Ana Jorge Carvalho de Soares Almeida

Dynamic Typing

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2020

Ana Jorge Carvalho de Soares Almeida

Dynamic Typing

Relatório de Projecto

Orientador: António Mário da Silva Marcos Florido

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2020

Abstract

A programming language is said to be dynamically typed if the type of every expression is checked at runtime. It is said to be statically typed if types are checked at compile-time. Dynamic typing enables the use of programming languages for quick prototyping. Examples include Python and JavaScript.

In order to achieve safety in a dynamically typed language, a careful use of type casts, which allow conversion from a type to another type, is mandatory.

In this project, we implemented an interpreter for a language based on the lambda calculus using explicit type casts. We also implemented a parser for the initial language, which produces the corresponding abstract syntax tree.

Contents

Chapter 1

Introduction

Programming languages are divided in two main groups. Statically typed languages, which mostly typecheck at compile-time, although some also typecheck at runtime in order to ensure safety, and dynamically typed languages, when typechecking occurs at runtime. But, what does typechecking at compile-time and runtime mean? Each language assumes the presence of a type system, and the strategy it uses to analyse that system determines whether it is statically or dynamically typechecked.

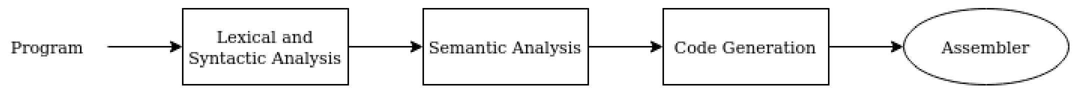

Figure 1.1: Main phases of a compiler.

Let us look at figure 1.1, the lexical and syntactic analysis receives the source code and delivers an abstract syntax tree to the semantic analysis, where static typechecking is done. Then, after arriving to the code generation stage, it produces an assembly language, which is given to the assembler. In the assembler, our assembly language will be turned into machine code. Thus, static typechecking at compile-time means it will happen in the semantic analysis stage, where our code's syntax is checked. Typechecking at runtime, in a compiled language, is done in the assembler, in a much more advanced stage. However, dynamic typing is mostly done in interpreted languages, where the interpreter itself will typecheck (which is what happens in our interpreter). It would be wrong to assume one is better than the other since they have complementary strengths, which will be further studied.

In order to tie both of these typing disciplines into a single language, we use gradual typing [9], which considers the existence of type casts. Type casts are used to convert a type into another type, mainly considering type consistency. A type is consistent with another type if one is a subset of the other, with `Dyn` being consistent with all types.

The objective of this project was to implement a simple language that uses explicit type casts and an interpreter that dynamically typechecks. We have implemented a parser for a functional language in `Haskell` and an interpreter of that same language. The full implementation is available at:
`https://github.com/anathegrey/dynamic-type-checking`

This report is organized in four chapters. **State of the Art**, chapter 2, where we introduce all relevant concepts studied before we started implementing, **Implementation**, chapter 3, where our project is described and where we present the results, and, finally, **Conclusion**, chapter 4, where we summarize what was done and talk about our future goals.

Chapter 2

State of the Art

2.1 Typed Lambda Calculus

The Lambda Calculus can be viewed as both a simple programming language in which computations can be described and as a mathematical object about which rigorous statements can be proved. It is described as a formal system in which all computation is reduced to the basic operations of function definition and application. It has seen widespread use in the specification of programming language features, language design and implementation, and the study of type systems [7].

Procedural abstraction is a key feature of most programming languages [7], as we write procedures or functions that perform calculations abstractly instead of writing the same calculation over and over again. For instance, we can represent the `factorial` function as follows.

Example 2.1.1

$\texttt{factorial}(\texttt{n}) \;=\; \texttt{if}\ \texttt{n} = 0\ \texttt{then}\ 1\ \texttt{else}\ \texttt{n} * \texttt{factorial}(\texttt{n} - 1)$

Example 2.1.2

$\texttt{factorial} \;=\; \lambda\texttt{n}.\ \texttt{if}\ \texttt{n} = 0\ \texttt{then}\ 1\ \texttt{else}\ \texttt{n} * \texttt{factorial}(\texttt{n} - 1)$

There appears to be a syntactic difference between both examples. The mathematical system presented in example 2.1.2 was created by Alonzo Church, and it states that everything is a function, the arguments accepted by functions are themselves functions and the result returned by a function is another function. It defines and applies a function in a pure form. This example uses a form of untyped lambda calculus, which we do not intend to lengthen, since we wish to study its typed form.

Definition 2.1.1
The set of simple types over the atomic type `Bool` *is generated by the following grammar:*

`T` ::=	*(types...)*
`T` $\rightarrow$ `T`	*type of functions*
`Bool`	*type of booleans*

Definition 2.1.2
The abstract syntax of simply typed lambda-terms (with booleans and conditional) is defined by the following grammar:

`t` ::=	*(terms...)*
`x`	*variable*
λ`x` : `T.t`	*abstraction*
`t t`	*application*
`true`	*constant true*
`false`	*constant false*
`if t then t else t`	*conditional*

There are two different presentation styles that are commonly used when representing type systems. The implicit typed systems, *Curry-style*, where the pure (untyped) lambda-calculus is used as the term language, the typing rules define a relation between untyped terms and the types that classify them. And the explicitly typed systems, *Church-style*, where the term language itself is refined so that terms carry some type information within them (the bound variables in function abstractions are always annotated with the type of the expected parameter, for example), the type system relates typed terms and their types [7].

To a large degree, the choice is a matter of taste, though explicitly typed systems generally pose fewer algorithmic problems for typecheckers [7]. Throughout this report we will adopt an explicitly typed presentation.

2.2 Dynamic and Static Types

A program variable can assume a range of values during the execution of a program. In typed languages, there is a component, a type system, that keeps track of the types of variables, and an untyped language is nothing more than a special case of typed code where every term has a dynamic type. The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program [3], that are generally formulated as collections of rules for checking the absence of runtime errors [7]. We can divide all programming languages as being statically typed, when the typechecking is done at compile-time, or dynamically typed when it is done at runtime.

	Statically checked	Dynamically checked
Strongly typed	ML, Haskell	Lisp, Scheme
Weakly typed	C, C++	Perl

Table 2.1: Examples of programming languages.

Saying a language is typechecked at compile-time means that it checks when our program is converted to an assembly language program. This guarantees the absence of *untrapped errors*. As previously mentioned, since a typed language contains a type system, there will only exist *trapped errors*, that happen when a variable or expression does not fit in a type system and immediately stops our execution.

The main difference between these two errors is that a trapped error is tracked down and an untrapped error is not. A trapped error is relatively easy to correct since we are given all its information. Untrapped errors are common when a language is typechecked during runtime, a property of dynamically typed languages. It is often hard to locate an untrapped error since we do not have the necessary information to track it.

But, for instance, there are statically typed languages that also type check dynamically, statically typed languages without type annotations or even dynamically typed languages where untrapped errors do not occur. For example, C is considered to be statically typed but has a well-known error that we often get, the segmentation fault, which is an untrapped error. This means that C must also type check dynamically, otherwise this error could not exist, and it tells us that C is a weakly-typed or ill-behaved language.

