Book of Magne

INF222 Crashcourse 2022

Sander Wiig

INF222 Crashcourse for v2022.
Some sections are adapted from Anya's INF225 notes and course material.

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

Institute for Informatics
University of Bergen
Norway
May 19, 2022

Contents

1 Introduction

> We're building something here, detective. We're building it from scratch. All the pieces matter.
>
> *Lester Freemont*
> *The Wire*

TODO: #1 Write introduction

2 The Basics

2.1 What is a programming language

A programming language

- is an artifical language(i.e made by us humans on purpose)
- used to tell machines what to do

More formally a programming language is a set of rules that converts some input, like strings, into instructions that the computer can follow. This is of course a very general description and it, therefore, follows that there are many different types of programming languages. IT therefore should come as little surprise that we group languages by features and properties.

2.1.1 Types of languages

There are many ways of grouping languages. They can be grouped by Purpose, typing, paradigm, Generality vs. Specificity, and many more. For now, we're going to group them by paradigm, and Generality vs. Specificity.

Generality vs. Specificity

Languages are usually grouped into two categories when based on their specificity.

- DSL
- GPL

Domain Specific Languages are as the name suggests languages with a "specific" domain. DSLs usually have limited scope and use. Examples are JSON and SQL. A Domain-Specific Language is a programming language with a higher level of abstraction optimized for a specific class of problems. Optimized for a certain problem/domain. DSLs can be further subdivided into external DSL(separate programming languages), and internal DLS(language-like interface as a library.)

General Purpose Languages however are more general and can be used to solve many different problems in many different situations. These languages have a wide array of uses and are usually what we think of when we hear the words programming language. Examples of GPLs are Java and Haskell.

Characteristic	DSL vs. GPL
Domain	DSLs have a small and well-defined domain
Size	GPLs are large, DSLs are usually small
Lifespan	GPLs last for years to decades, DSLs typicaly live for shorter periors.

Figure 1: Some more comparisons between GPLs and DSLs

Paradigm

We can also classify languages by programming paradigm, some of these are;

- Imperative Languages, i.e C
- Functional Languages, i.e Haskell
- Object-oriented Languages, i.e Java, C#, C++
- Logic Languages

Write something about this also *Write body*

Syntax and Semantics

All programming languages have two parts; the **Syntax**, and the **Semantics**.
Syntax is the study of *structure*, just as semantics is the study of *meaning*. Or in other words the syntax tells us *how* to write legal programs, the semantics tells us *what* those programs do.

2.1.2 The Interpreter process

An **Interpreter** is a computer program that directly executes instructions written in a programming language. This differs from **Compiler**s since they don't have to translate the instructions into machine code to run it. We use interpreters to define the semantics of our **Abstract Syntax Tree**(AST). They are relatively straightforward to implement since since each construct of the AST defines the semantics of that construct.

An interpreter is composed of several phases.
The first is the lexical analyzer. The lexical analyzer takes the actual characters that the code is made up of and divides it up into its lexical tokens(by the tokenizer) using the concrete syntax of the program[1]. It then sends that token stream to the next part of the interpreter. The static analyzer.
The static analyzer builds a parse tree out of the tokens and the concrete syntax. This is then converted into an AST by the abstract syntax rules. This AST is then given to the next part, the Semantic Analyser. The AST is type-checked, checked for well-formedness, names are resolved, types are inferred, and much more is discussed in chapter 3. The new AST, now with added information, is then given to the evaluator so that it can be evaluated and produce a result. See figure 2. for a visual description of the steps.

[1]Not covered by this course, so you can safely ignore how this works.

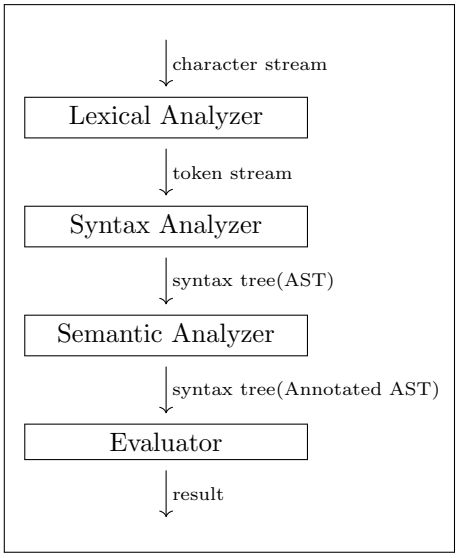

Figure 2: Phases of an interpreter

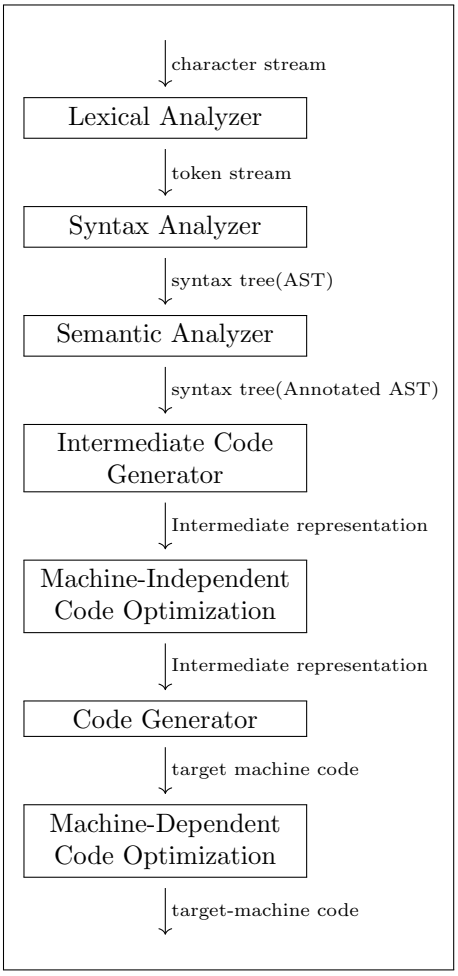

Figure 3: Phases of a "normal" compiler

Figure 4: Please appreciate the figures above, they were hard to make

2.2 Meta Programming

One of the harder things in the course is **Meta-Programming**. INF222 is usually the first time you've encountered meta-programming and it can be hard a hard concept to grasp. Meta-programming is programming *about* programming. More properly meta-programms treat other programms as data. When you see a datastructure like **Basic Signature Language** or **Basic Imperative Programing Language** in Haskell it represents a program.

