A Compiler for an Object-Oriented Language

Candidate Number: 1015723

University of Oxford

Trinity 2020

Project Supervisor:

Dr Michael Spivey

Word count: 9564

Abstract

Object-oriented languages are the most popular style of programming language for software development and are used extensively across many different areas. However, most of the commercial-grade object-oriented languages that are used are not pure object-oriented. “Pure” in this case means that everything in the language is an object, and “not pure” means that primitive types such as integers and Booleans are treated differently than objects.

The goals for this project were to design and build a compiler for an object-oriented language that would take a pure object-oriented approach in its design to analyse the feasibility of such a language – in particular, analysing the performance consequences of using objects for primitive types by comparing the performance to other languages. In this report I lay out the design for my compiler for Baozi (bau-tze), a language of my own design, targeting a simple stack-based abstract machine called Keiko (Spivey, 2017). Baozi has a number of core object-oriented features such as: classes and objects, inheritance, polymorphism and object identity.

Contents

Acknowledgements

Mike. Ed. Jenn.

Introduction

The majority of this report is dedicated to explaining the design and implementation of my compiler for the Baozi language – in particular explaining how the object-oriented features of Baozi are achieved and giving short examples demonstrating the desired behaviour. Following that will be a discussion of how efficient Baozi is, giving comparisons to Java, picoPascal and Scala.

Baozi

To start, we should introduce Baozi. Baozi is a strict, static typed, pure object-oriented language. As stated earlier, “Pure” means that everything is an object. In Baozi no primitive values are stored on the stack or as properties (or instance variables) of an object. Instead, primitive types such as integers or Booleans are “boxed” – that is to say, stored in records in heap memory.

The compiler is written in OCaml and has several passes over the Baozi source file in order to create object code. The compiler first creates an abstract syntax tree from the Baozi code, it then annotates the tree, transforms the annotated tree into Keiko code, optimises the Keiko code using a peephole optimiser, and finally the Keiko code is interpreted into machine code that can be executed. The compiler was tested by writing example programs that demonstrate the various features of the language and checking that they produced the correct output/error message.

Here we have an example of a simple program in Baozi:

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        p : Person = New Person.
        p->name = "Baozi".
        p->SayHello<-{}.
        Return.
    <$
<$

Define Person
$>
    Properties
    $>
        name: String,
    <$

    Method SayHello {} => {}
    $>
        Output->String<-{"Hello, World! I am "}.
        Output->StringLn<-{My->name}.
        Return.
    <$
<$
```

Two classes are defined in this program, one called “Program” and the other called “Person”. The Program class has one static method called Main. Main is also a keyword which identifies this method as the start point for the program. The brackets and arrows after the method name indicate the type of the method. In the case of Main, it takes no arguments and produces no output. The Person class has one method called “SayHello”. The SayHello method also takes no arguments and produces no result. The body of the Main method creates a new object of type Person, assigns the string “Baozi” to the “name” property of that object, and then calls the SayHello method on that object. The SayHello method calls the static methods “String” and “StringLn” on the class Output, with a single argument each, the string constants “Hello, World! I am ” and the object’s name property respectively. The output is shown below:

```
--> Hello, World! I am Baozi
```

Using the `Define` keyword, you can create as many classes as you desire. Each class then serves as a type that a variable can have or as part of a more complex type such an array type or a generic type. For simplicity, the whole program is in one file. Separate compilation is a project for future work.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        rect : Rectangle = New Rectangle.  -- rect has type Rectangle
        rect->width = 5.
        rect->height = 10.

        circ : Circle = New Circle.        -- circ has type Circle
        circ->radius = 7.

        rects : Array Of Rectangle = New Array Of Rectangle<-[3]
                                           -- rects has type Array of
                                                 Rectangle
    <$
<$

Define Rectangle                           -- Defines the class Rectangle
$>
    Properties
    $>
        width: Int,
        height: Int,
    <$
<$

Define Circle                              -- Defines the class Circle
$>
    Properties
    $>
        radius : Int,
    <$
<$
```

Classes serve as templates for objects to be created. Classes can define a number of properties and methods that can be accessed on objects. For clarity, in Baozi, properties are the same as instance variables and it does not define explicit getter and setter methods.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        v1 : Vector = New Vector.
        v1->x = 3.
        v1->y = 4.

        v2 : Vector = New Vector.
        v2->x = 5.
        v2->y = 12.

        Output->String<-{"v1 has x="}. Output->IntLn<-{v1->x}.
        Output->String<-{"v2 has x="}. Output->IntLn<-{v2->x}.
        Return.
    <$
<$

Define Vector
$>
    Properties
    $>
        x : Int,
        y : Int,
    <$
<$

--> v1 has x=3
    v2 has x=5
```

Methods in a class can also reference the object that receives the method. This is done through the use of the “Me” or “My” keyword.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        a1 : A = New A.
        a1->p = 7.

        a2: A = New A.
        a2->p = 43.

        a1->PrintSelf<-{}.
        a2->PrintSelf<-{}.
        Return.
    <$
<$
```

```
Define A
$>
    Properties
    $>
        p : Int,
    <$

    Method PrintSelf {} => {}
    $>
        Output->String<-{"My property is "}.
        Output->IntLn<-{My->p}.
    <$
<$

--> My property is 7
    My property is 43
```

A class can also inherit from a single other class. Doing this allows the subclass to use all the same properties and methods as their parent's class, as well as adding their own. Furthermore, you can always substitute an instance of the subclass into a context that expects an instance of the parent class, and it will implement the expected protocol. This is called the Liskov substitution principle (Liskov & Wing, 1994).

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        d : Dog = New Dog.
        a : Animal = New Dog.     -- valid, as Dog is a subtype of Animal

        d->size = 5.              -- Dog inherits size from animal
        d->breed = "Labrador".    -- Dog adds a new property, breed

        d->eat<-{}.               -- Dog inherits eat from animal
        d->bark<-{}.              -- Dog adds a new method, bark

        Program->Feed<-{d}        -- Can pass a dog to a method that
                                                    expects an animal
        Return.
    <$

    ClassMethod Feed {an : Animal} => {}
    $>
        an->eat<-{}.
        Return.
    <$
<$

Define Animal
$>
    Properties
    $>
        size : Int,
```

```
    <$

    Method eat {} => {}
    $>
        Output->StringLn<-{"Yum!"}.
        Return.
    <$
<$

Define Dog <- Animal            -- Dog inherits from Animal
$>
    Properties
    $>
        breed: String,
    <$

    Method bark {} => {}
    $>
        Output->StringLn<-{"Bark!"}.
        Return.
    <$
<$

--> Yum!
    Bark!
    Yum!
```

Subclasses can also override methods that they inherit from their parent class. In addition, the `Parent` keyword can be used to call the parent's version of a method.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        d : Dog = New Dog.
        a : Animal = New Animal.

        a->eat<-{}.
        d->eat<-{}.
        d->animalEat<-{}.
        Return.
    <$
<$

Define Animal
$>
    Method eat {} => {}
    $>
        Output->StringLn<-{"Eating food!"}.
        Return.
    <$
<$
```

```
Define Dog <- Animal
$>
    ReplaceMethod eat {} => {} -- Dog overrides the eat method from Animal
    $>
        Output->StringLn<-{"Eating dog food!!"}.
        Return.
    <$

    Method animalEat {} => {}
    $>
        Parent->eat<-{}.       -- Call Animal's eat method
        Return.
    <$
<$

--> Eating food!
    Eating dog food!!
    Eating food!
```

The exact method called on an object is determined at runtime. This is called dynamic binding.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        p : Person = New Person.
        p->SayHello<-{}.

        p = New Child.
        p->SayHello<-{}.
        Return.
    <$
<$

Define Person
$>
    Method SayHello {} => {}
    $>
        Output->StringLn<-{"Hello, World!"}.
        Return.
    <$
<$

Define Child <- Person
$>
    ReplaceMethod SayHello {} => {}
    $>
        Output->StringLn<-{"Hi!"}.
        Return.
    <$
<$

--> Hello, World!
```

```
    Hi!
```

You can downcast a type to a subtype using the "`Cast`" notation. A procedure will be performed at runtime to check that the cast is valid.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        r : Rectangle = New Rectangle.
        r->width = 11.
        r->height = 4.

        Program->printArea<-{r}.
        Return.
    <$

    ClassMethod printArea {s : Shape} => {}
    $>
        rect : Rectangle = Cast(s)->Rectangle.
        Output->IntLn<-{rect->width * rect->height}.
        Return.
    <$
<$

Define Shape
$>
<$

Define Rectangle <- Shape
$>
    Properties
    $>
        width : Int,
        height : Int,
    <$
<$

--> 44
```

You can create arrays using the "`Array Of`" notation. Arrays are initialised with a size and elements of the array can be accessed using the `->[i]` notation.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        xs : Array Of Int = New Array Of Int<-[3].
        xs->[0] = 1.
        xs->[1] = 4.
        xs->[2] = 9.

        For i : Int = 0. Step i = i + 1. Test i < xs->Length.
```

```
        $>
            Output->Int<-{xs->[i]}.
            Output->String<-{", "}.
        <$
        Output->Ln<-{}.
        Return.
    <$
<$

--> 1, 4, 9,
```

Generics can be created using the “Using” and “With” keywords.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        l1 : List With (Int) = New List With (Int).
        l1->value = 12.

        l2 : List With (String) = New List With (String).
        l2->value = "Hello!".

        Output->IntLn<-{l1->value}.
        Output->StringLn<-{l2->value}.
        Return.
    <$
<$

Define List Using T
$>
    Properties
    $>
        value : T,
        next : List With (T),
    <$
<$

--> 12
    Hello!
```

And finally, you can condition on the concrete type of an object at runtime.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        x : A = New A.
        y : B = New B.

        If x->GetType<-{} == TypeOf A Then
        $>
            Output->StringLn<-{"x has type A."}.
```

```
        <$
        If y->GetType<-{} == TypeOf A Then
        $>
            Output->StringLn<-{"This will not be printed."}.
        <$
        If y InstanceOf TypeOf A Then
        $>
            Output->StringLn<-{"y is a subtype of A."}.
        <$
        Return.
    <$
<$

Define A
$>
<$

Define B <- A
$>
<$

--> x has type A
    y is a subtype of A
```

Keiko

Keiko is a stack-based abstract machine originally developed by Dr Michael Spivey for the Oxford Oberon compiler (OBC). I chose to target Keiko for the Baozi compiler for several reasons. Firstly, since one does not need to deal with registers when working with Keiko, it was a lot easier to generate code for the abstract machine. Secondly, there was existing Keiko infrastructure, including a garbage collector, that I could make use of. Finally, the existing OBC compiler provided many examples of code that could be implemented in Baozi or used for reference when implementing certain features.

Keiko consists of a set of instructions that are executed in order and push or pop values onto or from a stack. For example

```
CONST 5  -- Push 5 onto the stack
CONST 6  -- Push 6 onto the stack
PLUS     -- Pop two values, add them together and push the result
```

The above Keiko code would leave 11 on the top of the stack. Memory addresses can also be pushed to the stack. For example:

```
LOCAL 12 -- Push the address that is offset 12 from the base pointer
LOADW    -- Pop the address, push the 4-byte value found at that address
```

Keiko has three types of memory: stack, heap and the data segment. By convention, the stack is used for records about the active procedure calls, the heap is used for dynamically allocating memory for data structures, and the data segment is used for global variables and constants. Data segment memory can be laid out using the Keiko assembler directives `DEFINE` and `WORD`. For example:

```
DEFINE A
WORD 4
WORD 2

DEFINE B
WORD A
```

Creates two blocks of memory that can be referenced using the names A and B. A points to an address in memory than has the value 4 at offset 0, and 2 at offset 4. B points to another address in memory that contains a pointer to A at offset 0. We can access these addresses using the GLOBAL keyword:

```
GLOBAL A        -- Get the address of A
CONST 4         -- Push 4
OFFSET          -- Offset 4 from the address of A
LOADW           -- Load from that address
```

This code would result in 2 being on the top of the stack. Although this may seem like a lot of instructions for a simple operation, this is for demonstration purposes. In the compiler, the peephole optimiser would replace these common patterns with single specialised instructions. In the above example the lines CONST 4 / OFFSET / LOADW would be replaced with LDNW 4, which does the same thing.

Procedures can be defined in Keiko using the PROC keyword. These procedures are globally accessible and cannot be nested. Bytecode for procedures is stored in the data segment. Procedures are called by first placing all the arguments onto the stack in reverse order, then pushing the address of the procedure, and finally using the CALLW keyword with number of arguments. If the procedure returns a value, it will be pushed onto the stack when the procedure returns.

```
PROC func 0 0 0 -- 0 bytes for locals, no maximum depth, frame map of 0
LDLW 12         -- Load the first argument (offset 12 in the frame)
LDLW 16         -- Load the second argument (offset 16 in the frame)
PLUS            -- Pop two values, add them together and push the result
RETURNW         -- Pop from the stack and return that value
END             -- The end of the procedure.

...
CONST 4         -- Push 4
CONST 5         -- Push 5
GLOBAL func     -- Push the address of the func procedure
CALLW 2         -- Pop the address and two arguments, then call func
```

After the CALLW 2 instruction, 9 would be on the top of the stack. The layout of the stack frame is as follows:

Arg n	+4(n+2)
…	
Arg 1	+12
Context Pointer	+8
Return address	+4
Dynamic link	Base pointer
Local 1	-4
…	
Local m	-4m

Figure 1

`LOCAL n` gets the address of the base pointer + n and `LDLW n` is single instruction that does the same as `LOCAL n / LOADW`.

Implementation

Compiler phases

The compiler acts to transform the Baozi code to Keiko code in five distinct phases.

Lexical Analysis – The lexer takes the raw text file containing the Baozi code and turns it into a series of tokens. These tokens are keywords, identifiers, operators and punctuation. This allows the next stage of the compiler to identify the language constructs that are in the code. The lexer is written using OCamllex.

Syntax Analysis – The syntax analysis is done using a parser made with OCamllyacc, which takes the stream of tokens from the lexer and creates an abstract syntax tree (AST) using a context free grammar (CFG). The AST for Baozi is a set of mutually recursive data types that describe the components of the language – the classes, methods, properties, statements, expressions, etc – in a structure that is easy to recurse over in later stages of the compiler. The code for the AST for Baozi is given below:

```
type ident = string

(* Temporary type before annotation *)
and ttype =
    Ident of ident
  | Array of ttype
  | Generic of ident * ttype list

(* Origin of method *)
and origin =
    Mine
  | Inherited of string

(* Language constructs *)
and expr =
    Name of name
  | Sub of expr * expr
  | Nil
  | MethodCall of expr * name * expr list
  | Property of expr * name
```

```
  | Constant of int * def_type
  | String of symbol * string
  | TypeOf of name
  | New of name
  | NewArray of name * expr
  | Cast of expr * name
  | Parent

and stmt =
  { s_guts: stmt_guts;
    s_line: int }

and stmt_guts =
    Assign of expr * expr
  | Delc of name * def_type * expr
  | Call of expr
  | Return of expr option
  | Seq of stmt list
  | IfStmt of expr * stmt * stmt
  | WhileStmt of expr * stmt
  | ForStmt of stmt * stmt * expr * stmt
  | Nop

and property = Prop of name * def_type

and m_method =
  { m_name: name;
    mutable m_type: def_type;
    m_static: bool;
    m_main: bool;
    m_replace: bool;
    mutable m_size: int;
    mutable m_arguments: property list;
    mutable m_body: stmt;
    m_origin: origin }

and c_class =
  { c_name: name;
    mutable c_ptype: def_type;
    mutable c_size: int;
    mutable c_properties: property list;
    mutable c_methods: m_method list;
    mutable c_ancestors: c_class list;
    c_generics: name list }

and program =
  Program of c_class list

(* Kind of a name *)
and def_kind =
  | ClassDef
  | VariableDef of int
  | PropertyDef of int
  | MethodDef of int * bool
```

```
  | NoneKind

(* Name defintion *)
and def =
  { d_kind: def_kind;
    mutable d_type: def_type}

(* Type of name *)
and def_type =
  | ClassType of c_class
  | ArrayType of def_type
  | GenericClassType of c_class * (ident * def_type) list
  | GenericType of ident * def_type
  | TempType of ttype
  | VoidType
  | NilType

(* Name for a class, method, property or variable *)
and name =
  { x_name: ident;
    mutable x_def: def}
```

The code for the AST demonstrates the components of the language constructs. The basic element of a Baozi program is the class. A class contains properties, methods, and other relevant information. Methods have a type and contain arguments and a method body made up of statements and expressions.

The parser also translates away syntactic sugar. For example, since integers in Baozi are objects, primitive operations on integers are translated by the parser into method calls.

`5 + 4` is translated to `5->add<-{4}`

To remove ambiguity in statements the CFG for the parser parses expressions in such a way as to enforce order of operations (Ambiguity, 2007).

Semantic Analysis: Annotations – The purpose of this stage of the compiler is to annotate all the names in the AST with relevant semantic information such as the type, what kind of thing the name refers to and, if needed, where in the stack frame it is located. Annotating the program is split into four stages. The four passes are needed as each later phase depends on the ones that come before. As we describe the four stages, we will demonstrate what is annotated on the following program:

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        p : Person = New Person.
        p->name = "Baozi".
        p->SayHello<-{}.
        Return.
    <$
<$

Define Person
$>
```

```
    Properties
    $>
        name: String,
    <$

    Method SayHello {} => {}
    $>
        Output->String<-{"Hello, World! I am "}.
        Output->StringLn<-{My->name}.
        Return.
    <$
<$
```

The job of the first stage is to annotate the names of all the classes in the program and then add them to an environment. The environment is a map from identifiers to definitions, where a definition is a pair of a type and a flag indicating the what kind of thing the definition refers to.

```
(* Environment of idents to definitions *)
module IdMap =
  Map.Make(struct
    type t = ident
    let compare = compare
  end)

type environment = Env of def IdMap.t
```

The list of classes is also the complete list of simple types for the program. All other types can be derived from this list of classes. Thus, once we have annotated the class names, we will be able to annotate everything else with a type.

In our example program the names “Program” and “Person” would be annotated with a definition that marks them as classes. The two classes would also be added to the environment, referenced by their idents, “Program” and “Person” respectively. The library classes such as Output, String, and Object would also be added to the environment.

Next, we annotate the parents of all classes. The parent of a class will be initially set to a `TempType ttype`. The `ttype` should contain the name of the parent class. We then search the environment to find a class with same name. With this parent class, the compiler then recursively updates the parent information on the parent class that was found, and then it updates subclass to add in all the properties and methods that it inherits or overrides from the parent.

In our example program neither Person nor Program declare a parent, and thus they would both inherit from Object.

