Tofu: A simple object-oriented programming language

Jake Elliott Callaghan
Oriel College
University of Oxford

Supervisor:
Dr Michael Spivey

A report submitted for the degree of
Computer Science
Trinity 2016

1. Introduction

The aim of this project is to design and implement the simplest possible object-oriented language, named Tofu, on the Keiko [1] virtual machine without compromising the expressivity of the language.

Tofu is a statically typed, class based programming language. This means the type checking system we employ is based on the semantic analysis of the source code at compile time, and that programs written in the language will be based around instantiations of class definitions.

The compiler produces object code that targets Keiko - essentially an idealized low-level stack-based abstract machine, whereby interaction with registers is abstracted away, allowing the focus of the project to be on the semantic analysis and implementation of Tofu, rather than machine code.

The compiler is written entirely in the OCaml programming language [2] . This gives clarity as to Tofu's semantics, through its provision of algebraic data types, as well as some objective features that assist the semantic analysis phases. Throughout the compilation process we will perform operations over abstract-syntax trees *(ASTs)* [3, 4] that represent the input program.

The project explores the implementation of standard object-oriented programming (OOP) paradigms such as information hiding (encapsulation), single-inheritance class hierarchies, dynamic dispatch, polymorphism (through sub-typing), code re-use, virtual methods and open recursion. Tofu is purely object-oriented, meaning that everything is an Object of some type and that there are no primitive values – so all numeric and boolean values are boxed as objects.

2. An Overview of Tofu

To help understand what we are going to implement, let’s consider the following Tofu code,

```
class Student {
  var iq : Integer;
  def init() : Unit = { iq = 0; }
  def learn(x : Integer) = { iq = iq.add(x); }
  def speak() : Unit = {}
}

class Mathematician extends Student {
  var number : Integer;
  def setNumber(x : Integer) = { number = x; }
  def init() : Unit = { iq = 130; number = 0; }
  def speak() : Unit = { iq.add(number).print(); }
}

class ComputerScientist extends Mathematician {
  var coding : Boolean;
  def coding() : Boolean = { return coding; }
  def init() : Unit = { iq = 256; number = 1024; coding = true; }
  def speak() : Unit = {
              var x : Integer = number;
              while (coding) {
                  0.print(); 1.print(); 0.print(); 0.print();
                  if (x.isGreaterThan(1)) { x = x.divide(2); }
                  else                    { coding = false;  }
               };
               coding = true;             (*‘reset’ the coding field*)
}
```

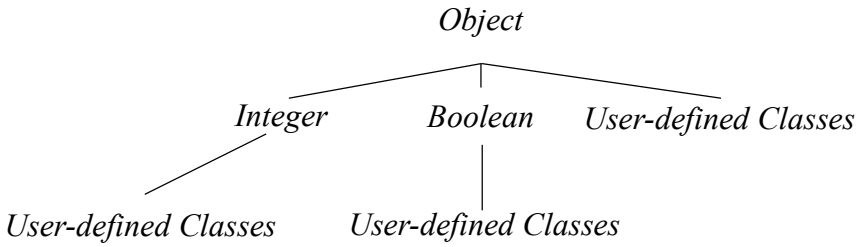
(Figure 1. The Student tofu type)

Figure 1 defines a class *Student* with field *iq* of type *Integer*, and methods *init()*, *learn(x:Integer) and speak()*. A *Mathematician* is a subclass of *Student*, introducing a field and method to represent her favourite number, and *ComputerScientist* a subclass of *Mathematician,* adding another field and method to determine if the student is currently coding or not.

In standard OOP fashion, this defines a class hierarchy where we consider a *ComputerScientist* a subtype of *Mathematician,* and a *Mathematician* a subtype of *Student.* We take this to mean that an object of type *Mathematician* inherits the field *iq,* and methods *init(), learn() and speak()* from *Student*. The class definition may do as it pleases to the definitions of these inherited features, but not their types (the possibility of manipulating inherited features' types is explored in chapter *7)*. The *Mathematician* class exerts its ability to override the methods *init()* and *speak()*, noting that the methods' signatures are the same as they appear in *Student.* Similarly, a *ComputerScientist* acquires access to all features from the *Mathematician* class (and subsequently the *Student* class*)*.

We define types in the language to be the names of all defined classes. As Tofu is *pure* and without primitive values, expressions such as *'256'* and *'false'* in figure *1* are of types that correspond to Tofu classes. That is to say these (apparently constant) values are actually wrapped in objects of type *Integer* and *Boolean,* respectively. This implementation detail is discussed later on.

For simplicity we make all class definitions, and consequently all types, of global scope. The special type keyword Unit can be used in a method's signature to declare that it does not return a value, and is thus called only for its side effects. A class that does not explicitly extend something will automatically subtype the *Object* Tofu class (which we define later). This means that any Tofu program will have a class structure with *Object* at its root:



(Figure 1.1 Tofu class hierarchy)

Note, that for expressions such as *iq.add(number)* and *x.isGreaterThan(1)*, we could introduce syntactic sugar such as "*iq + number"* and "*x > 1"* - but we refrain from doing so for clarity's sake.

The entry point for a Tofu program is a sequence of statements enclosed in a *main* declaration,

```
main {
     var ada : Student;
     var alan : Student;

     ada = new ComputerScientist;
     alan = new Mathematician;

     ada.speak();
```

```
    alan.speak();
}
```

(Figure 1.2 Tofu main deceleration)

The first two statements in figure *1.2* declare variables both to be of static type *Student.* We then proceed to assign these variables to newly instantiated *ComputerScientist* and *Mathematician* objects, respectively. It is a convention in Tofu that if a class defines a method named *init* with signature *init() : Unit,* this method is called immediately upon instantiation – providing simple initial assignments to class fields. Thus, both of our fresh objects in figure *1.2* will have assigned values to their respective fields.

We are to implement dynamic-dispatch, so the invocations of the method *speak()* in the final two lines of figure *1.2* will depend on the dynamic types of *ada* and *alan.* In effect,

- *ada.speak()* prints to stdout: *0 1 0 0 0 1 0 0 0 1 0 0 ... 0 1 0 0*
- *alan.speak()* prints to stdout: 130

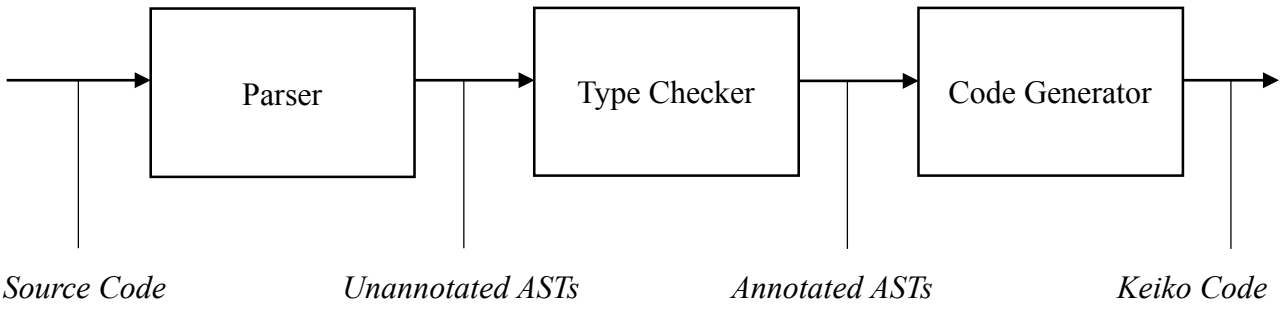
Note, ada's output is actually line separated (*I.e.* "0\n" "1\n" …).

There are several points to be made about Tofu, from the examples above,

I. Comments can be made within the tokens "(*" and "*)".
II. The main declaration is defined alongside the class declarations, and is not considered to be a method in the same way that a class defines one. It is simply a sequence of statements.
III. Because the constants *true* and *false,* and *n* (where *n* is some integer) are actually boxed as objects of the corresponding classes, we can invoke methods upon them directly that are defined by their respective classes. *E.g. 0.print()* calls the *print()* method defined by the *Integer* class on an *Integer* object with value 0.
IV. We support open recursion. This means that calls from one method can invoke another method of the same object, equivalent to *"this.method()"* in the Java programming language [8]. Due to late-binding, a method may call another that is in fact defined later in some subclass.

Point III and IV's implementations are explained in depth, later on.

3. Design



(Figure 2. A high-level overview of Tofu-c)

The Tofu compiler, Tofu-c, will perform three main tasks (as shown above in *Figure 2),* performing one or more passes over the input program's abstract syntax tree:

1. Syntactic analysis of raw textual input, producing unannotated abstract-syntax trees.
2. Semantic analysis of the abstract-syntax trees, annotating them with type information.
3. Conversion from the intermediate program representation (ASTs) to a sequence of machine instructions for Keiko.

