* in the program—it could be listed before the current class definition or after.
- The rule CAST says that an expression can be type cast if it is well typed and its type is castable to the type of the cast (using the `castable` function you wrote earlier). The resulting type of the cast is the cast-to type.
- The rule INVOKE says that for a method invocation to be well-typed, the receiver object expression and all arguments must be well-typed. Also, the types of the arguments must be subtypes of the method type we find by looking up id_m in whatever class corresponds to the receiver object, using the `lookup_meth` function you wrote earlier.

  Notice that we don't need to do anything else here, e.g., we don't need to look inside the method body. That's because we'll separately check that the method actually has the type the `lookup_meth` function says it has. Neat!
- The rule METHOD says that for a method definition to be well typed inside of class id_c, it must be that if we typecheck the method body with locals assigned their types as given, parameters assigned their types as given, and `self` given type id_c, the body of the method has a type T' that is a subtype of the declared method return type. Note the order here says that locals may shadow parameters, which may shadow `self`; however, we will *not* test whether you implement this shadowing.
- The rule CLASS says that for a class definition to be well typed, all its method definitions must be well-typed, and the class's type annotations must be consistent with its superclasses, according to the `check_class` function you wrote above. There must also be no definitions of `Object`, `Integer`, `String`, or `Bot`.
- Finally, rule PROGRAM says that for a program to be well-typed, all of its classes must be well-typed, and its top-level expression must be well-typed under the empty environment. Notice this means the top-level expression cannot even refer to `self`!

When you're writing `tc_prog`, you'll probably want to write several other functions, including

```
val tc_expr : prog -> environment -> expr -> typ
```

that type checks an expression. It's up to you to choose the type for `environment`, but something simple like `(string * typ) list` should work just fine. (Functions like `List.assoc` will be handy in working with this type.)

Your type checking functions can raise any exception to signal that a program is ill-typed. We've supplied the exception `Type_error` as a convenient exception to throw, but you're not required to use it. In `main.ml`, there's code that calls your `tc_prog` function and either prints `yes` or `no`, depending on whether `tc_prog` type checked the input or found a type error (i.e., raised an exception), respectively.

Part 5: Design

In the last part of the project, you must extend the type system to include *optional* and *named* arguments. Optional arguments should have a default value that is assumed if no argument is given. Named arguments should be able to be passed out of order.

The exact design of this feature is up to you. You should probably look at how OCaml handles optional and named arguments, and how Ruby handles optional arguments. (Ruby does not have named arguments—Ruby programs use hashes for the same purpose.)

To facilitate grading, we've added two components to the `meth_args` field of type `meth`: The `expr option`, if present, indicates an optional argument initialized by default to the given expression, and the `string option`, if present, indicates a named argument with the given name.

You should consider the following issues/questions:

- What is the syntax for denoting optional and named arugments?
- Type checking will need to ensure the default value for an optional argument has the right type.
- Can optional or named arguments appear anywhere in a parameter list, or are there restrictions?
- If there is a method with some unnamed and some named arguments, can the named arguments be put anywhere in a call, or are there restrictions?
- What happens if there are several named arguments with the same names?
- Is there such a thing as a named optional argument?

Implement your extension to support optional and named, and also include a writeup `part5.txt` in your submission describing your implementation and any design issues that arose in building it. **Be sure to include examples** illustrating how your language extension works.

Academic Integrity

The Campus Senate has adopted a policy asking students to include the following statement on each assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently your program is requested to contain this pledge in a comment near the top.

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus—please review it at this time.