The next stage of the annotation is to annotate the names of all the methods and properties of each class. Much like when we annotated the parents, the initial type of the methods and properties is a `TempType ttype`. `ttype` is a recursive data structure that contains the information for what type the object should be. At the base level of the data structure will be an `Ident`. Upon reaching an `Ident n` at the bottom of a `ttype` we search the environment for a class with the same name as `n` and build the correct type from that.

```
(* Get proper type from temp type *)
let rec get_temp_type t env =
  match t with
  | Ident n -> let d = lookup n env in d.d_type (* Find the name n in the
                                                   environment *)
  | Array t' -> ArrayType (get_temp_type t' env) (* Recurse on the type of
                                                    the array elements *)
  | Generic (n, ts) ->
      let c = get_class (lookup n env).d_type in (* Find the name in in
                                                    the environment *)
        (* Then do the same for all the types that are passed to the type
              variables *)
        try GenericClassType (c,
           List.combine (List.map (fun g -> g.x_name) c.c_generics)
           (List.map (fun x -> get_temp_type x env) ts))
          with Invalid_argument _ -> raise InvalidGeneric
```

Since we have previously added all the classes to the environment, if the type given is a valid type, we should be able to create it by looking in the environment. When a method, property or variable has a generic type variable as its type, it's flagged differently to allow later stages of the compiler to match the concrete type parameters of a generic with the type variables.

In our example program, the methods "Main" and "SayHello" would be annotated with definitions that: say that they are methods, say that they are void methods, say that Main is static and SayHello is not, and have their offsets in the method tables of their respective classes. The property "name" in Person would be annotated as a property and annotated as being at offset 4 (the place for the first property) in the object record. The methods "String" and "StringLn" already have correct annotations as they are part of the library class Output.

The fourth and final stage of annotation is to annotate all the names in the method bodies. Since all the method names, property names, and class names have already been annotated, we can refer to these from anywhere within any method. This allows for mutual recursion. Before we annotate the method body, we annotate the arguments with their type and their offset from the frame pointer and add them to the environment that will be used when annotating the method body. We then annotate everything in the method body. When we come across a variable declaration, we add it to the environment. When we see a variable reference, we search for it in the environment and annotate it as a variable. When we come across a method call, or property access, we can get the relevant class from the type of the receiving object and look in that class to find the method or property with the correct name. The code for annotating a method call is demonstrated below.

```
let find_method meth t =
  match t with
  | ClassType c ->
      begin (* Find a method in c with the same name as meth *)
        try (List.find (fun m -> meth = m.m_name.x_name)
              c.c_methods).m_name.x_def with
          Not_found -> raise (UnknownName meth)
       end
  | ArrayType _ ->
      begin (* Find a method in array_class with the same name as meth *)
```

```
        try (List.find (fun m -> meth = m.m_name.x_name)
              array_class.c_methods).m_name.x_def with
          Not_found -> raise (UnknownName meth)
      end
  | GenericClassType (c, _) ->
     begin (* Find a method in c with the same name as meth *)
        try (List.find (fun m -> meth = m.m_name.x_name)
              c.c_methods).m_name.x_def with
          Not_found -> raise (UnknownName meth)
      end
  | GenericType (_, d) ->
      let c = get_class d in
      begin (* Find a method in c with the same name as meth *)
        try (List.find (fun m -> meth = m.m_name.x_name)
               c.c_methods).m_name.x_def with
          Not_found -> raise (UnknownName meth)
      end
  | VoidType -> raise VoidOperation
  | NilType -> raise (UnknownName meth)
  | TempType n -> raise (UnannotatedName (print_temp_type n))

let rec annotate_expr expr env =
  match expr with
  ...
  | MethodCall (e, m, args) ->
      (* Annotate the receiving object *)
      let c = annotate_expr e env in
      (* Find method that is being called *)
      m.x_def <- find_method m.x_name c;
      (* Annotate all the arguments *)
      List.iter (fun x -> ignore(annotate_expr x env)) args;
     (* Return the type of the method *)
      m.x_def.d_type
```

In our example program, the names in Main (“p”, “name” and “SayHello”) and the names in SayHello (“Me”, “Output”, “String”, “StringLn” and “name”) would all be annotated. “p” and “Me” would be annotated as variables with their type and their offset in the stack frame. “Output” would be found in the environment and annotated as a class. “name” would be found in Person and annotated as a property with the same definition as was annotated to “name” in stage three of the annotator. “SayHello”, “String and “StringLn” would be found in their respective classes and annotated with the same annotations as the corresponding methods were given in stage three of the annotator.

Semantic Analysis: Type Checking – This next phase of the compiler ensures that all statements and expressions are well typed. It recurses over the AST, and where there is a method call, an assignment or a conditional expression, it gets the types of the expressions involved by recursively calling the type checker on the component expressions. At the base of the recursion will be a name (of a property, variable or method) or a constant. Constants have an obvious type, while names will have a type which will have been annotated onto them. Having established the types of the expressions involved in an expression or statement, the type checker then verifies that the types are compatible. This important bit of this phase is done by the following function:

```
let check_compatible type1 type2 =
  let pt1 = ref VoidType and pt2 = ref VoidType in
  ...
    let rec check t1 t2 =
      match (t1, t2) with
      | (ClassType c1, ClassType c2) ->
          if c1.c_name.x_name == c2.c_name.x_name then ()
          else check t1 c2.c_ptype (* Recurse and check t1 against t2's
                                                        parent type *)
      ...
      | _ -> raise (TypeError((print_type !pt1), (print_type !pt2)))
    in pt1 := type1; pt2 := type2; check type1 type2
```

For compatibility between more complex types such as generics and arrays, compatibility is checked on the constituent parts of those types. At the base level, however, will always be `ClassType`s, and these are checked by first comparing their names and, if they are different, recursively searching up the second type's parents, to see if a match can be found. Since all classes must be defined globally, we can check that two classes are the same by comparing their names. This way of checking type compatibility enforces the Liskov substitution principle. If you have a statement such as `x = y.`, the `check_compatible` method will check that y's type is equals to x's type, or y's type is a subtype of x's type.

Code generation – The last step of the compiler is to produce Keiko code from the annotated AST. The code generator produces code for each of the methods in the defined classes and creates descriptors for the classes using the Keiko `DEFINE` and `WORD` commands. The code is also passed through a peephole optimiser to optimise it. The details for how code is generated for the various features of the language are expanded upon in later sections.

Classes and Objects

At the core of Baozi are classes and objects. In Baozi, a class consists of a name, a set of properties, and a set of methods. When we do code generation, we produce class descriptors for each object. Class descriptors are areas of the data segment laid out with information about the class. If we have the following Baozi code:

```
Define Rectangle
$>
    Properties
    $>
        width: Int,
        height: Int
    <$

    Method GetArea {} => Int
    $>
        Return My->width * My->height.
    <$

    Method GetPerimeter {} => Int
    $>
        Return 2 * (My->width + My->height).
    <$
<$
```

Then in code generation, a class descriptor for this class would be generated as follows:

```
DEFINE Rectangle.%desc          --Makes this address globally accessible
WORD 0x...                      --Garbage collector map
WORD Rectangle.%anc             --Pointer to ancestor table
WORD Rectangle.%string          --Pointer to class string
...                             --Inherited methods will be here
WORD Rectangle.GetArea          --Pointer to bytecode for GetArea
WORD Rectangle.GetPerimeter     --Pointer to bytecode for GetPerimeter

DEFINE Rectangle.%anc           --Creates ancestor table
WORD 2                          --Number of items in ancestor table
WORD Rectangle.%desc            --Pointer to Class descriptor
...                             --Rest of ancestors
```

Key information about the class is stored at fixed offsets from the descriptor address. The components are: the GC map, which tells the garbage collector that it has properties that are pointers, the ancestors table that contains pointers to all the class descriptors for Rectangle's ancestors, a string for "Rectangle", that is used in printing (more on that later), and the virtual method table (vtable) – the list of pointers to all of Rectangle's methods.

Given this, if wanted to call a method that belongs to Rectangle, we could do:

```
GLOBAL Rectangle.%desc    --Get the address of the descriptor
CONST 28                  --Add 28 to that address to get the address
OFFSET                        of the Rectangle.GetArea method
LOAD                      --Load from that address, so the method
                              address is on the stack
CALLW 1                   --Call the method
```

The above code is not the exact sequence used to call methods, but it is useful for demonstrating how the vtable works. The class descriptor does not differentiate between static and non-static methods. The differences between them come about in how they are called and their arguments. This will be discussed more at a later point.

Objects are records, created on the heap, and contain all of the objects properties as well as a pointer to the class descriptor. Creating objects is done using the Keiko library function `lib.newgc` which in turn calls a C function that at allocates a block of memory of the correct size and places the class descriptor as the first item in the record. If we have the following code:

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        r : Rectangle = New Rectangle.
        r->width = 5.
        r->height = 7.
        Return.
    <$
<$

Define Rectangle
$>
```

```
    Properties
    $>
        width: Int,
        height: Int,
    <$

    Method GetArea {} => Int
    $>
        Return My->width * My->height.
    <$

    Method GetPerimeter {} => Int
    $>
        Return 2 * (My->width + My->height).
    <$
<$
```

The memory at the end of the program would look something like this:

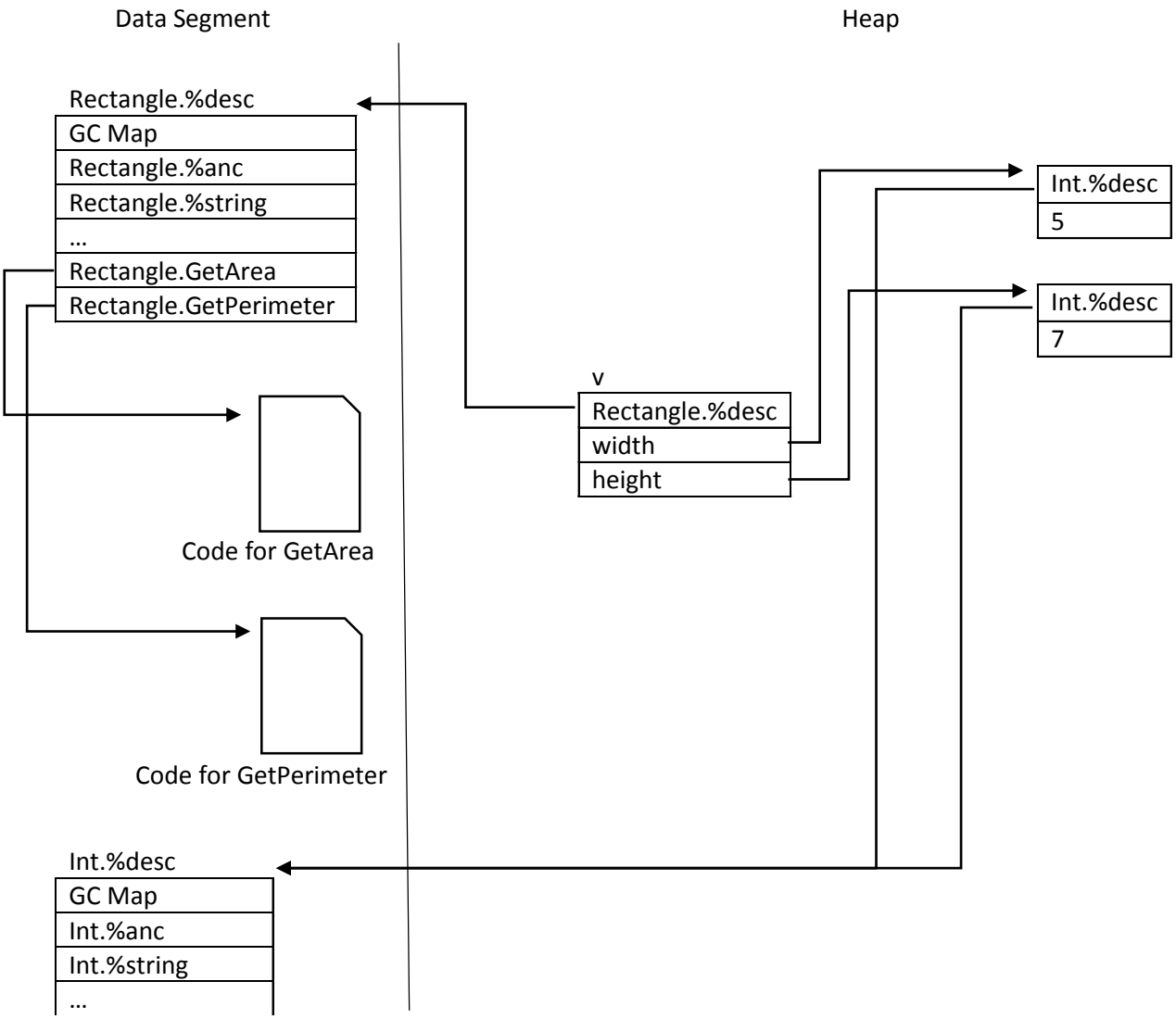

Figure 1

The reasons for this design are two-fold. Firstly, it means that objects use up little space, they only need 4 bytes of overhead. It is likely that in any program there are going to be many more objects than there are classes, so making classes larger in exchange for lightweight objects is worthwhile trade-off. Secondly, it allows for dynamic binding of methods. The method called at runtime is determined exclusively by the value of the class descriptor in the object. If the objects did not have a class pointer, and all methods were called directly by their name, then we could not have dynamic binding. As such, the four bytes of overhead is a small cost for such an important feature. Dynamic binding is expanded upon in a later section.

In the code generation, if we have an expression of the form `New class,` we generate the following code using the following function:

```
and gen_expr e =
    match e with
    ...
    | New n ->
        SEQ [
          (* get the address of the class descriptor *)
          get_name n;
          (* need space for all the properties + class descriptor *)
          CONST (4 + (get_size n));
          (* Call the make proceduure *)
          GLOBAL "baozi.%make";
          CALLW 2;
        ]
```

The code generated from this calls a procedure `baozi.%make`. This procedure creates the object using `lib.newgc` and fills all the objects properties with Nil.

Methods and Dynamic Binding

When calling a static method, we simply get the address of the method – which is globally accessible – and call the method.

```
and gen_static_call expr meth args =
  match expr with
  | Name n ->
      SEQ [
        (* evaluate that arguments *)
        SEQ (List.map gen_expr (List.rev args));
        (* get the address of the method *)
        GLOBAL (n.x_name ^"." ^ meth.x_name);
        (* Call the method *)
        CALLW (List.length args)
      ]
  | _ -> raise IncorrectSyntaxError
```

If the method is not static, then follow the class descriptor pointer – the first item in the object record – and then offset by a fixed amount to find the pointer to the desired method. The amount that we need to offset by was annotated on the method name in the annotation phase of the compiler. In addition, the receiving object is passed as an argument to the object.

```
and gen_name_addr n =
  match n.x_def.d_kind with
  ...
  | MethodDef (off, _) -> SEQ [CONST off; OFFSET]

and gen_call expr meth args =
  match expr with
  ...
  | _ ->
      SEQ [
        (* evaluate the arguments *)
        SEQ (List.map gen_expr (List.rev args));
        (* evaluate the calling object *)
        gen_expr expr;
        (* duplicate, as the receiving object is also an argument *)
        DUP 0;
        (* Load the class descriptor from the object *)
        LOADW;
        (* find the offset for the method *)
        gen_name_addr meth;
        (* load the method *)
        LOADW;
        (* call the method *)
        CALLW (List.length args + 1)
      ]
```

When inheriting methods, the inherited methods will be given the same index in the vtable for the subclass as in the vtable for the parent class. This means that the code that calls methods can be type-agnostic – the type of the receiving object at runtime does not matter to how the method is called. This allows for a key object-oriented feature – dynamic binding. With dynamic binding the method called is determined at run time, not compile time. The following demonstrates this feature:

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        !! a is a type of A !!
        a : A = New A.
        a->SayHello<-{}.

        !! Set a to be a type of B !!
        a = New B.
        a->SayHello<-{}.

        Return.
    <$
<$

Define A
$>
    Method SayHello {} => {}
    $>
        Output->StringLn<-{"Hello! I am an A!"}.
```

```
        Return.
    <$
<$

Define B <- A
$>
    ReplaceMethod SayHello {} => {}
    $>
        Output->StringLn<-{"Hello! I am a B!"}.
        Return.
    <$
<$

--> Hello! I am an A!
    Hello! I am a B!
```

B inherits from A and replaces the SayHello method with something new. When a is set to a type of A, it runs A's version of SayHello, when a is then set to a type of B, it runs B's SayHello.

For non-static methods, the receiving object is added as the first argument to the function. This can be referenced in the method body with the use of the Me/My keyword. In the annotation phase, non-static methods, have an additional argument added to them, with a name "Me". The keywords Me and My are translated by the parser into a normal variable reference for "Me".

```
let create_me cls =
  Prop({x_name="Me"; x_def=(create_def NoneKind VoidType)}, TempType
   (Ident cls.c_name.x_name))


let modify_non_static c =
  let modify m =
    if not m.m_static then m.m_arguments <- (create_me c) :: m.m_arguments
    else ()
  in List.iter modify c.c_methods
```

Inheritance

Inheritance is the ability for a class to copy the methods and properties of a parent class into a subclass as well as for instances of the subclass to be considered as being of the same type as the parent class. In Baozi a class can inherit from a single other class by use of the <- operator. A subclass can also override methods that they inherit from their parent, replacing the method with a new method that has that same name. In Baozi this can be done by flagging a method with ReplaceMethod. For simplicity, Baozi does not have the concept of a virtual method and all methods can be overridden.

In the compiler, inheritance is done by the annotation phase. A class will inherit all of its parent's methods and properties. It is ensured that inherited methods will have the same offset in the parent's vtable as they do in the subclass' vtable. Methods that replace an inherited method will also have the same offset in the vtable as the method they are replacing. This ensures type-agnosticism in the method calling. Properties work in a similar way, but without the complication of overriding. The properties of the parent class are prepended onto the list of properties of the subclass. This

ensures that if a property has a given offset in an object of the parent type, the same property will have the same offset in an object of the subtype. This ensures type-agnosticism for property access.

When a method is replaced, the type checker will ensure that the new method is compatible with the one it is replacing. Compatibility between two methods m_1 and m_2 (where m_1 is replacing m_2) is defined using two criteria:

- Contravariance of arguments: m_1 and m_2 have the same number of arguments and the type of each argument of m_1 is a supertype of the corresponding argument of m_2
- Covariance of result: The return type of m_1 must be a subtype of the return type of m_2

This specification is consistent with the semantics of functional types described in (Cardelli, 1988). However, the issue of whether contravariance of arguments is better than covariance of arguments is an ongoing debate (Castagna, 1995), (Ducournau, 2002). A investigation of this issue however was beyond the scope of this project.