2.3 Q&A

3 ASTs & Static Analysis

3.1 AST

An **Abstract Syntax Tree** is the tree representation of the syntactic structure of a program; The abstract syntax describes the structure of the abstract syntax tree - it can be defined using a regular tree grammar, or an algebraic data type or term (in Rascal, ML, Prolog, ...), or an object-oriented inheritance hierarchy of node classes (Java, C++, ...), or as an S-expression (in Lisp languages). The abstract syntax tree can be used as an internal representation in a language processor, but it is not the only possible representation.
An abstract syntax can be generated by a grammar in the following way: For every non-terminal type, there is a corresponding abstract syntax type. Each type has one constructor (or node type) corresponding to each production in the grammar, with one child for every symbol in the production that is not a literal token (e.g, punctuation, keywords, or spaces). If a constructor has only one child, of the same type, it can be removed (e.g., this would be the case for a parenthesis expression). You can do this process entirely based on the information contained in a parse tree. Translating a parse tree into a corresponding abstract syntax tree is called imploding the parse tree. Given an abstract syntax tree, it is possible to reconstruct a parse tree or program text, given the original grammar - though the resulting program may be slightly different in terms of spaces and punctuation. This is called unparsing or pretty-printing (particularly if the output is nicely formatted). Parsing, imploding, pretty-printing, and then reparsing may not yield the exact same parse tree as the original tree, but it should still implode to the same abstract syntax tree (otherwise there is a bug in your toolchain!).

Various phases in a language processor may change the abstract syntax tree, or use slightly different versions of the abstract syntax (e.g., after type checking, the nodes for variables include the type of the variable) - it is also possible to decorate or annotate the AST as processing proceeds. This adds extra information to the nodes in the AST, without impacting the structure of the abstract syntax. Important abstract syntax design considerations are;

- Simplicity. Generally, your compiler tools will do a lot of work on AST, and the fewer different cases you have to worry about, the better. For For example, if the processing of overloaded functions and operators is the same (which it is to some degree in C++), you may want to have only one AST node type to cover both. Having a lot of unnecessary nodes in the tree can be annoying as well, and may make processing slower.
- Good correspondence with the constructs of the language.
- Availability of information during processing. Some information that can be computed from the tree (such as type information) and might be encoded directly in the tree (at least at later stages) for easy processing.
- Being end-user friendly or familiar to most programmers isn't an important consideration - the abstract syntax may be radically different from the surface concrete syntax if that helps the compiler writer.

3.1.1 Sum of products

You may have encountered the term **Sum of Products**, lets's quickly run over why we use the terms *sum* and *product* to describe the data types, and show some examples in both Haskell and Java.

Haskell

```haskell
data SomeType = A Bool Bool Bool
              | B Bool
              | C
```

Here the type SomeType has 3 constructors, A, B, and C, where A takes 3 parameters, B takes one, and C zero. The type of SomeType could be expressed algebraically as

$$\underbrace{(\text{Bool} \times \text{Bool} \times \text{Bool})}_{A} + \underbrace{\text{Bool}}_{B} + \underbrace{1}_{C}$$

The Bool type can take on 2 different values (False and True), so the constructor can construct $2 * 2 * 2 = 8$ different values, since there are 8 different combinations you can make from 3 booleans (e.g. A True False False is one example). The constructor B can produce 2 different values, and C can only produce one (not zero!). Thus, the total numbers of values of type SomeType is $8 + 2 + 1 = 11$, as the data type is the sum of the three products we've just described.
In short, a sum type denotes "one of" its constituent types (if your function takes SomeType as input, it will get either (A b1 b2 b3), (B b1) or C for some boolean values b1 . . .), while a product type denotes "all of" its constituent types (e.g. a value constructed with A will have all 3 booleans present.) Another way to express a product type in Haskell is with tuples, e.g.

```haskell
type MyTriple = (Bool, Int , Char)
```

Java

In Java, we can model the same kind of data types using classes and inheritance. The instance variables of a class determine a "product" type, e.g.

```java
class SomeClass {
    boolean a;
    boolean b;
    boolean c;
}
```

Similar to the constructor for A in the previous section, there are $2 * 2 * 2 = 8$ different values an object of class SomeClass can have. Add another boolean, and you get 16 different values. Technically, a variable of type SomeClass can take one $8 + 1$ different values, since null is also

a valid value for all object variables in Java, but sometimes we ignore this fact and tell the users of our functions to kindly not pass in null as an argument where we expect an actual object. Now, to get sum types, we might use class hierarchies in Java:

```
interface SomeType {}

class A implements SomeType {
    boolean a;
    boolean b;
    boolean c ;
}
class B implements SomeType {
    boolean a;
}
class C implements SomeType {
}
```

Now, an object of type SomeType can (ignoring null) take on $8 + 2 + 1$ different values, just like in the Haskell example above.

3.2 Static Analysis

3.2.1 Type Checking

Programming language typing always falls into one or two categories; static, and dynamic typing. Static typing is when type checking happens at compile-time, while dynamic typing happens during runtime. We will focus on static typing. Statically typed languages typically have these properties.

- Variables and data structures must be declared before use.
- Variables and data structure fields can only hold values of the declared type.
- Operations(i.e functions, procedures, methods) and types must be declared.
- Declaring the exact types of variables and operations isn't always needed. Some languages use type inference (Discussed in 3.2.3).

What is a type checker and how does it work? A type checker is a meta-program that checks that verifies that the type of some construct(lists, expressions, etc.) matches what's expected of it. For example, a type checker will check that the Plus **Expression** takes two Integers. This lets the type checker discover and report certain errors before the program runs. To do this a type checker needs to know;

- How the language should look.
- The language types.
- Rules for assigning types to the constructs.