The global scope of our types means that mention of them may occur anywhere within a class' declaration, with some restriction placed on class extension; namely, that extension is only permitted of a class that is declared before the sub class' declaration. This prevents cycles in the inheritance hierarchy and simplifies our passes over the AST. Open recursion means that a method's body may refer to features belonging to the same class. This, in turn, means that we will need to maintain a record of all class declarations prior to examining method bodies, fields of these classes, and the main declaration. Thus, for an elegant and simple implementation, we require multi-passes over the program's AST: first, to define the types and their relationships with respect to inheritance. Second, to define the methods that belong to these types. Third, to inspect the statements and expressions that exist in these methods' definitions, as well as the sequence of statements in one's *main*. And, finally, to generate the code.

3.1 Parsing

The first phase is performed in two stages: first, the tokenization of keywords (lexing), and second, the groupings of these tokens into nodes that will form the ASTs (syntactic analysis). These tasks are routine, and are handled by OCamllex and OCamlyacc [6] respectively. These packages require that we provide constructs to represent the syntax of our language and that one specifies how these relate in a syntactic sense. To express the syntax of Tofu, we define the following types depicted in *Figure 2.1,*

```
type expr =
  | This
  | Number of int
  | Boolean of bool
  | Variable of variable_desc
  | NewObject of ctype
  | Call of expr_desc * string * expr_desc list

and stmt =
    Skip
  | Seq of stmt list
  | UnitCall of expr_desc * string * expr_desc list
  | LocalVarDecl of variable_desc * ctype
  | AssignStmt of variable_desc * expr_desc
  | ReturnStmt of expr_desc
  | IfStmt of expr_desc * stmt * stmt
  | WhileStmt of expr_desc * stmt
  | Newline
```

(Figure 2.1 Tofu expressions and statements)

These constructs will appear throughout the bodies of methods and within the program's main method (discussed later), representing expressible values and language statements.

E.g. A method call such as *ObjectA.method(ObjectB, ObjectC)* will be parsed as node

Call(exprDescA,"method",[exprDescB, exprDescC]), where exprDescX is an expression descriptor that is to be annotated throughout compilation.Such descriptors are the topic of the proceeding chapter.

3.2 Type Checking

The second phase annotates the ASTs with type information. As Tofu is a statically-typed language, all variables must have a valid declared type; thus allowing this part of the compilation to ensure type safety.
In accordance with the OOP style, the following typing paradigms are used in Tofu:

- ***Liskov's Substitution principle*** [5]
  If ***S*** is a subtype of ***T***, then objects of type ***T*** may be replaced with objects of type ***S*** (safely).

- ***Dynamic Binding*** [4, p. 279]
  An object's dynamic type at run-time determines which method definition is called, as opposed to its static type.

- ***Single Inheritance*** [6]
  A class ***S*** can subtype another (single) class ***T***, defining it to be a subtype of ***T***, and inherit all methods and variables from ***T***.

- ***Virtual Methods*** [4, p. 279]
  All methods are inheritable from a parent class and can be over ridden by subclasses.

To help implement these ideas, we introduce descriptor types that are instantiated during the parsing phase, and annotated during type-checking. These will carry all relevant information for our classes, methods, variables and expressions. The type-checking process can be summarized as

1. Gather all defined types and determine the class hierarchy.
2. Check the features of these classes against the defined types from (*1).* Where a feature is either a method or field, belonging to a class, we check method signatures (method name, return type and argument types) and class fields' static types.
3. Check the method bodies conform to our defined types from (*1)* and *(2),* as well as the statements in the *main* declaration.

We define what it means to *check* a class, feature, or expression in chapter *4.*
Let's look at the types of these descriptors, starting at the innermost level of the process: expressions. We define an expression's descriptor,

```
type expr_desc = {
    expr_guts : expr;
    mutable expr_type : ctype option;
}
```

The type *expr_desc* contains two fields: the first (*expr_guts)* contains the expression it intends to describe (of type *expr* in *Figure 2.1),* and the second (*expr_type*) is the annotated type we give to the expressed value. *E.g.* The expression *Call(edescX, foo, edescs)* is annotated with type *Y,* given that method *foo* returns an object of type *Y*. The *expr_desc* would be { *expr_guts = Call(edescX, foo, edescs) , expr_type = Some Y }*.

Note, OCaml's type *'a option* represents values that may not have been assigned value of type *'a*, defined

```
type 'a option =
              | None
              | Some 'a
```

The type *variable_desc* provides a descriptor for any references to a variable, defined

```
type variable_desc = {
    variable_name : string;
    mutable variable_type : ctype option;
    mutable variable_kind : var_kind option;
    mutable offset : int;
}
```

The field *variable_type* represents the annotated type of the variable, and we define *variable_kind* below. Note that we will return to *offset* later, as this is concerned with the code generation phase in chapter *3.3*.

```
type var_kind =
  | Field
  | Local
  | Arg
  | Global
```

var_kind allows an annotation to express what kind of variable is being referenced. *Field* corresponds to a class variable, *Local* corresponds to a variable declared within a method body, *Arg* corresponds to a variable that was passed to a method as a parameter, and *Global* is used for variables declared within the main method.

Every declared class will have a parent class, and some methods and fields (both defined by the class and inherited from the parent). These fields are just variables with *var_kind = Some Field*, and each method is itself wrapped in an appropriate descriptor. We can make a descriptor

```
type class_desc =
  { class_name : string;
    parent_name : string;
    mutable parent_desc : class_desc option;
    mutable variables : variable_desc list;
    mutable method_table : method_desc list;
  }
```

The fields *class_name* and *parent_name* are both immediately available from the syntax, as a class either declares that it *extends* another (subclassing), or implicitly inherits from the *Object* Tofu class (discussed later). During analysis, we will can update the list of variable descriptors we see that are appropriate to this class, and similarly for *method_table*.

The descriptor for a method contains information about its arguments, its locally defined variables, its return type and its defining class,

```
type method_desc =
  { method_name : string;
    mutable defining_class : class_desc option;
    return_type : ctype;
    number_of_formals : int;
    formals : formal list
    body : stmt;
    mutable code : Keiko.code;
    mutable vtable_index : int;
    mutable locals : variable_desc list;
```

```
  }

and formal = Formal of string * ctype
```

A formal parameter can be described by its name and its static type. During the semantic analysis, we will set *defining_class* to be the *class_desc* of the class that actually implements the code found in *body*. It will often be the case that a class inherits a method, and thus its *method_desc*, which by definition is the exact same procedure; so we need only compile code for this implementation of the method once. *vtable_index* will play an important role in this idea, as is later explained in the next chapter.

Finally, we wrap the entire program in a type,

```
type program = Program of main_method_desc * class_decl list

and feature_decl =
    InstanceVarDecl of variable_desc * ctype
  | MethDecl of method_desc

and class_decl = ClassDecl of class_desc * feature_decl list

and main_method_desc = {
  mdesc : method_desc;
  mutable decls : Keiko.code
}
```

feature_decl helps to categorize the two types of declaration that may occur within a class: instance variables and methods. We treat the entry point of the Tofu programs as a special method ‘main’, and define a Tofu program to be a pair consisting of a descriptor for main, and a list of *class_decl*’s.

We define the types available in a program to be the set of globally defined classes. We have stated that all classes are of global scope. Thus, we can take *ctype* to be a class' name:

```
type ctype = string
```

The module Typecheck.ml is thus required to define a function (whose definition it’s the subject of chapter *4)* that annotates a program, with type

```
val annotate : program -> unit
```

3.3 Code Generation

The third and final phase of Tofu-c is required to produce Keiko instructions from annotated trees. A module Kgen.ml will implement a function *translate* that generates code and stores this in the relevant descriptors as it traverses the trees, with type

```
val translate : Tree.program -> unit
```

To implement our notions of inheritence and polymorphism on Keiko, we must consider the layout of an object in memory.

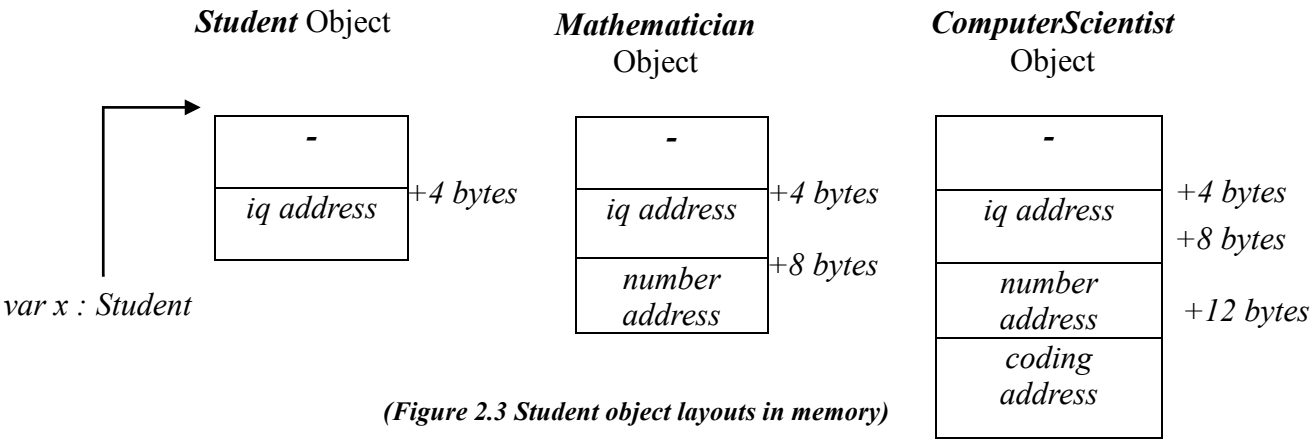
3.3.1 Memory Layout

To implement ***Single Inheritance***, we can ensure that all objects in the same class hierarchy share a

common layout in memory amongst their fields. This leads us to the technique of *prefixing* [4], defined as:

*"if **S** extends **T**, the fields of **S** that are inherited from **T** are to be laid out at the beginning of its record, in the exact same order that they appear in **T**'s records"*

Prefixing is advantageous as it allows easy substitution of subtypes (***Liskov Substitution***) with respect to code generation. Let us consider the memory layout of some objects of types *Student, Mathematician* and *ComputerScientist*.



