Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Type checker for the Yon language

Bachelor Thesis

2013

Tomáš Belan

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Type checker for the Yon language

Bachelor Thesis

Study programme: Informatics
Study field: 2508 Informatics
Department: Department of Computer Science
Supervisor: RNDr. Peter Borovanský, PhD.

Bratislava, 2013

Tomáš Belan

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Tomáš Belan
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Type checker for the Yon language

Aim: The main aim of this thesis should be a type extension of the existing programming language called Yon.
A new statically-based type system (implemented by a new type-checker) is expected as a result of the thesis.
The designed type-checker should have the following aspects:
- any variable or expression is statically typed during the compilation time,
- incremental compilation of modules is supported,
- this language extension supports the a sub-type polymorphism (object inheritance with interfaces), and also
a parametric polymorphism (i.e. generics).
The type inference of this type system should be considered in this thesis.

Annotation: The programming language Yon is Michal Antonic's master thesis result , already used in the education process.
The original language design provided a non-incremental type-checker, tool slow for a real application development.
This thesis should concentrate to the following aspects:
- tiny language extensions or necessary modifications of some language constructions,
- an implementation of the incremental type-checker.

Supervisor:	RNDr. Peter Borovanský, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. PhDr. Ján Rybár, PhD.
Assigned:	11.10.2012
Approved:	24.10.2012

doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

Student

Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Tomáš Belan
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Typechecker pre jazyk Yon

Cieľ: Rozšírenie jazyka Yon a jeho kompilátora o typový systém a typovú kontrolu. Jeho hlavné rysy budú pokrývať nasledujúce aspekty:
- pre každý výraz a premennú skontroluje počas kompilácie statický typ,
- dokáže inkrementálne typovať zdrojové súbory (t.j. po jednom, nie vždy len celý program naraz),
- podporuje podtypový polymorfizmus (objekty implementujúce spoločné rozhranie),
- podporuje parametrický polymorfizmus (tzv. generics).

Cieľom práce je tiež preskúmať možnosti implementácie typovej inferencie v takomto typovom systéme.

Anotácia: Programovací jazyk Yon je výsledkom diplomovej práce Michal Antoniča. Experimentálne bol nasadený do výuky. Napriek tomu, že v pôvodnej autorovej implementácii bol riešený problém typovej kontroly, implementovaný algoritmus nie je inkrementálny, a pre realne použitie pri práci s jazykom je teda nepoužitelne pomalý. Navrhovaná práca má práve riešiť tento problém, a to v dvoch rovinách: jemnou úpravou jazyka Yon a implementáciou inkrementálneho typecheckera.

Vedúci: RNDr. Peter Borovanský, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. PhDr. Ján Rybár, PhD.

Dátum zadania: 11.10.2012

Dátum schválenia: 24.10.2012

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgement

I would like to thank my supervisor RNDr. Peter Borovanský, PhD. for his guidance and advice.

Abstract

In this thesis we extend the dynamically typed language Yon with a static type system, allowing checking the program at compilation. The dynamic language semantics are largely kept unchanged. The type system is structural and supports generics, inheritance and limited type inference.

Keywords: programming languages, type system, subtyping, polymorphism

Abstrakt

V tejto práci rozšírime dynamicky typovaný jazyk Yon o statický typový systém, umožňujúci kontrolu programu počas kompilácie. Pritom z veľkej časti zachováme dynamickú sémantiku jazyka. Typový systém je štrukturálny a podporuje generické typy, dedičnosť a limitované automatické odvodzovanie typov.

Kľúčové slová: programovacie jazyky, typový systém, podtypy, polymorfizmus

Contents

Introduction

Programming language designers face a dilemma: should the type system be static, or dynamic? In other words, should type checking be done at compile time, or run time? Dynamic type systems are said to allow faster prototyping, and enable various other features. However, the safety provided by static type systems helps to detect many errors early.

Yon [2] is a multi-paradigm programming language similar in semantics to dynamically typed languages such as Python and Lua. Though Yon is, for the most part, dynamically typed, here we extend it with a static type system, while attempting to keep keep the desirable properties of dynamic languages. Accordingly, the type system needs to be expressive (so that it can faithfully represent complex scenarios) and yet succint (so that it does not get in the way when it is not needed).

Though Yon already had an older static type checker, it turned out to be unsatisfactory for multiple reasons. One is that it didn not support parametric polymorphism (generics), so it could not infer or check types of items in Arrays etc. Another disadvantage of the old algorithm was that it was unable to operate on a per-file basis. The whole program had to be reprocessed if any module was modified.

The new type system is structural – type compatibility and equivalence is determined not by how a type is named, but its actual features. If different classes support the same set of methods, they are considered to be the same type. And a class does not have to explicitly declare it is implementing an interface – simply implementing the methods specified by the interface is enough.

The rest of this thesis is structured as follows:

Chapter 1 provides an overview of Yon and its runtime semantics, as implemented by the virtual machine.

Chapter 2 introduces the new type system used by Yon, and describes the relationships between types.

Chapter 3 details the class system built on top of the type system from Chapter 2, and explains Yon's support for object-oriented programming.

Chapter 4 gives an overview of the type checker's implementation and internal data structures.

Chapter 1

Overview of Yon

Yon [2] is an experimental programming language created by Michal Antonič in 2011. This chapter provides a short overview of its features and semantics.

Yon programs are first compiled into bytecode and then run by a virtual machine. This overview focuses on the runtime semantics of Yon, as implemented by the virtual machine. Yon's syntax and type system are discussed elsewhere.

A Yon program manipulates various values, or objects. Yon uses these terms synonymously – every Yon value is an object. Objects can be given names by assigning them to variables. Assignment does not create a copy of an object, only a reference to it. The same object can be in multiple variables.

Some languages distinguish between *records* (also called *structs*) and other values. In Yon, every object is a record – it contains a collection of *slots*, or fields, which hold other objects. Slots are unordered. They are indexed by identifiers, and can be assigned to. For instance, an object representing a person may have a slot named `age` which can be assigned to by writing `person.age := 7`.

Functions in Yon are also objects. Calling a function with the right number of arguments yields a return value. Because a function is an object, it can have slots. For example, `sqrt` is a function and `sqrt.string` is a string describing it.

Other objects can be callable as well. For example, classes (see Chapter 3) are also callable first-class objects. Calling a class creates an instance of that class. `p := Person("John Smith")` looks up the value of the variable `Person` (the class), calls it with a string argument, and stores the result (the new person object) in `p`.