Despite being statically and dynamically typed, we also catalog languages regarding its safety. A safe language is said to be strongly-typed, in which the typechecker proves the absence of runtime type errors rather than its existence, and an unsafe language is weakly-typed, in which the typechecker proves the existence of runtime errors rather than its absence [7]. This appears to be slightly complex but it really shows that languages may not be strictly statically or dynamically checked. But why do some languages need to combine both?

Even in a statically typed language, there is often the need to deal with data whose type cannot be determined at compile-time [1]. Not all languages have the same weaknesses and some may be weaker than others, for instance, C has many unsafe features, such as pointer arithmetic and casting, and Pascal is unsafe only when using untagged variant types and function parameters. Although these languages are also dynamically checked, they are still considered statically typed because the dynamic type tests are defined on the basis of the static type system, the dynamic tests for type equality are compatible with the algorithm that the typechecker uses to determine type equality at compile-time [3].

Dynamic typechecking may be used to ensure that some operations will be executed quicker than if executed statically, but a certain amount of dynamic checking must be performed in order to preserve type safety [1]. Contrarily to what may be obvious now, most dynamically typed languages are safe.

Unsafe languages often provide "best effort" static type checkers that help programmers remove the most obvious sort of slips, but such languages are generally not capable of offering any sort of guarantees that well-typed programs are strongly-typed (or well-behaved) [7]. C is deliberately unsafe because of performance considerations, the runtime checks needed to achieve safety are sometimes considered too expensive. The choice between a safe and unsafe language may be ultimately related to a trade-off between development and maintenance time, and execution time [3].

It cannot be said that the difference between dynamically and statically typed languages is the presence or absence of type annotations, since there are statically typed languages without type annotations, known as implicitly typed. Languages such as ML and Haskell support writing large program fragments where type information is omitted [3], but no mainstream language is purely implicitly typed. The accurate difference for statically and dynamically typed languages is the presence or absence of a type system, because the presence of a type system is a property of statically typed languages. Therefore, we can divide statically typed languages in explicitly typed, where all types are annotated, and implicitly typed, where not all types are annotated, considering there cannot be one purely implicitly typed language.

It is not certain if statically typed languages are better than dynamically typed languages and vice-versa because they have complementary strengths. Static typing guarantees the absence of type errors, facilitates the generation of efficient code, and provides machine-checked documentation, and dynamic typing enables rapid prototyping, flexible programming idioms, and fast adaptation to changing requirements [9].

2.3 Dynamic Typechecking

As we have already seen, there can be dynamic typechecking either in dynamically typed languages and in some statically typed languages. In systems that combine both static and dynamic typechecking, we add a new type, the dynamic type, that we will represent as `Dyn`, meaning the typechecking will be done both at compile-time and runtime. A dynamic type can be seen as a box that contains a value of any type and the representation of said type [2].

2.3.1 Type Casts

Type casts are used in programming languages to convert a type into another type, mainly used in dynamic typing, with the programmer controlling the degree of static checking by annotating function parameters with our usual types or `Dyn`.

Gradual typing is the term used for type systems that provide the capability of mixing dynamic and static typing[8]. Cecil, Boo, C# or the Bigloo dialect of Scheme are a few examples of languages that support gradual typing.

A gradually typed language can be thought of as a superset of two other languages, a fully static language and a fully dynamic. A fully annotated gradually typed language should behave the same as a statically typed language, and a program without type annotations should behave the same as a dynamically typed language. We say that a program is fully annotated if all variables have type annotations and if the type `Dyn` does not occur [9].

Since a fully static program using gradual typing should behave the same as if it was written in a statically typed language, it guarantees the absence of type errors at runtime, that is, untrapped errors. But, what happens when our program is partially typed [9]?

Consider the following algorithm for computing the modular inverse.

```
def modinv(a, m)
  if g != 1: raise Exception()
  else: return x \% m
```

Since the parameters of `modinv` are not annotated, it is dynamically typed, but suppose it calls the statically typed GCD_{3b}, what happens if someone forgets a conversion and passes a string `m` of `modinv` [9]?

The string cannot flow into the gcd function and trigger a runtime error in the expression b % a, because gcd is statically typed and guaranteed to be free of runtime errors, of both the trapped and untrapped variety [9]. In this example, there is a cast error in `modinv` just before the call to gcd, since with gradual typing, the runtime system protects the static typing assumptions by casting values as they flow between statically and dynamically typed code and, in fact, it ensures that statically typed regions of code are free of runtime type errors [9].

GCD_{1a}

```
def gcd(a, b):
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x   (b // a) * y,
   y)
```

GCD_{1b}

```
def gcd(a, b):
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x   (b // a) * y)
```

GCD_{3a}

```
def gcd(a:int, b:int)  > Tuple[
    int,int,int]:
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x   (b // a) * y,
   y)
```

GCD_{3b}

```
def gcd(a:int, b:int)  > Tuple[
    int,int,int]:
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x   (b // a) * y)
```

Figure 2.1: Variants of the extended greatest-common divisor algorithm.

Let us now consider the following fully annotated version of `modinv` with a call to the dynamically typed GCD_{1a}. This function refers to a variable (gcd) that is dynamic, so it makes this program only partially typed.

```
def modinv(a:int, m:int):
    (g, x, y) = gcd(a, m)
    if g != 1: raise Exception()
    else: return x % m
```

We analyse the implicit casts in order to understand which parts of the program are safe and which might result in a runtime type error. It is always guaranteed that upcasts are safe and, if there is a type error, it happens in a downcast. When there is a cast from a subset to a larger set, we are talking about an upcast and, in this example, in the call `gcd(a,m)` there is an upcast, because the arguments are cast from Int to Dyn. But, when there is a cast from a set to its subsets, a downcast, it is an unsafe operation since it is not guaranteed that the cast is safe, for example, the return type of GCD$_{1a}$ is unspecified, so it defaults to Dyn, and the assignment to the tuple $(\mathtt{g}, \mathtt{x}, \mathtt{y})$ requires a downcast. Usually, if there is a cast error it occurs in a downcast.

2.3.2 The Cast Calculus

We will now review the fully annotated gradually typed program with the unknown type `Dyn`.

`Base Types` $B ::= Int \mid Float \mid Bool$

`Ground Types` $G ::= B \mid Dyn \to Dyn$

`Expressions` $f ::= k \mid x \mid \lambda x : T.f \mid f f \mid f : T \Rightarrow^{l} T \mid blame_T\ l$

`Values` $v ::= k \mid \lambda x : T.f \mid v : T_1 \to T_2 \Rightarrow^{l} T_3 \to T_4 \mid v : G \Rightarrow^{l} Dyn$

`Results` $r ::= v \mid blame_T l$

`Frames` $F ::= \square f \mid v\square \mid \square : T_1 \Rightarrow^{l} T_2$

Dynamic Semantics

$(\lambda x : T.f)v \ \to\ [x := v]f$	Beta
$v : B \Leftarrow^{l} B \ \to\ v$	IdBase
$v : Dyn \Leftarrow^{l} Dyn \ \to\ v$	IdStar
$v : G \Leftarrow^{l_2} Dyn \Leftarrow^{l_1} G \ \to\ v$	Succeed
$v : G_2 \Leftarrow^{l_2} Dyn \Leftarrow^{l_2} G_1 \ \to\ blame_{G_2} l_2 \quad if\ G_1 \neq G_2$	Fail
$\big(v_1 : T_3 \to T_4 \Leftarrow^{l} T_1 \to T_2\big)\, v_2 \ \to\ v_1 \big(v_2 : T_1 \Leftarrow^{l} T_3\big) : T_4 \Leftarrow^{l} T_2$	AppCast
$v : Dyn \Leftarrow^{l} T \ \to\ v : Dyn \Leftarrow^{l} G \Leftarrow^{l} T \quad if\ T \neq Dyn, T \neq G, T \sim G$	Ground
$v : T \Leftarrow^{l} Dyn \ \to\ v : T \Leftarrow^{l} G \Leftarrow^{l} Dyn \quad if\ T \neq Dyn, T \neq G, T \sim G$	Expand
$F[f] \ \to\ F[f'] \quad if\ f \to f'$	Cong
$F[blame_{T_1} l] \ \to\ blame_{T_2} l \quad if \vdash F : T_2 \Leftarrow T_1$	Blame

---

Figure 2.2: Cast insertion and the internal cast calculus.