When inheriting methods without replacing, the methods are tagged as `Inherited of string,` where the `string` is the name of the class that's being inherited from, as opposed to `Mine` if they are new or replacing. When placing pointers to methods in the class' vtable, inherited methods will point the original.

A class' ancestors also form part of the class descriptor that is stored in the data segment. Each class descriptor has an entry `classname.%anc` that points to memory containing the number of ancestors followed by all the class descriptors for itself and the class' ancestors. For example, `Define B <- A` would produce:

```
DEFINE B.%desc
WORD 0x...
WORD B.%anc
WORD B.%string
...

DEFINE B.%anc
WORD 3
WORD B.%desc
WORD A.%desc
WORD Object.%desc
```

This ancestors table is used in the `InstanceOf` operator and in run-time checks for downcasts.

The `InstanceOf` operator and the `GetType` methods, are two ways of conditioning on the concrete type of an object at runtime. `x->GetType<-{}` gets the concrete type of the object, while `x InstanceOf TypeOf A` will return true if `x`'s type is a subclass of `A`.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        a : A = New A.
        b : B = New B.

        If a->GetType<-{} == TypeOf A Then Output->StringLn<-{"a has type
            A."}.
```

```
        If b->GetType<-{} == TypeOf A Then Output->StringLn<-{"This will
            not print."}.
        If b InstanceOf TypeOf A Then Output->StringLn<-{"b has type B, a
            subtype of A"}.
        Return.
    <$
<$

Define A
$>
<$

Define B <- A
$>
<$


--> a has type A
    b has type B, a subtype of A
```

The line `a->GetType<-{}` returns an object of type Type which contains the class pointer stored in a's object record. `TypeOf A` returns a similar Type object containing a pointer to `A.%desc`. The `==` operator checks if these two Type objects contain the same class pointer. The `InstanceOf` operator in `b InstanceOf TypeOf A` searches through b's ancestor table to find an entry that matches `A.%desc`.

When doing a downcast, a check must be performed at runtime to ensure that the type of the object you are casting is compatible with the type you are casting to. Baozi does this type check by calling a method that searches through the object's ancestor table to find an entry that matches the class descriptor of the type it is being cast to. If it cannot find a match it will throw a runtime error.

Arrays

Baozi allows you to define a variable as an array of any type. This is done using the `Array Of` notation.

```
v : Array Of Int = New Array of Int<-[5]
```

In the annotation phase of the compiler, array types are annotated with a type `ArrayClassType(d)` where `d` is the type of the elements in the array (in this example, `Int`). This is used in the various stages of the compiler in determining the type of sub (`->[i]`) expressions.

Like everything in Baozi, arrays are objects. They are created on the heap and have a class pointer that points to their class descriptor.

The result of the code `v : Array Of Int = New Array of Int<-[5].` would, in memory, look like:

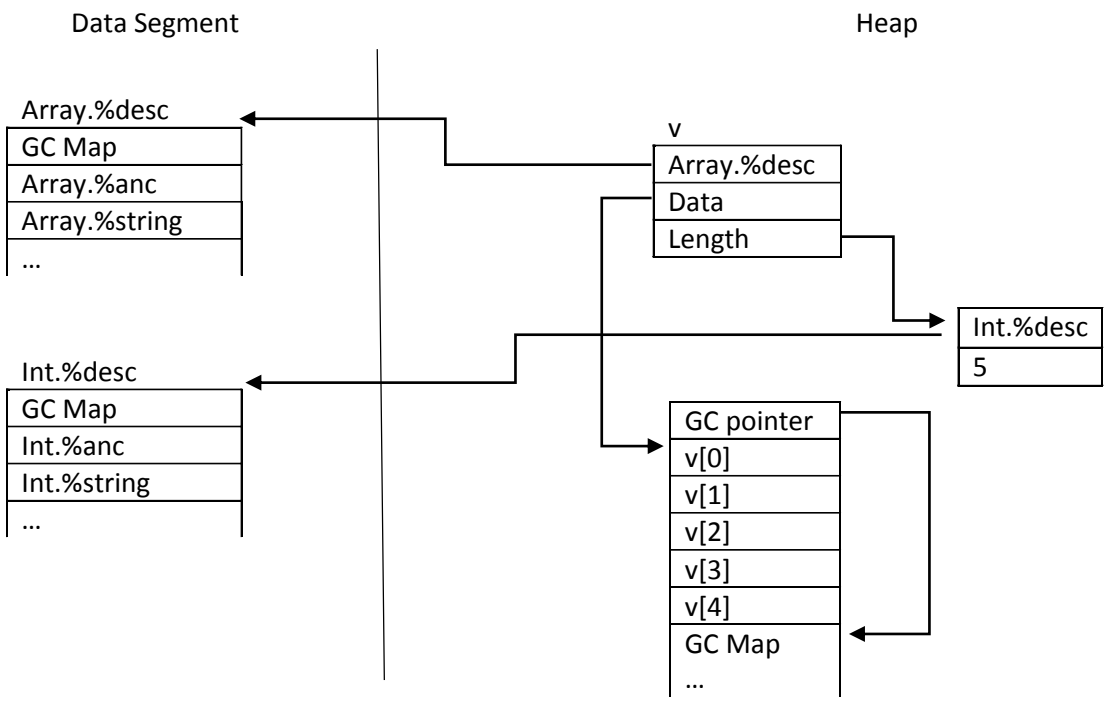

Figure 2

There are a couple reasons I went with this design. Firstly, the goal for Baozi was to make a pure object-oriented language, therefore arrays must be objects too. In the above example, v looks very similar to an object, the only difference is that it has the array data as a "property". Splitting the array object and the array data also makes it easier to compute the garbage collection maps, as there can be a separate one for the array object, and for the array data. Finally, having the "Data" pointer point to the first element of the array makes it very easy to get the address of an element given an offset, especially considering that Keiko has special instructions for popping a value and loading/storing from an offset four times that value.

The library class Array serves as the parent class for all arrays and contains the Length property which all arrays inherit. The `Array.%desc` class descriptor serves as the class descriptor for all array objects.

When creating an array, a procedure, `baozi.%makeArray` is called which allocates the correct amount of space, puts in the length, fills the array with Nils, and creates the GC map.

To access an element of an array, using the sub notion `v->[i]` we follow the pointer at "Data" in the object and then offset by 4i, to find the correct address of the element.

```
and gen_addr expr =
  match expr with
  ...
  | Sub (e1, e2) ->
      SEQ [
      (* Get the array that we are accessing *)
      gen_expr e1;
      (* Get the "Data" *)
      CONST 4;
      OFFSET;
      LOADW;
```

```
        (* Get the index *)
        gen_expr e2;
        unbox;
        (* The element we want is at offset 4*i *)
        CONST 4;
        BINOP Times;
        OFFSET;
      ]
```

A problem that is encountered when doing arrays is the question of how they interact with subtypes. It is natural to want arrays to be covariant, that is to say, that if A is a subtype of B, then Array of A is a subtype of Array of B. However, a problem occurs when you assign something to an index in the array.

Consider the situation where you have classes A, B, and C where B and C are subclasses of A. Let's say you create an Array of B. Now, we put this array in a context where is considered to be a type of Array of A. Since C is a subtype of A, we can add an object of type C to the array, however, the array is really a type of Array of B, and an object of type C should not be allowed in. In Java this problem is solved by having a type check done at run-time to ensure that the type of the object going into the array has a compatible type with the concrete type of the elements of the array. (Thorup, 1997)

In Baozi, we could implement a similar solution. As part of the semantic analysis, the compiler could create a list of all the array types mentioned in the program. For each of these array types, the compiler would create class descriptors. The type of the elements in the array would be stored at a fixed offset in the class descriptor. For example, if somewhere in the program we have `New Array of Int<-[3].` , this would cause the following class descriptors to be made:

```
! Descriptor for Array of Int
DEFINE array.%Int.%desc
WORD 0xF
WORD array.%Int.%anc
WORD array.%Int.%string
WORD Int.%desc
...

! Ancestor table for Array of Int
DEFINE array.%Int.%anc
WORD 3
WORD array.%Int.%desc
WORD array.%Object.%desc
WORD Object.%desc

! Descriptor for Array of Object
DEFINE array.%Object.%desc
WORD 0xF
WORD array.%Object.%anc
WORD array.%Object.%string
WORD Object.%desc
...

! Ancestor table for Array of Object
DEFINE array.%Object.%anc
WORD 2
```

```
WORD array.%Object.%desc
WORD Object.%desc
```

These class descriptors would then serve as the class descriptor for the array objects. When you assign something to an index in the array, the same procedure that type checks cast expressions could be called on the object you are adding and the class descriptor for the element type of the array. This procedure would return an error if the object you are trying to add is not compatible with the type of the elements in the array.

However, there is a significant trade-off if we go with this approach. The way that generics are implemented is Baozi is as a form of syntactic sugar, meaning that on the level of machine code, there is no distinction between `List With (Int)` and `List With (String)` – The class pointers for both objects point to the class descriptor for `List`. This could lead to the follow situation:

```
lsa : Array Of List With (String) = New Array Of List With (String)<-[4].
oa : Array Of Object = lsa.
li : List With (Int) = New List With (Int).
li->value = 3.
oa->[1] = li.               -- This will pass the run-time type check
                                 as both oa->[1] and li have a type List
s : String = lsa[1].value.  -- Error: Value is really and Int
```

Therefore, given our implementation of generics, if we wanted to have covariant arrays, we would have to disallow arrays of generics. This is largely the decision that Java makes, although it is complicated by the existence of wildcards (Bracha, 2004). In this report, however, we talk about the version of the compiler that has arrays as invariant. This is also the version of the compiler laid out in Appendix A. However, in Appendix B I lay out the necessary changes to make arrays covariant with the limitations as described above.

Generics

Generics is the ability to create classes that have “type variables” that are assigned when an object of that class is declared. For example:

```
Define List Using T
$>
    Properties
    $>
        value : T,
        next : List With (T),
    <$
<$
```

Defines a new class, List, which has one type variable T. This T is then used as a type within the body of the class. In this example, value is defined as being of type T. When a type of list is later declared, such as in `l : List With (Int) = New List With (Int).`, the T, for that object, is set. In this is example, T is set to `Int`, meaning that `l->value` would have type `Int`.

Generics are largely the work of the semantic analysis phase and are not represented in the Keiko code. You can think of the above example as being syntactic sugar for:

```
Define List
$>
```

```
    Properties
    $>
        value : Object,
        next : List,
    <$
<$
```

As the Keiko code that results would be identical. This decision was made as it vastly simplifies the implementation of generics. One downside of this approach, as noted earlier, is that if we wanted to have covariant arrays, we would not be able to have arrays of generics. Another downside is that you cannot cast to generic types. Otherwise, you could have a situation like this, where `Collection` is a supertype of `List Using T`.

```
il : New List With (Int) = New List With (Int).
il->value = 5.
ic : Collection = il.
sl : List With (String) = Cast(il)->List With (String)
  -- This would pass the run-time type check as List with (String) has a
  concrete type List, as does List With (Int)
s : String = sl->value              -- Error: value is really an Int
```

Generic classes are defined with the `Using` keyword, followed by a list of generic type variables. The type variables can also employ the `<-` operator to give a restriction on the types that are allowable. `T <- Person`, would declare that the type supplied to T must be a subtype of `Person` (assuming `Person` is a type that has been defined). This also makes `Person` the most general type of T.

In the annotation phase, before the members of a generic class are annotated, the type variables are added to the environment along with their most general type. The most general type for that type variable and is what is used when determining what methods and properties can be referenced. For example, in the following, T has a most general type of Object:

```
Define List Using T
$>
    Properties
    $>
        value : T,
        next : List With (T),
    <$

    Method ListMethod {} => {}
    $>
        My->value->DoSomething<-{}.
        !! Invalid as Object does not have a method DoSomething !!
    <$
<$

Define A
$>
    Method DoSomething {} => {}
    $>
        ...
    <$
<$
```

However, if we require that T is a subtype of A, then the most general type of T becomes A:

```
Define List Using T <- A
$>
    Properties
    $>
        value : T,
        next : List With (T),
    <$

    Method ListMethod {} => {}
    $>
        My->value->DoSomething<-{}.
        !! Valid, as A does have a method DoSomething !!
    <$
<$

Define A
$>
    Method DoSomething {} => {}
    $>
        ...
    <$
<$
```

When creating an object of a generic class, it is annotated with the type `GenericClassType (class, type list)` where `class` references the generic class and `type list` is the list of all concrete types assigned to the type variables. The type checker will ensure that all the types supplied meet the restrictions for the type variables. In the case where a class inherits from a generic class, the type checker will also ensure that the types given meet the restrictions for the type variables of the generic class. When refencing a method or property from the generic class that has a type variable as its type, type checker works out what the actual type is using the type of the receiving object and the name of the type variable.

The compiler also will work if the most general type for a type variable is another type variable, for example: `Define Gen Using T, U <-T`, defines a generic class that takes two types, and requires that the second type is a subtype of the first.

As with arrays, an issue comes up regarding the subtyping of generic classes. If A is a subtype of B, is List With (A) a subtype of List With (B)? One might hope that we could implement a similar solution to the one we suggested for arrays. However, we run into a number of difficulties. Firstly, since a generic can have any number of type variables, and any of them can be used in any capacity in a property or method, we would have to run a series of type checks every time we assigned to a property or called a method that used the type variables. This would increase the running time. A second problem would be that since generics can have many type variables, the number of ancestors they can have is must larger, as all of their type variables will have parents. For example, consider `Collection with (A, A, A)` where `A` has a parent `B`. This would have the ancestors: `Collection With (A, A, B)`, `Collection With (A, B, A)`, etc. The number of ancestors for a generic would be exponential in the number of type variables. There are some solutions to this problem, java for example has a version of generic subtyping using wildcards (Cameron, Drossspoulou, & Ernst, 2008), and other more general solutions have been proposed for other

object-oriented languages (Zólyomi, Porkoláb, & Kozsik, 2003). However, implementing these solutions were beyond the scope of this project. Therefore, generics in Baozi are invariant. This invariance can be seen in the way that the type checker checks type compatibility.

```
(* Check to see if two types are compatible with each other *)
let check_compatible type1 type2 =
  let pt1 = ref VoidType and pt2 = ref VoidType in
    (* Check strict compatibility between types *)
    let rec check_strict t1 t2 =
      match (t1, t2) with
      | (ClassType c1, ClassType c2) ->
          if c1.c_name.x_name == c2.c_name.x_name then () (* Classes must
                                                            be identical *)
          else raise (TypeError((print_type t1), (print_type t2)))
      ...
    let rec check t1 t2 =
      match (t1, t2) with
      ...
      | (GenericClassType (c1, ts1), GenericClassType (c2, ts2)) ->
          check (ClassType c1) (ClassType c2);
          (* Check the classes are compatible *)
          begin
            (* Then check that the type arguments are the same *)
            try List.iter2 (fun (_, x) (_, y) -> check_strict x y) ts1 ts2
            with Invalid_argument _ -> raise (TypeError((print_type !pt1),
                                                (print_type !pt2)))
          end
      ...
      | _ -> raise (TypeError((print_type !pt1), (print_type !pt2)))
    in pt1 := type1; pt2 := type2; check type1 type2
```

Garbage Collection

For memory management, Keiko uses an automatic stop and copy garbage collector (GC). The GC starts with the assumption that nothing is a pointer, and so all pointers must be identified to the GC, to ensure reference checking. If this is not done, we risk running out of memory or crashing the program. The GC use 32-bit bitmaps to indicate what is and isn't a pointer. The 32-bit integers, when viewed as an array of bits is treated as a list of flags. A flag is set to 1 if the corresponding word in memory contains a pointer. The least significant bit is used to determine if the map is empty – this is called the bitmap flag.

The GC uses maps in three places: object maps which indicate the pointers among object properties, frame maps for identifying the pointers among the parameters and local variables in a procedure, and finally stack maps which identify pointers on the evaluation stack when a procedure is called that are not covered by one of the two previous types of GC map.

Since everything in Baozi is an object, all instance variables are pointers to objects. Therefore, the GC maps can be very straight forward. Object maps are stored as the first word in the class descriptor for the object and consist of an integer representing a string of 1s, the length of the number of properties (plus the bitmap flag). For example, a class with three properties would have the object map, 15 ≡ 111|1. A class with seven properties would have an object map 255 ≡ 1111111|1.

Frame maps are a little more complicated and have the following structure:

Params	Frame head	Locals	Bitmap flag
000000000000	000	0000000000000000	1

A method with 4 locals and 3 parameters would have a GC map of 7462913 ≡ 111|000|1111000000000000|1.

Stack maps track pointers that are on the evaluation stack when a method is called but have not been covered by another GC map. For example, consider the following code fragment `A->DoSomething<-{New A, New B}`. This would be translated (initially) to the Keiko code:

```
CONST ...
GLOBAL A.%desc
GLOBAL baozi.%make
CALLW 2             --First call statement
CONST ...
GLOBAL B.%desc
GLOBAL baozi.%make
CALLW 2             --Second call statement
GLOBAL A.DoSomething
CALLW 2
```

At the second call statement, the stack consists of: The address of `baozi.%make`, the address of `B.%desc`, a constant, and the pointer to the new object of type A. The first thing on the stack is the address of the procedure and this does not concern us. The second thing on the stack is the address of `B.%desc` and is a parameter for the `baozi.%make` procedure and therefore will be covered by the frame map for the `baozi.%make` procedure . The third thing on the stack is a constant and is a parameter for `baozi.%make`. However, the fourth thing, the pointer the new object of type A, is not involved in the procedure call and thus not covered by the frame map of the `baozi.%make` procedure. Therefore, we need to have stack map of 9 ≡ 100|1 right before the second call statement to ensure the pointer to the new object of type A is correctly identified to the GC. The resulting Keiko becomes:

```
CONST ...
GLOBAL A.%desc
GLOBAL baozi.%make
CALLW 2
CONST ...
GLOBAL B.%desc
GLOBAL baozi.%make
STKMAP 0x9             --Stack map
CALLW 2
GLOBAL A.DoSomething
CALLW 2
```

The way this is handled in the compiler is to simulate the evaluation stack for the Keiko generated by each expression and inserting the correct `STKMAP` expressions where needed.

Due to the fact that arrays have a dynamically allocated size, their GC maps need to be more complicated. The GC maps for arrays are stored with their elements and have the following structure:

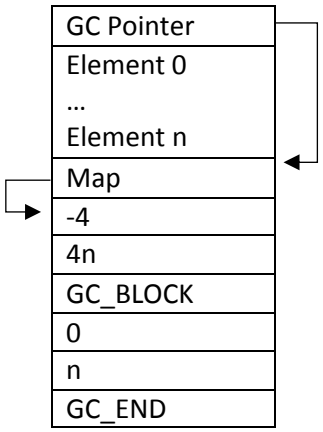

Figure 3

The garbage collector requires that the first word in any heap allocated record is a pointer to the GC map. A quirk of the garbage collector further requires that the first word of the GC map for dynamically sized objects is another pointer. When garbage collection happens, all heap allocated memory gets moved, therefore internal pointers such as GC Pointer and Map must be identified to the GC so that they can be updated. The words -4 and 4n tell the GC that there are two pointers at offset -4 and 4n from the first element of the array. GC_BLOCK is an integer flag that tells the garbage collector that we are starting a GC map for a block. The 0 and n words tells the GC that the block of pointers starts at offset 0 from the first element of the array and contains n pointers. The GC_END flag signals the end of the GC map. If we have `v : Array Of Int = New Array Of Int<-[3]`, the result in memory would look like:

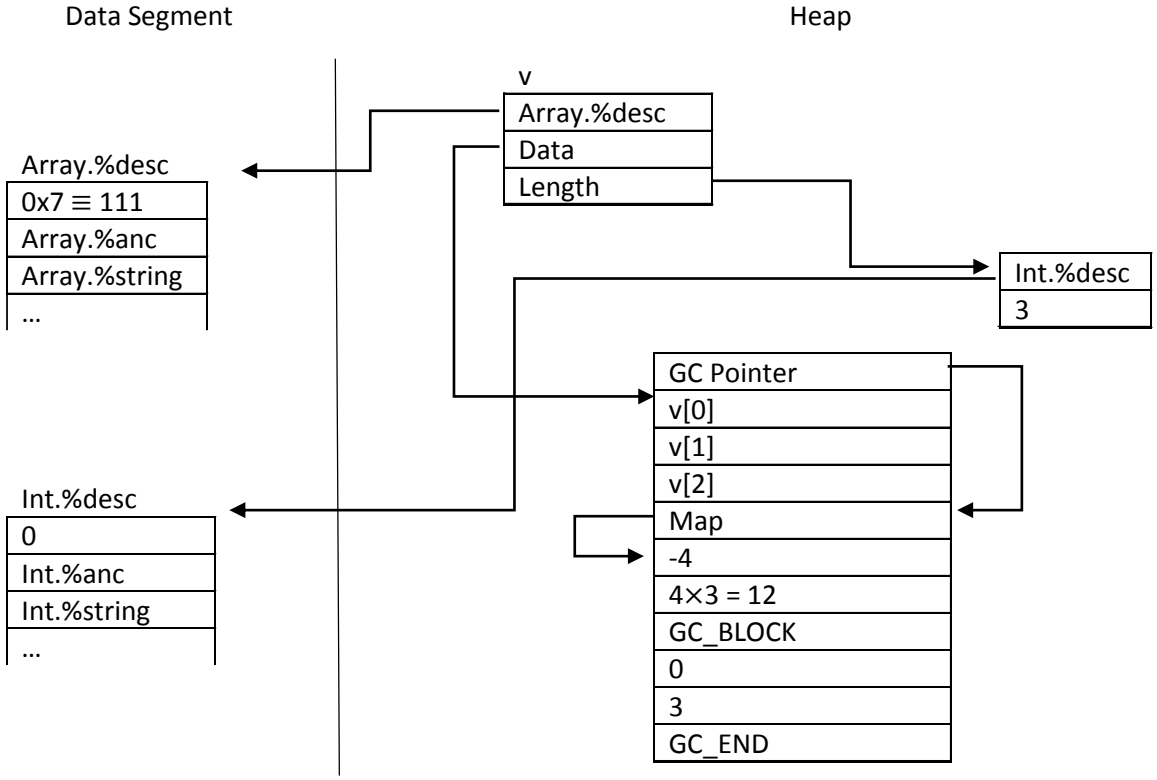

Figure 4

Primitive values and Library

As we have noted several times, everything in Baozi is an object. Therefore, even primitive values such as integers and Booleans are treated as objects. This done by "boxing" primitive values. A primitive in Baozi is in fact, a pointer to a heap allocated block of memory containing a class pointer to the class descriptor for the primitive type, and the actual primitive value. This the only place in which an object contains something other than a pointer. The primitive classes Object, Array, Type, Int, Bool, and String, are provide as libraries for every Baozi program. In addition, there is a procedure `baozi.%makePrim` which takes two arguments, a pointer to a primitive class descriptor and a value, and creates a boxed primitive. Therefore, a constant, such as `5`, is translated to the Keiko:

```
CONST 5
GLOBAL Int.%desc
GLOBAL baozi.%makePrim
CALLW 2
```

Since all primitive values are objects, operations on primitive values are in fact method calls. For example: `5 + 4` is syntactic sugar for `5->add<-{4}` and would be translated to:

```
CONST 4
GLOBAL Int.%desc
GLOBAL baozi.%makePrim
CALLW 2                --Create the 4

CONST 5
GLOBAL Int.%desc
GLOBAL baozi.%makePrim
STKMAP 0x9
CALLW 2                --Create the 5

DUP 0
LOADW                -- Load the class descriptor for Int
CONST 28             -- Get the address of Int.add
OFFSET
LOADW
CALLW 2              -- Call the add method
```

The library classes for Keiko contain methods for all primitive operations. These primitive methods take in a number of arguments, unbox their values, perform the operation, and then create a new primitive with the resulting value. This ensures that primitive types behave as though they are passed by value, even though they are passed by pointer. The Keiko for the add method is:

```
PROC Int.add 0 0 0x300001
LDLW 12                        -- Load first argument
LDNW 4                         -- Unbox
LDLW 16                        -- Load second argument
LDNW 4                         -- Unbox
PLUS                           -- Add together
GLOBAL Int.%desc
GLOBAL baozi.%makePrim   -- Create a new integer
CALLW 2
RETURNW                  -- Return the new integer
```

```
END
```

The peephole optimiser will optimise away situations where we immediately unbox an integer after boxing it, so

```
CONST 16
GLOBAL Int.%desc
GLOBAL baozi.%makePrim
CALLW 2                         -- Create boxed integer
LDNW 4                          -- Unbox
```

Would become:

```
CONST 16
```

The class Object is another library class that serves as an ancestor class for all classes. Any class that does not have an explicit parent class will have Object as their parent class. The Object class has four methods that are inherited by all other classes: equals, GetType, InstanceOf, and Print.

The equals method is a method that checks equality between two objects. The operator == is syntactic sugar for calling the equals method. By default, the method returns true if the objects (which are pointers) are pointers to the same address. In this way we can consider the identity of an object to be given the value of the pointer to the object record.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        c1 : Circle = New Circle.
        c1->radius = 7.

        c2 : Circle = New Circle.
        c1->radius = 7.

        If c1 == c2 Then
        $>
            Output->StringLn<-{"This will not print"}.
        <$
        Return.
    <$

<$

Define Circle
$>
    Properties
    $>
        radius : Int,
    <$
<$

-->
```

The expression `c1 == c2` is syntactic sugar for `c1->equals<-{c2}` which will call Object's equals method, which will return false as `c1` and `c2` are different pointers. Like all inherited methods, however, equals can be overridden. This is what the integer class does, to cause the == operator to return true if the integers have the same value. It is also possible to override this equals method in created classes:

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        c1 : Circle = New Circle.
        c1->radius = 7.

        c2 : Circle = New Circle.
        c1->radius = 7.

        If c1 == c2 Then
        $>
            Output->StringLn<-{"This will print now!"}.
        <$
        Return.
    <$

<$

Define Circle
$>
    Properties
    $>
        radius : Int,
    <$

    ReplaceMethod equals {o : Object} => Bool
    $>
        circ : Circle = Cast(o)->Circle.
        Return circ->radius == My->radius.
    <$
<$

--> This will print now!
```

GetType is a method that returns the type of an object while InstanceOf is the method called in expressions of the form `x InstanceOf TypeOf y`. The use of these has been discussed in a previous section. Finally, we have the Print method. By default the print method prints the string stored at classname.%string in the class descriptor. This string is set by the code generator to be the string of the class's name.

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        v : Vector = New Vector.
```

```
        v->Print<-{}
    <$
<$

--> Vector
```

Therefore, when creating the class descriptor, a class, the code generator will also create:

```
DEFINE classname.%string
STRING ...00
```

To store the name of the class that will be used in the default print method.

Nil is also provided as part of the library functionality. Nil is not a class, but in fact a specific object created on the data segment rather than on the heap:

```
DEFINE Nil
WORD Object.%desc
```

In terms of typing, Nil is given a special type by type semantic analyser, `NilType`. Any object can be set to Nil, but Nil cannot be assigned anything. When an object is set to Nil, it is set to point to the globally accessibly Nil pointer. In terms of calling methods on Nil, it is considered an Object, and thus has access to the four methods that are defined in Object. This includes the `equals` method, and thus expression like `Nil == x` ≡ `Nil->equals<-{x}` are valid, and would return true, if x is the same pointer as Nil – i.e. if x has been set to Nil.

Performance Comparison

To look at the performance of Baozi, I wrote the same program in Baozi, Java using standard primitive values, Java using boxed primitive values, picoPascal using standard primitive values, picoPascal where primitives are boxed, and Scala. Java is interesting to look at because it allows you to distinguish between boxed and unboxed primitive values. picoPascal is relevant because it also compiles to Keiko and thus it can use the same Keiko interpreter to run. Scala is interesting because it is also a pure object-oriented language and it runs on the JVM, same as Java (Odersky, et al., 2004). The program in question finds 2^n in a way that runs in time $O(2^n)$. The code for all four programs is show below:

Baozi

```
Define Program
$>
    ClassMethod Main {} => {}
    $>
        x : Int = 10.
        r : Int = Program->twoPow<-{x}.
        Output->String<-{"Result: "}.
        Output->IntLn<-{r}.
        Return.
    <$

    ClassMethod twoPow {x : Int} => Int
    $>
```

```
        If x == 0 Then Return 1.
        Else Return Program->twoPow<-{x - 1} + Program->twoPow<-{x - 1}.
    <$
<$
```

Java (unboxed)

```
class TwoPow
{
    public static void main (String[] args)
    {
        long start_time = System.nanoTime();
        int x = 10;
        int r = twoPow(x);
        System.out.print("Result: ");
        System.out.println(r);
        long end_time = System.nanoTime();
        double duration = (double) (end_time - start_time) / 1000000000;
        System.out.println("Time spent: " + duration + " seconds.");
    }

    public static int twoPow (int x)
    {
        if (x == 0) return 1;
        else return twoPow(x-1) + twoPow(x-1);
    }
}
```

Java (boxed)

```
class TwoPowBox
{
    public static void main (String[] args)
    {
        long start_time = System.nanoTime();
        Integer x = new Integer(10);
        Integer r = twoPow(x);
        System.out.print("Result: ");
        System.out.println(r);
        long end_time = System.nanoTime();
        double duration = (double) (end_time - start_time) / 1000000000;
        System.out.println("Time spent: " + duration + " seconds.");
    }

    public static Integer twoPow (Integer x)
    {
        if (x.equals(new Integer(0))) return new Integer(1);
        else return new Integer(twoPow(new Integer(x - new Integer(1))) +
            twoPow(new Integer(x - new Integer(1))));
    }
}
```

Scala

```
object Program {
  def main(args: Array[String]) {
    val start_time : Double = System.nanoTime();
    var x : Int = 10;
    var r : Int = twoPow(x);
    print("Result: ");
    println(r);
    val end_time : Double = System.nanoTime();
    println("Time spent: " + (end_time - start_time)/1000000000 + "
seconds");
  }

  def twoPow(n: Int) : Int = {
    if(n==0) {
      return 1;
    } else {
      return twoPow(n-1) + twoPow(n-1);
    }
  }
}
```

PicoPascal

```
proc twoPow(x : integer) : integer;
begin
  if x = 0 then return 1
  else return twoPow(x-1) + twoPow(x-1)
  end
end;

var x, r: integer;

begin
  x := 10;
  r := twoPow(x);
  print_num(x);
  print_string(" factorial is "); print_num(r);
  newline()
end.
```

PicoPascal (boxed)

```
type
  intbox = pointer to primint;
  boolbox = pointer to primbool;
  primint = record val: integer end;
  primbool = record val : boolean end;

proc buildint(n: integer): intbox;
  var b: intbox;
begin
```

```
  new(b);
  b^.val := n;
  return b
end;

proc buildbool(n: boolean): boolbox;
  var b: boolbox;
begin
  new(b);
  b^.val := n;
  return b
end;

proc plus(x: intbox; y: intbox): intbox;
  var b: intbox;
begin
  b := buildint(x^.val + y^.val);
  return b
end;

proc minus(x: intbox; y: intbox): intbox;
  var b: intbox;
begin
  b := buildint(x^.val - y^.val);
  return b
end;

proc equals(x: intbox; y: intbox): boolbox;
  var b: boolbox;
begin
  b := buildbool(x^.val = y^.val);
  return b
end;

proc twoPow(n: intbox) : intbox;
  var e : boolbox;
begin
  e := equals(n, buildint(0));
  if e^.val then return buildint(1)
  else
    return plus(twoPow(minus(n, buildint(1))), twoPow(minus(n,
buildint(1))))
  end
end;

var x: intbox; r : intbox;

begin
  x := buildint(10);
  r := twoPow(x);
  print_string("Result: "); print_num(r^.val); newline()
end.
```

For Baozi, and PicoPascal, the timing for the program was handled by adding the needed functionality to the Keiko.

A series of tests were conducted. Each program was run with several different values of x. For each value of x, the program was run three times, measuring the running time in seconds, and an average was taken. The results are as follows:

	10	12	14	16	18	20	22	24	26
Baozi (unopt)	0.002	0.007	0.029	0.115	0.453	1.835	7.285	29.527	116.366
Java	0.002	0.003	0.009	0.003	0.004	0.007	0.024	0.067	0.266
Java (Boxed)	0.003	0.010	0.017	0.031	0.038	0.058	0.123	0.352	1.286
picoPascal	0.000	0.001	0.002	0.008	0.032	0.125	0.491	1.988	7.881
picoPascal (boxed)	0.002	0.007	0.027	0.110	0.446	1.774	6.945	27.824	111.295
Scala	0.299	0.314	0.300	0.318	0.290	0.304	0.323	0.361	0.566

Table 1

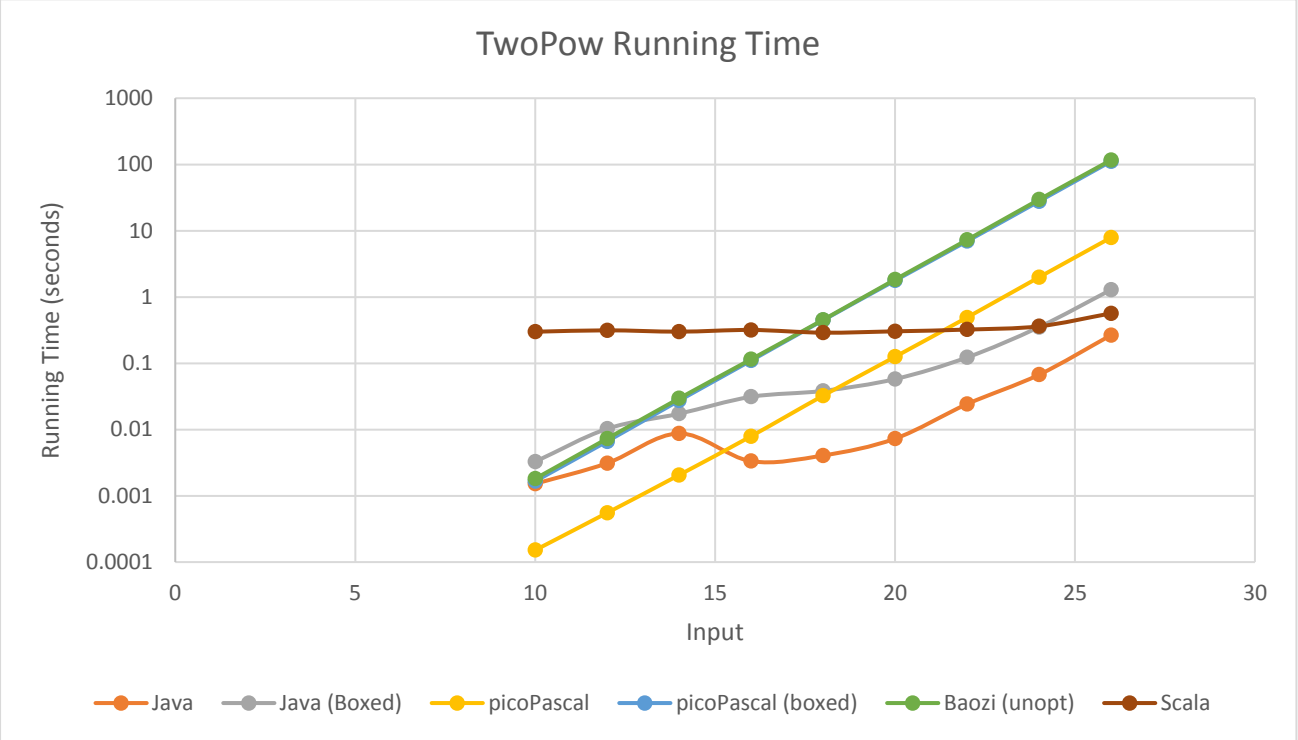

Figure 5

Baozi is by far the slowest, with running time approximately 14.8 times longer than the picoPascal (unboxed) version. The Java implementations are faster again with the boxed version being 90.5 times faster the Baozi at the large inputs and the unboxed 437.5 times faster. The Scala version, at the largest input is 198.6 times faster than Baozi. Interestingly, the Java versions seem slower than one might expect at smaller inputs. This is likely due to Java having more preamble work than Baozi and picoPascal to do before the main work of the program. The Scala version is even more extreme in this regard, with a near constant running time across all the inputs. It is likely that preamble work that Scala has to do is much more extensive that Java, but it also seems to be very good optimising the main body of the program. The running time for the picoPascal (boxed) version is almost exactly identical to the running time for Baozi. This clearly demonstrates that it is the boxing and unboxing of primitive values that causes the increase in running time for Baozi.

One way of improving the running time of Baozi, is to store small integers on the data segment so that they don't have to be created new each time. For constants between -10 and 10, we create constants on the data segment, e.g.

```
DEFINE baozi.%const.%2
WORD Int.%desc
WORD 2
```

We replace all uses of the `baozi.%makePrim` where we are making an Integer, with a new procedure `baozi.%makeInt`. This procedure checks whether the value is between -10 and 10, and if it is, we return a pointer to the constant that is stored in the data segment. Otherwise, we do the normal integer creation using `baozi.%makePrim` . We also adjust the peephole optimiser to replace the code for creating integer constants that are between -10 and 10 with `GLOBAL baozi.%const.%n`. The Boolean constants for True and False are also stored in the data segment. Any code that was originally creating a Boolean primitive using `baozi.%makePrim` now just uses a conditional jump to determine if it should return `GLOBAL baozi.%const.%true` or `GLOBAL baozi.%const.%false`.