The majority of a normal Yon program consists of reading and assigning to variables and slots, constructing literals (including function literals or lambdas) and calling functions or other callables. There are no bytecode instructions for calling a method or using an operator. Calling a method such as `person.move("London")` is just accessing the `person.move` slot and calling the value retrieved from the slot with one string argument. And operators

such as "+", "-", "!=" are just specially named methods with a different syntax. From the virtual machine's point of view, they are called like any other. This includes the array access operators "[]" and "[]:=". The bytecode generated for the expression "`arr[idx] := val`" reads the `[]:=` slot of `arr`, and then calls the slot value with two arguments, `idx` and `val`.

1.1 Operations

The *operations* of an object are the things that can be done to it. We have seen that objects are records with named slots. So if an object has a slot named *s*, accessing (reading from or writing to) that slot is an operation. Calling a callable object is also an operation. The list of an object's operations does not directly include method calls, because method calls are not something the virtual machine knows about – calling a method is just reading a slot and then by calling the retrieved value. But those two are operations of their own.

Here are some examples:

- `{10, 20, 30}` is a list of numbers. It has a slot named `size`. Thus, accessing a slot named `size` is one of the operations it supports.
- `sqrt` is a function. Calling it with one argument is one of its operations. It also supports other operations, such as accessing a slot named `string`, because functions are records too.
- `3` is a number. The programmer can read its slot named + and call the slot value with another number, yielding their sum. So accessing a slot named + is one of `3`'s operations, and the operations of the slot's value include being called with one argument.
- `3` is a number. It is not callable, but it is a record – it has slots such as +, - and many others. Furthermore, built-in functions provided directly by the virtual machine, such as `sqrt`, can access its actual unboxed value and do computations with it. Therefore, we count "accessing the primitive numeric value" amongst its operations.

  Even if the programmer constructs a "fake number" object that has exactly the same slots as numbers, built-in functions such as `sqrt` can recognize that it is not really a number. Only the virtual machine can construct real number objects.

Operations are important because they describe the external interface of objects. An object can be replaced with another one with the same set of operations, even if their origins or implementations differ. Though the above description of operations is still only on the level of the virtual machine, and the type checker uses a modified definition of operations, this principle is still preserved. Whether a value belongs to a type depends on its operations, not on the implementation of its methods or other factors. Section 2.2 goes into more detail.

1.2 Functions

Some languages have a separate namespace for functions and for variables – a function and a variable can have a same name and yet be different things. But Yon functions are first-class values stored in variables. A Yon expression such as `sqrt(PI)` involves reading the global variables `sqrt` and `PI` and calling the value of `sqrt` (a function) with the value of `PI` (a number) as an argument.

Functions can defined and named using function literals:

```
function add_five(x:Number)->Number
    print "adding five"
    x + 5
```

But this is actually syntactic sugar for declaring a variable, creating an anonymous function and assigning it to the variable:

```
var add_five (function(Number)->Number)

add_five := function (x:Number)->Number
    print "adding five"
    x + 5
```

The above examples also contain type annotations that specify the type of the argument `x`, the type of the function's return value, and the type of the `add_five` function itself. Yon's type system is the subject of Chapter 2.

Yon functions are lexical closures, as seen in Scheme [4] and other languages. When a function is created, it keeps a reference to the lexical environment it was created in, and can read and assign to its variables. The following Yon code implements the classic example of a counter made with closures:

```
function makeCounter()->function()->Number
    var value := 0
    return function ()->Number
        value := value + 1
        return value

var c1 := makeCounter()
var c2 := makeCounter()

c1()   # returns 1
```

```
c1()   # returns 2
c1()   # returns 3
c2()   # returns 1
c1()   # returns 4
```

Each call of `makeCounter` creates a new stack frame with a new lexical environment, in which the variable `value` is created. The anonymous function is then bound to this enviroment, returned and stored in either `c1` or `c2`. Thus, `c1` and `c2` are functions with the same body, but bound to different environments with different `value`s in them.

By default, a function returns the value of its last expression. One can return from a function early by using a return expression, analogous to return statements in other languages. When `return e` is evaluated, the execution of the function stops and `e` becomes its return value.

The `yield` keyword provides support for coroutines. Evaluating `yield e` suspends the function and returns `e` to the caller. When the suspended function is called again, execution resumes after the `yield` instead of starting at the beginning.

The expressions `return e from f` and `yield e from f` allow returning values from other functions deeper on the call stack. Further discussion is in the original thesis [2].

1.3 Prototypes

The old version of Yon presented in Michal Antonič's thesis used a prototype-based inheritance model. In this model, every object could be linked with a *prototype*. When reading a non-existent slot of an object, that slot access would be delegated to the object's prototype. If the prototype also did not contain such a slot, its prototype would be checked in turn, etc.

Here is an example to clarify the behavior of objects with prototypes. We start with two empty objects `o` and `p`, with `p` being `o`'s prototype.

```
o.a := 10    # slot "a" is created in o
o.b := 20    # slot "b" is created in o
p.b := 30    # slot "b" is created in p
p.c := 40    # slot "c" is created in p
o.a          # evaluates to 10
o.b          # evaluates to 20
o.c          # evaluates to 40 -- the value of p.c,
             # because slot "c" is not present in o
p.a          # evaluates to nil (the slot is not found)
o.d          # evaluates to nil (the slot is not found)
```

```
o.c := 50    # slot "c" is created in o, but
             # the value in p remains unchanged
o.c          # evaluates to 50
p.c          # still evaluates to 40
p.d := 60    # slot "d" is created in p
o.d          # evaluates to 60 (the slot is now found in p)
```

Though the language internally used prototypes, this was mostly invisible to the programmer. Yon programs used a class-based system built on top of prototypes.

However, because of the requirements of the new type system, prototypes had to be replaced with another mechanism – proxy assignments. Though the programmer can still use a similar class-based model, it now uses proxy assignments instead of prototypes as a different building block.

1.4 Proxy assignment

For the most part, Yon uses the call-by-value evaluation strategy. When evaluating an expression composed of subexpressions, the subexpressions are evaluated first. For example, when calling a function, the function and the arguments are evaluated before the function is called. Similarly, for classic assignments of the form `lvalue := expression` (where the lvalue is either a variable or an object slot) the expression is evaluated and then stored in the lvalue.