We use the blame tracking technique of Findler and Felleisen [4] in order to determine whether a cast is safe. Upcasts never result in blame, only downcasts or cross-casts [9]. The blame label l represents source position information from the parser, so blame labels are unique. The typing rules for constants, variables and functions are the same as in fully annotated gradually typed languages. Thus, we have the consistency relation, $\mathtt{T}_1 \sim \mathtt{T}_2$, that is used where the fully annotated programs would check for type equality. The consistency relation is more liberal when it comes to the unknown type: it relates any type to the unknown type, for example, $\mathtt{int} \sim \mathtt{Dyn}$ and $\mathtt{bool} \nsim \mathtt{int}$. Consistency is symmetric but not transitive [9].

The dynamic semantics of the gradually typed languages is defined by translating into an internal cast much like the Blame Calculus [10]. The internal cast calculus extends the fully annotated gradually typed languages with the unknown type `Dyn` but it replaces the implicit casts with explicit casts [9]. The dynamic semantics of the cast calculus is given in figure 2.2. Our language of expression consists on an annotated version of the Lambda Calculus [5, 7].

The following examples will be used later in chapter 3 in order for us to evaluate the performance of our program. We can identify the rules `Appcast` and `Expand`.

Example 2.3.1

$\big(\lambda \mathtt{y} : \mathtt{Int}.\,(\mathtt{if}\ (\mathtt{y} > 0)\ \mathtt{then}\ (\mathtt{y} * 3 - \mathtt{y} - 1)\ \mathtt{else\ none}) : \mathtt{float} \to \mathtt{float} \Leftarrow^{\mathtt{l}} \mathtt{Dyn} \to \mathtt{Dyn}\big)\ 3.01$
$\to ((\lambda \mathtt{y} : \mathtt{Int}.\,(\mathtt{if}\ (\mathtt{y} > 0)\ \mathtt{then}\ (\mathtt{y} * 3 - \mathtt{y} - 1)\ \mathtt{else\ none}))$
$\big(3.01 : \mathtt{Dyn} \Leftarrow^{\mathtt{l}} \mathtt{float})\big) : \mathtt{float} \Leftarrow^{\mathtt{l}} \mathtt{Dyn}$
$\to^* 5.02 : \mathtt{float} \Leftarrow^{\mathtt{l}} \mathtt{Dyn} \Leftarrow^{\mathtt{l}} \mathtt{float}\ \to\ 5.02$

Example 2.3.2

$\big(\lambda \mathtt{y} : \mathtt{Int}.\,(\mathtt{if}\ (\mathtt{y} > 0)\ \mathtt{then}\ (\mathtt{y} * 3 - \mathtt{y} - 1)\ \mathtt{else\ none}) : \mathtt{float} \to \mathtt{float} \Leftarrow^{\mathtt{l}} \mathtt{Dyn} \to \mathtt{Dyn}\big) - 2.99$
$\to ((\lambda \mathtt{y} : \mathtt{Int}.\,(\mathtt{if}\ (\mathtt{y} > 0)\ \mathtt{then}\ (\mathtt{y} * 3 - \mathtt{y} - 1)\mathtt{else\ none}))$
$\big(-2.99 : \mathtt{Dyn} \Leftarrow^{\mathtt{l}} \mathtt{float})\big) : \mathtt{float} \Leftarrow^{\mathtt{l}} \mathtt{Dyn}$
$\to^*\ \mathtt{None} : \mathtt{float} \Leftarrow^{\mathtt{l}} \mathtt{Dyn} \Leftarrow^{\mathtt{l}} \mathtt{NoneType} \to \mathtt{blame}_{\mathtt{float}}\ \mathtt{l}$

Chapter 3

Implementation

The goal of this project was to create an interpreter that would reproduce a language based on the Lambda Calculus using explicit type casts.

Our project was implemented in Haskell.

3.1 Overall Architecture

Figure 3.1 offers us a simple diagram that will help us perceive the stages of our program and how information is shaped.

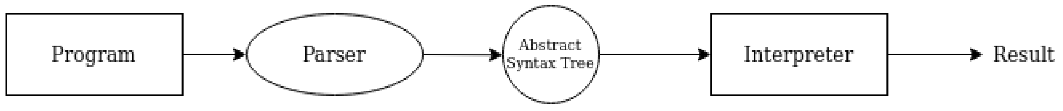

Figure 3.1: Overview.

Our parser receives a string, which is converted to an abstract syntax tree, and then passed on to the interpreter. It will compute the input and give us a result. The parser's and interpreter's behaviour is explained in detail in section 3.2 and section 3.3, respectively.

3.2 Parser

Our parser converts a string into an abstract syntax tree, that is passed on to our interpreter according to the following data.

```haskell
module CastData where

        data Type = TInt
                  | TBool
                  | TFloat
                  | FuncT Type Type
                  | Dyn
                  | NoneType
                  deriving (Show, Eq)

        type Label = String

        data Expr = ConstI Int Type
                  | ConstB Bool Type
                  | ConstF Float Type
                  | Minus Expr
                  | VarE String
                  | Add Expr Expr
                  | Mul Expr Expr
                  | Sub Expr Expr
                  | Div Expr Expr
                  | Less Expr Expr
                  | Bigger Expr Expr
                  | LessEq Expr Expr
                  | BiggerEq Expr Expr
                  | Eq Expr Expr
                  | If Expr Expr Expr
                  | FuncE String Type Expr
                  | AppE Expr Expr
                  | ExprC Expr Type Type Label
                  | Blame Type Label
                  | None
```