Let's do a practical example

Here's the abstract syntax for our Simple Typed Language or STL for short.

```
type VarDecl = [(String, ExprType)]

data Expr =
    Var String
    | I Int
    | B Bool
    | BinOp Op Expr Expr
    | UnOp Op Expr
    | Choice Expr Expr Expr deriving (Show, Eq)

data Op = Plus | Mult | Or | And | Not | Eq deriving (Show, Eq
```

Before we can continue we need to define the types that are allowed. We decide to use two types, Integers, and Booleans.

```
data ExprType = Integer | Boolean deriving (Show, Eq)
```

Now to the meat of the exercise, the type checker itself. The usual way to do this in Haskell(and most other languages I've experienced) is to define a series of recursive functions, one for each expression/operand.

```
            typeCheck :: VarDecl->Expr->ExprType
```

I have found it easiest, to begin with, the cases that form the basic "building blocks" of a language, the literals. It is easy to know the type of Int Literals(Integer obviously), and Bool Literals(Boolean).

```
typeCheck _ (I _) = Integer
typeCheck _ (B _) = Boolean
```

After these "base" cases we add a case for Vars, this is slightly more complicated since the type is dependent on the list of var declarations, so we need to check the VarDecl list.

```
typeCheck vars (Var name) = case lookup name vars of
            Just tp -> tp
            Nothing -> error $ "No variable could be found named "++name
```

We use a pretty clever Haskell expression called "case". This lets us pattern match the result of the function call in lookup. I strongly recommend that you all learn how to use these since they are extremely useful and I'll be using them liberally during this example.
We now move on to the ops.

```
typeCheck vars (UnOp Not expr) =
            case typeCheck vars expr of
                Boolean-> Boolean
                _-> error "Argument not boolean"
```

```
5
6  typeCheck vars (BinOp Plus left right) =
7                case (typeCheck vars left, typeCheck vars right) of
8                       (Integer, Integer)->Integer
9                       _->error "One of the args is not an Integer"
10
11 typeCheck vars (BinOp Mult left right) =
12               case (typeCheck vars left, typeCheck vars right) of
13                      (Integer, Integer)->Integer
14                      _->error "One of the args is not an Integer"
15
16 typeCheck vars (BinOp Or left right) =
17               case (typeCheck vars left, typeCheck vars right) of
18                      (Boolean, Boolean)->Boolean
19                      _->error "One of the args is not a Boolean"
20
21 typeCheck vars (BinOp And left right) =
22               case (typeCheck vars left, typeCheck vars right) of
23                      (Boolean, Boolean)->Boolean
24                      _->error "One of the args is not a Boolean"
25 typeCheck vars (BinOp Eq left right) = Boolean
```

These all more-or-less follow the same pattern [2]. They all check that the arguments are of the correct type and return the operand type if so, if not they raise an error. Eq is the odd one out since it returns a boolean no matter what since it checks if two expressions are the same.
The last case is the most complex. Choice tests a boolean condition and returns one of the two expressions depending on the value. The problem is that we don't know which branch will return. We have therefore decided both branches need to be of the same type.

```
1  typeCheck vars (Choice test left right) =
2         case typeCheck vars test of
3                Boolean | l == r -> r
4                        | otherwise -> error "Args did not match"
5                _-> error "Test condition is not a Boolean"
6         where
7         l = typeCheck vars left
8         r = typeCheck vars right
```

Here we begin by checking that the test evals to a boolean type, if so we check that both branches have the same type and return that type. Note that even though an evaluator would only evaluate one of the two branches, we still type check them both.

3.2.2 Wellformedness

For a program to be **Wellformed**it needs to satisfy all the constraints(kinda like rules) on it. This means that the program follows all the rules for it like;

- The program conforms to the AST.

[2] Maya made me use fancier words, apparently "pretty similar" isn't good enough.

- It is typed correctly
- All procedure calls/declarations are wellformed.
- and much more.

To check if a program is wellformed we usually implement a so-called constraint checker. These are pretty much just unit tests. The recipe for a constrain checker is as follows;

- **Negative test cases** - Designate one negative test case for each constraint that should be checked. Ideally, each such test case should violate only one constraint.
- **Reporting** - Choose an approach to "reporting". The result of constraint violation may be communicated either through a boolean value, as a list of errors, or by throwing an exception.
- **Modularity** - Implement each constraint in a separate function, thereby allowing modularity and testing.
- **Testing** - The constraint violations must be correctly detected for the negative test cases. The positive test case must pass.

3.2.3 Type inference

Some languages don't specify the type of expressions and therefore need to be inferred. See example. Some of the expressions in the language below have different types depending on the input. We, therefore, need to create a type inferred(real word?).

```
type ShittyEnviroment = [(String, Expr)]

data Expr =
       Var String
       | IntLit Integer
       | StrLit Str
       | Plus Expr Expr
       | Mult Expr Expr
```

We decide on some rules for those pesky ambiguous expressions. A string plus another string produces a string, an integer, and a string also produces a string. We also decide that it should be possible to multiply a string with a number. Using these rules we start making our type inferior. Like previously we start with the literals and vars.

```
inferType _ (IntLit _) = Integer
inferType _ (StrLit _) = String
inferType env (Var name) = case lookup name env of
                      Just expr -> inferType env expr
                      _ -> error "Can't find var"
```

We then implement Plus and Mult expressions based on the rules.

```
inferType env (Plus left right) = case (inferType env left, inferType env right) of
                              (Integer, Integer)->Integer
```

```
                                (_,_)->String --If it is not two ints it's a string!

ingerType env (Mult left right) = case (inferType env left, inferType env right) of
                                (String, Integer)-> String
                                (Integer, Integer)->Integer
                                _->error "invalid arguments"
```

We decide on the return type by looking at the argument types.

3.3 Q&A

4 State & Scoping

4.1 State

4.1.1 Store

When running a program we often want to remember intermediary values or have variables. For us to do this we need some way of storing values. To do this we create a **Store**. A store is very simply an array, usually containing either bytes or ValueTypes. The store is the program's memory(this is true in C). To access a value located at i in our store we simply access the value at array index i. Worth noting that the store is what we call the program heap and by tradition, it grows upwards. This means that new entries are stored at the highest available index in our store. This is because the stack grows upwards and gives us the best possible use of the memory.