There are some exceptions to this rule. The `if` expression has three subexpressions: the condition, the "then" clause and the "else" clause. Only the condition is evaluated immediately, and the result of this evaluation decides which of the other clauses is evaluated then.

Proxy assignment is another such exception. It is similar to classic assignment, but the assigned expression is not evaluated until the lvalue is read. With classic assignment, the right-hand expression is evaluted and its value is stored. Reading the variable or slot returns the value, until it is overwritten with another assignment. With proxy assignment, the right-hand expression is not evaluated before storing it. Instead, every time the variable or slot is read, the right-hand expression is evaluated again, until the variable is overwritten again (using either classic or proxy assignment).

Although there is an instruction for proxy assignment in Yon's bytecode, the facility is not directly available for the programmer. There is no syntax that would produce the proxy assignment instruction. It is only created indirectly as part of class definitions. But even if proxy assignment has no syntax, it is helpful to invent a way of writing it for the purposes of this text, so that its semantics can be better explained and demonstrated. Therefore, we will write proxy assignment as `lvalue :=& expression`.

We will now show an example of proxy assignment behavior.

```
var a (Number)
var b (Number)
a := 2
b :=& a + 1    # the addition is not evaluated yet
b              # evaluates to 3 (by computing 2 + 1)
b              # evaluates to 3 (computing 2 + 1 again)
a := 5
b              # evaluates to 6

b := 11        # b was overwritten -- no longer linked to a
b              # evaluates to 11
a := 7
b              # still evaluates to 11

function logged(x:Number)->Number
    print "hello"
    x
b := logged(a)
b              # prints "hello" and returns 7
b              # prints "hello" again and returns 7

b :=& b
b              # loops forever
```

The reader may notice that the delayed evaluation caused by proxy assignment behaves similarly to function literals. The body of a function is only executed when it is called – and similarly, the right-hand expression of proxy assignment is only executed when the variable is read. This is actually how proxy assignment is implemented. With classic assignment, the virtual machine evaluates the right-hand expression and stores it in the variable. With proxy assignment, the virtual machine stores an anonymous function, marked as a *proxy*, in the variable. Then, when the virtual machine reads a variable and finds a proxy in it, it calls it and returns its return value instead of the proxy.

Because proxy assignment creates an anonymous function, all the behavior exhibited by functions (as detailed in Section 1.2) is present. The right-hand side of proxy assignment (the body of the anonymous function) can access local variables from the environment, because functions can be closures.

1.5 Old type system

The original thesis on Yon by Michal Antonič presented a type checker based on the Cartesian Product Algorithm [1]. The set of types considered by this algorithm is simply the set of classes in the program. Every value has a specific type. However, the algorithm still supports a form of generics and polymorphism by allowing variables to contain values of different types. Instead of inferring one type for each variable and then making sure only that type is ever assigned to it, the algorithm infers the set of all types the variable can hold during program execution. Function arguments can have multiple types too, and the return type of a function can also change depending on the argument types.

For instance, consider "`function add(x, y) [x + y]`". The arguments `x`, `y` can hold various types. For instance, one part of the program might use `add(1, 2)` and another might use `add("hello ", "world")`. CPA correctly infers that for the first invocation, the result is a number, while for the second one, it is a string. Thus, `add` exhibits a degree of polymorphism.

One of the problems of this algorithm is that it cannot check modules or libraries one by one. The whole program must be checked at once. To compute the type set information for a function, the CPA algorithm must process every place the function is called from.

Incremental compilation is one of the main reasons for splitting a program into modules and libraries. In C and similar languages, the compilation happens in two steps: first, each source file is compiled to an object file, then, the object files are linked together. When a single file changes, the build time is much less than what would be needed to rebuild the whole program. All that is needed is to rebuild the changed module and link everything together.

Even languages with type inference, such as ML or Haskell, generally run their type inference algorithms separately for each source file. When compiling a module, all type information of its dependencies is already known, but the module cannot rely on information given by other modules that depend on it.

But for Yon's old CPA-based type checker, something like this is impossible, or at least very difficult. If even a single file changes, this can change the type information of its dependencies. CPA cannot be used incrementally, module by module.

Support for incremental type checking is one of the main goals for the new type checker. Further motivation is explored in Section 2.1.

Chapter 2

New type system

The main aim of this thesis is to replace the CPA-based type system with a new static type system. Though the details are specific to Yon, the lessons learned can be applied to other languages with similar semantics.

2.1 Goals

In the old type system, there was no interface inheritance, only class-based single inheritance. There were also no type schemes and generic classes such as `Array<T>`. A type was simply a class name. But this inflexibility with regard to the allowed definitions of types and their relations allowed the use of a type inference algorithm.

The new type system takes a different set of tradeoffs. Though it requires explicit type annotations, it provides a richer "language of types". The design goals of the new type system are as follows:

- Though Yon's semantics can be changed when necessary, we want to preserve them as much as possible. Yon is a dynamic language, and the runtime allows even things such as modifying a method of a class or a specific instance. It is important to preserve as much of this dynamic behavior as possible and provide type checking for it.
- As already mentioned, a big disadvantage of the old type system is its inability to run incrementally, module by module. The new type system must be able to do this.
- Support for polymorphic functions is a must. But we cannot typecheck the body of a function again every time it is used with new argument types, as was done in the old type checker. Since values now have specific types instead of type sets, the type system has to be able to express universal type.
- Generic classes and interfaces (type schemes) must also be supported.

2.2 Types

The purpose of a static type checker is to detect a certain class of program errors without needing to run the program. Of course, static program analysis tools cannot detect all errors, because the halting problem is undecidable.

Yon's type checker cannot find infinite loops or off-by-one errors, but it can still verify that the operations (as introduced in Section 1.1) used by the program are valid for the values they are performed on. The programmer has to prove to the type checker that every slot access and function call in the program is valid. In other words, if the program attempts to read `p.name`, the type checker verifies whether `p` contains a slot named `name`. If it cannot prove that such a slot exists using the information given, the program will be rejected.

Most values support many operations. If the program accesses `p.name`, `p.age` and `p.location`, the programmer has to prove that `p` has all three slots. Because "the set of values whose operations include accessing slots named `name`, `age` and `location`" is needlessly long, Yon allows the programmer to give a symbolic name to this list of operations and simply use "the set of all `Person`s". This is called defining a type.