When we apply these changes, we get the following running times, for Baozi:

	10	**12**	**14**	**16**	**18**	**20**	**22**	**24**	**26**
Baozi (opt 1)	0.001	0.003	0.013	0.052	0.201	0.802	3.211	12.758	51.108

Table 2

This cuts the running time by more than half – a marked improvement. This clearly demonstrates that it is the allocating of memory of primitive objects that causes the massive increase in the running time for this Baozi programs. It is possible that creating more integers on the data segment could reduce this time even more. However, there is one other thing we could do to improve the running time. When a Boolean expression is used in an if statement, the value of the expression is evaluated to a boxed Boolean, the Boolean value is then immediately unboxed and used for a condition jump. This a waste of instructions and could be simplified.

To implement this, we first require that we place fake Keiko instructions `TYPE s` in our code before each method call. These instructions tell the peephole optimiser the type of the object calling the method. If we see that an object of type Int is calling a method that returns a Boolean (such as equals) and following that is an unbox operation and a condition jump, we replace all the instructions with two unbox instructions and a conditional jump.

```
GLOBAL baozi.%const.%0
LDLW 12
DUP 0
LOADW
LDNW 12           -- Load the equals method
! TYPE Int        -- The TYPE instruction would be here
CALLW 2
LDNW 4            -- unbox
JEQZ 16           -- Jump if equal to 0
```

Becomes,

```
GLOBAL baozi.%const.%0
LDLW 12
LDNW 4
```

```
SWAP
LDNW 4
JNEQ 16             -- Pop two values and jump if they are not equal
```

When we apply these changes, we get the following running times, for Baozi:

	10	12	14	16	18	20	22	24	26
Baozi (opt 2)	0.001	0.002	0.010	0.040	0.157	0.630	2.544	10.146	40.046

Table 3

A slight, but not insignificant improvement over the previous version. This shows that the unboxing of Booleans for conditional jumps is a minor cause in the increased running time for Baozi programs. Put all the data together and we get:

	10	12	14	16	18	20	22	24	26
Baozi (unopt)	0.002	0.007	0.029	0.115	0.453	1.835	7.285	29.527	116.366
Java	0.002	0.003	0.009	0.003	0.004	0.007	0.024	0.067	0.266
Java (Boxed)	0.003	0.010	0.017	0.031	0.038	0.058	0.123	0.352	1.286
picoPascal	0.000	0.001	0.002	0.008	0.032	0.125	0.491	1.988	7.881
picoPascal (boxed)	0.002	0.007	0.027	0.110	0.446	1.774	6.945	27.824	111.295
Scala	0.299	0.314	0.300	0.318	0.290	0.304	0.323	0.361	0.566
Baozi (opt 1)	0.001	0.003	0.013	0.052	0.201	0.802	3.211	12.758	51.108
Baozi (opt 2)	0.001	0.002	0.010	0.040	0.157	0.630	2.544	10.146	40.046

Table 4

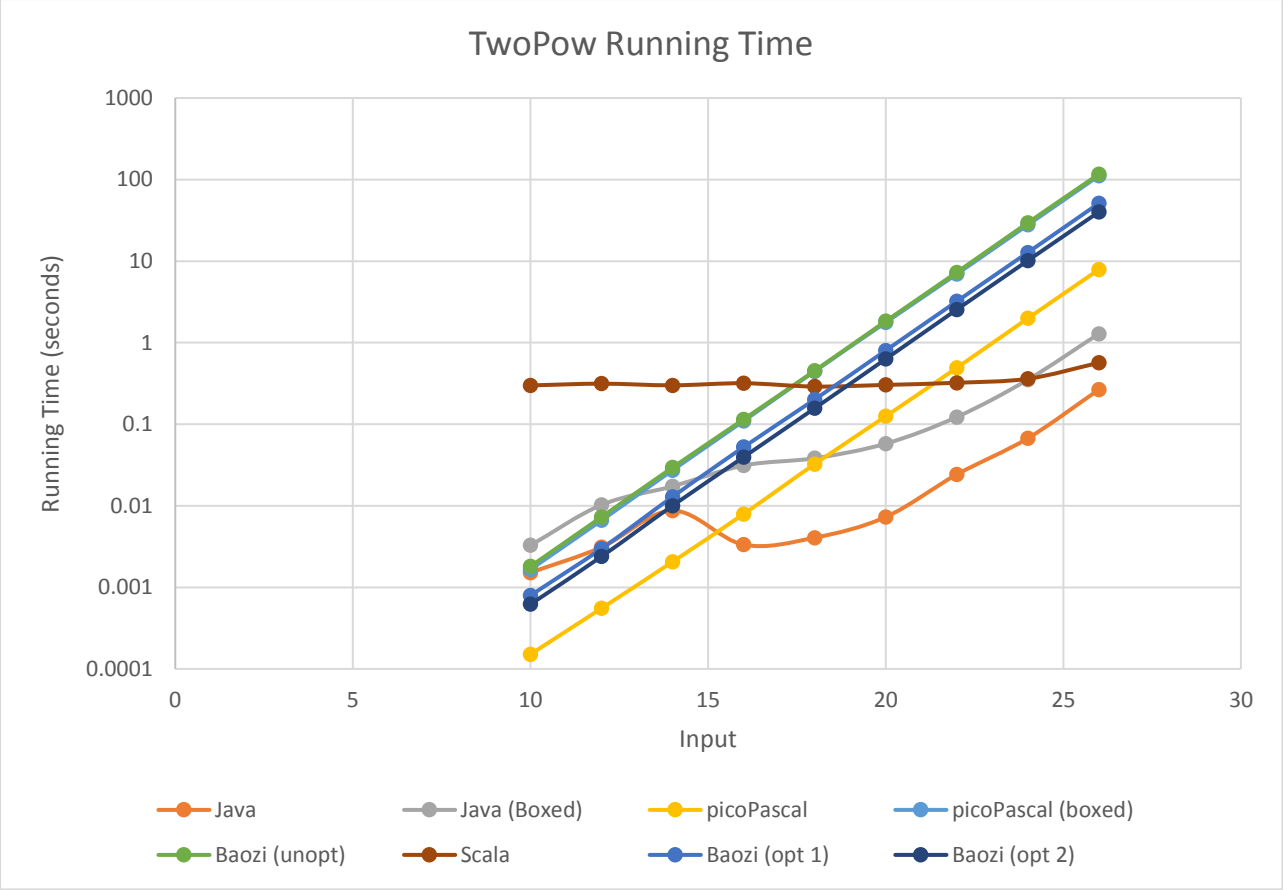

Figure 6

The Java (unboxed) version is about 4.81 times faster than the Java (boxed) version. The unboxed version of picoPascal is about 5.08 times faster than Baozi. This shows is that Baozi is about as optimal for Keiko as the boxed version of Java is for the JVM.

In general, we see that despite a number of optimisations that we were able to do, boxing primitive values leads to a significant increase in the running time for programs. This would indicate that the pure object-oriented approach is not a particularly useful one.

Extension

Due to time limitations, there are a number of desired features that I was not able to add to Baozi.

Access Modifiers

Access modifiers are the ability to restrict where properties and methods can be accessed from. In Java, for example, you can designate a property as “private” meaning that it can not be accessed from outside that class.

To implement this in Baozi, I would need to change the way that I look up methods and properties in the relevant class when they are called. All methods and properties would need to be flagged with their access modifier, and the annotator would have to keep of check of what class it was in and what the class' ancestors are at all times.

Multiple Inheritance

Currently in Baozi you can only inherit from a single class, however allowing multiple inheritance can be very useful in object-oriented programming. Multiple inheritance would allow you to inherit from a variety of classes and create more complicated type relationships.

To do this in Baozi, would require a large overhaul of the type system and require reworking how inheritance is done. To allow for the Liskov substitution principle, the type checker would have to be updated to check all the parents of a particular class. In the annotation phase, all the properties and methods from all of a classes parents would have to be inherited and the system that replaces methods with new methods that have the `ReplaceMethod` flag, would have to check all the inherited methods. The vtable in class descriptors would also need to be changed to support multiple inheritance. If you had `A <- B, C` and `D <- C, E, B`, you would have to ensure that you provide a type-agonistic way of accessing the `B`'s methods (for example) regardless of whether the concrete type of the object was `B`, `A` or `D`.

Overloading

Another extremely useful feature of object-oriented languages is the ability to overload methods. This is the ability to create multiple methods with the same name, but different arguments. The compiler then would work out which method to call depending on the arguments provided.

In order to implement this, I would need to change how the semantic analyser attaches a method to a method call. Currently it only searches through the list of methods until it finds one with a matching name. To implement overloading, it would additionally have to check the types of the arguments provided and find a method that matches them as well.

Performance Improvements

As our performance analysis showed, Baozi is rather slow when it comes to working with primitive types. Further work could be done to improve the performance, in particular reducing the time spent allocating space for new objects, and the amount of time spent unboxing primitive objects.

Making these changes would largely mean adding new optimisation rules to the peephole optimiser. These rules could detect when redundant work is being done and replace the offending Keiko code with something more efficient. It may also be a good idea to do some optimisation earlier in the compiler process, such as pruning the AST to reduce work.

Better Error Reporting

While Baozi does produce compile errors for incorrect syntax and incompatible types, it is unable to provide any help in locating or fixing those errors. For example, missing a "`.`" On the final line before a "`<$`" will result in the error: `Syntax Error: <$` – which is rather unhelpful as it does not tell you where the error can be found. Similarly, if there is a type error, Baozi will only print the names of the conflicting types, and not the line where the error was found.

To fix this would require keeping track at all-times what line of the program the compiler is currently working on and being able to print the offending line in the event of an error.

Conclusion

In conclusion, this compiler achieves the goals that it set out to complete. It is a working pure object-oriented language that implements many of the key features of such languages, such as inheritance, polymorphism and identity. The implementation of the compiler presents a clear transformation of the Baozi code to the Keiko abstract machine.

The comparison with picoPascal, however, tells us that the efficiency trade-offs involved in using a pure object-oriented approach are not worth it, and a hybrid approach that treats primitive types as fundamentally different than objects is the preferred route when implementing this style of programming language.

References

Ambiguity. (2007). In A. V. Aho, M. S. Lam, R. Sethi, & J. D. Ullman, *Compilers Principles, Techniques & Tools* (p. 49). Pearson Education, Inc.

Bracha, G. (2004). Generics in the Java programming language. McGraw-Hill/Osborne.

Cameron, N., Drossspoulou, S., & Ernst, E. (2008). A Model for Java with Wildcards. *European Conference on Object-Oriented Programming*, (pp. 2-26).

Cardelli, L. (1988). A Semantics of Multiple Inheritance. *Information and Computation*, 138-164.

Castagna, G. (1995). Convarience and Contravarience: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 431-447.

Ducournau, R. (2002). ``Real World'' as an Argument for Covariant Specialization in Programming and Modeling. *Advances in Object-Oriented Information Systems* (pp. 3-13). Berlin: Springer Berlin Heidelberg.

Liskov, B. H., & Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 1811-1841.

Odersky, M., Altherr, P., Crement, V., Emir, B., Maneth, S., Micheloud, S., . . . Zenger, M. (2004). An Overview of the Scala Programming Language.

Spivey, M. (2017, November 18). *Keiko instructions in stages (Compilers)*. Retrieved from Spivey's Corner: http://spivey.oriel.ox.ac.uk/corner/Keiko_instructions_in_stages_(Compilers)

Thorup, K. K. (1997). Genericity in java with virtual types. *European Conference on Object-Oriented Programming* (pp. 444-471). Springer.

Zólyomi, I., Porkoláb, Z., & Kozsik, T. (2003). An Extension to the Subtype Relationship in C++ Implemented with Template Metaprogramming. *International Conference on Generative Programming and Component Engineering*, (pp. 209-227).

Appendix A

The core code for the Baozi compiler.

tree.ml

```
open Printf
open Errors
open Keiko

type ident = string

(* Temporary type before annotation *)
and ttype =
    Ident of ident
  | Array of ttype
  | Generic of ident * ttype list

(* Origin of method *)
and origin =
    Mine
  | Inherited of string

(* Language constructs *)
and expr =
    Name of name
  | Sub of expr * expr
  | Nil
  | MethodCall of expr * name * expr list
  | Property of expr * name
  | Constant of int * def_type
  | String of symbol * string
  | TypeOf of name
  | New of name
  | NewArray of name * expr
  | Cast of expr * name
  | Parent

and stmt =
  { s_guts: stmt_guts;
    s_line: int }

and stmt_guts =
    Assign of expr * expr
  | Delc of name * def_type * expr
  | Call of expr
  | Return of expr option
  | Seq of stmt list
  | IfStmt of expr * stmt * stmt
  | WhileStmt of expr * stmt
  | ForStmt of stmt * stmt * expr * stmt
  | Nop

and property = Prop of name * def_type
```

```
and m_method =
  { m_name: name;
    mutable m_type: def_type;
    m_static: bool;
    m_main: bool;
    m_replace: bool;
    mutable m_size: int;
    mutable m_arguments: property list;
    mutable m_body: stmt;
    m_origin: origin }

and c_class =
  { c_name: name;
    mutable c_ptype: def_type;
    mutable c_size: int;
    mutable c_properties: property list;
    mutable c_methods: m_method list;
    mutable c_ancestors: c_class list;
    c_generics: name list }

and program =
  Program of c_class list

(* Kind of a name *)
and def_kind =
  | ClassDef
  | VariableDef of int
  | PropertyDef of int
  | MethodDef of int * bool
  | NoneKind

(* Name definition *)
and def =
  { d_kind: def_kind;
    mutable d_type: def_type}

(* Type of name *)
and def_type =
  | ClassType of c_class
  | ArrayType of def_type
  | GenericClassType of c_class * (ident * def_type) list
  | GenericType of ident * def_type
  | TempType of ttype
  | VoidType
  | NilType

(* Name for a class, method, property or variable *)
and name =
  { x_name: ident;
    mutable x_def: def}

(* Environment of idents to definitions *)
module IdMap =
  Map.Make(struct
```

```
    type t = ident
    let compare = compare
  end)

type environment = Env of def IdMap.t

let lookup id (Env(ids)) =
  try IdMap.find id ids with
    Not_found -> raise (UnknownName id)

let add_def id classDef n = IdMap.add id classDef n

let define id d (Env(ids)) = Env(add_def id d ids)

let can f x = try f x; true with Not_found -> false

let empty = Env(IdMap.empty)

let empty_def = {d_kind=NoneKind; d_type=VoidType}

(* create functions for parser *)
let createStmt s =
  { s_guts=s; s_line=(!Source.lineno) }

let createEmptyStmt s =
  { s_guts=s; s_line=0 }

let seq =
  function
    | [] -> createEmptyStmt Nop       (* Use Nop in place of Seq [] *)
    | [s] -> s                        (* Don't use a Seq node for one
element *)
    | ss -> createEmptyStmt (Seq ss)

let createClass (n, p, props, meths, generics) =
  { c_name=n; c_ptype=p; c_size=0; c_properties=props; c_methods=meths;
c_ancestors=[]; c_generics=generics }

let createMethod (n, static, args, t, stmt, main, replace) =
  { m_name=n; m_type=t; m_static=static; m_size=0; m_arguments=args;
m_body=stmt; m_main=main; m_replace=replace; m_origin=Mine }

let createName n =
  { x_name=n; x_def=empty_def }

let createGeneric n t =
  { x_name=n; x_def={d_kind=NoneKind; d_type=t} }

let createTypeName t =
  { x_name="type_delc"; x_def={d_kind=NoneKind; d_type=t} }
```

lexer.mll

```
{
  open Parser
  open Keiko
  open Source
  open Errors

  (* Create a hash type and add values to it *)
  let make_hash n ps =
    let t = Hashtbl.create n in
    List.iter (fun (k, v) -> Hashtbl.add t k v) ps; t

  let next_line lexbuf =
    incr lineno; Source.note_line !lineno lexbuf

  (* Keyword hash table *)
  let kwtable = make_hash 64
  [
    "Define", K_DEFINE;
    "Using", K_USING;

    "ClassMethod", K_CLASSMETHOD;
    "Main", K_MAIN;
    "Method", K_METHOD;
    "Properties", K_PROPERTIES;
    "ReplaceMethod", K_REPLACE;
    "ConstructorMethod", K_CONSTRUCTOR;

    "Me", K_ME;
    "My", K_ME;
    "Myself", K_ME;
    "Nil", K_NIL;
    "Parent", K_PARENT;

    "True", C_TRUE;
    "False", C_FALSE;
    "and", O_AND;
    "or", O_OR;
    "not", O_NOT;
    "InstanceOf", O_IS;
    "TypeOf", O_TYPEOF;

    "If", K_IF;
    "Then", K_THEN;
    "Else", K_ELSE;
    "While", K_WHILE;
    "For", K_FOR;
    "Step", K_STEP;
    "Test", K_TEST;
    "Cast", K_CAST;

    "Array", K_ARRAY;
    "New", K_NEW;
    "Of", K_OF;
    "Return", K_RETURN;
```

```
    "With", K_WITH;
  ]

  let idtable = Hashtbl.create 64

  (* Lookup a string in the keyword table *)
  let lookup s =
    try Hashtbl.find kwtable s with
      Not_found ->
        Hashtbl.replace idtable s ();
        IDENT s

  (* List of all string constants in the program *)
  let strtable = ref []

  let get_string s =
    let lab = gen_sym () in
      strtable := (lab, s)::!strtable;
      C_STR (lab, s)
}

let letter = ['A'-'Z''a'-'z']
let digit = ['0'-'9']
let qq = '"'
let notqq = [^'"''\n']

rule token = parse
    letter (letter | digit | '_')* as s
                          { lookup s }
  | digit+ as s           { C_NUMBER (int_of_string s) }
  | qq (notqq* as s) qq { get_string s }

  | "+"                   { O_PLUS }
  | "*"                   { O_TIMES }
  | "-"                   { O_MINUS}
  | "/"                   { O_DIV }
  | "%"                   { O_MOD }
  | "="                   { O_ASSIGN }
  | "->"                  { O_RIGHTARROW }
  | "<-"                  { O_LEFTARROW }
  | "=="                  { O_EQUALS }
  | "/="                  { O_NOTEQUALS }
  | "<"                   { O_LESSTHAN }
  | ">"                   { O_GREATERTHAN }
  | "<="                  { O_LESSTHANEQ }
  | ">="                  { O_GREATERTHANEQ }

  | "."                   { P_DOT }
  | ":"                   { P_COLON }
  | ","                   { P_COMMA }
  | "("                   { P_LPAR }
  | ")"                   { P_RPAR }
  | "["                   { P_LSQUARE }
  | "]"                   { P_RSQUARE }
```

```
  | "{"                 { P_LCURL }
  | "}"                 { P_RCURL }
  | "=>"                { P_DOUBLEARROW }
  | "$>"                { P_START }
  | "<$"                { P_END }

  | "!!"                { comment lexbuf; token lexbuf}
  | [' ''\t']+          { token lexbuf }
  | "\r"                { token lexbuf }
  | "\n"                { next_line lexbuf; token lexbuf }
  | _                   { BADTOKEN }
  | eof                 { EOF }

and comment =
  parse
    | "!!"              { () }
    | "\n"              { next_line lexbuf; comment lexbuf}
    | _                 { comment lexbuf }
    | eof               { raise UnexpectedEnd }
```