4.1.2 Variables and Enviroment

We now have a place to store stuff, but how do we know where it is in the store. This is where **Enviroment**s comes into play.
An environment is a map between variables and their location in the store.

4.2 Scoping

A **Scope** is a collection of identifier bindings - i.e, what is captured by the environment at some point in the code or in time.
The scope of a declaration includes all the points in the code where that declaration exists(binding). In lexical scoping the scope of a declaration is usually either included in the declaration constructor or extends to the nearest scoping container(think. Haskells let, where — Java curly-braces). In dynamic scoping, the scope of a declaration is determined at runtime and lasts until the program exits the scoping construct. Bindings can also be shadowed by declaring a new variable with the same name as an in-scope variable. The results in the outer variables passing out of scope while the inner variable is in-scope. The shadowed variable still exists, but it's not accessible until the shadowing variable passes out of scope.
Scoping is especially important when it comes to procedures and they usually have their own environments.
Speaking of procedures...

4.3 Q&A

5 Procedures

A procedure is a programming construct that abstracts away the implementation of an algorithm. Instead of writing 20 lines of code every time you want to run an algorithm we simply call a procedure with those 20 lines of code inside it. This increases the readability of our code and makes it easier to write it.

5.1 Procedure Declarations

In this course, we have a simplified procedure declaration inspired by PASCAL. Our procedure declarations have a list of parameters(variables that pass in/out of the procedure), another list of local variables(variables that are used within), and a single statement(the algorithm itself). If we were to translate this abstract syntax into something more concrete then a typical procedure might look something like this.

```
procedure p ( upd a: integer , out b: integer ; obs c : boolean ) ;
    var x: integer ;
        y: boolean ;
        z : boolean ;
    begin
        if c then begin b := a ; x := b ; a := x end;
        y := not c ;
        c := y or c ;
    end;
```

The abstract syntax would be something like this in Haskell;

```
-- | Procedure declaration
data ProcedureDeclaration = Proc
    String -- Name of the procedure
    [Parameter] -- Parameter list
    [VarDecl] -- Local variables
    Stmt -- Statement part
    deriving (Show, Eq, Read)

-- | Procedure parameters: mode and variable declaration
type Parameter = (Mode, VarDecl)

-- | Parameter modes: observe, update, output
data Mode = Obs | Upd | Out
           deriving (Show, Eq, Read)

-- | Variable declaration: variable name and its typ
type VarDecl = (Var,Type)
```

5.1.1 Parameters & Local Variables

A **Local Variable** is a variable that only exits in a specific part of the program. We're going to talk about them in the context of procedures. These local variables are declared or defined and only used within that procedure. Since the procedure exists in a vacuum, the local variable is allowed to have the same name as variables in other parts of the program. Since the variable is stored at a different store location any change to the local variable wouldn't change the variable outside.

Parameter are the variables the procedure uses to communicate to the outside world. They are variables, but with one key difference; They also specify *how* it communicates with the outside world, the parameters specify if the variable is an output variable(write), observed only(read), or updatable(is this a word?)(read and write).
Another way of looking at parameters is that parameters are local variables that are initialized with the passed arguments at invocation time.

5.1.2 Performing a Procedure

When performing a function we can usually assume that the parameters have already been added to the environment and initialized and that except for those parameters we have a clean environment to work with. The performing of a procedure consists of the following steps;

1. The first is that we need to somehow remember the current environment or delete all the local variables when we're done. The best way is to get the current **Stackframe**.
2. We then add all the local variables.
3. We then execute the procedure statements.
4. We then reset the environment back to how it looked before using the saved stack frame from (1). This ensures that all local variables that shouldn't exist outside the procedure are removed.

5.2 Executing a Procedure Call

We now know how to actually perform a procedure, but for us to even be able to perform a procedure in the first place we need to prepare the environment and program for it. There are two main ways of making the program ready to perform a procedure. These are dependent on what type of **Argument** passing we are doing.

5.3 Arg Passing

Argument passing fall into one of two camps.

5.3.1 Copy Semantics

The first is **Copy Semantics**. In copy semantics, all the arguments are passed to the parameters by copying the value into them. With copy semantics, the procedure arguments are first evaluated and then bound to the parameters when they are added. The procedure is then performed, and the results are obtained. Those are then copied back to all the arguments that are upd or out. To further explain let's look at a trivial example to understand what happens in-store. Say we have a procedure that simply adds 5 to any number.

```
1 procedure add5(obs x: integer, out res: integer)
2 begin
3     res = x + 5;
4 end;
```

Now let's look at the main procedure

```
1 procedure main()
2 var a: integer;
3 var b: integer;
4 begin
5     a = 42;
6     add5(a, b);
7 end;
```

Figure 5 shows us what the state looks like before we start executing the procedure call. The

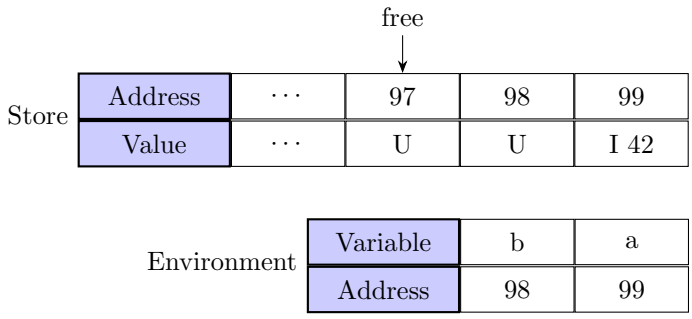

Figure 5: State before the procedure call

first thing we do when entering the procedure call is to save the stackframe so we can restore the environment to its proper place once we're done. We then clear the environment, ensuring that the only variable declared by the procedure is in scope. We then add all the parameters(in our case we add x and res) and copy the values of a and b into x and res respectively [3]. Figure 6 shows what the state looks like at this point.