A type is the binding of a *type name* (such as `Person`) to a *type definition* – a list of operations. This list of operations defines a set of values that belong to the type. A value belongs to a type if and only if it supports all the operations specified by the type definition. It can also support other operations. So with a `Person` type defined as above, the values that belong to type `Person` are those that have a `name`, an `age` and a `location`, and optionally support other operations (they can have other slots, or they might be callable, etc).

Yon does not have type aliases – a type name cannot just point to a different type. Every type has a name and a definition. The namespace of type names is separate from the namespace of variables. A type and a variable can have the same name.

Usually, a type name is simply an identifier – a string of letters and numbers, such as `Person`. Unlike variable names, type names can also contain dots. But in later sections, we will see complex type names such as `Pair<String,Number>`. In this case, the whole string `Pair<String,Number>` is the type name, not just `Pair`.

When the programmer declares a variable (or a function argument), they have to specify its type. The type checker uses this information to ensure all operations with the variable are correct:

```
var p (Person)
print p.name    # valid
print p.namw    # invalid!
```

Because the programmer declared that `p` will only contain values that belong to type `Person`, the type checker knows that accessing `p.name` is valid and will not cause a runtime

error. But because `Person` does not specify a slot named `namw`, programs that attempt to use `p.namw` are rejected.

But what about compound expressions, such as `p.name.size`? The type checker can verify that `p` has a `name` slot – accessing a slot named `name` is one of its supported operations. But our old definition of operations does not specify anything about the value stored in the slot. The type checker cannot verify that `p.name` is an object with a `size` slot.

Clearly, the original description of operations from Section 1.1 is not enough. We need to extend it with type information. By the previous description, "accessing a slot named s" was an operation. The operation also needs to specify the type T of the value stored in the slot. The other kinds of operations need to be modified analogously.

By the extended definition, an operation is one of the following:

- access a slot named s containg values of type T (for reading and writing),
- call with n arguments of types $a_1, \ldots, a_n$, returning a value of type r,
- access primitive numeric/string/etc. object value (see Section 1.1).

The old `Person` type definition specified that a `Person` value had to have slots named `name`, `age` and `location`. But the value in `name` could be of any type, so the type checker could not prove that `p.name.size` will always be valid.

Now that we redefined operations to include type information, we can say that `Person` must have slots `name` and `location` of type `String` and a slot `age` of type `Number`. The type definition of `Person` consists of these three operations.

With this additional information, the type checker knows that if `p` is a `Person`, `p.name` is a `String` and `p.name.size` is a `Number` (according the type definition of `String`, which is built into the language).

Some types are built into the language, others are defined by the programmer. The programmer can define new types using the `interface` keyword. The `Person` type from above can be defined like this:

```
interface Person
    slot name (String)
    slot location (String)
    slot age (Number)
```

The keyword `interface` is used to emphasize that whether a value belongs to a type depends only on the set of operations it supports – the value's external interface. It does not

matter how the value was constructed or what the slots actually contain (e.g. the implementation of its functions).

Type definitions can also recursively refer to themselves, and mutual recursion between multiple types is supported as well.

```
interface LinkedListOfStrings
    slot value (String)
    slot next (LinkedListOfStrings)
```

A type definition can "inherit" operations from another type, while adding more:

```
interface Employee(Person)
    slot company (String)
```

This is just syntactic sugar for copying the slots of `Person` into the definition of `Employee`. The two types do not share any special link because one was defined using the other.

This feature can be used to create something like a type alias. If an `interface` block adds no new operations, both type will have the same definition. But they are still considered to be distinct types.

Yon also provides several built-in types. Some of them, such as `Number` and `String`, correspond to special objects created by the virtual machine that can be processed by primitive functions such as `sqrt`. The type definition of `Number` includes the operation of "accessing the object's primitive numeric value", so only objects that really have a numeric value can be given to `sqrt` as an argument.

The base type is named `Top`. It is built-in and its definition is empty – it has no operations. All values belong to it, but nothing can be done with a `Top` variable.

There is also a type named `Object`, but this is actually the root of the class hierarchy, as described in Chapter 3. There are some values that are not of the type `Object`, but everything belongs to `Top`.

Yon also provides built-in types for functions and callable objects. Until now, all the type names we have seen were just simple identifiers like `String` and `Person`. The type names of functions and callables have a more complex syntax. For example, the string "`callable(String)->Person`" names a valid type, and the definition of this type has a single operation: the value must support being called with a `String` and returning a `Person`. So all callables that can accept a string and return a person belong to this type.

In general: if `A1`, . . ., `An`, `R` are valid type names of existing types (whether built-in or programmer defined), `callable(A1,...,An)->R` is also a type name. All values that can be called with arguments of types $A1, ..., An$ and return a value of type R belong to it.

For example, the standard library includes a variable named `sqrt`. The variable is declared with type `callable(Number)->Number`. The type checker thus knows that calling the value of this variable with a `Number` argument is valid, and the result will be another `Number`.

Type names of the form `function(A1,...,An)->R` behave similarly, but the definition of such a type also contains additional slots that are present on functions, but might not be on all callable objects. As all objects, not just functions, can be callable (see Chapter 1), this distinction can be important. For example, the standard library gives all functions a `Boolean` slot named `isFrozen`, which specifies whether execution of the function is currently suspended with a `yield` statement. Normal Yon functions have this slot, but it is not necessarily present in other callable objects.

2.3 Type constructors

Type constructors allow building new types from old ones. Applying a type constructor to suitable type arguments results in a new type. The new type's list of operations is obtained by substituting the given types for the type arguments in the type constructor's definition. For instance, if the program contains the following `interface`:

```
interface Pair<A, B>
    slot first (A)
    slot second (B)
```

then `Pair<Number,String>` is a valid type name, and its definition has two operations: accessing the slot `first` of type `Number`, and accessing the slot `second` of type `String`.

The type constructor is named `Pair`, but on its own, `Pair` is not a type name, or at least, not a name of an existing type. A variable might be a `Pair<Number,String>`, but it cannot be just `Pair`.

The arguments of a type constructor must be valid type names. `Foo<Pair>` is invalid, because `Pair` is just the type constructor, not a full type name. But anything that is a type name can be used, even type names that use a type constructor. `Pair<Bool,Bool>` is a valid type name, so `Pair<Pair<Bool,Bool>,Number>` is valid too.

Some define type constructors to be anything that allows building complex types from more basic ones. By this definition, `callable(...)->R` and `function(...)->R` described in Section 2.2 could also be considered type constructors. We will not use this definition. In this work, the term *type constructors* will only mean things like `Pair`, not `callable` and `function`.