The following code is the grammar defined in our parser (the full implementation can be found in appendix C).

```
Expr : Expr1 { $1 }
     | ExprArith { $1 }
     | ExprBool { $1 }

Expr1 : int { ConstI $1 TInt }
      | ' ' int { Minus (ConstI $2 TInt) }
      | '[' int ']' { ConstI $2 Dyn }
      | '[' ' ' int ']' { Minus (ConstI $3 Dyn) }
      | float { ConstF $1 TFloat }
      | ' ' float { Minus (ConstF $2 TFloat) }
      | '[' float ']' { ConstF $2 Dyn }
      | '[' ' ' float ']' { Minus (ConstF $3 Dyn) }
      | bool { ConstB $1 TBool }
      | '[' bool ']' { ConstB $2 Dyn }
      | var { VarE $1 }
      | '(' Expr ')' { $2 }
      | '(' Expr ')''(' Expr ')' { AppE $2 $5 }
      | if ExprBool then Expr else Expr { If $2 $4 $6 }
      |  '\\' var ':' Type '.' Expr { FuncE $2 $4 $6 }
      | '<' Type "<=" Type ',' var '>' Expr { ExprC $8 $4 $2 $6 }
      | none { None }

ExprArith : Expr '+' Expr1 { Add $1 $3 }
          | Expr ' ' Expr1 { Sub $1 $3 }
          | Expr '*' Expr1 { Mul $1 $3 }
          | Expr '/' Expr1 { Div $1 $3 }

ExprBool : Expr "<=" Expr1 { LessEq $1 $3 }
         | Expr ">=" Expr1 { BiggerEq $1 $3 }
         | Expr '<' Expr1 { Less $1 $3 }
         | Expr '>' Expr1 { Bigger $1 $3 }
         | Expr "==" Expr1 { Eq $1 $3 }

Type : "Int" { TInt }
     | "Float" { TFloat }
     | "Bool" { TBool }
     | "Dyn" { Dyn }
     | Type " >" Type1 { FuncT $1 $3  }

Type1 : "Int" { TInt }
      | "Float" { TFloat }
```

```
        | "Bool"  { TBool }
        | "Dyn"  { Dyn }
```

The parser will make our code more friendly for the programmer.

Example 3.2.1
`AppE (FuncE "x" TInt (VarE "x")) (ConstF 2.4 TFloat)`

Example 3.2.2
`(\\x : Int.x) (2.4)`

Instead of presenting our input as in example 3.2.1, we can write as shown in example 3.2.2 and the parser will convert the string into the previous example so it can be sent to the main program, the interpreter.

The parser is implemented using Happy, which is a parser generator for Haskell [6].

```
parse :: String  > Expr
parse s = calc(lexer s)
```

We defined a function called `parse`, the parser's main function, that receives an input and sends the input through the `lexer`. This function will cover all tokens present in the input, connect them to the proper rules and, if there is a rule that defines them, it sends them to the interpreter's data. If they are not defined by a rule we will get a parse error.

This allows us to work with a pleasant syntax that will transform our code into something the interpreter can recognise.

3.3 Interpreter

The interpreter is the main program of our project (the full implementation can be found in appendixes A and B), here we have implemented all the rules and defined how our language should behave.

```
        run :: String  > Expr
        run s = interp (Parser.parse s)
```

When we wish to pass an input through our program, we start by calling the `run` function, from our interpreter. This function will send our input to the parser so it can be transformed into a language our program can perceive and, when it is ready to be reduced, it enters our `interp`. It will then search all the rules in this function through pattern matching.

Our interpreter implements the rules in figure 2.2.

The rules `Ground` and `Expand` (in figure 2.2) are simplified because they are non-deterministic, and since we only work with `Int`, `Float`, `Bool` and `Dyn`, we cannot build a compatible type for each of these types since they are only compatible with `Dyn`.

3.4 Examples

Examples 3.4.1 and 3.4.2 correspond to the examples 2.2.1 and 2.2.2 from chapter 2.

Although the input's syntax is processed by the parser, the output's syntax is the raw abstract syntax tree of the result (we intend to implement a pretty printer as future work).

Example 3.4.1

$< \texttt{Float} \ \Leftarrow \ \texttt{Dyn}, \ \texttt{l} > (\backslash\backslash \texttt{y} : \texttt{Int}.(\texttt{if y} > \texttt{0 then} \ (\texttt{y} * 3 - \texttt{y} - 1) \ \texttt{else none}))$
$(< \texttt{Dyn} \ \Leftarrow \ \texttt{Float}, \ \texttt{m} > \ [3.01])$
$\rightarrow^* \ \texttt{ConstF 5.0199995 Dyn}$

Example 3.4.2

$< \texttt{Float} \ \Leftarrow \ \texttt{Dyn}, \ \texttt{l} > ((\texttt{y} : \texttt{Int}.(\texttt{if y} > \texttt{0 then} \ (\texttt{y} * 3 - \texttt{y} - 1) \ \texttt{else none}))$
$(< \texttt{Dyn} \ \Leftarrow \ \texttt{Float}, \ \texttt{m} > \ [-2.99])$
$\rightarrow^* \ \texttt{Blame TFloat} \ ''\texttt{l}''$

Example 3.4.3
$(< \texttt{Dyn} \ \rightarrow \ \texttt{Dyn} \ \Leftarrow \ \texttt{Int} \ \rightarrow \ \texttt{Int}, \ \texttt{l} > (\backslash\backslash \texttt{y} : \texttt{Int.y} + 1)) \ ([2])$
$\rightarrow^{*}$ `ConstI 3 Dyn`

Example 3.4.4
$(\backslash\backslash \texttt{x} : \texttt{Int.x}) \ (2)$
$\rightarrow^{*}$ `ConstI 2 TInt`

Examples 3.4.1 and 3.4.2 correspond to the examples 2.3.1 and 2.3.2 from chapter 2.

Chapter 4

Conclusion

In this project, we implemented an interpreter for a core functional language that typechecks at runtime. The typechecker was implemented for a core language, and the next step would be to apply these ideas to other constructs as `let` expressions and explicitly recursive functions.

As future work, one can define a compiling method that introduces type casts into the original programs as done in gradual typing [8].

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM transactions on programming languages and systems (TOPLAS)*, 13(2):237–268, 1991.

[2] Arthur I Baars and S Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166, 2002.

[3] Luca Cardelli. Type systems. *ACM Computing Surveys (CSUR)*, 28(1):263–264, 1996.

[4] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN; International Conference on Functional Programming (ICFP'02)*, pages 48–59, 2002.

[5] C. Hankin. *Lambda Calculi: A Guide for Computer Scientists.* Graduate texts in computer science. Clarendon Press, 1994.

[6] Simon Marlow and Andy Gill. Happy user guide. *Glasgow University, December*, 1997.

[7] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[8] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

[9] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in*

Programming Languages (SNAPL 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[10] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, pages 1–16. Springer, 2009.