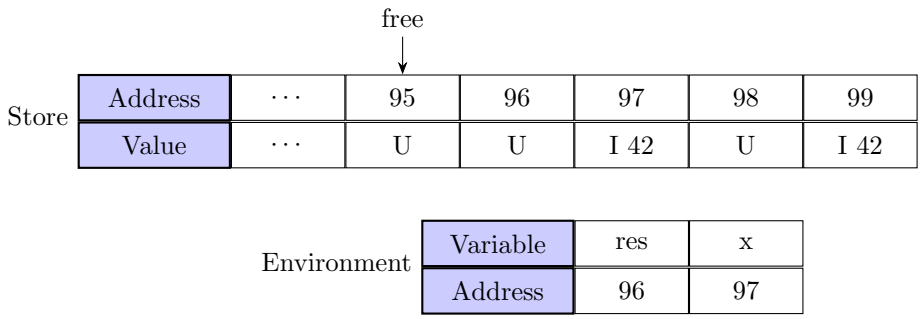

Figure 6: State when entering add5 (copy semantics).

We have now prepared the procedure so that it can be performed. Figure 7 shows how the state looks after its been performed.

[3] since res is out it isn't initialized

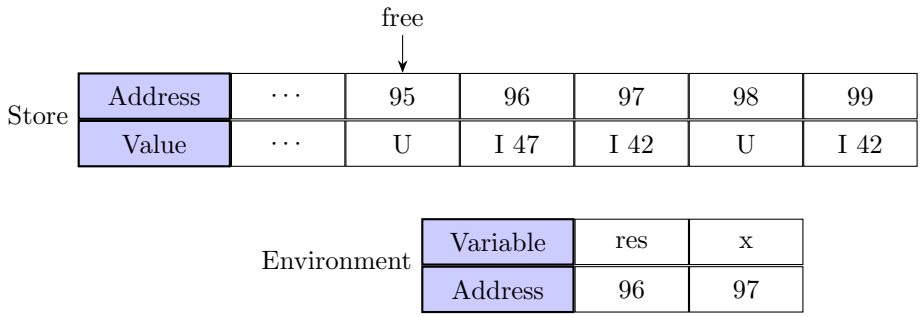

Figure 7: State after res = x +5 in add5 (copy semantics).

Now comes the fun part, *cleaning*! Just performing the function isn't enough, if we were to just drop it here we would have an environment that is drastically different than then when we started. We could just reset the stackframe to what it was when we started, but then b wouldn't get its new value. What we need to do is copy the values of the out/upd params back to their argument variables. Then we can reset the stackframe making our state look like fig. 5. As you can see the free pointer now points to where x used to point and b now has the same value as a res. That means that the next time we add something to the environment it will overwrite those values in the store.

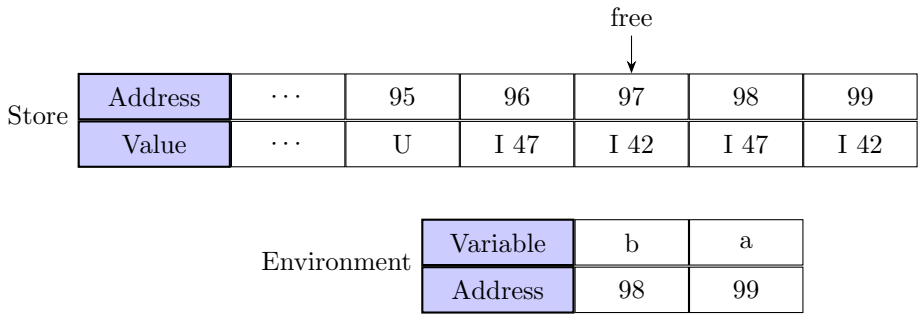

Figure 8: State after add5(a, b) in main (copy semantics).

5.3.2 Reference Semantics

Now, what if `add5` could instead write *directly* to b in `main` instead of having to allocate its own copy which is later copied? This is the main advantage of **Reference Semantics**. Lets reuse the example from before. As you can see the from fig. 9 the state before we enter the procedure call is the same as during copy semantics.

It is now we encounter our first change. Like before we get the stackframe before clearing the environment and adding the parameters, but here comes the change. instead of allocating space and copying the value of the args to the upd/out params we instead set their address to the address of the corresponding argument. We now get a state that looks like fig. 10. Since x is an obs parameter

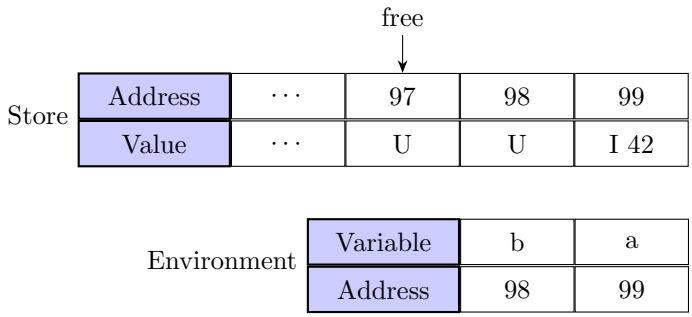

Figure 9: State before the procedure call

it is still allocated and copied to as normal.

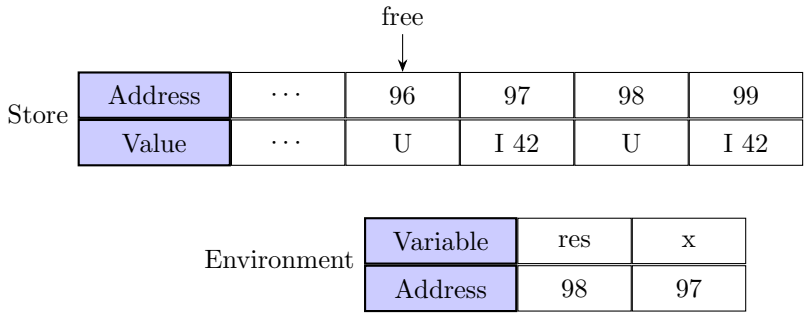

Figure 10: State when entering add5 (reference out).

Since res now points to the same place as b, any changes to ref will also be reflected by b. This makes cleanup much easier! infact all we have to do after performing the procedure is to reset the enviroment back to where it was and call it a day makng the state look like fig. 11.