parser.mly

```
%{
  open Tree
%}

/* Token names use underscores to denote their "type" for clarity of
reading
    - K : Keyword
    - P : Punctuation
    - O : Operator
    - C : Constant
*/

/* Primitives */
%token <Tree.ident> IDENT
%token <int> C_NUMBER
%token <Keiko.symbol * string> C_STR

/* Keywords */
%token K_DEFINE
%token K_AS
%token K_ARRAY
%token K_OF
%token K_USING
%token K_WITH

%token K_CLASSMETHOD
%token K_MAIN
%token K_METHOD
%token K_PROPERTIES
%token K_REPLACE
%token K_CONSTRUCTOR
```

```
%token K_ME
%token K_NIL
%token K_NEW
%token K_RETURN
%token K_PARENT

%token K_IF
%token K_THEN
%token K_ELSE
%token K_WHILE
%token K_FOR
%token K_STEP
%token K_TEST
%token K_CAST

/* Boolean Constants */
%token C_TRUE
%token C_FALSE

/* Operators */
%token O_PLUS
%token O_TIMES
%token O_MINUS
%token O_DIV
%token O_MOD
%token O_ASSIGN
%token O_RIGHTARROW
%token O_LEFTARROW
%token O_EQUALS
%token O_NOTEQUALS
%token O_LESSTHAN
%token O_GREATERTHAN
%token O_LESSTHANEQ
%token O_GREATERTHANEQ
%token O_AND
%token O_OR
%token O_NOT
%token O_IS
%token O_TYPEOF
%nonassoc O_UMINUS

/* Punctution */
%token P_DOT
%token P_COLON
%token P_COMMA
%token P_LPAR
%token P_RPAR
%token P_LSQUARE
%token P_RSQUARE
%token P_LCURL
%token P_RCURL
%token P_DOUBLEARROW
%token P_START
```

```
%token P_END

%token BADTOKEN
%token EOF

/* Program */
%type <Tree.program> program
%start program

%%

program :
  |  classes
      { Program($1) } ;

classes :
  | cls
      { [$1] }
  | cls classes
      { $1 :: $2 } ;

cls :
  | K_DEFINE name parent generics P_START properties methods P_END
      { createClass($2, $3, $6, $7, $4) }

/* Generic types for class */
generics :
  | /* empty */
      { [] }
  | K_USING generic_list
      { $2 }

/* List of generic definitions */
generic_list :
  | IDENT
      { [createGeneric $1 (TempType (Ident "Object"))] }
  | IDENT O_LEFTARROW ttype
      { [createGeneric $1 (TempType $3)] }
  | IDENT P_COMMA generic_list
      { (createGeneric $1 (TempType (Ident "Object"))) :: $3 }
  | IDENT O_LEFTARROW ttype P_COMMA generic_list
      { (createGeneric $1 (TempType $3)) :: $5 }

/* Parent type of class */
parent :
  | /* empty */
      { TempType (Ident "Object") }
  | O_LEFTARROW ttype
      { TempType $2 };

/* Properties for class */
properties :
  | /* empty */
      { [] }
```

```
  | K_PROPERTIES P_START pairs P_END
      { $3 } ;

/* List of methods */
methods :
  | /* empty */
      { [] }
  | meth methods
      { $1 :: $2 } ;

meth :
  | K_CLASSMETHOD K_MAIN P_LCURL P_RCURL P_DOUBLEARROW P_LCURL P_RCURL
P_START stmts P_END
      { createMethod(createName "Main", true, [], VoidType, $9, true,
false) }
  | K_METHOD name P_LCURL pairs P_RCURL P_DOUBLEARROW mtype P_START stmts
P_END
      { createMethod($2, false, $4, $7, $9, false, false) }
  | K_CLASSMETHOD name P_LCURL pairs P_RCURL P_DOUBLEARROW mtype P_START
stmts P_END
      { createMethod($2, true, $4, $7, $9, false, false) }
  | K_REPLACE name P_LCURL pairs P_RCURL P_DOUBLEARROW mtype P_START stmts
P_END
      { createMethod($2, false, $4, $7, $9, false, true) }

/* Type of a method */
mtype :
  | ttype
      { TempType $1 }
  | P_LCURL P_RCURL
      { VoidType }

/* Temporary type */
ttype :
  | IDENT
      { Ident $1 }
  | K_ARRAY K_OF ttype
      { Array $3}
  | IDENT K_WITH P_LPAR type_list P_RPAR
      { Generic ($1, $4) }

/* List of types for generic type */
type_list :
  | ttype
      { [$1] }
  | ttype P_COMMA type_list
      { $1 :: $3 }

/* Pair of a name and a type */
pair:
  | name P_COLON ttype
      { Prop ($1, TempType $3) }

/* List of pairs */
```

```
pairs:
  | /* empty */
      { [] }
  | pair
      { [$1] }
  | pair P_COMMA pairs
      { $1 :: $3 } ;

stmts :
  | stmt_list
      { seq $1 } ;

stmt_list :
  | stmt
      { [$1] }
  | stmt stmt_list
      { $1 :: $2} ;

stmt :
  | stmt1
      { createStmt $1 }

stmt1 :
  |  expr O_ASSIGN expr P_DOT
      { Assign($1, $3) }
  | name P_COLON ttype O_ASSIGN expr P_DOT
      { Delc($1, TempType $3, $5) }
  | K_RETURN expr P_DOT
      { Return (Some $2) }
  | K_RETURN P_DOT
      { Return None }
  | expr P_DOT
      { Call($1) }
  | K_IF expr K_THEN body elses
      { IfStmt($2, $4, $5) }
  | K_WHILE expr P_DOT body
      { WhileStmt($2, $4) }
  | K_FOR for_stmt K_STEP for_stmt K_TEST expr P_DOT body
      { ForStmt($2, $4, $6, $8) } ;

/* Body of if/for/while statments */
body :
  | stmt
    { $1 }
  | P_START stmts P_END
    { $2 }

/* Initial and step statments for for loop */
for_stmt :
  | /* empty */
      { createEmptyStmt Nop }
  | stmt
      { $1 } ;
```

```
/* else statment for if */
elses :
  | /* empty */
      { createEmptyStmt Nop }
  | K_ELSE body
      { $2 };

expr:
  | simple
      { $1 }
  | simple O_EQUALS simple
      { MethodCall($1, createName "equals", [$3]) }
  | simple O_NOTEQUALS simple
      { MethodCall($1, createName "notEquals", [$3]) }
  | simple O_LESSTHAN simple
      { MethodCall($1, createName "lessThan", [$3]) }
  | simple O_GREATERTHAN simple
      { MethodCall($1, createName "greaterThan", [$3]) }
  | simple O_LESSTHANEQ simple
      { MethodCall($1, createName "lessThanEq", [$3]) }
  | simple O_GREATERTHANEQ simple
      { MethodCall($1, createName "greaterThanEq", [$3]) }
  | simple O_AND simple
      { MethodCall($1, createName "and", [$3]) }
  | simple O_OR simple
      { MethodCall($1, createName "or", [$3]) }
  | simple O_IS simple
      { MethodCall($1, createName "InstanceOf", [$3]) }

simple :
  | term
      { $1 }
  | O_MINUS term %prec O_UMINUS
      { MethodCall($2, createName "uminus", []) }
  | simple O_PLUS term
      { MethodCall($1, createName "add", [$3]) }
  | simple O_MINUS term
      { MethodCall($1, createName "sub", [$3]) }
  | O_TYPEOF name
      { TypeOf $2 }

term :
  | factor
      { $1 }
  | term O_TIMES factor
      { MethodCall($1, createName "times", [$3]) }
  | term O_DIV factor
      { MethodCall($1, createName "div", [$3]) }
  | term O_MOD factor
      { MethodCall($1, createName "mod", [$3]) }

factor :
  | name
      { Name $1 }
```

```
  | C_NUMBER
      { Constant ($1, TempType (Ident "Int")) }
  | C_STR
      { let (lab, s) = $1 in String (lab, s) }
  | factor O_RIGHTARROW name O_LEFTARROW P_LCURL arguments P_RCURL
      { MethodCall ($1, $3, $6) }
  | factor O_RIGHTARROW name
      { Property($1, $3) }
  | factor O_RIGHTARROW P_LSQUARE expr P_RSQUARE
      { Sub($1, $4) }
  | K_CAST P_LPAR expr P_RPAR O_RIGHTARROW name
      { Cast($3, $6)}
  | O_NOT factor
      { MethodCall($2, createName "not", []) }
  | K_NIL
      { Nil }
  | K_NEW ttype O_LEFTARROW O_LSQUARE expr P_RSQUARE
      { NewArray (createTypeName (TempType $2), $4) }
  | K_NEW ttype
      { New (createTypeName (TempType $2)) }
  | K_ME
      { Name (createName "Me") }
  | K_PARENT
      { Parent }
  | C_TRUE
      { Constant (1, TempType (Ident "Bool")) }
  | C_FALSE
      { Constant (0, TempType (Ident "Bool")) }
  | P_LPAR expr P_RPAR
      { $2 }


arguments :
  | /* empty */
      { [] }
  | expr
      { [$1] }
  | expr P_COMMA arguments
      { $1 :: $3 } ;

name :
  | IDENT
      { createName $1 } ;
```

analyse.ml

```
open Syntax.Tree
open Errors
open Lib.Lib_all
open Lib.Type
open Lib.String
open Lib.Object
open Lib.Array
```

```
open Printf

let propertyOffset = 4 (* offset of properties in an object record *)
let argumentOffset = 12 (*offset of arguments in stack frame *)
let variableOffset = -4 (*offset of variables in stack frame *)
let vtableOffset = 12 (* offset of the vtable in the class descriptor *)
let variableIndex = ref 0 (* number of local variables *)
let p_type = ref VoidType (* parent type of class *)
let mainMethod = ref "" (* Main method *)

let create_def kind t = { d_kind=kind; d_type = t }

(* Replace inheirited method with new method *)
let create_method i r =
  { m_name=i.m_name; m_type=r.m_type; m_static=i.m_static;
m_size=i.m_size; m_arguments=r.m_arguments; m_body=r.m_body; m_main=false;
m_replace=true; m_origin=Mine }

(* Get the origin of a method *)
let create_origin i c =
  match i.m_origin with
  | Mine -> Inherited c.c_name.x_name
  | o -> o

(* copy inheirited method *)
let copy i cls =
  { m_name=i.m_name; m_type=i.m_type; m_static=i.m_static;
m_size=i.m_size; m_arguments=i.m_arguments; m_body=i.m_body; m_main=false;
m_replace=false; m_origin = create_origin i cls }

let create_me cls =
  Prop({x_name="Me"; x_def=(create_def NoneKind VoidType)}, TempType
(Ident cls.c_name.x_name))

let rec get_class d =
  match d with
  | ClassType c -> c
  | ArrayType _ -> array_class
  | GenericClassType (c, _) -> c
  | GenericType (_, d) -> get_class d
  | _ -> raise InvalidExpression

(* Get proper type from temp type *)
let rec get_temp_type t env =
  match t with
  | Ident n -> let d = lookup n env in d.d_type (* Find the name n in the
enviroment *)
  | Array t' -> ArrayType (get_temp_type t' env) (* Recurse on the type of
the array elements *)
  | Generic (n, ts) ->
      let c = get_class (lookup n env).d_type in (* Find the name in in
the enviroment *)
        (* Then do the same for all the types that are passed to the type
variables *)
```

```
        try GenericClassType (c, List.combine (List.map (fun g ->
g.x_name) c.c_generics) (List.map (fun x -> get_temp_type x env) ts))
          with Invalid_argument _ -> raise InvalidGeneric

let get_type t env =
  match t with
  | TempType d -> get_temp_type d env
  | _ -> t

let rec print_temp_type t =
  match t with
  | Ident n -> n
  | Array t' -> sprintf "Array of %s" (print_temp_type t')
  | Generic (n, ts) -> sprintf "%s With %s" n (List.fold_right (fun t s ->
sprintf "%s, %s" (print_temp_type t) s) ts "")

(* Find a method within an class *)
let find_method meth cls =
  match cls with
  | ClassType c ->
      begin (*Find a method in c with same name as meth *)
        try (List.find (fun m -> meth = m.m_name.x_name)
c.c_methods).m_name.x_def with
          Not_found -> raise (UnknownName meth)
      end
  | ArrayType _ ->
      (* Arrays only have methods from Array *)
      begin (*Find a method in Array with same name as meth *)
        try (List.find (fun m -> meth = m.m_name.x_name)
array_class.c_methods).m_name.x_def with
          Not_found -> raise (UnknownName meth)
      end
  | GenericClassType (c, _) ->
      begin
        try (List.find (fun m -> meth = m.m_name.x_name)
c.c_methods).m_name.x_def with
          Not_found -> raise (UnknownName meth)
      end
  | GenericType (_, d) ->
      (* Generic types are treated as their most general type *)
      let c = get_class d in
      begin (*Find a method in c with same name as meth *)
        try (List.find (fun m -> meth = m.m_name.x_name)
c.c_methods).m_name.x_def with
          Not_found -> raise (UnknownName meth)
      end
  | VoidType -> raise VoidOperation
  | NilType -> raise (UnknownName meth)
  | TempType n -> raise (UnannotatedName (print_temp_type n))

(* Find a property within an class *)
let find_properties prop t =
  match t with
  | ClassType c ->
```

```
      begin (* Find a property in c with the same name as prop *)
        try (List.find (fun n -> prop.x_name = n.x_name) (List.map (fun
(Prop(n, _)) -> n) c.c_properties)).x_def with
          Not_found -> raise (UnknownName prop.x_name)
      end
  | ArrayType _ ->
      (* Arrays only have properties from Array *)
      begin (*Find a property in Array with same name as prop *)
        try (List.find (fun n -> prop.x_name = n.x_name) (List.map (fun
(Prop(n, _)) -> n) array_class.c_properties)).x_def with
          Not_found -> raise (UnknownName prop.x_name)
      end
  | GenericClassType (c, _) ->
      begin (* Find a property in c with the same name as prop *)
        try (List.find (fun n -> prop.x_name = n.x_name) (List.map (fun
(Prop(n, _)) -> n) c.c_properties)).x_def with
          Not_found -> raise (UnknownName prop.x_name)
      end
  | GenericType (_, d) ->
      (* Generic types are treated as their most general type *)
      let c = get_class d in
      begin (* Find a property in c with the same name as prop *)
        try (List.find (fun n -> prop.x_name = n.x_name) (List.map (fun
(Prop(n, _)) -> n) c.c_properties)).x_def with
          Not_found -> raise (UnknownName prop.x_name)
      end
  | VoidType -> raise VoidOperation
  | TempType n -> raise (UnannotatedName (print_temp_type n))
  | NilType -> raise (UnknownName prop.x_name)

(* Add classes to environment *)
let annotate_classes classes env =
  let annotate c e =
    (* Classes are marked as ClassDef and have a type of themselves*)
    c.c_name.x_def <- create_def ClassDef (ClassType c);
    define c.c_name.x_name c.c_name.x_def e
  in List.fold_right annotate classes env

(* Add Arguments to environment *)
let annotate_arguments args env =
  let rec annotate a i e =
    match a with
    | (Prop(x, t))::props ->
        (* Arguments are marked as Variables and have a type depending on
their temporary type *)
        x.x_def <- create_def (VariableDef(argumentOffset + i)) (get_type
t e);
        let env' = define x.x_name x.x_def e in
          annotate props (i+4) env'
    | _ -> e
  in annotate args 0 env

let annotate_properties properties env =
  let rec annotate ps i e =
```

```
    match ps with
    | (Prop(x, t))::props ->
        (* Properties are marked as Properties and have a type depending
on their temporary type *)
        x.x_def <- create_def (PropertyDef(propertyOffset + i)) (get_type
t e);
        annotate props (i+4) e
    | _ -> ()
  in annotate properties 0 env

let rec annotate_expr expr env =
  match expr with
  | Name n ->  let d = lookup n.x_name env in n.x_def <- d; n.x_def.d_type
(* Find the name in the environment *)
  | Constant (_, d) -> get_type d env
  | String _ -> string_def.d_type;
  | TypeOf _ -> type_def.d_type
  | MethodCall (e, m, args) ->
      (* Annotate the calling object *)
      let c = annotate_expr e env in
      (* Find method that is being called *)
      m.x_def <- find_method m.x_name c;
      (* Annotate all the arguments *)
      List.iter (fun x -> ignore(annotate_expr x env)) args;
      (* Return the type of the method *)
      m.x_def.d_type
  | Property (e, n) ->
      (* Annotate the calling object *)
      let c = annotate_expr e env in
      (* Find the property in the calls *)
      n.x_def <- find_properties n c;
      (* Return the type of the property *)
      n.x_def.d_type
  | Sub (e1, e2) ->
      (* Annotate the index *)
      ignore (annotate_expr e2 env);
      begin
        match annotate_expr e1 env with
        | ArrayType d -> d        (* Cannot use sub notation on anything
expect arrays *)
        | _ -> raise InvalidSub
      end
  | New n ->
      let d = get_type n.x_def.d_type env in
        n.x_def <- create_def ClassDef d; d (* Names following New are
classes *)
  | NewArray (n, e) ->
      ignore(annotate_expr e env);
      let d = get_type n.x_def.d_type env in
        n.x_def <- create_def ClassDef d; d (* Names following New are
classes *)
  | Cast (e, n) ->
      ignore(annotate_expr e env);
      let d = lookup n.x_name env in
```

```
        n.x_def <- d;
        d.d_type
  | Parent -> !p_type
  | Nil -> object_def.d_type

let rec annotate_stmt stmt env =
  match stmt.s_guts with
  | Assign (e1, e2) ->
      (* Annotate both sides of the assignment *)
      ignore(annotate_expr e1 env); ignore(annotate_expr e2 env); env
  | Delc (n, t, e) ->
      (* Annotate the assinged value *)
      ignore(annotate_expr e env);
      (* Variables are marked as Variables and have a type depending on
their temporary type *)
      n.x_def <- create_def (VariableDef(variableOffset - !variableIndex))
(get_type t env);
      variableIndex := !variableIndex + 4;
      (* Add this new variable to the environment *)
      define n.x_name n.x_def env
  | Call e -> ignore(annotate_expr e env); env
  | Return r ->
      begin
        match r with
        | Some e -> ignore(annotate_expr e env); env
        | None -> env
      end
  | IfStmt (e, ts, fs) ->
      (* Annotate all constituent parts *)
      ignore(annotate_expr e env);
      ignore(annotate_stmt ts env);
      ignore(annotate_stmt fs env);
      env
  | WhileStmt (e, s) ->
      (* Annotate all constituent parts *)
      ignore(annotate_expr e env);
      ignore(annotate_stmt s env);
      env
  | ForStmt (init, step, test, body) ->
      (* Annotate all constituent parts *)
      let env' = annotate_stmt init env in
        ignore(annotate_stmt step env');
        ignore(annotate_expr test env');
        ignore(annotate_stmt body env');
      env
  | Seq stmts -> List.fold_left (fun env' s -> annotate_stmt s env') env
stmts
  | Nop -> env

let annotate_generics env generic =
  let d = get_type generic.x_def.d_type env in
    generic.x_def <- create_def ClassDef (GenericType (generic.x_name,
d));
```

```
    (* Create new generic type in enviroment with it's mpost general type
*)
    define generic.x_name generic.x_def env

let annotate_methods methods env=
  let rec annotate meths i e =
    match meths with
    | m::ms ->
        (*  Methods are marked as Methods and have a type depending on
their temporary type *)
        m.m_name.x_def <- create_def (MethodDef(vtableOffset + i,
m.m_static)) (get_type m.m_type e);
        annotate ms (i + 4) e
    | _ -> ()
  in annotate methods 0 env

let annotate_body meth cls env =
  match meth.m_origin with
  | Mine -> (* Only annotate if this is a new method *)
      if meth.m_main then mainMethod := cls.c_name.x_name ^ "." ^
meth.m_name.x_name;
      let env' = List.fold_left annotate_generics env cls.c_generics in (*
Add generic types to environment *)
      let newEnv = annotate_arguments meth.m_arguments env' in        (*
Add arguments to environment *)
      variableIndex := 0;
      p_type := ClassType cls;
      ignore(annotate_stmt meth.m_body newEnv);
      meth.m_size <- !variableIndex
  | _ -> ()

let annotate_bodies cls env =
  List.iter (fun m -> ignore(annotate_body m cls env)) cls.c_methods

(* Find a method with same name as r, and replaced it with r *)
let rec replace_method r inherited =
  match inherited with
  | i::is ->
      let same_method r i = r.m_name.x_name = i.m_name.x_name in
        if (same_method r i) then (create_method i r) :: is
        else i :: (replace_method r is)
  | [] -> []

(* Split methods into replacing and normal *)
let rec split methods =
  match methods with
  | m::ms ->
      let (x,y) = split ms in
        if m.m_replace then (m::x, y)
        else (x, m::y)
  | [] -> ([], [])

(* Annotate the methods and properties of a class *)
let annotate_members cls env =
```

```
  let env' = List.fold_left annotate_generics env cls.c_generics in
    let generics =
      function
        | GenericClassType (_, ts) -> ts
        | _ -> [] in
      (* Add all the generic types to the environment *)
      let env'' = List.fold_left (fun e (i, t) -> define i (create_def
ClassDef t) e) env' (generics cls.c_ptype) in
        annotate_methods cls.c_methods env'';
        annotate_properties cls.c_properties env''

let rec annotate_parent cls env =
  match cls.c_ptype with
  | TempType _ -> (* If the parent has not been annotated *)
      let parent = get_type cls.c_ptype env in
        annotate_parent (get_class parent) env; (* Recursively annotate
the parent of the parent *)
        cls.c_ptype <- parent;                  (* Update own parent *)
        let (r, n) = split cls.c_methods and p = get_class parent in
          (* Use the replacing methods to replace inheirited methods *)
          cls.c_methods <- (List.fold_right replace_method r (List.map
(fun i -> copy i p) p.c_methods)) @ n;
          cls.c_properties <- p.c_properties @ cls.c_properties; (*
inheirit all the properties *)
          cls.c_size <- 4 * (List.length cls.c_properties);
          cls.c_ancestors <- p :: p.c_ancestors
  | _ -> ()

let modify_non_static c =
  let modify m =
    (* Add Me to the list of arguments *)
    if not m.m_static then m.m_arguments <- (create_me c) :: m.m_arguments
    else ()
  in List.iter modify c.c_methods

(* Annotate the whole program *)
let annotate_program (Program(cs)) =
  let env = annotate_classes cs start_env in
    List.iter modify_non_static cs;
    List.iter (fun c -> annotate_parent c env) cs;
    List.iter (fun c -> annotate_members c env) cs;
    List.iter (fun c -> annotate_bodies c env) cs
```