Appendix A

CastData.hs

```haskell
module CastData where

        data Type = TInt
                  | TBool
                  | TFloat
                  | FuncT Type Type
                  | Dyn
                  | NoneType
                  deriving (Show, Eq)

        type Label = String

        data Expr = ConstI Int Type
                  | ConstB Bool Type
                  | ConstF Float Type
                  | Minus Expr
                  | VarE String
                  | Add Expr Expr
                  | Mul Expr Expr
                  | Sub Expr Expr
                  | Div Expr Expr
                  | Less Expr Expr
                  | Bigger Expr Expr
                  | LessEq Expr Expr
                  | BiggerEq Expr Expr
                  | Eq Expr Expr
```

```
                   | If Expr Expr Expr
                   | FuncE String Type Expr
                   | AppE Expr Expr
                   | ExprC Expr Type Type Label
                   | Blame Type Label
                   | None
                   deriving (Show, Eq)

    isGround :: Type  > Bool
    isGround TInt = True
    isGround TBool = True
    isGround TFloat = True
    isGround (FuncT Dyn Dyn) = True
    isGround _ = False

    isValue :: Expr  > Bool
    isValue (ConstI x TInt) = True
    isValue (Minus (ConstI x TInt)) = True
    isValue (ConstI x Dyn) = True
    isValue (Minus (ConstI x Dyn)) = True
    isValue (ConstB x TBool) = True
    isValue (ConstB x Dyn) = True
    isValue (ConstF x TFloat) = True
    isValue (Minus (ConstF x TFloat)) = True
    isValue (ConstF x Dyn) = True
    isValue (Minus (ConstF x Dyn)) = True
    isValue (VarE x) = True
    isValue (FuncE x t1 exp) = True
    isValue (ExprC v (FuncT t1 t2) (FuncT t3 t4) l) = True
    isValue (ExprC v t1 Dyn l) = if (isGround t1) then True else
False
    isValue _ = False

    takeInt :: Expr  > Int
    takeInt (ConstI n TInt) = n
    takeInt (Minus (ConstI n TInt)) = ( n)
    takeInt (ConstI n Dyn) = n
    takeInt (Minus (ConstI n Dyn)) = ( n)

    isInt :: Expr  > Bool
    isInt (ConstI n TInt) = True
    isInt (Minus (ConstI n TInt)) = True
    isInt _ = False

    fromInt :: Int  > Float
    fromInt n = fromInteger (toInteger n)

    takeFloat :: Expr  > Float
    takeFloat (ConstF n TFloat) = n
    takeFloat (Minus (ConstF n TFloat)) = ( n)
    takeFloat (ConstF n Dyn) = n
    takeFloat (Minus (ConstF n Dyn)) = ( n)

    isFloat :: Expr  > Bool
    isFloat (ConstF n TFloat) = True
    isFloat (Minus (ConstF n TFloat)) = True
    isFloat _ = False

    takeBool :: Expr  > Bool
    takeBool (ConstB n TBool) = n
    takeBool (ConstB n Dyn) = n

    isBool :: Expr  > Bool
    isBool (ConstB n TBool) = True
    isBool _ = False

    isDynInt :: Expr  > Bool
    isDynInt (ConstI n Dyn) = True
    isDynInt (Minus (ConstI n Dyn)) = True
    isDynInt _ = False

    isDynFloat :: Expr  > Bool
    isDynFloat (ConstF n Dyn) = True
    isDynFloat (Minus (ConstF n Dyn)) = True
    isDynFloat _ = False

    isDynBool :: Expr  > Bool
    isDynBool (ConstB n Dyn) = True
    isDynBool _ = False
```