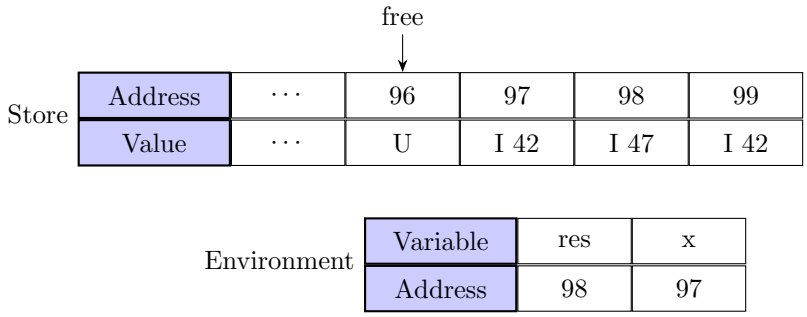

Figure 11: State after res = x+5 in add5 (reference out).

5.4 Q&A

6 Generics

To explain generics we'll demonstrate an example where generics are useful. Say we wanted to test if some algebraic type was a ring. It would be inefficient to create a unique test for each algebraic type we wanted to check. This is where generics are useful. A generic mechanism would let us write only one of these tests. Any generic mechanism needs three components, a Lets start with showing the ASR and datastructures we're going to be using.

```
type Enviroment valuedomain= [(String, ExprAST valuedomain)]
data ExprAST valuedomain
    = Lit valuedomain TypeName
    | Fun FunName [ExprAST valuedomain]
    | Var VarName
    | Assert (ExprAST valuedomain)
    deriving (Eq, Read, Show)
```

Now for an algebraic structure to be a ring, certain properties need to hold. We, therefore, add some tests to check for these[4].

```
--Tests that carrier set is an abelian group
testAssoc :: ExprAST valuedomain
testAddComm ::ExprAST valuedomain
testAddID ::ExprAST valuedomain
testAddInv ::ExprAST valuedomain

--Tests that carrier set is a monoid under multiplication
testMultAssoc::ExprAST valuedomain
testMultID ::ExprAST valuedomain

--Test that multiplication is distributive
testLDist :: ExprAST valuedomain
testRDist :: ExprAST valuedomain

--List of tests
tests = [testAssoc, testAddComm, testAddID, testAddInv, testMultAssoc, testMultID,
    testLDist, testRDist]
```

6.1 Signature

To represent the different types(of algebraic structures) we introduce the concept of a signature.

```
type Signature = ([TypeDeclaration],[FunDeclaration])
```

A signature is a data structure that contains a list of types, and the operations(functions, etc.) that operate on those types. Signatures have other applications other than just generics. Defining

[4]The implementation is found in Appendix B

functions in signatures means that if we want to expand our language with a new intrinsic function we won't have to change the code anywhere else than in the signature and the accompanying semantic implementation.
Let's take a look at the signature for the Integers.

```
int ::Signature
int = (["Int", "Bool"],[
               ("Add", ["Int","Int"],"Int"),
               ("Mult", ["Int","Int"], "Int"),
               ("Neg", ["Int"], "Int"),
               ("Eq", ["Int", "Int"], "Bool"),
               ("LEq", ["Int", "Int"], "Bool")
       ])
```

As you see the integers have 5 functions, the first two are binary operators that are needed for a ring, the `neg` function is to represent the negative numbers. `Eq` and `LEq` are needed so we can compare the numbers and check transitivity. Since we have multiple types defined in our signatures we need to tell Haskell about them, this is why we have `valuedomain` everywhere. But since we have multiple types in the signature we need to create a custom data structure to represent it.

```
-- | The semantic domain for the rings sigs
data Ring = BoolDom Bool
       | NatDom Int Bool
       | IntDom Int Bool
       | RealDom Float Bool
       deriving(Show, Eq, Ord)
```

Now that we have defined what functions these algebraic structures have we need some way of defining their semantics, i.e what they do. To do this we create a `FunctionModel` for each function implemented by a signature.

```
type FunModel valuedomain = FunName -> [valuedomain] -> valuedomain
```

Here is the semantics of our IntegerRing

```
--Semantics of a ring(when carrier set is a int)
intRingModel ::FunModel Ring
intRingModel "Add" arg = IntDom (foldl (+) 0 (map unpackInt arg)) False
intRingModel "Mult" arg = IntDom (foldl (*) 1 (map unpackInt arg)) False
intRingModel "Neg" [x] = IntDom (negate (unpackInt x)) $ getBool x
intRingModel "Eq" [x, y] = BoolDom (x==y)
intRingModel "LEq" [x, y] = BoolDom (x<=y)
```

We now have all we need to generify the evaluator.

```
-- | Eval
eval ::FunModel Ring->Signature->Enviroment Ring->ExprAST Ring->Ring
eval mod sig env (Lit i _) = i
```

```
eval mod sig env (Var name) = case lookup name env of
                          Just x -> eval mod sig env x
                          Nothing -> error "No such var in env"

eval mod (types, funcs) env (Fun name args) | elem name (map (\(x,y,z)->x) funcs) =mod
    name (map (eval mod (types, funcs) env) args)
                                        | otherwise= error "Can't find function in
                                            signature"

eval mod (sig) env (Assert expr) = let res@(BoolDom b) = eval mod sig env expr in if b
    then res else error $"Assert did not hold"++ (show expr)
```

Instead of the evaluator handling the evaluation of functions we pass it to the function model letting it deal with it according to the implementation defined there.

6.2 Property based assertions

In the example above we used asserts to check if an algebraic structure was a ring or not. Having asserts in a language can be extremely useful since it lets us check if certain conditions hold, and if they don't, to stop execution. A program crash is often the best course of action to prevent illegal states.