check.ml

```
open Syntax.Tree
open Errors
open Lib.Int
open Lib.Bool
open Lib.Type
open Lib.String
open Lib.Array
open Printf
```

```
let p_type = ref VoidType (* Parent type *)

(* Get a string from Temporary type *)
let rec print_temp_type t =
  match t with
  | Ident n -> n
  | Array t' -> sprintf "Array of %s" (print_temp_type t')
  | Generic (n, ts) -> sprintf "%s With %s" n (List.fold_right (fun t s ->
sprintf "%s, %s" (print_temp_type t) s) ts "")

(* Get a string from a type *)
let rec print_type t =
  match t with
  | ClassType c -> c.c_name.x_name
  | ArrayType d -> sprintf "Array of %s" (print_type d)
  | GenericClassType (c, ts) -> (sprintf "%s with " c.c_name.x_name) ^
(List.fold_right (fun (n, t') s -> (print_type t') ^ s) ts "")
  | GenericType (n, d) -> sprintf "%s as %s" n (print_type d)
  | VoidType -> "Void"
  | TempType d -> sprintf "Temp %s" (print_temp_type d)
  | NilType -> "Nil"

(* Check to see if two types are compatible with each other *)
let check_compatible type1 type2 =
  let pt1 = ref VoidType and pt2 = ref VoidType in
    (* Check strict compatibility between types *)
    let rec check_strict t1 t2 =
      match (t1, t2) with
      | (ClassType c1, ClassType c2) ->
          if c1.c_name.x_name == c2.c_name.x_name then () (* Classes must
be identical *)
          else raise (TypeError((print_type t1), (print_type t2)))
      | (ArrayType d1, ArrayType d2) -> check_strict d1 d2 (* Recurse on
the type of the array elements *)
      | (GenericType (n1, _), GenericType (n2, _)) ->
          if n1 == n2 then () (* Generic types must have the same name *)
          else raise (TypeError((print_type t1), (print_type t2)))
      | (_, NilType) -> () (* Nil is compatible with anything *)
      | _ -> raise (TypeError((print_type !pt1), (print_type !pt2))) in
    let rec check t1 t2 =
      match (t1, t2) with
      | (ClassType c1, ClassType c2) ->
          if c1.c_name.x_name == c2.c_name.x_name then ()
          else check t1 c2.c_ptype (* Recurse and check t1 against t2's
paraent type *)
      | (GenericClassType (c1, ts1), GenericClassType (c2, ts2)) ->
          check (ClassType c1) (ClassType c2); (* Check the classes are
compatible *)
          begin
            (* Then check that the type arguments are the same *)
            try List.iter2 (fun (_, x) (_, y) -> check_strict x y) ts1 ts2
            with Invalid_argument _ -> raise (TypeError((print_type !pt1),
(print_type !pt2)))
```

```
          end
      | (ClassType c1, GenericClassType (c2,_)) ->
          if c1.c_name.x_name == c2.c_name.x_name then () (* Generic class
can be compatible with normal class *)
          else check t1 c2.c_ptype
      | (ArrayType d1, ArrayType d2) -> check_strict d1 d2  (* Arrays are
invarient *)
      | (ClassType c, ArrayType _ ) ->
          if c.c_name.x_name == "Object" then () (* Arrays are subtypes of
Object *)
          else raise (TypeError((print_type !pt1), (print_type !pt2)))
      | (GenericType (n1, d1), GenericType (n2, d2)) ->
          if n1 == n2 then () (* Generic types should have the same name
or be in a subtype relation *)
          else check t1 d2
      | (ClassType c, GenericType(_, t)) -> check t1 t (* Check the most-
general type of the generic against the class type *)
      | (_, NilType) -> () (* Nil is compatible with anything *)
      | (VoidType, VoidType) -> () (* Void is compatible with itself *)
      | _ -> raise (TypeError((print_type !pt1), (print_type !pt2)))
    in pt1 := type1; pt2 := type2; check type1 type2

(* Validate that the types supplied to a generic do not violate the
restrictions *)
let validate_generics generics =
  let rec validate prev =
    let validate_pair t gt =
      match gt with
      | GenericType (n1, d1) ->
          let (n, d) = List.find (fun (n2, d2) -> n1 == n2) prev in
            check_compatible d t
      | pt -> check_compatible pt t
      in function
        | (t, g)::gs ->
            validate_pair t g.x_def.d_type;
            validate ((g.x_name, g.x_def.d_type)::prev) gs
        | _ -> ()
  in validate [] generics

(* Make sure a type is valid *)
let rec validate_type t =
  match t with
  | GenericClassType (c, ts) ->
      let ds = List.map (fun (_,y) -> y) ts in
        List.iter validate_type ds; validate_generics (List.combine ds
c.c_generics)
  | _ -> ()

(* Get the true type of something with a generic type *)
let get_type ct t =
  let rec get_type_with_generic ts =
    function
    | GenericType (n, _) ->  let (_, d) = List.find (fun (i, _) -> i = n)
ts in d
```

```
    | GenericClassType (c, ts') -> GenericClassType (c, List.map (fun
(i,t') -> (i, get_type_with_generic ts t')) ts')
    | ArrayType d -> ArrayType (get_type_with_generic ts d)
    | t' -> t'
  in match (ct, t) with
  | (GenericClassType (_, ts), _) -> get_type_with_generic ts t
  | _ -> t

(* Get the class from a type *)
let rec get_class d =
  match d with
  | ClassType c -> c
  | ArrayType _ -> array_class
  | GenericClassType (c, _) -> c
  | GenericType (_, d) -> get_class d
  | _ -> raise InvalidExpression

(* Find a method in class *)
let find_method meth cls =
  match cls with
  | ClassType c ->
      begin (*Find a method in c with same name as meth *)
        try List.find (fun m -> meth.x_name = m.m_name.x_name) c.c_methods
with
          Not_found -> raise (UnknownName meth.x_name)
      end
  | ArrayType _ ->
      (* Arrays only have methods from Array *)
      begin (*Find a method in Array with same name as meth *)
        try List.find (fun m -> meth.x_name = m.m_name.x_name)
array_class.c_methods with
          Not_found -> raise (UnknownName meth.x_name)
      end
  | GenericClassType (c, _) ->
      begin
        try List.find (fun m -> meth.x_name = m.m_name.x_name) c.c_methods
with
          Not_found -> raise (UnknownName meth.x_name)
      end
  | GenericType (_, d) ->
      (* Generic types are treated as their most general type *)
      let c = get_class d in
      begin (*Find a method in c with same name as meth *)
        try List.find (fun m -> meth.x_name = m.m_name.x_name) c.c_methods
with
          Not_found -> raise (UnknownName meth.x_name)
      end
  | VoidType -> raise VoidOperation
  | TempType n -> raise (UnannotatedName (print_temp_type n))
  | NilType -> raise (UnknownName meth.x_name)

(* Type check the args supplied to a method call *)
let rec check_args args margs =
  match (args, margs) with
```

```
  | (t::ts, m::ms) ->
      check_compatible m t;
      check_args ts ms
  | ([], []) -> ()
  | _ -> raise IncorrectArgumentCount

(* Get the type of an expression *)
and check_expr e =
  match e with
  | Name n -> n.x_def.d_type
  | Constant (_, d) ->
    begin
      match d with
      | TempType (Ident "Int") -> integer_def.d_type
      | TempType (Ident "Bool") -> bool_def.d_type
      | _ -> raise UnknownConstant
    end
  | String _ -> string_def.d_type
  | TypeOf _ -> type_def.d_type
  | MethodCall (e1, m, args) ->
      let t = check_expr e1 in (* Get the type of the receiving object *)
        let meth = find_method m t in (* Find the method *)
          let margs = List.map (fun (Prop(x, _)) -> get_type t
x.x_def.d_type) meth.m_arguments in (* check the arguments *)
            if meth.m_static then check_args (List.map check_expr args)
margs
            else  check_args (t::(List.map check_expr args)) margs;
            get_type t m.x_def.d_type (* Get the type of the result *)
  | Property (e1, n) ->
      let t = check_expr e1 in (* Get the type of the object *)
      get_type t n.x_def.d_type (* Get the type of the property *)
  | Sub (e1, e2) ->
      begin
        match check_expr e1 with (* Make sure we are calling Sub on an
array *)
        | ArrayType d1 ->
            let d2 = check_expr e2 in
              check_compatible d2 integer_def.d_type; d1 (* Check that the
index is an integer *)
        | _ -> raise InvalidSub
      end
  | New n -> let d = n.x_def.d_type in validate_type d; d (* Validate the
type of the new object *)
  | NewArray (n, e) ->
      let t = check_expr e and d = n.x_def.d_type in
        validate_type d; (* Validate the type of the new object *)
        check_compatible t integer_def.d_type; d (* Check that the length
is an integer *)
  | Cast (e, n) ->
      let t = check_expr e and d = n.x_def in
      begin
        match d.d_kind with
        | ClassDef -> check_compatible t d.d_type; d.d_type
        | _ -> raise InvalidExpression
```

```
      end
  | Parent -> !p_type
  | Nil -> NilType

(* Check that the method returns a value of the correct type *)
let check_return r ret =
  match (r, ret) with
  | (Some e, _) ->
      let t = check_expr e in check_compatible t ret
  | (None, VoidType) -> ()
  | _ -> raise InvalidReturn

(* Type-check a statement *)
let rec check_stmt s ret =
  match s.s_guts with
  | Assign (e1, e2) ->
      (* Check that the LHS of an assignment is compatible with the RHS *)
      let (t1, t2) = ((check_expr e1), (check_expr e2)) in
        check_compatible t1 t2
  | Delc (x, _, e) ->
      (* Check that the RHS of an declaration is compatible with the
declared type*)
      let t = check_expr e in
        check_compatible x.x_def.d_type t
  | Call e ->
      begin
        match e with
        | MethodCall(_, _, _) -> ignore (check_expr e)
        | _ -> raise IncompleteStatement
      end
  | Return r -> check_return r ret; (* Check the returned value *)
  | IfStmt (test, ts, fs) ->
      let t = check_expr test in
        check_compatible t bool_def.d_type; (* Check that the condition is
a Boolean *)
        check_stmt ts ret;
        check_stmt fs ret
  | WhileStmt (test, stmt) ->
      let t = check_expr test in
        check_compatible t bool_def.d_type; (* Check that the condition is
a Boolean *)
        check_stmt stmt ret
  | ForStmt (init, step, test, body) ->
      let t = check_expr test in
        check_stmt init ret;
        check_stmt step ret;
        check_compatible t bool_def.d_type; (* Check that the condition is
a Boolean *)
        check_stmt body ret
  | Seq(ss) -> List.iter (fun st -> check_stmt st ret) ss
  | Nop -> ()

(* Type-check a method *)
let check_method meth parent =
```

```
  if meth.m_replace then (* If it replaces a method, make sure it is
compatible with what it is replacing *)
  begin
    let p_meth = find_method meth.m_name parent and get_types args =
List.map (fun (Prop(x, _)) -> get_type VoidType x.x_def.d_type) args in
      check_compatible p_meth.m_name.x_def.d_type
meth.m_name.x_def.d_type;
      check_args (get_types (List.tl p_meth.m_arguments)) (get_types
(List.tl meth.m_arguments));
  end;
  (* Check the body of the method *)
  check_stmt meth.m_body meth.m_name.x_def.d_type

(* Type-check a class *)
let check_class c =
  validate_type c.c_ptype;
  p_type := c.c_ptype;
  List.iter (fun m -> check_method m c.c_ptype) c.c_methods

(* Type check the whole program *)
let check_program (Program(cs)) =
  List.iter check_class cs
```