Appendix B

CastCalculus.hs

```
module CastCalculus where
        import CastData
        import Parser

        succeed :: Expr  > Expr   implements rules Succeed and Fail from figure 2.2
        succeed (ExprC (ExprC v g1 Dyn l1) Dyn g2 l2)
                 | (isValue v) && (isGround g1) && (isGround g2) = if g1 == g2 then v else (Blame g2 l2)
                 | otherwise = interp (ExprC (ExprC v g1 Dyn l1) Dyn g2 l2)

        appcast :: Expr  > Expr   implements rule Appcast from figure 2.2
        appcast (AppE (ExprC v1 (FuncT t1 t2) (FuncT t3 t4) l) v2)
                 | (isValue v1) && (isValue v2) = ExprC (AppE v1 (ExprC v2 t3 t1 l)) t2 t4 l
                 | otherwise = interp (AppE (ExprC v1 (FuncT t1 t2) (FuncT t3 t4) l) v2)

        buildCompatible :: Type  > Type   auxiliary function for functions ground and expand
        buildCompatible (FuncT t1 t2) = FuncT t1 Dyn

        ground :: Expr  > Expr   implements rule Ground from figure 2.2
```

```
    ground (ExprC v t Dyn l)
            | (isValue v) && t /= Dyn && (isGround t) == False = let g = (buildCompatible t) in ExprC (ExprC v t g l) g Dyn l
            | otherwise = if (isValue v) then v else (ExprC v Dyn t l)

    expand :: Expr  > Expr   implements rule Expand from figure 2.2
    expand (ExprC v Dyn t l)
            | (isValue v) && t /= Dyn && (isGround t) == False = let g = (buildCompatible t) in ExprC (ExprC v Dyn g l) g t l
            | otherwise = if (isValue v) then v else (ExprC v Dyn t l)

    subst :: Expr  > String  > Expr  > Expr    implements rule Beta from figure 2.2
    subst (ConstI n TInt) x v = (ConstI n TInt)
    subst (ConstI n Dyn) x v = (ConstI n Dyn)
    subst (ConstB n TBool) x v = (ConstB n TBool)
    subst (ConstB n Dyn) x v = (ConstB n Dyn)
    subst (ConstF n TFloat) x v = (ConstF n TFloat)
    subst (ConstF n Dyn) x v = (ConstF n Dyn)
    subst (VarE y) x v = if x == y then v else (VarE y)
    subst (FuncE y t f) x v = if x == y then (FuncE y t f) else (FuncE y t (subst f x v))
    subst (Add e1 e2) x v = Add (subst e1 x v) (subst e2 x v)
    subst (Sub e1 e2) x v = Sub (subst e1 x v) (subst e2 x v)
    subst (Mul e1 e2) x v = Mul (subst e1 x v) (subst e2 x v)
    subst (Div e1 e2) x v = Div (subst e1 x v) (subst e2 x v)
    subst (Less e1 e2) x v = Less (subst e1 x v) (subst e2 x v)
    subst (LessEq e1 e2) x v = LessEq (subst e1 x v) (subst e2 x v)
    subst (Bigger e1 e2) x v = Bigger (subst e1 x v) (subst e2 x v)
    subst (BiggerEq e1 e2) x v = BiggerEq (subst e1 x v) (subst e2 x v)
    subst (If e1 e2 e3) x v = If (subst e1 x v) (subst e2 x v) (subst e3 x v)
    subst (ExprC expr t1 t2 l) x v = (ExprC (subst expr x v) t1 t2 l)
    subst (AppE e1 e2) x v = AppE (subst e1 x v) (subst e2 x v)
```

```
      subst (Blame t l) x v = (Blame t l)
      subst None x v = None


  interpreter
      interp :: Expr > Expr
      interp (AppE (FuncE x t f) v) = if (isValue v) then interp (subst f x v) else interp (AppE (FuncE x t f) (interp v))   calls function subst
      interp (ExprC (ExprC v1 g1 Dyn l1) Dyn g2 l2) = interp (succeed (ExprC (ExprC v1 g1 Dyn l1) Dyn g2 l2))   calls function succeed
      interp (ExprC (Blame t l) t1 t2 l1) = if (t == t1) then (Blame t2 l) else interp (ExprC (Blame t l) t1 t2 l1)   implements rule Blame from figure 2.2
      interp (ExprC None t1 t2 l1) = if (t1 == Dyn && t2 /= Dyn) then (Blame t2 l1) else (Blame t1 l1)   in case value in cast is None, it will generate blame
      interp (AppE (ExprC v1 (FuncT t1 t2) (FuncT t3 t4) l) v2) = interp (appcast (AppE (ExprC v1 (FuncT t1 t2) (FuncT t3 t4) l) v2))   calls function appcast
      interp (AppE expr1 expr2) = (AppE (interp expr1) (interp expr2))   implements rule Cong from figure 2.2
      interp (ExprC expr t1 t2 l)
              | (isValue expr) && t1 == t2 && t1 /= (FuncT Dyn Dyn) && t1 /= Dyn = expr   implements rule idBase from figure 2.2
              | (isValue expr) && t1 == t2 && t1 == Dyn = expr   implements rule idStar from figure 2.2
              | t1 /= Dyn && t2 == Dyn = interp (ground (ExprC (interp expr) t1 Dyn l))   calls function ground
              | t1 == Dyn && t2 /= Dyn = interp (expand (ExprC (interp expr) Dyn t2 l))   calls function expand
              | otherwise = interp (ExprC expr t1 t2 l)
        reduction of constants
      interp (ConstI x TInt) = (ConstI x TInt)
      interp (Minus (ConstI x TInt)) = Minus (ConstI x TInt)
      interp (ConstI x Dyn) = (ConstI x Dyn)
      interp (Minus (ConstI x Dyn)) = Minus (ConstI x Dyn)
      interp (ConstB x TBool) = (ConstB x TBool)
      interp (ConstB x Dyn) = (ConstB x Dyn)
      interp (ConstF x TFloat) = (ConstF x TFloat)
```

```
    interp (Minus (ConstF x TFloat)) = Minus (ConstF x TFloat)
    interp (ConstF x Dyn) = (ConstF x Dyn)
    interp (Minus (ConstF x Dyn)) = Minus (ConstF x Dyn)
    interp (VarE y) = VarE y
      reduction of arithmetic expressions
    interp (Add e1 e2)
            | isInt expr1 = x
            | isFloat expr1 = y
            | isDynInt expr1 = w
            | isDynFloat expr1 = z
            where
              x = case x of
                    x | (isFloat expr2)   > ConstF (fromInt (takeInt expr1) + takeFloat expr2) TFloat
                      | isDynFloat expr2  > ConstF (fromInt (takeInt expr1) + takeFloat expr2) Dyn
                      | isInt expr2       > ConstI (takeInt expr1 + takeInt expr2) TInt
                      | isDynInt expr2    > ConstI (takeInt expr1 + takeInt expr2) Dyn
              y = case y of
                    y | isFloat expr2     > ConstF (takeFloat expr1 + takeFloat expr2) TFloat
                      | isDynFloat expr2  > ConstF (takeFloat expr1 + takeFloat expr2) Dyn
                      | isInt expr2       > ConstF (takeFloat expr1 + fromInt (takeInt expr2)) TFloat
                      | isDynInt expr2    > ConstF (takeFloat expr1 + fromInt (takeInt expr2)) Dyn
              w = case w of
                    w | isFloat expr2 || isDynFloat expr2  > ConstF (fromInt (takeInt expr1) + takeFloat expr2) Dyn
                      | isInt expr2 || isDynInt expr2      > ConstI (takeInt expr1 + takeInt expr2) Dyn
              z = case z of
                    z | isFloat expr2 || isDynFloat expr2  > ConstF (takeFloat expr1 + takeFloat expr2) Dyn
                      | isInt expr2 || isDynInt expr2      > ConstF (takeFloat expr1 + fromInt (takeInt expr2)) Dyn
              expr1 = interp e1
              expr2 = interp e2
    interp (Sub e1 e2)
            | isInt expr1 = x
            | isFloat expr1 = y
            | isDynInt expr1 = w
            | isDynFloat expr1 = z
            where
              x = case x of
                    x | isFloat expr2    > ConstF (fromInt (
   takeInt expr1)   takeFloat expr2) TFloat
                      | isDynFloat expr2 > ConstF (fromInt (
   takeInt expr1)   takeFloat expr2) Dyn
                      | isInt expr2      > ConstI (takeInt expr1
    takeInt expr2) TInt
                      | isDynInt expr2   > ConstI (takeInt expr1
    takeInt expr2) Dyn

              y = case y of
                    y | isFloat expr2    > ConstF (takeFloat expr1
      takeFloat expr2) TFloat
                      | isDynFloat expr2 > ConstF (takeFloat expr1
      takeFloat expr2) Dyn
                      | isInt expr2      > ConstF (takeFloat expr1
      fromInt (takeInt expr2)) TFloat
                      | isDynInt expr2   > ConstF (takeFloat expr1
      fromInt (takeInt expr2)) Dyn

              w = case w of
                    w | isFloat expr2 || isDynFloat expr2  > ConstF
    (fromInt (takeInt expr1)   takeFloat expr2) Dyn
                      | isInt expr2 || isDynInt expr2      > ConstI
    (takeInt expr1   takeInt expr2) Dyn

              z = case z of
                    z | isFloat expr2 || isDynFloat expr2  > ConstF
    (takeFloat expr1   takeFloat expr2) Dyn
                      | isInt expr2 || isDynInt expr2      > ConstF
    (takeFloat expr1   fromInt (takeInt expr2)) Dyn
              expr1 = interp e1
              expr2 = interp e2
    interp (Mul e1 e2)
            | isInt expr1 = x
```

```
              | isFloat expr1 = y
              | isDynInt expr1 = w
              | isDynFloat expr1 = z
              where
                x = case x of
                      x | isFloat expr2     > ConstF (fromInt (
    takeInt expr1) * takeFloat expr2) TFloat
                        | isDynFloat expr2  > ConstF (fromInt (
    takeInt expr1) * takeFloat expr2) Dyn
                        | isInt expr2       > ConstI (takeInt expr1 *
     takeInt expr2) TInt
                        | isDynInt expr2    > ConstI (takeInt expr1 *
     takeInt expr2) Dyn

                y = case y of
                      y | isFloat expr2     > ConstF (takeFloat expr1
     * takeFloat expr2) TFloat
                        | isDynFloat expr2  > ConstF (takeFloat expr1
     * takeFloat expr2) Dyn
                        | isInt expr2       > ConstF (takeFloat expr1