6.3 Q&A

7 Language Standards

7.1 Software Engineering Implications of Languages

The choice of language can have a massive impact on software development. The move to higher abstraction languages has caused a corresponding increase in efficiency since the developer can focus all their efforts on what the program should accomplish instead of having to work on the details. **Software Language Engineering** has many applications within software engineering like;

- Design, implementation, and usage of DSLs that are tailor-made for a specific problem or technical domains, like, UI, web services, configurations, testing, date exchange, interoperability, deployment, and distribution.
- Software reverse engineering and re-engineering in many forms, for example, analysis of projects regarding their dependence on open source software, integration of systems, and migration of systems constrained by legislation or technology.
- Data extraction in the context of data mining, information retrieval, machine learning, big data analytics, social science, digital forensics, and AI, with diverse input, artifacts to be parsed interchange formats to conform to.

Software languages can also impact the security of any product. If a language has a fundamental flaw or creates unknown pitfalls that might be hard to spot due to the inherent design of the language could then be exploited by malicious actors.

7.2 Reading spesifications

7.2.1 Backus-Naur Form

Backus-Naur Form is a format for describing context-free grammar. We use it to describe the syntax of languages. Although it is probably not part of the curriculum it's still important to know since most language specification describes their languages using some variation of BNF most common of which is the Extended Backus-Naur form(EBNF). An EBNF consists of terminal symbols and non-terminal production rules which are the restrictions governing how terminal symbols can be combined into a legal sequence. Examples of terminal symbols include alphanumeric characters, punctuation marks, and whitespace characters. These constructions often end up looking like Haskell data structures, this should hopefully help you understand them.

```
<symbol> ::= __expression__
```

Here is the complete PASCAL-like language that only allows assignments. in its EBNF form.

```
(* a simple program syntax in EBNF - Wikipedia *)
program = 'PROGRAM', white space, identifier, white space,
          'BEGIN', white space,
          { assignment, ";", white space },
          'END.' ;
identifier = alphabetic character, { alphabetic character | digit } ;
number = [ "-" ], digit, { digit } ;
string = '"' , { all characters - '"' }, '"' ;
assignment = identifier , ":=" , ( number | identifier | string ) ;
alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white space = ? white space characters ? ;
all characters = ? all visible characters ? ;
```

EBNF also includes regex like syntax for expressing repetition(*,+), optionality(?,[]), and alternatives(—). If the above looks confusing or unintuitive I wouldn't worry. The syntax of EBNF is fairly intuitive and most standards have fairly little of it at once.

7.3 Q&A

8 Misc.

8.1 Datastructures in memory

8.1.1 Arrays

Say we have an array of n elements of type T, and the type T takes up 4 bytes of memory. Lets also assume that the header is 8 bytes The array would then look like this in the store.

8.1.2 Records

8.1.3 Pointers & linked lists

8.2 Q&A

A Glossary

Glossary

Abstract Syntax Tree An Abstract Syntax Tree(AST) is a tree representation of the syntactic structure of our program. Can be represented by using trees or terms, and described by an algebraic data type or regular tree grammar.. 4, 6

Argument An argument is a value provided to the procedure when it is run. When the procedure is run the parameters of the procedure is initialized with its corresponding argument.. 14

Backus-Naur Form Formal notation for describing grammars. Used to describe the syntax of a language.. 22

Basic Imperative Programing Language BIPL is a trivial language to explain basic imperative programming concepts like mutable vars, assignments, control-flow, loops, and iteration. 5

Basic Signature Language Simple language to illustrate signatures. 5

Compiler A compiler is a program that translates computer code(*the source language*) into another language(*the target language*) Compilers usually convert some high level language to some lower level language.. 4

Copy Semantics Type of argument passing where the parameters are initialized with the value of the arguments. 14

Domain Specific Languages A language(i.e not just a library) with abstractions targetet at a specific problem domain.

- *External DSL* - A DSL defined as a seperate programming language.
- *Internal/Embedded DSL* - A DSL defined as a language-like interface to a library.

. 3

Enviroment Map describing where things are located in the **Store**. Kinda like a phonebook. 12

Expression An expression is a syntactic construct that can be evaluated in order to obtain its value. The resulting value is usually one of the program's types.. 8

General Purpose Languages A language suited for a wide variety of problems and situations but lacks specialized features to deal with specific programs like a DSL.. 3

Interpreter An interpreter is a program that directly executes instructions written in a programming language.. 4

Local Variable A local variable is a variable that only exists within a limited scope.. 14

Meta-Programming Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running.. 5

Parameter A parameter is a localvariable that is initialized with the arguments. Also often contains how those args are to be treated.. 14

Reference Semantics Type of argument passing where the parameters point to the the address as the argument.. 17

Scope A collection of identifier bindings . i,e what's captured by the eniroment at some point in the code.. 12

Semantics The semantics of a program concerns the meaning of the program. i.e what it actually does. It can take many forms, sometimes we're only interested in the result of the program. Other cases concern the steps taken by the program to reach said output.. 4

Software Language Engineering Software Language Engineering is the scientific field that researches language development, and mainteinace of formal descriptios, and tooling of software lanugages. 21

Stackframe The stackframe is a snapshot of how the program enviroment looks at a certain point in time. The stackframe saves things that could be changed by running the procedure and lets us restore the program to the previous state without all the changes made by the procedure.. 14

Store Program memory, byte/value array, grows uppwards.. 12, 23

Sum of Products . 7

Syntax Syntax refers to the rules that define the structure of a language. Syntax in computer programming means the rules that control the structure of the symbols, punctuation, and words of a programming language.. 4

Wellformed Wellformedness is when a program is following all the rules. smileemoji. 10