codegen.ml

```
open Syntax.Tree
open Syntax.Keiko
open Syntax.Lexer
open Errors
open Semantics
open Gc
open Printf

let p_name = ref "" (* Name of the parent class *)

(* Get the string indicating a type *)
let type_name t =
  match t with
  | ClassType c -> c.c_name.x_name
  | _ -> "Other"

(* Get the size of a class *)
let get_size x =
  match x.x_def.d_type with
  | ClassType c -> c.c_size
  | GenericClassType (c, _) -> c.c_size
  | _ -> raise (InvalidNew x.x_name)

(* Get a pointer to a class descriptor *)
let get_name n =
  match n.x_def.d_type with
    | ClassType c -> GLOBAL (c.c_name.x_name ^ ".%desc")
    | GenericClassType (c, _) -> GLOBAL (c.c_name.x_name ^ ".%desc")
```

```
    | _ -> raise (InvalidNew n.x_name)

(* Get the name of the parent class from the type *)
let get_pname t =
  match t with
    | ClassType c -> c.c_name.x_name
    | GenericClassType (c, _) -> c.c_name.x_name
    | _ -> raise IncorrectSyntaxError

(* Check if a method is static *)
let is_static m =
  match m.x_def.d_kind with
  | MethodDef(_, s) -> s
  | _ -> raise (UnknownName m.x_name)

(* Unbox a primitive *)
let unbox = SEQ [CONST 4; OFFSET; LOADW]

(* Generate code for getting the address assoicated with a name *)
let rec gen_name_addr n =
  match n.x_def.d_kind with
  | ClassDef -> SEQ [GLOBAL (n.x_name ^ ".%desc")]
  | VariableDef off -> SEQ [LOCAL off]
  | PropertyDef off -> SEQ [CONST off; OFFSET]
  | MethodDef (off, _) -> SEQ [CONST off; OFFSET]
  | NoneKind -> raise (NoneKindError n.x_name)

(* Generate code for an address *)
and gen_addr expr =
  match expr with
  | Name n -> gen_name_addr n
  | Property (e, n) -> SEQ[gen_expr e; gen_name_addr n]
  | Sub (e1, e2) ->
      SEQ [
        (* Get the array the we are indexing from *)
        gen_expr e1;
        (* Get the "Data" *)
        CONST 4;
        OFFSET;
        LOADW;
        (* Get the index *)
        gen_expr e2;
        unbox;
        (* The element we want is at offset 4*i *)
        CONST 4;
        BINOP Times;
        OFFSET;
      ]
    | _ -> raise InvalidAssigment

(* Generate code for an expression *)
and gen_expr e =
  match e with
  | Name n -> SEQ [gen_name_addr n; LOADW]
```

```
  | Constant (x, d) ->
    begin
      match d with
      | TempType (Ident "Int") -> SEQ [CONST x; GLOBAL "baozi.%makeInt";
CALLW 1]
      | TempType (Ident "Bool") ->
          if x = 0 then GLOBAL "baozi.%const.%false"
          else GLOBAL "baozi.%const.%true"
      | _ -> raise UnknownConstant
    end
  | String (lab, s) -> (* A String constant *)
      SEQ [
        GLOBAL lab;
        CONST (String.length s);
        GLOBAL "baozi.%makeString";
        CALLW 2
      ]
  | TypeOf n -> (* A TypeOf expression creates a Type object *)
      SEQ [
        GLOBAL (n.x_name ^ ".%desc");
        GLOBAL "Type.%desc";
        GLOBAL "baozi.%makePrim";
        CALLW 2;
      ]
  | MethodCall (e1, m, args) ->
      if is_static m then gen_static_call e1 m args
      else gen_call e1 m args
  | Property (_, _) -> SEQ [gen_addr e; LOADW]
  | Sub (_, _) -> SEQ [gen_addr e; LOADW]
  | New n ->
      SEQ [
        (* get the address of the class descriptor *)
        get_name n;
        (* need space for all the properties + class descriptor *)
        CONST (4 + (get_size n));
        (* Call the make proceduure *)
        GLOBAL "baozi.%make";
        CALLW 2;
      ]
  | NewArray (_, e) ->
      SEQ [
        gen_expr e;
        GLOBAL "baozi.%makeArray";
        CALLW 1;
      ]
  | Cast (e, n) ->
      SEQ [
        gen_expr e;
        DUP 0;
        GLOBAL (n.x_name ^ ".%desc");
        GLOBAL "baozi.%typeCheck";
        CALL 2;
      ]
  | Parent -> SEQ [LOCAL 12; LOADW]
```

```
  | Nil -> LDG "Nil"

(* Generate code for a static call *)
and gen_static_call expr meth args =
  match expr with
  | Name n ->
      SEQ [
        (* evaluate that arguments *)
        SEQ (List.map gen_expr (List.rev args));
        (* get the address of the method *)
        GLOBAL (n.x_name ^"." ^ meth.x_name);
        (* Call the method *)
        CALLW (List.length args)
      ]
  | _ -> raise IncorrectSyntaxError

(* Generate the code for a non-static method call *)
and gen_call expr meth args =
  match expr with
  | Parent -> (* If the method is called on Parent, use the parent's
method *)
      SEQ [
        (* evaluate the arugments *)
        SEQ (List.map gen_expr (List.rev args));
        (* evaluate the calling object as an argument *)
        gen_expr expr;
        (* get the method from the parent *)
        GLOBAL (!p_name ^ "." ^ meth.x_name);
        (* call the method *)
        CALLW (List.length args + 1)
      ]
  | _ ->
      SEQ [
        (* evaluate the arugments *)
        SEQ (List.map gen_expr (List.rev args));
        (* evaluate the calling object *)
        gen_expr expr;
        (* duplicate, as the calling object is also and argument *)
        DUP 0;
        (* Load the class descriptor from the object *)
        LOADW;
        (* find the offset for the method *)
        gen_name_addr meth;
        (* load the method *)
        LOADW;
        (* call the method *)
        TYPE (type_name (Check.check_expr expr));
        CALLW (List.length args + 1)
      ]

(* Generate code for a conditional jump *)
and gen_cond tlab flab test =
  SEQ [
    gen_stack_maps (gen_expr test);
```

```
    unbox;
    CONST 0;
    JUMPC (Neq, tlab);
    JUMP flab
  ]

(*Generate code for a statement *)
and gen_stmt stmt =
  let rec code s =
    match s.s_guts with
    | Assign (e1, e2) -> SEQ [gen_stack_maps (SEQ [gen_expr e2; gen_addr
e1]); STOREW]
    | Delc (n, _, e) -> SEQ [gen_stack_maps (gen_expr e); gen_name_addr n;
STOREW]
    | Call e  -> gen_stack_maps (gen_expr e)
    | Return r ->
      begin
        match r with
        | Some e -> SEQ [gen_stack_maps (gen_expr e); RETURN 1]
        | None -> SEQ [RETURN 0]
      end
    | IfStmt (e, ts, fs) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [
          gen_cond lab1 lab2 e; (* Generate code for the condition *)
          LABEL lab1;
          gen_stmt ts; (* Generate code for the body *)
          JUMP lab3;
          LABEL lab2;
          gen_stmt fs; (* Generate code for the else body *)
          LABEL lab3;
        ]
    | WhileStmt (test, stmt) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [
          JUMP lab2;
          LABEL lab1;
          gen_stmt stmt; (* Generate code for  the body *)
          LABEL lab2;
          gen_cond lab1 lab3 test; (* Generate code for the condition *)
          LABEL lab3
        ]
    | ForStmt (init, step, test, body) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
          SEQ [
            code init; (* Generate code for the initial statement *)
            JUMP lab2;
            LABEL lab1;
            gen_stmt body; (* Generate code for the body *)
            gen_stmt step; (* Generate code for the step *)
            LABEL lab2;
            gen_cond lab1 lab3 test;
            LABEL lab3
          ]
```

```
    | Seq ss -> SEQ (List.map gen_stmt ss)
    | Nop -> NOP
  in SEQ [ LINE stmt.s_line; code stmt ]

(* Generate code for a method *)
and gen_method classname m =
  match m.m_origin with
  | Mine -> (* If it is a new method, generate the code *)
      SEQ [
        PROC (classname ^ "." ^ m.m_name.x_name, m.m_size, 0,
gen_proc_gc_map m.m_size (List.length m.m_arguments));
        Peepopt.optimise (gen_stmt m.m_body); (* Optimise the code *)
        END
      ]
  | Inherited _ -> NOP

(* Generate code for all methods in a class *)
and gen_methods c = p_name := get_pname c.c_ptype; SEQ (List.map
(gen_method c.c_name.x_name) c.c_methods)

(* Create a pointer to the correct procedure *)
and gen_method_name meth cls =
  match meth.m_origin with
  | Mine -> WORD (SYMBOL (cls.c_name.x_name ^ "." ^ meth.m_name.x_name))
  | Inherited n -> WORD (SYMBOL (n ^ "." ^ meth.m_name.x_name))

(* Generate a hex string from a char *)
and gen_hex_string c =
  let chr = Char.code c and hex = "0123456789ABCDEF" in
    (Char.escaped (hex.[chr / 16])) ^ (Char.escaped (hex.[chr mod 16]))

(* Create a hex string from a string *)
and fold_string s =
  match s with
  | "" -> ""
  | _ -> (gen_hex_string (s.[0])) ^ (fold_string (String.sub s 1
((String.length s) - 1)))

(* Generate a definintion in the data segment for a string constant *)
and gen_string (lab, s) =
  let strings = split_string s in
    let string_code = List.map (fun s -> STRING (fold_string s)) strings
in
      SEQ [COMMENT (sprintf "String \"%s\"" s); DEFINE lab; SEQ
string_code]

(* Split the a string into 32 bit chunks *)
and split_string s =
  let n = String.length s in
    if n > 31 then
      (String.sub s 0 32) :: (split_string (String.sub s 32 (n-32)))
    else [s ^ "\000"]

(* Generate code for a class *)
```

```
and gen_class c =
  let name = c.c_name.x_name in
  SEQ [
    COMMENT (sprintf "Descriptor for %s" name);
    DEFINE (name ^ ".%desc");
    WORD (SYMBOL (gen_class_gc_map c.c_size)); (* Create GC Map *)
    WORD (SYMBOL (name ^ ".%anc"));
    WORD (SYMBOL (name ^ ".%string"));
    SEQ (List.map (fun m -> gen_method_name m c) c.c_methods);
    COMMENT (sprintf "Ancestor Table for %s" name);
    DEFINE (name ^ ".%anc"); (* Create ancestor table *)
    WORD (DEC (1 + (List.length c.c_ancestors)));
    WORD (SYMBOL (name ^ ".%desc"));
    SEQ (List.map (fun c -> WORD (SYMBOL (c.c_name.x_name ^ ".%desc")))
c.c_ancestors);
    gen_string (name ^ ".%string", name);
  ]

(* Generate code for the whole program *)
and gen_program (Program(cs)) =
  SEQ [
    SEQ (List.map gen_methods cs);
    SEQ (List.map gen_class cs);
    SEQ (List.map gen_string !strtable);
    (* Create the MAIN method *)
    PROC ("MAIN", 0, 0, 0);
    GLOBAL !Analyse.mainMethod;
    CALLW 0;
    RETURN 0;
    END;
  ]
```

gc.ml

```
open Printf
open Syntax.Keiko
open Errors

(* Get 2^(n/4) - 1 *)
let rec gen_ones n =
  match n with
  | 0 -> 0
  | _ -> ((gen_ones (n-4)) lsl 1) + 1

(* Create the GC Map for a class *)
let gen_class_gc_map n =
  if n > 0 then sprintf "0x%X" (gen_ones (n+4))
  else "0"

(* Create the GC map for a procedure *)
let gen_proc_gc_map locals params =
  let local_off = 17 - (locals/4) and param_off = 20 in
```

```
    let gc = ((gen_ones locals) lsl local_off) lor ((gen_ones (params *
4)) lsl param_off) in
      if gc > 0 then gc+1 (* GC map must have 1 in the least-significant
bit *)
      else 0

(* Create a GC map from an evaluation stack, for a STKMAP instruction *)
let gen_stack_gc n stack =
  let rec hex m stk =
      match stk with
      | [] -> 0
      | true::st ->
          if m < 1 then ((hex m st) lsl 1) + 1
          else (hex (m-1) st) lsl 1
      | false::st -> (hex (m-1) st) lsl 1
  in let h = hex n stack in
    if h == 0 then NOP
    else STKMAP ((h lsl 1) + 1) (* GC map must have 1 in the least-
significant bit *)

(* Swap two values on the evaluation stack *)
let swap stk =
  match stk with
  | x::y::st -> y::x::st
  | _ -> raise InvalidExpression

(* duplicate a value on the evaluation stack *)
let dup n stk =
  let rec dup' n' stk' =
    match (n', stk') with
    | (0, s::_) -> s
    | (m, _::st) -> dup' (m-1) st
    | _ -> raise InvalidExpression
  in (dup' n stk)::stk

(* Simulate the evaluation stack and place any STKMAP instructions that
are needed *)
let gen_stack_maps code =
  let rec gen_map cd stack =
    match cd with
    | CONST _ | GLOBAL _ -> (cd, false::stack)
    | LOCAL _ -> (cd, true::stack)
    | LOADW -> (cd, true::(List.tl stack))
    | STOREW -> (cd, drop 2 stack)
    | CALLW n -> (* At call instruction, we may need to add a STKMAP *)
        let stkmap = gen_stack_gc n (List.tl stack) in
          (SEQ [stkmap; cd], true::(drop (n+1) stack))
    | CALL n -> (* At call instruction, we may need to add a STKMAP *)
        let stkmap = gen_stack_gc n (List.tl stack) in
          (SEQ [stkmap; cd], (drop (n+1) stack))
    | BINOP _ -> (cd, false::(drop 2 stack))
    | MONOP _ -> (cd, false::(List.tl stack))
    | OFFSET -> (cd, true::(drop 2 stack))
    | SWAP -> (cd, swap stack)
```

```
    | DUP n -> (cd, dup n stack)
    | SEQ ss ->
        let (kcode, stk) = List.fold_left (fun (k', s) k -> let (nk, s') =
gen_map k s in (k' @ [nk], s')) ([], stack) ss in
          (SEQ kcode, stk)
    | NOP | TYPE _ -> (cd, stack)
    | s -> print_keiko s; raise InvalidExpression
  in let (k, _) = gen_map code [] in k
```

Appendix B

The changes required to make arrays covariant.

arrays.ml

```
open Syntax.Tree
open Syntax.Keiko
open Errors
open Printf

(* Set of all arrays in the program *)
module ArraySet = Set.Make(
  struct
    let compare = compare
    type t = def_type
  end )

let arraytypes = ref ArraySet.empty

(* Generate the ancestor types of an array *)
let rec anc_types t =
  match t with
  | ClassType c | GenericClassType (c, _) -> t :: (anc_types c.c_ptype)
  | ArrayType d -> List.map (fun t' -> ArrayType t') (anc_types d)
  | VoidType -> []
  | _ -> raise IncorrectSyntaxError

(* Add and array type the set *)
let rec add_type t =
  match t with
  | ClassType _ -> ()
  | ArrayType d ->
      arraytypes := List.fold_right ArraySet.add (anc_types t)
!arraytypes; (* Need to add all the ancestors *)
      add_type d (*Need to add the elements of the array *)
  | GenericType (_, _) | GenericClassType (_, _) -> raise
GenericArrayError (* Cannot have arrays of generics *)
  | _ -> raise IncorrectSyntaxError

(* Make a name for an array class descriptor *)
let rec make_name t =
  match t with
```

```
  | ClassType c | GenericClassType (c, _) -> c.c_name.x_name
  | ArrayType d -> sprintf "array.%%%s" (make_name d)
  | _ -> raise IncorrectSyntaxError

(* Generate the names of the ancestor table *)
let rec gen_anc t =
  match t with
  | ClassType c | GenericClassType (c, _) -> c.c_name.x_name :: (gen_anc
c.c_ptype)
  | ArrayType d -> List.map (fun s -> sprintf "array.%%%s" s) (gen_anc d)
  | GenericType (_, d) -> gen_anc d
  | VoidType -> []
  | _ -> raise IncorrectSyntaxError

(* Generate a hex string from a char *)
and gen_hex_string c =
  let chr = Char.code c and hex = "0123456789ABCDEF" in
    (Char.escaped (hex.[chr / 16])) ^ (Char.escaped (hex.[chr mod 16]))

(* Create a hex string from a string *)
and fold_string s =
  match s with
  | "" -> ""
  | _ -> (gen_hex_string (s.[0])) ^ (fold_string (String.sub s 1
((String.length s) - 1)))

(* Generate a definintion in the data segment for a string constant *)
and gen_string (lab, s) =
  let strings = split_string s in
    let string_code = List.map (fun s -> STRING (fold_string s)) strings
in
      SEQ [COMMENT (sprintf "String \"%s\"" s); DEFINE lab; SEQ
string_code]

(* Split the a string into 32 bit chunks *)
and split_string s =
  let n = String.length s in
    if n > 31 then
      (String.sub s 0 32) :: (split_string (String.sub s 32 (n-32)))
    else [s ^ "\000"]

(* Generate the code for the array class descriptors *)
let create_fake_classes () =
  let gen_descriptor (ArrayType t) =
    let tn = (make_name t) and anc = gen_anc (ArrayType t) and pn =
Check.print_type (ArrayType t) in
      let n = "array.%" ^ tn in
        SEQ [
          COMMENT ("Descriptor for " ^ pn);
          DEFINE (n ^ ".%desc");
          WORD (SYMBOL "0xF");
          WORD (SYMBOL (n ^ ".%anc"));
          WORD (SYMBOL (n ^ ".%string"));
          WORD (SYMBOL (tn ^ ".%desc"));
```

```
          WORD (SYMBOL "Object.equals");
          WORD (SYMBOL "Object.GetType");
          WORD (SYMBOL "Object.Is");
          WORD (SYMBOL "Object.Print");
          COMMENT ("Ancestor table for " ^ pn);
          DEFINE (n ^ ".%anc");
          WORD (DEC ((List.length anc) + 1));
          SEQ (List.map (fun s -> WORD (SYMBOL (s ^ ".%desc"))) anc);
          WORD (SYMBOL ("Object.%desc"));
          gen_string (n ^ ".%string", pn)
        ]
  in SEQ (List.map gen_descriptor (ArraySet.elements !arraytypes))
```

Changes to analyse.ml

```
...
let vtableOffset = 16 (* offset of the vtable in the class descriptor *)
...
  | NewArray (n, e) ->
      ignore(annotate_expr e env);
      let d = get_type n.x_def.d_type env in
        (* Create fake classes from d *)
        add_type d;
        n.x_def <- create_def ClassDef d; d (* Names following New are
classes *)
...
```

Changes to check.ml

```
...
let check_compatible type1 type2 =
  let pt1 = ref VoidType and pt2 = ref VoidType in
  ...
    let rec check t1 t2 =
      match (t1, t2) with
      ...
      | (ArrayType d1, ArrayType d2) -> check d1 d2 (* Arrays are
covariant *)
      ...
      | _ -> raise (TypeError((print_type !pt1), (print_type !pt2)))
    in pt1 := type1; pt2 := type2; check type1 type2
  ...
```

Changes to codegen.ml

```
...
and gen_addr expr =
  let expr =
    match expr with
    ...
    | Sub (e3, e4) ->
        SEQ [
          (* Get the array the we are indexing from *)
          gen_expr e3;
          (* Type check the element we are adding *)
```

```
          DUP 1;
          DUP 1;
          LOADW;
          CONST 12;
          OFFSET;
          LOADW;
          GLOBAL "baozi.%typeCheck";
          CALL 2;
          (* Get the "Data" *)
          CONST 4;
          OFFSET;
          LOADW;
          (* Get the index *)
          gen_expr e4;
          unbox;
          (* The element we want is at offset 4*i *)
          CONST 4;
          BINOP Times;
          OFFSET;
        ]
...
and gen_class c =
  let name = c.c_name.x_name in
  SEQ [
    COMMENT (sprintf "Descriptor for %s" name);
    DEFINE (name ^ ".%desc");
    WORD (SYMBOL (gen_class_gc_map c.c_size)); (* Create GC Map *)
    WORD (SYMBOL (name ^ ".%anc"));
    WORD (SYMBOL (name ^ ".%string"));
    WORD (DEC 0);
    SEQ (List.map (fun m -> gen_method_name m c) c.c_methods);
    COMMENT (sprintf "Ancestor Table for %s" name);
    DEFINE (name ^ ".%anc"); (* Create ancestor table *)
    WORD (DEC (1 + (List.length c.c_ancestors)));
    WORD (SYMBOL (name ^ ".%desc"));
    SEQ (List.map (fun c -> WORD (SYMBOL (c.c_name.x_name ^ ".%desc")))
c.c_ancestors);
    Arrays.gen_string (name ^ ".%string", name);
  ]

and gen_program (Program(cs)) =
  SEQ [
    SEQ (List.map gen_methods cs);
    SEQ (List.map gen_class cs);
    SEQ (List.map Arrays.gen_string !strtable);
    Arrays.create_fake_classes ();
    (* Create the MAIN method *)
    COMMENT "MAIN procedure";
    PROC ("MAIN", 0, 0, 0);
    GLOBAL !Analyse.mainMethod;
    CALLW 0;
    RETURN 0;
    END;
  ]
```