     * fromInt (takeInt expr2)) TFloat
                        | isDynInt expr2    > ConstF (takeFloat expr1
     * fromInt (takeInt expr2)) Dyn
                w = case w of
                      w | isFloat expr2 || isDynFloat expr2  > ConstF
     (fromInt (takeInt expr1) * takeFloat expr2) Dyn
                        | isInt expr2 || isDynInt expr2      > ConstI
     (takeInt expr1 * takeInt expr2) Dyn
                z = case z of
                      z | isFloat expr2 || isDynFloat expr2  > ConstF
     (takeFloat expr1 * takeFloat expr2) Dyn
                        | isInt expr2 || isDynInt expr2      > ConstF
     (takeFloat expr1 * fromInt (takeInt expr2)) Dyn
                expr1 = interp e1
                expr2 = interp e2
       interp (Div e1 e2)
              | isInt expr1 = x
              | isFloat expr1 = y
              | isDynInt expr1 = w
              | isDynFloat expr1 = z
              where
```

```
             x = case x of
                   x | isFloat expr2     > ConstF (fromInt (takeInt expr1) / takeFloat expr2) TFloat
                     | isDynFloat expr2  > ConstF (fromInt (takeInt expr1) / takeFloat expr2) Dyn
                     | isInt expr2       > ConstF (fromInt (takeInt expr1) / fromInt (takeInt expr2)) TFloat
                     | isDynInt expr2    > ConstF (fromInt (takeInt expr1) / fromInt (takeInt expr2)) Dyn

             y = case y of
                   y | isFloat expr2     > ConstF (takeFloat expr1 / takeFloat expr2) TFloat
                     | isDynFloat expr2  > ConstF (takeFloat expr1 / takeFloat expr2) Dyn
                     | isInt expr2       > ConstF (takeFloat expr1 / fromInt (takeInt expr2)) TFloat
                     | isDynInt expr2    > ConstF (takeFloat expr1 / fromInt (takeInt expr2)) Dyn
             w = case w of
                   w | isFloat expr2 || isDynFloat expr2  > ConstF (fromInt (takeInt expr1) / takeFloat expr2) Dyn
                     | isInt expr2 || isDynInt expr2      > ConstF (fromInt (takeInt expr1) / fromInt (takeInt expr2)) Dyn
             z = case z of
                   z | isFloat expr2 || isDynFloat expr2  > ConstF (takeFloat expr1 / takeFloat expr2) Dyn
                     | isInt expr2 || isDynInt expr2      > ConstF (takeFloat expr1 / fromInt (takeInt expr2)) Dyn
             expr1 = interp e1
             expr2 = interp e2
    interp (Less e1 e2)
           | isInt expr1 = x
           | isFloat expr1 = y
           | isDynInt expr1 = w
           | isDynFloat expr1 = z
           where
             x = case x of
                   x | isFloat expr2 || isDynFloat expr2  > ConstB (fromInt (takeInt expr1) < takeFloat expr2) TBool
                     | isInt expr2 || isDynInt expr2      > ConstB
```

```
(takeInt expr1 < takeInt expr2) TBool
            y = case y of
                  y | isFloat expr2 || isDynFloat expr2  > ConstB
(takeFloat expr1 < takeFloat expr2) TBool
                    | isInt expr2 || isDynInt expr2      > ConstB
(takeFloat expr1 < fromInt (takeInt expr2)) TBool
            w = case w of
                  w | isFloat expr2 || isDynFloat expr2  > ConstB
(fromInt (takeInt expr1) < takeFloat expr2) TBool
                    | isInt expr2 || isDynInt expr2      > ConstB
(takeInt expr1 < takeInt expr2) TBool
            z = case z of
                  z | isFloat expr2 || isDynFloat expr2  > ConstB
(takeFloat expr1 < takeFloat expr2) TBool
                    | isInt expr2 || isDynInt expr2      > ConstB
(takeFloat expr1 < fromInt (takeInt expr2)) TBool
            expr1 = interp e1
            expr2 = interp e2
   interp (Bigger e1 e2)
          | isInt expr1 = x
          | isFloat expr1 = y
          | isDynInt expr1 = w
          | isDynFloat expr1 = z
          where
            x = case x of
                  x | isFloat expr2 || isDynFloat expr2  > ConstB
(fromInt (takeInt expr1) > takeFloat expr2) TBool
                    | isInt expr2 || isDynInt expr2      > ConstB
(takeInt expr1 > takeInt expr2) TBool
            y = case y of
                  y | isFloat expr2 || isDynFloat expr2  > ConstB
(takeFloat expr1 > takeFloat expr2) TBool
                    | isInt expr2 || isDynInt expr2      > ConstB
(takeFloat expr1 > fromInt (takeInt expr2)) TBool
            w = case w of
                  w | isFloat expr2 || isDynFloat expr2  > ConstB
(fromInt (takeInt expr1) > takeFloat expr2) TBool
                    | isInt expr2 || isDynInt expr2      > ConstB
(takeInt expr1 > takeInt expr2) TBool
            z = case z of
                  z | isFloat expr2 || isDynFloat expr2  > ConstB
```

```
(takeFloat expr1 > takeFloat expr2) TBool
                          | isInt expr2 || isDynInt expr2      > ConstB
(takeFloat expr1 > fromInt (takeInt expr2)) TBool
              expr1 = interp e1
              expr2 = interp e2
   interp (LessEq e1 e2)
            | isInt expr1 = x
            | isFloat expr1 = y
            | isDynInt expr1 = w
            | isDynFloat expr1 = z
            where
              x = case x of
                      x | isFloat expr2 || isDynFloat expr2  > ConstB
(fromInt (takeInt expr1) <= takeFloat expr2) TBool
                        | isInt expr2 || isDynInt expr2      > ConstB
(takeInt expr1 <= takeInt expr2) TBool
              y = case y of
                      y | isFloat expr2 || isDynFloat expr2  > ConstB
(takeFloat expr1 <= takeFloat expr2) TBool
                        | isInt expr2 || isDynInt expr2      > ConstB
(takeFloat expr1 <= fromInt (takeInt expr2)) TBool
              w = case w of
                      w | isFloat expr2 || isDynFloat expr2  > ConstB
(fromInt (takeInt expr1) <= takeFloat expr2) TBool
                        | isInt expr2 || isDynInt expr2      > ConstB
(takeInt expr1 <= takeInt expr2) TBool
              z = case z of
                      z | isFloat expr2 || isDynFloat expr2  > ConstB
(takeFloat expr1 <= takeFloat expr2) TBool
                        | isInt expr2 || isDynInt expr2      > ConstB
(takeFloat expr1 <= fromInt (takeInt expr2)) TBool
              expr1 = interp e1
              expr2 = interp e2
   interp (Eq e1 e2)
            | isInt expr1 = x
            | isFloat expr1 = y
            | isDynInt expr1 = w
            | isDynFloat expr1 = z
            where
              x = case x of
                      x | isFloat expr2 || isDynFloat expr2  > ConstB
```

```
    (fromInt (takeInt expr1) == takeFloat expr2) TBool
                          | isInt expr2 || isDynInt expr2      > ConstB
    (takeInt expr1 == takeInt expr2) TBool
                  y = case y of
                         y | isFloat expr2 || isDynFloat expr2  > ConstB
    (takeFloat expr1 == takeFloat expr2) TBool
                          | isInt expr2 || isDynInt expr2      > ConstB
    (takeFloat expr1 == fromInt (takeInt expr2)) TBool
                  w = case w of
                         w | isFloat expr2 || isDynFloat expr2  > ConstB
    (fromInt (takeInt expr1) == takeFloat expr2) TBool
                          | isInt expr2 || isDynInt expr2      > ConstB
    (takeInt expr1 == takeInt expr2) TBool
                  z = case z of
                         z | isFloat expr2 || isDynFloat expr2  > ConstB
    (takeFloat expr1 == takeFloat expr2) TBool
                          | isInt expr2 || isDynInt expr2      > ConstB
    (takeFloat expr1 == fromInt (takeInt expr2)) TBool
                  expr1 = interp e1
                  expr2 = interp e2
      interp (BiggerEq e1 e2)
              | isInt expr1 = x
              | isFloat expr1 = y
              | isDynInt expr1 = w
              | isDynFloat expr1 = z
              where
                x = case x of
                         x | isFloat expr2 || isDynFloat expr2  > ConstB
    (fromInt (takeInt expr1) >= takeFloat expr2) TBool
                          | isInt expr2 || isDynInt expr2      > ConstB
    (takeInt expr1 >= takeInt expr2) TBool
                  y = case y of
                         y | isFloat expr2 || isDynFloat expr2  > ConstB
    (takeFloat expr1 >= takeFloat expr2) TBool
                          | isInt expr2 || isDynInt expr2      > ConstB
    (takeFloat expr1 >= fromInt (takeInt expr2)) TBool
                  w = case w of
                         w | isFloat expr2 || isDynFloat expr2  > ConstB
    (fromInt (takeInt expr1) >= takeFloat expr2) TBool
                          | isInt expr2 || isDynInt expr2      > ConstB
    (takeInt expr1 >= takeInt expr2) TBool
```

```
             z = case z of
                   z | isFloat expr2 || isDynFloat expr2  > ConstB (takeFloat expr1 >= takeFloat expr2) TBool
                     | isInt expr2 || isDynInt expr2      > ConstB (takeFloat expr1 >= fromInt (takeInt expr2)) TBool
             expr1 = interp e1
             expr2 = interp e2
    interp (If e1 e2 e3) = if (takeBool (interp e1)) then (interp e2) else (interp e3)
    interp None = None

    run :: String  > Expr
    run s = interp (Parser.parse s)
```