2.4 Polymorphism

Yon supports polymorphic functions (and other callable objects). The arguments of normal functions are expressions. Polymorphic functions also have *type arguments*. When calling a polymorphic function, the caller has to provide suitable type names. The function can use the type arguments in its body and the argument list.

Normal arguments are written in parentheses. Type arguments are written in angle brackets before the normal argument list.

The identity function is an example of a polymorphic function:

```
function id<T>(a:T)->T
    a


id<String>("x")    # returns "x"
id<Number>(3)      # returns 3
```

`id` can be used with different types being substituted for `T`. The type of the function `id` is written as `function<T>(T)->T`.

We can use bounded quantification to limit the range of types that can be specified in a type argument:

```
interface NamedThing
    slot name (String)


function verboseId<Z is NamedThing>(a:Z)->Z
    print a.name
    a
```

The type checker will ensure that callers of `verboseId` always use a `Z` that is a subtype of `NamedThing`. Subtypes will be defined in Section 2.6.

The type checker allows accessing `a.name` because it knows `a` is a `NamedThing`. The type of the function `verboseId` is `function<Z is NamedThing>(Z)->Z`.

We have seen that function type names are of the form `function(A1,...,An)->R`. Now that we have polymorphic functions, we extend this to `function<T1 is B1,...,Tm is Bm>(A1,...,An)->R`, where `T1,...,Tm` are identifiers and `B1,...,Bm,A1,...,An,R` are other type names. The "`is Bi`" quantification bounds are optional. Polymorphic callables are similar, with the keyword `function` replaced by `callable`.

Polymorphic function types are also called "universal types". `id` is a polymorphic function that, *for all* types `T`, accepts `T` and returns `T`. In polymorphic lambda calculus, this would be written as $\forall T : (T, T) \to T$).

A polymorphic function call, such as `id<String>("x")` in the example above, is actually a composition of two operations: a *type instantiation* (also called *type application*), which turns a polymorphic function such as `function<T>(T)->T` into a specific callable type such as `callable(String)->String`, followed by a normal call.

(The result of instantiation is always just `callable`, never `function` – see below. This was done for ease of implementation and to avoid edge cases where instantiating `function` would cause problems.)

A type instantiation changes the type of an expression, but for the virtual machine, it is a no-op. It has no bytecode instruction. The virtual machine considers `id` and `id<String>` to be the exact same object – they only behave differently during type checking. `id` cannot be directly called, and `id<String>` cannot be instantiated with a different type argument anymore.

As type instantiation is a thing that can be done to some values and cannot be done to others, it is clear that the definition of operations from Section 2.2 needs to be extended once again. Normal function calls were already included – they are now joined by type instantiation.

Yon does not support polymorphic objects. More precisely, although the value that is being type instantiated can be an object (because after all, functions are also objects), the result of type instantiation is always just a "`callable(...)->...`". The type checker "forgets" about the slots the objects had.

Consider an object `c` which can be called as in `c<Number>(5)` and also has a slot `c.s`. Because `c<Number>` is of type `callable(Number)->Number`, which does not have a slot `s` in its definition, writing `c<Number>.s` would cause a type error.

2.5 α-equivalence

It is sometimes necessary to compare type names (as opposed to their definitions). However, type names that only differ in the specific identifiers used for quantification should be regarded as the same. This notion is captured by the *α-equivalence* relation.

Two type names are α-equivalent if both can be transformed to the same type name by repeatedly substituting fresh variables for variables bound in quantifiers.

For example, the types `callable<X,Y>(X)->Y` and `callable<Y,X>(Y)->X` are α-equivalent.

2.6 Subtyping

If a type S is a subtype of type T, all values that belong to S also belong to T. The subtype relation is denoted as "`S <: T`".

Subtyping is used for checking assignment and similar cases. If the program tries to assign the result of an expression of type `Employee` to a variable of type `Person`, the type checker can prove the validity of the assignment. Even though the types are not the same, if `Employee` is a subtype of `Person`, the type checker knows that the value of the right-hand side expression will be a `Person` and thus can be stored in a `Person` variable. Similarly, if a function expects an argument of type `Person`, it is sound to give it a value of type `Employee`.

In Yon, the definition of the subtyping relation is as follows: `S <: T` if and only if, for each operation in the definition of T, S also has the operation, or an α-equivalent one.

For example, `function<T>(T)->Number` is a subtype of `callable<Q>(Q)->Number`. The latter only supports one operation – it can be type instantiated with a type `Q`, yielding a function that receives a `Q` value and returns a number. `function<T>(T)->Number` can also do this. The type argument is named differently, but α-equivalence takes care of that.

`function<T>(T)->Number` also supports other operations, such as accessing a `Bool` slot named `isFrozen` (see Section 2.2). The definition of `callable<Q>(Q)->Number` does not contain such a slot. Therefore, `callable<Q>(Q)->Number` is not a subtype of `function<T>(T)->Number`.

Yon's definition of subtyping is more restrictive than usual. Other type systems, such as System $F_{<:}$ [3], have different definitions of subtyping.

Yon supports *record width subtyping*: adding more fields to a record (more slots to an object) produces a subtype. But it does not have *record depth subtyping*. If a slot `x` exists in both S and T, it must have the exact same type in both. The type names must match as well (barring α-renaming). If the types of `S.x` and `T.x` have the same definition, but different names, that is enough for S and T not to be subtypes of each other.

This was done to avoid issues with recursive types. Consider the following type definitions:

```
interface LinkedListA
    slot data (Number)
    slot next (LinkedListA)

interface LinkedListB
    slot data (Number)
    slot next (LinkedListB)
```

In a system that compares the structure of the slots, not just the names, the two types

should clearly be considered equivalent. But a naive implementation of this equality check would loop infinitely:

```
pseudocode of function typesEqual(t1, t2):
    if some slot exists in t1 but not in t2:
        return no
    if some slot exists in t2 but not in t1:
        return no
    for each slot s of type t1:
        if not typesEqual(t1.s, t2.s):
            return no
    return yes
```

If there is a finite number of types, checking their structural equivalence can be reduced to determining the equivalence of two regular languages. [3] However, Yon has type constructors (see Section 2.3) and so the number of types can be infinite. (For instance, an `interface Foo<T>` might contain a slot with type `Foo<Array<T>>`.)