(Figure 2.3 Student object layouts in memory)

Figure 2.3 raises several points that are worth noting:

- Keiko uses 32-bit (4 bytes) words for addresses
- Variables are references to objects, i.e. addresses.
- Contiguous memory is dynamically allocated for objects.
- The first 4 bytes of any object are reserved (see 2.3.2), and fields begin at offsets of +4 bytes from the object's address.

We can see that all objects in a common class hierarchy have the same layout, with sub-classing types appending their additional fields. The key point here is that we can use the same Keiko code to, say, load the value of the *iq* field, regardless of the dynamic type of a *Student* object. *E.g.* for an object *var x : Student,* we can load *x.iq* as follows:

```
'push address of x' / CONST 4 / BINOP PlusA / LOADW
```

Where,

- *CONST N* is an instruction to push the value N onto the stack
- *BINOP op* is a binary operation that pops the top two values x0,x1 and performs x0 `op` x1.
- *PlusA* denotes address arithmetic.
- *LOADW* pushes a word stored at the address on the stack (popping the address).

This will correctly load the address of the field *iq,* regardless of whether x is in fact a *Mathematician* or *ComputerScientist* at run-time.

If we require, say, the field *code* from a *ComputerScientist,* then the instructions would just change the value added to the variable's address, namely,

```
'push address of x' / CONST 12 / BINOP PlusA / LOADW
```

Which of course is only a valid sequence of instructions for objects of type *ComputerScientist.*

This value that represents the offset within the object's memory layout is what is stored in a *variable_desc*'s *offset* field, described in the previous section. *E.g.* The descriptor for a reference to the *code* field will be annotated *{ variable_name = "code"; variable_type = "Boolean"; kind = Some Field; offset = 12 }.*

3.3.2 Virtual Method Tables

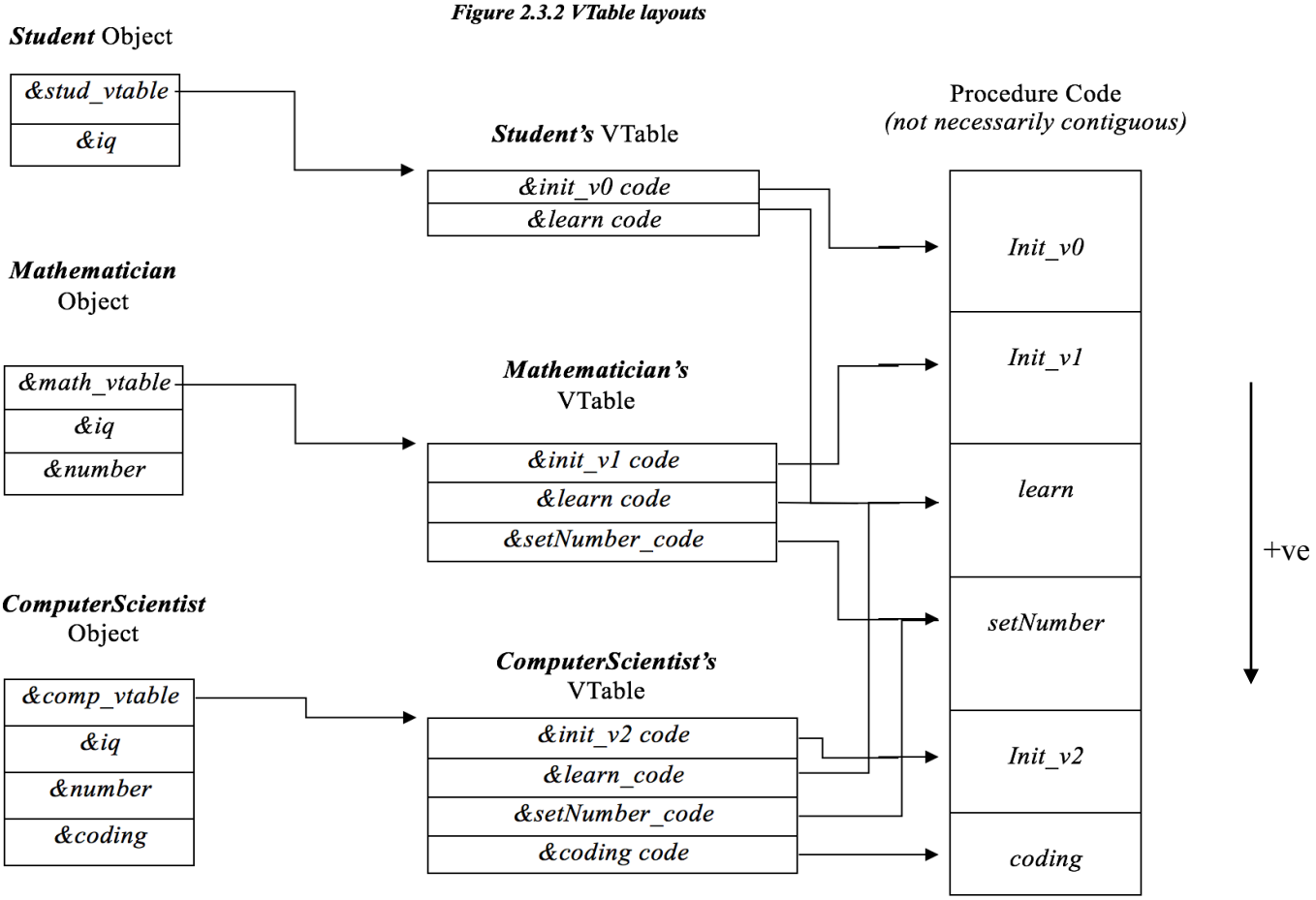
To support run-time binding of methods ***(dynamic-dispatch)*** we must provide a way for an object to lookup its methods' code. We also want to exploit the fact that inherited methods will use the same code throughout the class-hierarchies. This is where *virtual method tables* [4, p. 279] come in.

We will arrange that the first 4 bytes of each object's allocated memory will be a pointer to a table of pointers, which point to the respective methods that the particular object has access to. This suggests that we can use *prefixing* again, but with method addresses instead of field addresses. They will work in the same way, namely that all methods inherited from some class will appear at the same indexes in the virtual table for both the parent and child class; with the child class appending its own new methods. Note, any overridden methods (i.e. methods that are inherited with same method signature, and then redefined by the child class) will still appear at the same indexes in the table, but will point to different addresses.

Let's consider the method tables for types in the *Student* program, alongside their memory layouts from *Figure 2.3.* Let *&x* denote the address of x and each cell be 4 bytes apart,

(*The method speak() and any inherited methods from Object are omitted to give clarity)*

Figure 2.3.2 VTable layouts



The procedure code is efficient with regard to inherited methods, as methods such as *learn* and *setNumber* are used by multiple classes but only defined once. The index of a method in a class' virtual table corresponds to the *vtable_index* field in the aforementioned *method_desc* type.

Using the *vtable_index* field, pushing the address for a method with descriptor *mdesc* becomes:

```
"push address of variable" / LOADW
CONST ( 4 * mdesc.vtable_index ) / BINOP PlusA / LOADW
```

Note, Tofu does not support static methods - i.e. methods that are relative to all objects of a particular class, as opposed to instances of the class. This prospect is discussed in chapter *7*.