Appendix C

Parser.y

```
{
module Parser where
import CastData
import Data.Char
  }

%name calc
%tokentype { Token }
%error { parseError }

%token
int { TokenInt $$ }
float { TokenFloat $$ }
bool { TokenBool $$ }
var { TokenVar $$ }
"Int" { TokenStringInt }
"Float" { TokenStringFloat }
"Bool" { TokenStringBool }
"Dyn" { TokenStringDyn }
'\\' { TokenLambda }
"==" { TokenEq }
">=" { TokenBiggerEq }
"<=" { TokenLessEq }
" >" { TokenArrow }
'+' { TokenAdd }
' ' { TokenSub }
```

```
'*' { TokenMul }
'/' { TokenDiv }
'<' { TokenLess }
'>' { TokenBigger }
'.' { TokenDot }
':' { TokenColon }
',' { TokenComma }
'(' { TokenOBrack }
')' { TokenCBrack }
'[' { TokenOSquare }
']' { TokenCSquare }
if   { TokenIf }
then  { TokenThen }
else  { TokenElse }
none  { TokenNone }


%nonassoc '<' '>' "<=" ">="
%left '+' ' '
%left '*' '/'
%%

Expr : Expr1 { $1 }
     | ExprArith { $1 }
     | ExprBool { $1 }

Expr1 : int { ConstI $1 TInt }
      | ' ' int { Minus (ConstI $2 TInt) }
      | '[' int ']' { ConstI $2 Dyn }
      | '[' ' ' int ']' { Minus (ConstI $3 Dyn) }
      | float { ConstF $1 TFloat }
      | ' ' float { Minus (ConstF $2 TFloat) }
      | '[' float ']' { ConstF $2 Dyn }
      | '[' ' ' float ']' { Minus (ConstF $3 Dyn) }
      | bool { ConstB $1 TBool }
      | '[' bool ']' { ConstB $2 Dyn }
      | var { VarE $1 }
      | '(' Expr ')' { $2 }
      | '(' Expr ')''(' Expr ')' { AppE $2 $5 }
      | if ExprBool then Expr else Expr { If $2 $4 $6 }
      |  '\\' var ':' Type '.' Expr { FuncE $2 $4 $6 }
```

```
        | '<' Type "<=" Type ',' var '>' Expr { ExprC $8 $4 $2 $6 }
        | none { None }

ExprArith : Expr '+' Expr1 { Add $1 $3 }
          | Expr ' ' Expr1 { Sub $1 $3 }
          | Expr '*' Expr1 { Mul $1 $3 }
          | Expr '/' Expr1 { Div $1 $3 }

ExprBool : Expr "<=" Expr1 { LessEq $1 $3 }
         | Expr ">=" Expr1 { BiggerEq $1 $3 }
         | Expr '<' Expr1 { Less $1 $3 }
         | Expr '>' Expr1 { Bigger $1 $3 }
         | Expr "==" Expr1 { Eq $1 $3 }

Type : "Int" { TInt }
     | "Float" { TFloat }
     | "Bool" { TBool }
     | "Dyn" { Dyn }
     | Type " >" Type1 { FuncT $1 $3  }

Type1 : "Int" { TInt }
      | "Float" { TFloat }
      | "Bool" { TBool }
      | "Dyn" { Dyn }

{

parseError :: [Token]  > a
parseError _ = error "Parse error"

data Token
    = TokenInt Int
     | TokenFloat Float
     | TokenBool Bool
     | TokenVar String
     | TokenEq
     | TokenBiggerEq
     | TokenLessEq
     | TokenLess
     | TokenBigger
     | TokenLambda
```

```haskell
    | TokenArrow
    | TokenStringInt
    | TokenStringFloat
    | TokenStringDyn
    | TokenStringBool
    | TokenAdd
    | TokenSub
    | TokenMul
    | TokenDiv
    | TokenDot
    | TokenColon
    | TokenComma
    | TokenOBrack
    | TokenCBrack
    | TokenOSquare
    | TokenCSquare
    | TokenIf
    | TokenThen
    | TokenElse
    | TokenNone
    deriving (Show, Eq)

lexer :: String  > [Token]
lexer [] = []
lexer (c:cs)
      | isSpace c = lexer cs
      | isAlpha c = lexVar (c:cs)
      | isDigit c = lexNum (c:cs)
lexer ('=':'=':cs) = TokenEq : lexer cs
lexer ('<':'=':cs) = TokenLessEq : lexer cs
lexer ('>':'=':cs) = TokenBiggerEq : lexer cs
lexer ('<':cs) = TokenLess : lexer cs
lexer ('>':cs) = TokenBigger : lexer cs
lexer (' ':'>':cs) = TokenArrow : lexer cs
lexer ('+':cs) = TokenAdd : lexer cs
lexer (' ':cs) = TokenSub : lexer cs
lexer ('*':cs) = TokenMul : lexer cs
lexer ('/':cs) = TokenDiv : lexer cs
lexer ('\\':cs) = TokenLambda : lexer cs
lexer ('(':cs) = TokenOBrack : lexer cs
lexer (')':cs) = TokenCBrack : lexer cs
lexer (’[’:cs) = TokenOSquare : lexer cs
lexer (’]’:cs) = TokenCSquare : lexer cs
lexer (’:’:cs) = TokenColon : lexer cs
lexer (’.’:cs) = TokenDot : lexer cs
lexer (’,’:cs) = TokenComma : lexer cs

lexNum :: String > [Token]
lexNum cs = let (num, rest) = span isDigit cs in if rest == [] then [TokenInt (read num)] else if (head rest) == ’.’ then let (first, second) = span isDigit (tail rest) in (TokenFloat (read (num ++ ”.” ++ first)) : lexer second) else TokenInt (read num) : lexer rest

lexVar :: String > [Token]
lexVar cs =
     case span isAlpha cs of
     (”if”,rest) > TokenIf : lexer rest
     (”then”,rest) > TokenThen : lexer rest
     (”else”,rest) > TokenElse : lexer rest
     (”Int”,rest) > TokenStringInt : lexer rest
     (”Bool”,rest) > TokenStringBool : lexer rest
     (”Float”,rest) > TokenStringFloat : lexer rest
     (”Dyn”,rest) > TokenStringDyn : lexer rest
     (”none”,rest) > TokenNone : lexer rest
     (var,rest) > TokenVar var : lexer rest

parse :: String > Expr
parse s = calc(lexer s)

  }
```