The Java programming language solves this problem differently: it requires all interfaces to specify the interfaces they implement. Thus, the programmer gives the compiler the complete subtyping relation, and the compiler only needs to check whether the relation is self-consistent.

For Yon, it was decided to support a limited version of structural subtyping. Thus, the programmer does not need to declare which types are subtypes of which. Whether one type is a subtype of another is determined by comparing their structure – the operations they support. But beyond the "first level", the comparisons are based on type names instead of their structure.

2.7 Bottom and Nil

The special built-in type `Bottom` is defined to be a subtype of all other types. Values of type `Bottom` can be assigned to any variable and used as any other type.

The built-in function `error` has a return value of type `Bottom`. This is safe, because the function never really returns. It stops the whole program with an error. Thus, the type checker can safely allow using the return value of `error` in any context. `return` expressions also have the `Bottom` type, for the same reason.

Yon has a null value called `nil`. The virtual machine uses it as the default value of newly defined variables and non-existent slots. Because any variable might contain `nil`, the type

checker does not check for `nil`-related errors and leaves them to be found at runtime. The `nil` constant also has the type `Bottom`, so assigning `v := nil` is valid regardless of the type of `v`.

Implementation-wise, `nil` is an instance of a class named `Nil`. It has the same slots as its parent, `Object`. See Chapter 3 for a general description of classes.

Chapter 3

Classes and methods

Classes provide infrastructure for creating objects. Though an object can be created without using a class, the class-based approach is usually more convenient – especially when many similar objects are to be created.

This chapter will show how to use Yon classes, and how they are implemented using the primitives introduced in previous chapters – mainly Section 1.4 and Chapter 2.

3.1 Normal slots

We start with the following class definition:

```
class Person
    slot name (String) := "Anonymous"
    slot location (String)
    slot age (Number)
```

The programmer would like a simple way to create `Person` objects and their default `name` should be the string `"Anonymous"`.

```
var p (Person) := Person()
p.name              # evaluates to "Anonymous"
p.location          # evaluates to nil
p.age := 7
p.age               # evaluates to 7
```

How does the compiler implement the class definition? How could it be done by hand?

First, we create the `interface Person`, needed so that the programmer can declare a variable `p` like in the above example. The definition is similar to the original class definition, but of course, the `name` slot no longer has a value.

```
interface Person
    slot name (String)
    slot location (String)
    slot age (Number)
```

Next, we need a object that will represent the class. It will be stored in the global variable `Person` and calling it will create a new instance. The type of this object will be named `Person.class`.

That is, the type of the variable named `p` is `Person`, and the type of the variable named `Person` is `Person.class`. This is a bit confusing, but it would be more confusing if the type of `p` was something different, like `Person.instance`.

`Person.class` needs to be callable. The way to do this is to give it a slot named `call` containing the appropriate function. The function itself will only call `Person.alloc()`, which will actually create the new instance.

```
interface Person.class
    slot alloc (function()->Person)
    slot call (function()->Person)
    slot prototype (Person.prototype)
```

We will also need an object that stores the default values for the instance slots (in this case, the string `"Anonymous"`). Storing them in a separate object will make it easier to deal with inheritance. This second object will be called `Person.prototype` and its type will also be `Person.prototype`.

```
interface Person.prototype
    slot name (String)
```

The other slots do not have a default value, so they do not need to be in the prototype interface.

Now that we have all the interfaces, we can create the actual objects. We will use the built-in function `createempty`. Calling `createempty<T>()` will create a new empty object of type `T`. Its slots will all be `nil`. Or rather, it will have no slots, and the virtual machine will default to returning `nil` when a missing slot is looked up.

```
var Person (Person.class)
Person := createempty<Person.class>()
Person.prototype := createempty<Person.prototype>()
Person.prototype.name := "Anonymous"
```

All that is left is to define `Person.alloc` and `Person.call`.

```
Person.alloc := function ()->Person
    var s (Person) := createempty<Person>()
    s.name := Person.prototype.name
    s
Person.call := function ()->Person
    s.alloc()
```

We can use `createempty` to create a `Person`, just like we used it to create `Person.class` and `Person.prototype` before.

This can be further improved by using `:=&` proxy assignment to initialize `s.name`. Then, a new `Person` instance will track changes in `Person.prototype` until it gets a `name` of its own.

```
var p1 (Person) := Person()
var p2 (Person) := Person()
p1.name    # "Anonymous"
p2.name    # "Anonymous"
Person.prototype.name := "suomynonA"
p1.name    # "suomynonA"
p2.name    # "suomynonA"
p1.name := "Joe"
p1.name    # "Joe"
p2.name    # "suomynonA"
```

3.2 Method slots

Method slots are a way to give a class dynamic slots that have different values depending on which instance they were read from.

```
class Person
    slot name (String) := "Anonymous"
    slot location (String)
    slot age (Number)

    method slot twiceMyAge (function(Person)->Number) :=
    function (self:Person)->Number
        self.age * 2
```

```
var p (Person) := Person()
p.age := 87
p.twiceMyAge    # evaluates to 174
```

This time, `p.twiceMyAge` and `Person.prototype.twiceMyAge` will be of different types: the first is a `Number`, and the second is a `function(Person)->Number` that will be called with `p` when `p.twiceMyAge` is accessed.

This can be implemented with a small change to the existing system:

```
interface Person
    # ...old slots...
    slot twiceMyAge (Number)

interface Person.prototype
    # ...old slots...
    slot twiceMyAge (function(Person)->Number)

Person.alloc := function ()->Person
    var s (Person) := createempty<Person>()
    s.name :=& Person.prototype.name
    s.twiceMyAge :=& Person.prototype.twiceMyAge(s)
    s
```

Because of the proxy assignment, every time `s.twiceMyAge` is read, the virtual machine will call `Person.prototype.twiceMyAge(s)`, which will return `s.age * 2`.

Of course, `Person.prototype.twiceMyAge` can be overridden with a different value of type `function(Person)->Number`, just like `Person.prototype.name` was overridden at the end of Section 3.1.

In other languages, `twiceMyAge` would probably be called a property. The reason this mechanism is called "method slots" is because it also allows defining methods:

```
class Person
    # ...old slots...
    method slot greet (function(Person)->function(String)->Nil) :=
    function (self:Person)->function(String)->Nil
        function (other:String)->Nil
            print "Hello " + other + ", I am " + self.name + "."
```

```
var p (Person) := Person()
p.greet("Jane")
```

`Person.prototype.twiceMyAge` is a function that takes a `self` argument and returns the "visible value" of `self.twiceMyAge`. Similarly, `Person.prototype.greet` is a function that takes a `self` argument and returns the "visible value" of `self.greet`. But in the second case, it is another function. The function is then called with a string argument.