Similar to field lookup, the prefixing supports simple method lookup. For example,

```
var alan : Mathematician; alan.setNumber(256);
```

The method call to set Alan's favourite number will produce the following Keiko instructions,

```
(1)     "push address of a new integer with value 256"
(2)     LOCAL 16 / LOADW / DUP
(3)     LOADW / CONST 8 / BINOP PlusA / LOADW
(4)     CONST 0 / PCALL 2
```

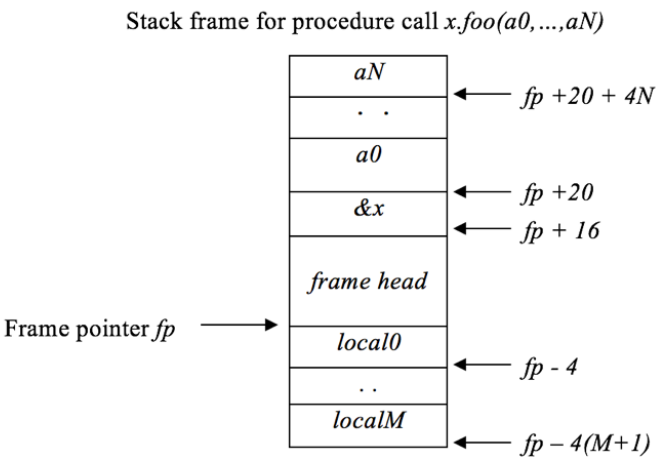
There are several points to made here,

- As Tofu is a *pure* language (there are no primitive values), we treat the argument 256 as var arg : Integer; arg = 256; arg. *I.e.* we wrap the number in an Integer object. This is of much debate in chapter *5*.

- The Keiko instructions introduced here do as follows,
  - *LOCAL n* = push the address that is +n bytes from the frame-pointer
  - *DUP* = duplicate the element on top of the stack and push this duplicate.
  - *PCALL n* = call the procedure whose address is atop the stack, with n arguments, and do not expect a word to be returned.
  - *PCALLW n* = same as *PCALL n* except a word is returned from the procedure.

- Line *2* pushes the address of object *alan* onto the stack twice (by duplication). We want to use this address to lookup the address of the procedure via the virtual table for the object ***(dynamic-dispatch)***, and then again as the first argument passed to the procedure. This is the standard calling convention we will use: that any method will have the address of the calling object as its first argument and that all arguments are passed on the stack. As shown below,

(Figure 2.3.3 procedure stack frame)



- Line *3* pushes the address of *Mathematician's* virtual table, pushes the offset of method *setNumber,* pushes the calculated address of the table entry that contains *&setNumber*, and finally loads *&setNumber*.

- *CONST 0* in line *4* pushes a static link for the procedure – supported by Keiko but not used in our language (due to the absence of nested procedures). It then calls the method with the 2 words below the static link as arguments passed to *setNumber*.

- Arguments are passed by *value* [3, pp. 424-429]. All objects are represented by values that are pointers to their (dynamically) allocated blocks of memory. This allows clean access to the features of an object by simply pushing its pointer, stored in the activation record for a procedure, onto the stack.

4. Implementation

We will now consider how to implement the ideas described in chapter *3,* through a collection of OCaml modules that handle (or provide support to) each of the three main phases in figure *2.* At the highest level, we will consider their correspondence to the following three functions:

```
1. val program : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Tree.program
2. val annotate : program -> unit
3. val translate : program -> unit
```

The function *program (1)*, despite its slightly more complex type, actually just produces an abstract syntax tree according to our design in section *3.1* - consisting of unannotated descriptors for every class, feature, and expression described in the source text, as well as the statements apparent in the *main* declaration. As the focus of Tofu's implementation concerns functions *(2)* and (*3)*, more so, we will omit detail as to how OCamlyacc and OCamllex [7] are implemented. Instead, there is excerpt of the grammar used (specified in Backus–Naur Form) to represent the syntax of Tofu, in appendix *A*.

annotate (2) implements our type-checking. It traverses the abstract syntax tree, through each class declaration and recursively into all of its features, filling in the missing type information in the descriptors from section *3.2.* We exploit the objective features of the OCaml language to create an e*nvironment*, described in detail in section *4.1,* from which we allow these descriptors (and thus their annotations) to be accessed at any point during a pass over the program's AST.

translate (3) operates in a very similar fashion to that of the *annotate* function, in that it traverses the tree recursively over the *Tree* algebraic data type representing the Tofu program. The key difference here is that we convert expressions and statements into Keiko object code, using the annotations from *(2)* to guide us.

4.1 Environments

As mentioned earlier, we would like access to class descriptions of type *class_desc* that correspond to the types our program uses. This leads us to the definiton of *Env,* a module of the following interface:

- `type environment`
- `val add_class : class_desc -> feature_decl list -> unit`

- `val find_class : ctype -> class_desc`
- `val is_subclass : ctype -> ctype -> bool`
- `val add_method : class_desc -> method_desc -> unit`
- `val find_method : class_desc -> string -> method_desc`
- `val add_instance_var : class_desc -> variable_desc -> unit`
- `val find_instance_var : class_desc -> string -> variable_desc`

Env provides a means of interaction with any of the class descriptors present during compilation. As a *class_desc* keeps record of all relevant information to a class, *Env* provides efficient addition and lookup of the features belonging to a class. We define,

```
type environment = (ctype, class_desc) Hashtbl.t
```

This defines an environment to be a hash-table* that maps a *ctype* (which are just class names) to their corresponding class descriptors. The module defines a (static) internal value, *env*, that is of type *environment,* upon which the interface's operations are performed (in effect implementing the state of this implicit data-structure),

```
Let env = Hashtbl.create 50
```

Definition for the methods *Env* provides are just operations on the Hash-table *env* or a *class_desc,* outlined below in a simplified pseudocode:

```
1. add_class cdesc =
         env.add cdesc.class_name cdesc;
         cdesc.parent_class <- Some (find_class cdesc.parent_name)
         Inherit features from the parent descriptor

2. find_class cname = env.find cname

3. is_subclass cname1 cname2 =
         Let cdesc1 = find_class cname1 in
               If cdesc1.class_name = cname2 then true else
               If cdesc1.class_name = "Object" then false
               Else is_subclass (cname1's parent's name) cname2

4. add_method cdesc mdesc =
         insert mdesc into cdesc.method_table
         Set mdesc.defining_class as 'appropriate'

5. find_method cdesc mname =

         find mdesc with name mname in cdesc.method_table
6. add_instance_var cdesc vdesc =
         Insert vdesc into cdesc.variables

7. find_instance_var cdesc vname =
         Find vdesc with name vname in cdesc.variables
```

is_subclass (3) works by searching up the program's class hierarchy. To determine if type *S* is a subtype of *T*, we check if *S* = *T* (as any type is a subtype of itself), if this fails then we consider *S's* parent type *P.* If *P* is a subtype of *T* then by transitivity we have that *S* subtypes *T.* We do this recursively until *P* = *Object*, *i.e.* we reach the top of the tree, if *T* is not *Object* then we return false as *S* does not subtype *T.*

add_method (4) inserts a class' *method_table,* ensuring that if this is a method being over-ridden

* http://caml.inria.fr/pub/docs/manual-ocaml/libref/Hashtbl.html

then it is placed at the same index as the original - otherwise it is simply appended. We note that all inherited methods will already be in the *method_table,* when *(4)* is called, due to *add_class'*s handling of the inheritance prior to any calls to *add_method* taking place.

4.2 Implementing Type Checking

Type checking involves many interactions with the *Env* module. To check that a program obeys our notion of ***Liskov Substitution,*** we will often ask if an expression's type is a subclass of a variable or arguments static type. Here is an outline of the definition of method *annotate : program -> unit*,

```
Let annotate prog =

   1. "Add library class descriptors to Env"
   2. List.iter add_class class_descs
   3. List.iter check_fields class_descs
   4. List.iter check_methods class_descs
   5. Check_main main_method_desc
```

For now, we will just accept that line *(1)* adds the required *class_desc*'s to allow for comprehensive type checking - where the details of how this occurs are the subject of the next sub-section (*4.2.2).* We note that the OCaml function *List.Iter :* ('a -> Unit) -> 'a list -> Unit is defined as,

```
Let rec Iter f xs = match xs with
      | [] -> ()
      | (x::xs) -> f x; iter f xs;
```

In effect, this sequentially applies a function f on the elements of a list – each time returning a value of the OCaml type Unit, namely *'()'* .

Line by line, *annotate* does as follows:

- Line *2* adds each declared class to the environment. This is handled by the *Env* module.

- Line *3* checks that the static type of each instance variable (for each class) is defined within *Env*,
  and that there are no clashes with variable names.

- Line *4* checks the methods in each class, *check_methods* does a follows:
  - Ensures there are no name clashes amongst method names in the class' *method_table*

  - For each formal parameter in each respective method signature, check that there are no name clashes and that each has a well-defined static type.

  - Checks the return types are all well-defined types.

  - Checks the method bodies by calling *check_stmt.*

  - Checks the absence (for *Unit* return types) or presence of a return statement.

- Line *5* checks the sequence of statements in *main* declaration, by calling *check_stmt*.

The function *check_stmt*, alongside its mutually recursive friend *check_expr*, performs the legwork of type checking all *Tree.stmt* and *Tree.expr* nodes (respectively) in the AST. We briefly outline the checks performed on each type of *Tree.stmt* and *Tree.expr,* to avoid overloading the reader with OCaml code, by functions -

check_stmt : method_desc -> Tree.stmt -> Unit

check_expr : method_desc -> expr_desc -> Unit

Note, the passed *method_desc* to these functions corresponds to the method where the expression/statement was found.