To reiterate: `Person.prototype.greet` is a function that returns a function. The first function (which receives a `Person` argument) is called internally in the proxy assignment in `Person.alloc`. It always has exactly one argument, and its type is the class being defined. The second function is called by the user. In this case, it received one `String` argument, but different methods can have different arguments.

Because the first function is common to almost all method slots, there is an easier way to create them. When the programmer uses just `method` instead of `method slot`, the wrapper with a `self` argument is created for them. The following example is equivalent to the previous ones:

```
class Person
# ...old slots...
method twiceMyAge (Number) := self.age * 2
method greet (function(String)->Nil) :=
function (other:String)->Nil
    print "Hello " + other + ", I am " + self.name + "."
```

3.3 Class slots

The class object we created above only contains `call`, `alloc` and `prototype`. It is often useful to give it more slots. This can be used for factory methods, and various functionality that should be grouped with the class, but does not need an instance to operate on (as opposed to methods).

```
class Person
    # ...old slots...
    class slot loadFromFile (function(String)->Person) :=
    function (filename:String)->Person
        var contents (String) := String.readFromFile(filename)
        var lines (Array<String>) := contents.lines
        var p (Person) := Person()
```

```
        p.name := lines[0]
        p.location := lines[1]
        p.age := lines[2].number
        p
```

The implementation of class slots is straightforward. The type of the class slot is simply added as-is to `interface Person.class` and the value is assigned to the class along with the other functions (`Person.alloc`, etc.).

```
interface Person.class
    slot alloc (function()->Person)
    slot call (function()->Person)
    slot prototype (Person.prototype)
    slot loadFromFile (function(String)->Person)

Person.loadFromFile := function (filename:String)->Person
    # ...the function body...
```

3.4 Constructors

Some classes may wish to customize the process of creating a new instance. While creating the instance itself and doing the requisite proxy assignments is done by `Person.alloc`, many classes need more control over the process.

We allow classes to define a *constructor*, a function that is called with every newly created instance of the class. This function must be stored in a class slot named `init`. Its first argument will always be an instance of the class. If it has other arguments, they can be given when creating the class.

If a class does not define `init`, it cannot be instantiated.

Here is how `init` might look like for our `Person` class:

```
class Person
    # ...old slots...
    class slot init (function(Person,String,String,Number)->Top) :=
    function (self:Person, name:String, location:String,
    age:Number)->Top
        self.name := name
        self.location := location
        self.age := age
```

The implementation of classes shown in Section 3.1 used two functions for creating new objects: `call` and `alloc`. We will add a third: `create`. It will allocate the new object by calling `alloc` and then pass it to `init`.

```
interface Person.class
    slot init (function(Person,String,String,Number)->Top)
    slot alloc (function()->Person)
    slot create (function(String,String,Number)->Person)
    slot call (function(String,String,Number)->Person)
    slot prototype (Person.prototype)

# ... create the class and prototype ...

Person.create :=
function (name:String, location:String, age:Number)->Person
    var s (Person) := Person.alloc()
    Person.init(s, name, location, age)
    s

Person.call :=
function (name:String, location:String, age:Number)->Person
    Person.create(name, location, age)

var p (Person) := Person("Richard", "Earth", 64)
# the above line calls Person.alloc()
# and Person.init(p, "Richard", "Earth", 64)
```

`init` is a class slot intead of a method to allow subclasses to define another `init` with different arguments.

3.5 Inheritance

Classes can inherit from each other:

```
class Employee(Person)
    slot company (String)

    method greet (function(String)->Nil) :=
```

```
    function (other:String)->Nil
        Person.prototype.greet(self)(other)   # super method
        print "And I am from " + self.company + "."

    class slot init (function(Employee,String,Number)->Top) :=
    function (self:Employee, name:String, age:Number)->Top
        Person.init(self, name, "ACME HQ", age)
        self.company := "ACME"
```

When a class inherits from another, the code generated from the above class definition changes as follows:

```
interface Employee(Person)
    slot greet (function(String)->Nil)

interface Employee.class
    slot init (function(Employee,String,Number)->Top)
    slot alloc (function()->Employee)
    slot create (function(String,Number)->Employee)
    slot call (function(String,Number)->Employee)
    slot prototype (Employee.prototype)

interface Employee.prototype
    slot name (String)
    slot twiceMyAge (function(Employee)->Number)
    slot greet (function(Employee)->function(String)->Nil)

var Employee (Employee.class)
Employee := createempty<Employee.class>()
Employee.prototype := createempty<Employee.prototype>()

Employee.prototype.name :=& Person.prototype.name

Employee.prototype.twiceMyAge :=
function (self:Employee)->Number
    Person.prototype.twiceMyAge(self)

Employee.prototype.greet :=
```

```
function (self:Employee)->function(String)->Nil
    function (other:String)->Nil
        Person.prototype.greet(self)(other)   # super method
        print "And I am from " + self.company + "."

Employee.alloc := function ()->Employee
    var s (Employee) := createempty<Employee>()
    s.name :=& Employee.prototype.name
    s.twiceMyAge :=& Employee.prototype.twiceMyAge(s)
    s.greet :=& Employee.prototype.greet(s)
    s

# Employee.init, Employee.create, Employee.call
# are created as before
```

`interface Employee` inherits from `interface Person`, but `Employee.prototype` is not a subtype of `Person.prototype`. This is because the method slots inside them are not compatible. For example, the `twiceMyAge` slot of `Person.prototype` is a value of type `function(Person)->Number`, while in `Employee.prototype`, it has a different type: `function(Employee)->Number`. The first argument is always the instance of the class that is being defined.

Because of this, `Employee.prototype.twiceMyAge` and other inherited method slots must be wrapped in another function whose `self` self argument has the right type.

Inherited normal slots (such as `name`) use proxy assignment, so that if the value stored in `Person.prototype.name` changes, it will spread to `Employee.prototype.name` and all the `Employee` instances.

Class slots are not inherited at all.

The root of the class hierarchy is named `Object`. It provides several basic slots, such as "=", the equality operator. All other classes inherit from another class. If a class does not specify its parent, it defaults to `Object`.