Recall, *type expr_desc = { expr_guts : expr , expr_type : Some ctype }.*

```
Let rec check_expr mdesc edesc = match edesc.guts with

|This -> edesc.expr_type <- the defining class of this method

|Number n -> edesc.expr_type <- Some "Integer"

|Boolean b -> edesc.expr_type <- Some "Boolean"

|Variable vdesc ->
      Local var in mdesc =>
            Vdesc.variable_kind <- Some Local
            Vdesc.variable_type <- "copy type from mdesc.locals"
            Vdesc.offset <- -4 * (index of vdesc in mdesc.locals + 1)

      Parameter in mdesc =>
            Vdesc.variable_kind <- Some Arg
            Vdesc.variable_type <- "copy from mdesc.formals"
            Vdesc.offset <- 20 + (4 * index of this var in mdesc.formals)

      Class field in mdesc's defining class =>
            Vdesc.variable_kind <- Some Field
            Vdesc.variable_type <- "copy from mdescs.defining_class.variables"
            Vdesc.offset <- 4 * (index in mdesc.defining_class.variables + 1)
      _ => variableNameError ...

|NewObject cname -> edesc.expr_type <- Some ((find_class cname).class_name)

|Call (edesc2,mname,arg_edescs) ->
      * check type of the object whose method is being called *
      check_expr mdesc edesc2;
      Let etype = edesc2.expr_type in

      * get the corresponding method descriptor *
      Let mdesc2 = find_method etype mname in ...

      "check the correct amount of arguments are present"

      "check that each argument's expression is a subtype of the corresponding
formal parameter's static type"
```

And for checking statements,

```
and check_stmt mdesc s = match s with

|Seq ss -> List.iter (check_stmt mdesc) ss

|UnitCall (edesc,mname,arg_edescs) ->
      "same as in check_expr call..."

|LocalVarDecl (vdesc,vtype) ->
      "check that this variable name isn't locally declared already"
      "check that this variable name doesn't clash with the mdesc's arguments"
      Vdesc.variable_type <- Some (find_class vtype)
      List.append mdesc.locals [vdesc]
```

```
|AssignStmt (vdesc,edesc) ->
      "check vdesc corresponds to a defined and assignable variable, i.e. a
       local variable or a class field in mdesc.defining_class "

      "annotate this variable reference using the details from its declaration
as a local variable or a class field "

      Check_expr mdesc edesc;
      Let etype = edesc.expr_type and let vtype = "type of LHS var" in
      If not (is_subclass etype vtype) then semanticError "..."

|ReturnStmt edesc ->
      Check_expr mdesc edesc;
      Let etype = edesc.expr_type in
      If not is_subclass etype mdesc.return type then semanticError "..."

|IfStmt (edesc, thenstmts, elsestmts) ->
      Check_expr mdesc edesc;
      Let etype = edesc.expr_type...
      If not is_subclass etype "Boolean" then semanticError "..."
      Else check_stmt mdesc thenstmts;
           Check_stmt mdesc elsestmts;

|WhileStmt (edesc,body) ->
      Check_expr mdesc edesc;
      Let etype = edesc.expr_type...
      If not is_subclass etype "Boolean" then semanticError "..."
      Else check_stmt mdesc body

|Newline -> ()
```

In order for the type-checking to work, we require that the *Env* module has knowledge of the Tofu library classes. Namely: *Object, Integer,* and *Boolean.* We do so by defining *class_desc* objects for each of them, all with appropriately annotated fields, that are linked into *Env* during *annotate*.

Here is an example of an assignment being checked, during *annotate*, taken from figure *1.2*:

"ada = new ComputerScientist". Where *ada* has a *variable_desc ada_vdesc* and the right-hand side has *expr_desc edesc = {expr_guts : NewObject "ComputerScientist", expr_type : None }*. Note, that the variable *ada* was declared to be of static type *Student*.

```
Check_stmt mdesc (AssignStmt (ada_vdesc, edesc)) =>
```

- *ada_vdesc* is checked to contain a well-defined variable, and annotated accordingly.
- *edesc* is checked and annotated by calling *check_expr mdesc edesc*. This results in the annotation: *edesc.expr_type* <- *Some "ComputerScientist"*.
- We then check to see if *etype = "ComputerScientist"* is a subclass of *vtype = "Student"*, which it is.

4.3 Implementing Code Generation

Code generation concerns the implementation of *translate : program -> unit*. Here is an outline of its definition:

```
Let translate prog =
```

- *"link library methods' code to their descriptors"*

- *List.iter gen_class cdescs*
- *gen_main_method main_mdesc*

The first task of *translate* is to link the code for the methods in each Tofu library class to each of their respective *class_desc*'s. The Keiko code for each of these is specified in separate files (*Object_methods.ml , ... etc)*. This allows their implementations to vary whilst still supporting the interface defined by their *class_desc*'s.

The second task is to generate code for each of the class descriptors in the program. This means generating code for each procedure defined by a class, and storing this object code in the method's descriptor – allowing us to dress procedures with the appropriate assembler directives at a later stage.

Finally, we generate the code for the *main*. This is to be treated differently than the methods defined in a class declaration; the main difference being that references to local variables will not be handled as if *main* were a method with local variables stored in an activation record, they are actually just global variables.

To assist us, a module *Lib.ml* is defined to handle sequences of code that will frequent our Tofu programs (*e.g.* object instantiation code), as well as some other useful functions.

4.3.1 Lib.ml

The interface of the *Lib* module is as follows:

- `val gen_object : ctype -> Keiko.code`
- `val gen_integer : int -> Keiko.code`
- `val gen_boolean : bool -> Keiko.code`
- `val sizeof_object : class_desc -> int`

The first of these methods, *gen_object ctype*, pushes a Keiko code sequence (onto the stack) that creates a new object of type *ctype*. Recall (from figure *2.3*) that the first 4 bytes of an object's allocated memory store the address of its class' method table, and that the addresses of its field objects follow thereafter (each being an address of size 4 bytes). Hence,

```
let gen_object cname = let cdesc = find_class cname in SEQ [

    (1)         CONST (sizeof_object cdesc);
    (2)         GLOBAL ("%"^cdesc.class_name);
    (3)         CONST 0;
    (4)         GLOBAL "_new";
    (5)         PCALLW 2

]
```

- Line *1* determines how many bytes are required to store an object of type *cname*. We define *sizeof_class* to be (number of class variables + 1) x 4, with the '+1' to account for the virtual table's address. Equivalently:

- *sizeof_object cdesc = fold_left (\acc v -> acc + 4) 4 cdesc.variables*

- Line *2* pushes the address of the class' virtual table.

- Line *4* calls the primitive *_new* that will allocate the appropriate amount of space for the object, determined by line *1*, in memory – setting the first 4 bytes to be the address pushed in line *2*.

The second method, *gen_integer n,* is used to box a constant integer *n*. This requires us to instantiate a new *Integer* object, assign the value *n* to it, and then return the address of this object.

```
let gen_integer n = SEQ [

      (1)     gen_object "Integer";
      (2)     DUP;
      (3)     CONST n;
      (4)     SWAP;
      (5)     CONST integer_value_offset;
      (6)     BINOP PlusA; STOREW
]
```

- Lines *1* to *4* instatiate a new *Integer* object *x*, duplicate the address of *x* that is returned, push the value *n*, and then swaps this value *n* with one the copies of *&x*. The stack at this point is:

&x
n
&x

- Lines *5* and *6* then calculate the address of the *x._value* and store *n* there (popping *&x* and *n)*, leaving *&x* on the stack to represent the value of the new *Integer* object.

A *Boolean* object has a field *_value* of type *Integer* such that for *Boolean* object *b*:

- *b._value.isEqual(0)* if and only if *b* represents *False*.
- Otherwise, b represents *True.*

The implementation of *gen_boolean b* is thus,

```
let gen_boolean b =

    let n = if b then 1 else 0 in SEQ [

              gen_object "Boolean";
              DUP;
              Gen_integer n;
              SWAP;
              CONST Boolean_value_offset;
              BINOP PlusA;
```

```
            STOREW
    ]
```

This works the same as *gen_integer*, except that instead of storing a constant value we are actually storing the address of a fresh *Integer* object. The implications of this are discussed in chapter *6.*

4.3.2 Kgen.ml

With reference to step two of *translate*: "*List.iter gen_class cdescs",* let us examine *gen_class*:

```
let gen_class cdesc = List.iter (gen_method cdesc) cdesc.method_table.methods
```

where

```
let gen_method cdesc mdesc =

  if "cdesc's class is the defining class for mdesc's method"
  then mdesc.code <- gen_stmt cdesc mdesc
```

This means that we only generate code for a procedure that is *defined* by the class we are currently considering. This corresponds to inherited (but not overridden) methods, like *learn(x:Integer)* in figure *2.3.2*, that will have several pointers that lead to their code (in memory) from various class' method tables.

Similar to our implementations of functions *check_stmt* and *check_expr*, we will perform recursion over the program's abstract syntax in mutually recursive functions *gen_stmt* and *gen_expr*.

gen_expr generates code to push the value of an expression onto the stack, which is just an address:

```
let gen_expr cdesc edesc = match edesc.expr_guts with

      | This -> SEQ [LOCAL 16; LOADW]

      | Number n -> gen_integer n

      | Boolean b -> gen_boolean b

      | Variable vdesc -> gen_var_addr mdesc vdesc

      | NewObject cname -> gen_object cname

      | Call (edesc2, mname, arg_edescs) ->

            gen_method_call mdesc edesc2 mname arg_edescs true
```

Recall that we always pass the address of the object whose method is being called as the first argument to one of its methods. The first argument resides at an offset of 16 from the frame-pointer (figure *2.3.2)*, thus *This* just pushes the address of that object. We must note that *This* will not arise in the *main* declaration's sequence of statements as we only use *This* to represent calls from within some class – and *main* is not considered to be within any class.

A variable's value is the address of an object that could be a parameter, a local, a class field, or even a global (if the statement is inside the *main* declaration*)*:

```
let gen_var_addr mdesc vdesc = match (unwrap vdesc.variable_kind) with
```

```
  | Field -> SEQ [LOCAL 16; LOADW; CONST vdesc.offset; BINOP PlusA; LOADW]

  | Local | Arg -> SEQ [LOCAL vdesc.offset; LOADW]

  | Global -> GLOBAL ("_"^vdesc.variable_name)
```

We handle a field variable by first pushing the address of the object it belongs to, then we add the annotated offset, and then we push the address of this object's object onto the stack.

Local variables and references to a method's formal parameter are handled simply by loading the address stored at some offset from the frame pointer.

We arrange that a local variable *x* in *main* is actually a global variable, named '*_x*'. We simply push the runtime address of *x*.

Generating the code for a procedure call is handled by another function, as there are two types of method call in Tofu: those that return a value, and those that do not – and these differ by only one instruction:

```
let rec gen_method_call mdesc edesc2 mname arg_edescs rw =

  let callOp = if rw then PCALLW else PCALL in SEQ [

      (1)     SEQ (List.rev_map (gen_expr mdesc) arg_edescs);
      (2)     gen_expr mdesc edesc2;
      (3)     DUP;
      (4)     LOADW;
      (5)     CONST (4 * mdesc2.vtable_index);
      (6)     BINOP PlusA; LOADW;
      (7)     CONST 0; callOp;
  ]
```

We use the Boolean *rw* to determine whether or not the method call should return a word or not. Note, the parameter *mdesc* corresponds to the method from which this method is being called, and *edesc2* is the expression who is calling the method *mname* . Line by line, *gen_method_call* does as follows:

- Line *1* pushes code to generate the expressions that are passed as arguments. Note, we evaluate the arguments from left to right but push them onto the stack in reverse. *rev_map f xs* is equivalent to *(rev · map f) xs*.
- Line *2* pushes generates code to push the value of *edesc2*'s expression, which is an address.
- Lines *3* to *6* calculate the address of the method being called. This is done by first pushing the address of the class' virtual method table *(4)* and then loading the relevant address of the code *(5 & 6)*.
- Line 7 calls the procedure in standard Keiko fashion.

Generating code for statements is slightly more intricate, as we will use *Labels* to represent points in the object code we would like to jump to:

```
and gen_stmt mdesc body = match body with
```

```
| Skip -> NOP
| Seq ss -> SEQ (List.map (gen_stmt mdesc) ss)
| UnitCall (edesc,mname,arg_edescs) ->
      gen_method_call edesc mname arg_edescs false
| LocalVarDecl (vdesc,cname) -> SEQ [
      gen_object cname; gen_init cname; LOCAL vdesc.offset; STOREW ]
| AssignStmt (vdesc,edesc) ->
      SEQ [ gen_expr mdesc edesc; gen_var_addr mdesc vdesc; STOREW ]
| ReturnStmt edesc ->
      SEQ [ gen_expr mdesc edesc; RETURNW ]
| IfStmt (edesc,true_ss,false_ss) ->
      let tlab = label () and flab = label () and exit = label () in
      SEQ [  gen_cond mdesc edesc tlab flab;
             LABEL tlab; gen_stmt mdesc true_ss; JUMP exit;
             LABEL flab; gen_stmt mdesc false_ss;
             LABEL exit ]
| WhileStmt (edesc,body) ->
      let start = label () and bodyLab = label () and exit = label () in
      SEQ [  LABEL start;
             gen_cond mdesc edesc bodyLab exit;
             LABEL bodyLab; gen_stmt mdesc body; JUMP start;
             LABEL exit ]
| Newline -> NOP
```

Firstly, we note the call to *gen_object* and *gen_init,* in the processing of a *LocalVarDecl* node. The latter determines whether or not the class has a method with the signature *init() : Unit*. If so, it pushes code to call this initializing function. This means that the declaration *var x : A* instantiates a fresh object (of type *A)* without the need of the assignment *x = new A.* This is a simple way to prevent unassigned values being used later on. We consider alternative conventions in chapter *7*.

The code generated for an *IfStmt* evaluates a *Boolean* expression and then makes a conditional jump to either label *tlab* or *flab*, depending on this *Boolean* value. To implement this we define,

```
let gen_cond mdesc edesc trueLab falseLab =
  SEQ [  gen_expr mdesc edesc;
```

```
"push the intrinsic value of the Boolean object"

CONST 0;

JUMPC (Eq,falseLab);

JUMP trueLab ]
```

gen_cond pushes code to make a conditional jump depending on the value of the *Boolean* expression corresponding to *edesc*. The labels, *trueLab* and *falseLab*, represent the points in the object code where we would like to continue execution: the former if the condition is true, the latter if not. We implement the check by comparing the *Boolean*'s intrinsic value against the constant 0, *JUMP (Op, Lab)* is a conditional jump that only jumps to label *Lab* if the two arguments atop the stack evaluate to true, when passed as arguments to *Op*. We check for equality. Failing this conditional jump, we jump directly to the true branch.

Thus, the *IfStmt* either branches to the *true* sequence of code generated by *gen_stmt mdesc true_ss* and then jumps to the exit label – where execution will continue with the statements following the *IfStmt*. Or, branches to the code generated by *gen_stmt mdesc false_ss* and then continues execution to the statements that follow.

To implement the *WhileStmt*, we repeatedly check a guard condition: if true, we execute the body of the loop and then jump back to the point at which we check the guard, otherwise we exit.

4.4 Compiler Output

The final task of the compiler is to output the generated object code, using the proper directives. At this stage, we have generated all code for the procedures and *main* – we need only organize the code as to be consistent with our design of the virtual method tables.

We define a module *Output.ml*, with interface:

```
val output : Tree.program -> unit
```

output will print (properly formatted) the Keiko instructions to *stdout*. The implementation is itself a simple pass over the abstract-syntax:

- Output each library class' procedure code and method tables.
- For each class descriptor *cdesc:*
  - output the code for all methods defined by *cdesc*, with appropriate headers.
  - output the method table
- Output the *main* declaration's object code, along with global variable declarations for the *main*'s locally instantiated variables.

5. Testing

To test our compiler, we can systematically compile some programs while checking for some particular output / runtime expectations. To assist us, functions to print an AST of type *Tree.program* are introduced, allowing for us to check the program at different stages throughout compilation.

Let us consider the compilation of the *Student* program described in figures *1* and *1.2,* by analyzing select output. This is beneficial as we can ensure that the requirements in our design are met by each of the three central functions that implement our compiler.

Recall the following three functions,

```
1. val program : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Tree.program
2. val annotate : program -> unit
3. val translate : program -> unit
```

After calling the function *program (1)* on the source text, we produce an AST with blank descriptors where *=None* denotes an annotation of type *'a option* that has yet to be given a value.

Program (Main (MethDecl (main,-,=None,0,[],-1,[],Seq
[LocalVarDecl (ada, Student,=None);

LocalVarDecl (alan, Student, =None);
Assign (Variable (ada,=None,=None,-1),
New (ComputerScientist,=None));
Assign (Variable (alan,=None,=None,-1),
New (Mathematician,=None));
Call (Variable(ada,=None,=None,-1),speak, []);
Call (Variable (alan,=None,=None,-1), speak, []);
Skip],NOP),NOP),

[ClassDecl (Student, Object, =None, [], VTable ([]));
ClassDecl (Mathematician, Student, =None, [], VTable ([]));
ClassDecl (ComputerScientist,Mathematician,=None, [],VTable ([]))])

(Figure 5.1 The initial AST for the Student program)

The output complies with our design of function *1*: namely, that we set up the skeleton of the AST with descriptors for our classes. We note that at this point, the methods have yet to be placed in the *vtable*'s of each descriptor – hence they do not appear in figure *5.1*. We can see the static types of the local variables *ada* and *alan* are declared to be *Student.*

The next phase in the compiler is to perform type checking and annotating, we will call *annotate (2)* on the *Tree.program* in figure *5.1*. This phase will link the descriptors to the appropriate *class_descs* in the *Env* (denoted *=ctype*), as well as annotate offsets appropriate to variables, and put *method_desc*'s in the method tables of each class descriptor.