3.6 Generic classes

A normal class such as `Person` creates instances of `interface Person`. Generic classes use type constructors (introduced in Section 2.3) instead. For example, a `class LinkedList<T is Person>` creates instances of `interface LinkedList<T>`.

```
class LinkedList<T is Person>
```

```
    slot data (T)
    slot next (LinkedList<T>)
    slot rank (Number) := -1
    method size (Number)
    class slot favoriteNumber (Number) := 7
    class slot init (function<T is Person>(LinkedList<T>)->Top)
```

The preceding class definition expands into the following interfaces:

```
interface LinkedList<T>
    slot data (T)
    slot next (LinkedList<T>)
    slot rank (Number)
    slot size (Number)

interface LinkedList.class
    slot init (function<T is Person>(LinkedList<T>)->Top)
    slot alloc (function<T is Person>()->LinkedList<T>)
    slot create (function<T is Person>()->LinkedList<T>)
    slot call (function<T is Person>()->LinkedList<T>)
    slot prototype (LinkedList.prototype)
    slot favoriteNumber (Number)

interface LinkedList.prototype
    slot name (String)
    slot size (function<T is Person>(LinkedList<T>)->Number)
```

Though `LinkedList.class` and `LinkedList.prototype` are still regular interfaces, not type constructors, many of their slots are now polymorphic functions. Namely, the auto-generated class slots (`alloc`, `create` and `call`) and the method slots in the prototype now take a type parameter `<T is Person>`.

Keeping the above interface definitions in mind, modifying the process from the previous sections to also work for generic classes is mostly straightforward. The template stays the same, except for the additional type instantiations.

It should be noted that a generic class can inherit from another generic class, and they don't have to have the same type arguments.

Chapter 4

Implementation

This chapter provides an overview of the implementation of the type checker.

Michal Antonič's Yon codebase [2] included a Yon compiler written in Yon. This compiler was modified to compile the new version of Yon, and extended with the type checker. However, the compiler itself is still written in the old Yon.

4.1 Compiler phases

The compilation of a Yon source file has the following phases:

- `scanner.yon`: The program is tokenized. This phase is mostly unchanged from the original Yon.
- `parser.yon`: The token stream is parsed into an abstract syntax tree. The parser now also does local type inference for nodes whose types can be computed without any additional information, such as number, string and function literals. Types of variables and slots initialized with such literals do not need to be specified.
- `compiler.yon`: The tree is scanned for import statements that request loading other files. They are compiled recursively.
- `analyzer.yon`: The interfaces of the file are analyzed and interface inheritance is resolved. The compiler now has the complete definition of all available interfaces.
- `transformer.yon`: Various transformations are performed on the abstract syntax tree. Most importantly, class definitions are translated into equivalent code based on `:=&` and `createempty`, as outlined in Chapter 3.
- `checker.yon`: The type checker processes the tree. It derives the resulting type of each node, and verifies that they are consistent.

- The top-level variable declarations of the file are collected so that they can be imported into other modules.

The last phase is handled by `generator.yon`, which has also seen little change. After all files are compiled, the generator processes the abstract syntax trees into bytecode and returns it as the compiler's output.

4.2 Data structures

The data structures used by the compiler and type checker provide a formalization of the concepts and descriptions from Chapter 2.

Though the compiler is written in the old Yon, this summary uses the same interface syntax used in the rest of this work.

```
interface TypeName
    slot name (String)
    slot polynames (Array<String>)
    slot polybounds (Array<TypeName>)
    slot args (Array<TypeName>)
    slot rettype (TypeName)
```

A `TypeName` object represents a type name.

For simple type names such as `Bool`, only the `name` is used, and the other slots are empty (i.e. either empty arrays or `nil`).

For types built with type constructors, such as `Pair<Bool,String>`, the `name` is the identifier specifying the type constructor (such as `Pair`), and `args` are the type names that are its arguments.

For callables and functions, the `name` is either `callable` or `function`, the `polynames` are the names of the type arguments (`T1,...,Tm` in Section 2.4), the `polybounds` are their bounds (`B1,...,Bm`), the `args` are the function arguments (`A1,...,An`) and `rettype` is the type of the return value.

```
interface TypeDefinition
    slot slots (Dictionary<String,TypeName>)
    slot primitives (Dictionary<String,Bool>)
    slot polycall (TypeName)
```

A `TypeDefinition` is a type's list of operations, as described in Section 1.1 and Section 2.2. `slots` contains the slot access operations this type supports. If `slots[s] = T`, the

type has a slot named `s` of type `T`. `primitives` are the primitive object values supported by this type. And if an object is callable (or instantiable), `polycall` contains a `TypeName` that specifies its `args`, `rettype`, etc.

```
interface DefinitionSet
    slot defs (Dictionary<String,TypeDefinition>)
    slot args (Dictionary<String,Array<String>>)
    slot get (function(TypeName)->TypeDefinition)
```

The `DefinitionSet` holds information about all the interfaces and type constructors in the program. The `args` dictionary contains the argument names of type constructors, while the `defs` dictionary contains the `TypeDefinition`s themselves. `function` and `callable` types are not in `defs`, but are generated by the `get` method as needed.

Conclusion

We have augmented the semantics of Yon to be more amenable to type checking and detailed them in the first chapter. We have then described a type system for the new version of Yon, and demonstrated its usability by building a class system on top of it to provide facilities for object-oriented programming. We have also created a working implementation of the type checker and summarized its high-level design.

The type checker met its goals outlined in Section 2.1. It can compile program modules one by one, and it is flexible enough to be able to express entities such as type constructors and polymorphic functions.

The class system built on top of the type system supports generic classes and inheritance, and allows class methods to be switched to another implementation even after instances of the class have been constructed.

Future work

Though this thesis has met its goals, avenues for future work remain. Some of them are:

- Explore the relation between Yon's type system and well-studied formal systems such as System F and System $F_{<:}$.
- Add support for more types, such as variant types and existential types.
- Study type inference algorithms and evaluate their potential to be included in a language like Yon.

Bibliography

[1] Ole Agesen. The cartesian product algorithm - simple and precise type inference of parametric polymorphism, 1995.

[2] Michal Antonič. Yon - design and implementation of a new programming language, 2011.

[3] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[4] Gerald Jay Sussman and Guy L Steele Jr. Scheme: An interpreter for extended lambda calculus. In *MEMO 349, MIT AI LAB*, 1975.