E.g.1 An excerpt from the resulting *main* descriptor:
Assign (Variable(ada,=Student,=Global,-4), New (ComputerScientist,=ComputerScientist));

We see that *ada* is recognized to be a *Global* variable, due to its declaration in the *main.* Because of this, the annotation -4 is actually meaningless – recall that we push *Global* address' via the instruction *GLOBAL name.* The RHS of this expression has been annotated with type *ComputerScientist*, and our checks have ensured that this of a subtype of the LHS (*Student*).

E.g.2 Some of the *ComputerScientist*'s descriptor:

```
ClassDecl (ComputerScientist, Mathematician,=Mathematician,
        [Variable(iq,=Integer,=Field,4);

         Variable(number,=Integer,=Field,8);

         Variable(coding,=Boolean,=Field,12)],

         VTable ([MethDecl (isEqual,Boolean,=Object,1, [Formal (that,Object)],0,[],Skip,NOP);

                 MethDecl (print,Unit,=Object,0,[],1,[],Skip,NOP);

                 MethDecl (init,Unit,=ComputerScientist,0,[],2,[],Seq
                         [Assign (Variable(iq,=Integer,=Field,4),Number (256,=Integer));
                          ..... ]);
                 MethDecl (learn,Unit,=Student,1,[Formal (x,Integer)],3,[],Seq [.....])    .... ] )
```

(Figure 5.2 The type-checked and annotated class_desc for class ComputerScientist)

Several features of our design can be seen in figure *5.2*:

- The memory layout of a *ComputerScientist* matches that of figure *2.3.*
- Inherited methods, such as *isEqual* and *learn,* can be seen in the *VTable* with annotations that indicate the defining class for these methods, thus allowing us to form proper method tables like the ones described in *figure 2.3.2*.
- *ComputerScientist* overrides the *init* method, but the method maintains its index in the table (because of our *prefixing*).

The final phase, performed by *translate (3)* alongside *Output.ml*, will annotate the AST further by replacing the *NOP* instructions seen in figure *5.2*'s method descriptors with the actual object code (if the class defines that method), and then print all of this as described in section *4.4*.

Here are the method tables for classes *Student*, *Mathematician*, and *ComputerScientist:*

```
DEFINE %Student
WORD Object.isEqual
WORD Object.print
WORD Student.init
WORD Student.learn
WORD Student.speak
```

```
DEFINE %Mathematician
WORD Object.isEqual
WORD Object.print
WORD Mathematician.init
WORD Student.learn
WORD Mathematician.speak
WORD Mathematician.setNumber
```

```
DEFINE %ComputerScientist
WORD Object.isEqual
WORD Object.print
WORD ComputerScientist.init
WORD Student.learn
WORD ComputerScientist.speak
WORD Mathematician.setNumber
WORD ComputerScientist.coding
```

(Figure 5.3 The method tables of user-defined classes in Student.tofu)

This corresponds nicely to the notion of inheritance that we implement through the virtual method tables: for example, inherited methods from *Object* in the table (that are not overridden) point to the procedure code produced by the *Object* class. Implementing our design in figure 2.3.2.

Here are some procedures that are pointed to by the tables in *Figure 5.3*,

```
PROC ComputerScientist.coding 0 0 0
LOCAL 16
LOADW
CONST 12
PLUSA
LOADW
RETURNW
END
```

```
PROC Object.print 0 0 0
LOCAL 16
LOADW
CONST 0
GLOBAL _print_num
PCALL 1
END
```

6. Evaluation

So far, the selected output has been nice and concise. Let's look at some of the object code for the *speak* method, defined in *ComputerScientist* (in figure *1)*, as it reveals several issues with our implementation of Tofu.

```
PROC ComputerScientist.speak 4 0 0
CONST 8
GLOBAL %Integer
CONST 0
GLOBAL _new
PCALLW 2
DUP 0
DUP 0
LOADW
CONST 40
PLUSA
LOADW
CONST 0
PCALL 1
LOCAL -4
STOREW
LOCAL 16
LOADW
CONST 8
PLUSA
LOADW
LOCAL -4
LOADW
STOREW
…
```

Var x : Integer;

x.init();

x = this.number;

(Figure 5.4 Object code for the speak() method)

The first set of bracketed instructions in Figure *5.4* corresponds to our convention that a local variable declaration always instantiates an object of the static type. The next grouping of instructions calls the *init() : Unit* method that is defined by the *Integer class*, initializing *x*'s intrinsic value to 0. Finally, the remaining code sequence deals with the assignment *x = number*. There are some important points to be made here:

- Because we are re-assigning a value to the variable x immediately, via the declaration *"var x : Integer =number",* we need not execute the first two groups of instructions in *Figure 5.4*. This is immediately amended by accounting for the fact this is not a "stand-alone" declaration (such as *var y : T;)* through extension of the *VarDecl* node. We can include an optional expression that is to denote it's initialized value. This will also remove the need for many (simple) initializations via the *init() : Unit* convention. I.e. we extend *VarDecl* in type *Tree.stmt* to be:

```
type stmt =
    …
    | VarDecl of vdesc * ctype * edesc
    …
```

  This requires only minor modifications to the parser, type-checker and code generator.

- Another issue is the assignment *x = number*. As variables are just pointers to objects, this assignment actually makes *x* point to the class field *number*. As a consequence, any modification to *x* (*i.e. x.divide(2) in Figure 1)* will change the field's intrinsic value. Implied by the fact that a variable was declared and then initialized to *number,* this may not have been the programmer's intention. As such, we should provide a means of copying (deep) an object. This will require the extension of the *class_desc*'s for *Object, Integer,* and *Boolean*. The implementation details are fairly trivial. Thus, given the previous extension and a *copy* method, the programmer could now write: *var x : Integer = number.copy()*. Otherwise, with clarity in mind, they should just use the *number* field directly. The typing of such a method would require some trickery, as technically (as it stands) *number.copy()* would actually return an object of type *Object*. A solution is outlined in chapter *7,* whereby we introduce a special type keyword *Self.*

- The final point to be made arises from the expression "*x.isGreaterThan(1)"* in *Figure 1*. The type signature for the invoked method is "*isGreaterThan(that : Integer) : Boolean",* this serves as a guard to the *WhileStmt*.

  Firstly, the implementation of using a constant *n* in place of an *Integer* is *n* implicitly boxed up inside a freshly instantiated *Integer* object (with intrinsic value *n*). The method then returns a fresh *Boolean* object that in turn contains an *Integer* value to represent its state. If the guard expression is evaluated *N* times:

  - We create *N* fresh *Integer* objects to represent '1'
  - We create *N* fresh *Boolean* objects to represent to the value of the expression, each creating a new *Integer* object to represent the state.
  - Thus, in total, we create *2N Integer* objects and *N* Booleans. This requires a total of $2N * 8 + N * 16 = 32N = \Theta(N)$ *bytes* [†] to store the objects and their intrinsic values, as Tofu has no garbage collector.

---

† *Integer* requires 4 bytes for the *vtable* address and 4 bytes for the intrinsic value, whereas a *Boolean* needs 4 bytes for the *vtable* address, 4 bytes for the address of its intrinsic object and then a further 8 bytes for the new integer it points to

This is worrying, as we would like such a simple element of the program to operate in constant space. There are several solutions to this, whilst maintaining the purity of the language:

1. We implement *Boolean* objects so that they store an intrinsic value, like that of the *Integer*. This halves the space required to store a fresh *Boolean*, but does not reduce the space complexity.

2. We allocate two words in memory to store the values *1* and *0*, at locations *&t* and *&f*. Any expression that yields a *Boolean*, like that of "*x.isGreaterThan(1)",* can just return *&t* or *&f* as appropriate. Any *Boolean* that is assigned a truth value will just reassign its *_value* field to be either *&t* or *&f*. We can do something similar for *Integer* objects, i.e. allocate fixed-space to store the boxed integer constants when they are used – however this will require extra care as we will need to determine a bound on the number of such boxing's that occur at any one time as well as keep track of which boxes are currently in use: a similar problem to register allocation. This would provide a constant space complexity, in the situation described earlier.

3. Implement a garbage collector. Although the collector would have to de-allocate a lot of objects, and we would need to take care in deciding when to collect, this idea coupled with a better implementation of the library classes would be considerably more efficient in terms of memory.

We should note that these solutions also reduce the amount of instructions in the object code, in turn producing faster Tofu programs.

The lack of some common built-in types certainly reduces the power of the language (caused by a lack of time on the author's part). In chapter *7* we discuss (briefly) the addition of arrays. These are fairly fundamental to implementing most serious algorithms and abstract data-types. Furthermore, an implementation of a *String* type would be most useful in giving feedback in a program.

7. Future

We will now consider some short-term developments of Tofu, that were not implemented due to time constraints, and then some more ambitious long-term ideas.

Immediately, we could introduce the notion of parameterized constructors for classes. That is to say, when a class *C* is instantiated and *C* defines a method *C(*x,y,…,z) : *C*, we must provide values for these arguments *e.g. var c : C = new C(10, true, .., new D).* This is similar to what the *init() : Unit* convention provides, but with the added expressivity of parameterization. We must take care typing these constructors, as our (current) notion of inheritance would mean they are inherited with return types of the parent class, and not the subclass. This can be solved by using the idea of a '*Self'* type, adapted from [7, p. 271], whereby the return type *Self* is used to denote the type of the class from which is was called. Similar to the keyword '*This'* in Java [8], but for types and not objects.

We could also formally introduce the idea of a class interface, to support the design of OO code. As it stands, the programmer could define a class with blank definitions and uninitialized fields that *acts* as an interface, *e.g.*

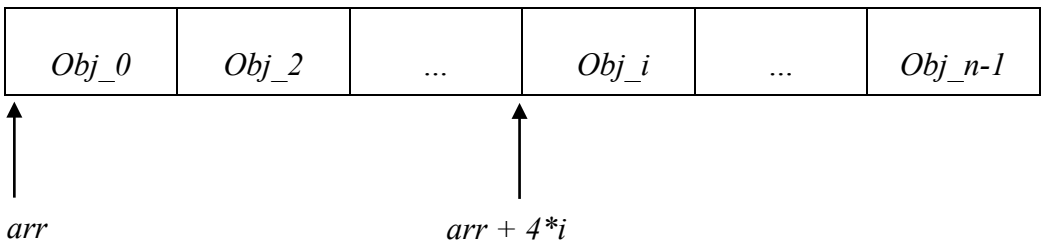
```
class PersonInterface { …
             var age : Integer;
             def dance(time : Integer) : Unit = {}
              … }
```

Introducing the keywords *implements* and *interface* (as well as some more intricate type checking) we could write programs that appeal to many OOP design ideas, like:

```
interface Person {
             age : Integer;
             gender : Gender;
             speak() : Unit;
             dance(time : Integer) : Unit;
}

class JohnTravolta implements Person {
             def dance(time : Integer) : Unit = { … }
             …
}
```

An array data type could be introduced. This would be implemented as a generic type such that the type *Array[T]* consists of a data-structure of some fixed size that contains *N* objects that are each of subtype *T*. The array itself is as a contiguous section of memory, where we store the value of each object at intervals of 4 bytes apart (these values are just the addresses of objects):



(Figure 7.1 An array object's layout in memory)

Figure *7.1* shows the contiguous space of an array, where the address of the *ith* object is stored at location *arr* + 4 * i , with *arr* being a variable pointing to the array. Some care would be needed to ensure that the bounds of an array are not exceeded.

Type checking an array would involve ensuring that the expression on the right hand side of any assignment is a sub-type of the Array's type parameter. We would also need to extend the permitted kinds of variables as a subscript of the form *arr[i]* denoting the *ith* element in array *arr* would appear in the left hand side of such an assignment.

To allow objects to be deallocated memory we could support an explicit deconstructor method or implement a garbage collector by some means of simple reference counts, whereby we deallocate an object's memory if there are no references to it in our program at some point.

Two final ideas - first: permit class fields and method return types to be altered by sub-classes, according to some rules, and second: following on from the introduction of the generic type *Array[T]* in *Figure 7.1*, we could permit any class to take type parameters so that when they are instantiated we require arguments of type *ctype*, in square brackets, *e.g. var x :*

GenericClass[A,B,C]. Both of these ideas need some restrictions, as described below:

We introduce the ideas of *covariant, invariant* and *contravariant* subtypes. Given types *A* and *B,* and some type transformation *f*, where *'≤'* denotes the relation 'subtypes':

- *f* is ***covariant*** if $A \leq B$ implies that $f(A) \leq f(B)$
- *f* is ***contravariant*** if $A \leq B$ implies that $f(B) \leq f(A)$
- *f* is ***invariant*** if neither of the above holds.

We could allow the programmer to specify the relationship of overridden types, with some careful consideration as to what should be allowed – as not all of these ensures type-safety!

We would say that the array data type is ***covariant***. *I.e. ComputerScientist ≤ Student* implies that *Array[ComputerScientist] ≤ Array[Student]*. Whereas, to ensure type-safety with *generic* types: we would say that all other *generics* are ***invariant***. This means that for a generic class e.g. *Box[C], Box[ComputerScientist]* neither subclasses *Box[Student]* nor super classes it.

Works Cited

[1] D. M. Spivey, "Keiko Specification," [Online]. Available: http://spivey.oriel.ox.ac.uk/corner/Specification_of_Keiko_(Compilers).

[2] OCaml.org, 2016. [Online]. Available: https://ocaml.org.

[3] R. S. J. D. U. Alfred V.Aho, Compilers, Principles, Techniques and Tools, Addison-Wesley , p. 49.

[4] A. W. Appel, "Moden Compiler Implementation in ML," Cambridge University Press, 1997, pp. 7-10.

[5] R. C. Martin, "The Liskov Substitution Principle," vol. 8, March 1996.

[6] K. B. Bruce, Foundations of Object-Oriented Languages, The MIT Press, 2002, pp. 22-26.

[7] J. B. Smith, Practical OCaml, Apress, 2007, pp. 193-211.

[8] J. G. D. H. Ken Arnold, The Java Programming Language, Fourth Edition, Addison Wesley Professional, 2005.

Appendix A

```
* parser.mly */

%token<string>                                          IDENT
%token<int>                                             NUMBER
%token                                                  TRUE FALSE
%token                                                  THIS
/** decleratives */
%token                                                  MAIN CLASS EXTENDS NEW VAR DEF
/** statements */
%token                                                  WHILE IF ELSE LCURL RCURL ASSIGN RETURN
/** punctuation */
%token                                                  DOT COMMA SEMI COLON EOF BADTOK NEWLINE LBRAC
RBRAC
/** main entry point */
%start                                  program
%type<Tree.program>             program

%{
                open Tree
                open Keiko
%}

%%

program :
MAIN LCURL stmts RCURL class_decl_list { Program ( mainDesc (methodDesc "main" "-" [] $3)  , $5 ) };

/********* Declerations **********/

class_decl_list :
                  /* empty */                                           { [] }
                | class_decl class_decl_list                            { $1 :: $2 };

class_decl :
                  CLASS IDENT LCURL feature_decl_list RCURL
                { ClassDecl (classDesc $2 "Object", $4) }
                | CLASS IDENT EXTENDS IDENT LCURL feature_decl_list RCURL  { ClassDecl (classDesc
$2 $4, $6) };

feature_decl_list :
                  /* empty */                       { [] }
                | feature_decl feature_decl_list    { $1 :: $2 };

feature_decl :
                  DEF IDENT formals COLON IDENT ASSIGN LCURL stmts RCURL { MethDecl (methodDesc $2
$5 $3 $8) }
                | VAR IDENT COLON IDENT SEMI
{ InstanceVarDecl ((variableDesc $2),$4) };

formals :
                  LBRAC RBRAC                       { [] }
                | LBRAC formal_list RBRAC           { $2 };

formal_list :
                  formal                        { [$1] }
                | formal COMMA formal_list      { $1 :: $3 };

formal :
                IDENT COLON IDENT                                       { Formal ($1, $3) };

/*********** Statements ************/

stmts :
                stmt_list                                       { seq $1 };

stmt_list :
                  stmt                                      { [$1] }
                | stmt SEMI stmt_list                       { $1 :: $3 };

stmt :
                  /* empty */                               { Skip }
                | VAR IDENT COLON IDENT                     { LocalVarDecl ((variableDesc $2),$4) }
                | IDENT ASSIGN expr                          { AssignStmt ((variableDesc $1), $3) }
                | RETURN expr                                { ReturnStmt $2 }
                | IF LBRAC expr RBRAC LCURL stmts RCURL    { IfStmt ($3, $6, Skip) }
```

```
                | IF LBRAC expr RBRAC LCURL stmts RCURL ELSE LCURL stmts RCURL
                { IfStmt($3, $6, $10) }
                | WHILE LBRAC expr RBRAC LCURL stmts RCURL                              { WhileStmt
($3, $6) }
                | NEWLINE                                                       { Newline }
                | expr DOT IDENT args                                                   { UnitCall
($1, $3, $4) };
                | IDENT args                                                    { UnitCall
(exprDesc This, $1, $2)}  /* implicit this.method(args) call */

/*********** Expressions ************/

expr :
                | THIS                                                                  { exprDesc
This }
                | NUMBER                                                        { exprDesc (Number
$1)  }
                | TRUE                                                                  { exprDesc
(Boolean true) }
                | FALSE                                                         { exprDesc (Boolean
false) }
                | IDENT                                                         { exprDesc
(Variable (variableDesc $1)) }
                | NEW IDENT                                        { exprDesc (NewObject $2) }
                | expr DOT IDENT args        { exprDesc (Call ($1, $3, $4)) };
                | IDENT args                 { exprDesc (Call (exprDesc This, $1, $2))}
                                           /* implicit this.method(args) call */

args :
                  LBRAC RBRAC                           { [] }
                | LBRAC expr_list RBRAC                 { $2 };

expr_list :
                  expr                                  { [$1] }
                | expr COMMA expr_list                  { $1 :: $3 };
```