B Code Examples

B.1 Type Check Example

```
1 module TypeCheckExample where
2 type VarDecl = [(String, ExprType)]
3
4 data Expr =
5       Var String
6       | I Int
7       | B Bool
```

```
8         | BinOp Op Expr Expr
9         | UnOp Op Expr
10        | Choice Expr Expr Expr deriving (Show, Eq)
11
12  data Op = Plus | Mult | Or | And | Not | Eq deriving (Show, Eq)
13
14  data ExprType = Integer | Boolean deriving (Show, Eq)
15
16  typeCheck ::VarDecl->Expr->ExprType
17  typeCheck _ (I _) = Integer
18  typeCheck _ (B _) = Boolean
19
20  typeCheck vars (Var name) =
21                case lookup name vars of
22                        Just tp -> tp
23                        Nothing -> error $ "No variable could be found named "++name
24
25  typeCheck vars (UnOp Not expr) =
26                        case typeCheck vars expr of
27                          Boolean-> Boolean
28                          _-> error "Argument not boolean"
29
30  typeCheck vars (BinOp Plus left right) =
31                        case (typeCheck vars left, typeCheck vars right) of
32                          (Integer, Integer)->Integer
33                          _->error "One of the args is not an Integer"
34
35  typeCheck vars (BinOp Mult left right) =
36                        case (typeCheck vars left, typeCheck vars right) of
37                          (Integer, Integer)->Integer
38                          _->error "One of the args is not an Integer"
39
40  typeCheck vars (BinOp Or left right) =
41                        case (typeCheck vars left, typeCheck vars right) of
42                          (Boolean, Boolean)->Boolean
43                          _->error "One of the args is not a Boolean"
44
45  typeCheck vars (BinOp And left right) =
46                        case (typeCheck vars left, typeCheck vars right) of
47                          (Boolean, Boolean)->Boolean
48                          _->error "One of the args is not a Boolean"
49
50  typeCheck vars (BinOp Eq left right) = Boolean
51
52  typeCheck vars (Choice test left right) =
53                        case typeCheck vars test of
54                          Boolean | l == r -> r
55                                  | otherwise -> error "Args did not match"
56                          _-> error "Test condition is not a Boolean"
57                        where
```

```
                              l = typeCheck vars left
                              r = typeCheck vars right
```

B.2 Type Inference Example

```
module TypeCheckExample where

type ShittyEnviroment = [(String, Expr)]

data Expr =
       Var String
       | IntLit Integer
       | StrLit Str
       | Plus Expr Expr
       | Mult Expr Expr

--New Rules
--Int + string = string
--String+String = string
--String * int = String

data ExprType = Integer | String deriving (Show, Eq)

inferType:: ShittyEnviroment->Expr->ExprType
inferType _ (IntLit _) = Integer
inferType _ (StrLit _) = String

inferType env (Var name) = case lookup name env of
                     Just expr -> inferType env expr
                     _ -> error "Can't find var"

inferType env (Plus left right) = case (inferType env left, inferType env right) of
                              (Integer, Integer)->Integer
                              (_,_)->String --If it's not two ints it's a string!

ingerType env (Mult left right) = case (inferType env left, inferType env right) of
                              (String, Integer)-> String
                              (Integer, Integer)->Integer
                              _->error "invalid arguments"
```

B.3 Generic Example

```
module GenericsExample where

type TypeName = String
type FunName = String
```

```
type VarName = String

type Enviroment valuedomain= [(String, ExprAST valuedomain)]
--Defines semantics of a signature
type FunModel valuedomain = FunName -> [valuedomain] -> valuedomain

type FunDeclaration = (FunName, [TypeName], TypeName)
type TypeDeclaration = TypeName

--Defines what functions types exists
type Signature = ([TypeDeclaration],[FunDeclaration])

--DS??
data ExprAST valuedomain
    = Lit valuedomain TypeName
    | Fun FunName [ExprAST valuedomain]
    | Var VarName
    | Assert (ExprAST valuedomain)
    deriving (Eq, Read, Show)


--DSL Describing Ints
int ::Signature
int = (["Int", "Bool"],[
               ("Add", ["Int","Int"],"Int"),
               ("Mult", ["Int","Int"], "Int"),
               ("Neg", ["Int"], "Int"),
               ("Eq", ["Int", "Int"], "Bool"),
               ("LEq", ["Int", "Int"], "Bool")
       ])

--DSL for Describing Reals
real :: Signature
real =(["Real", "Bool"],[
               ("Add", ["Real","Real"],"Real"),
               ("Mult", ["Real","Real"], "Real"),
               ("Neg", ["Real"],"Real"),
               ("Eq", ["Real", "Real"], "Bool"),
               ("LEq", ["Real", "Real"], "Bool")
    ])
--Natural nums
nat :: Signature
nat =(["Nat","Bool"],[
               ("Add", ["Nat","Nat"],"Nat"),
               ("Mult", ["Nat","Nat"], "Nat"),
               ("Eq", ["Nat", "Nat"], "Bool"),
               ("LEq", ["Nat", "Nat"], "Bool")
    ])
--Natural nums
bool :: Signature
```

```
bool =(["Bool"],[
           ("Add", ["Bool","Bool"],"Bool"),
           ("Mult", ["Bool","Bool"], "Bool"),
           ("Eq", ["Bool", "Bool"], "Bool"),
           ("LEq", ["Bool", "Bool"], "Bool")
       ])

-- | The semantic domain for the rings sigs
data Ring = BoolDom Bool
           | NatDom Int Bool
           | IntDom Int Bool
           | RealDom Float Bool
           deriving(Show, Eq, Ord)

unpackInt :: Ring->Int
unpackInt (IntDom i _) = i
unpackNat :: Ring->Int
unpackNat (NatDom n _) = n
unpackReal :: Ring->Float
unpackReal (RealDom x _) = x
unpackBool :: Ring->Bool
unpackBool (BoolDom q) = q

getBool ::Ring->Bool
getBool (IntDom _ q) = q
getBool (NatDom _ q) = q
getBool (RealDom _ q) = q
getBool (BoolDom q) = q


-- | Semantics

--Semantics of a ring(when carrier set is a int)
intRingModel ::FunModel Ring
intRingModel "Add" arg = IntDom (foldl (+) 0 (map unpackInt arg)) False
intRingModel "Mult" arg = IntDom (foldl (*) 1 (map unpackInt arg)) False
intRingModel "Neg" [x] = IntDom (negate (unpackInt x)) $ getBool x
intRingModel "Eq" [x, y] = BoolDom (x==y)
intRingModel "LEq" [x, y] = BoolDom (x<=y)

--Semantics of a ring(when carrier set is a nat)
natRingModel ::FunModel Ring
natRingModel "Add" arg = NatDom (foldl (+) 0 (map unpackNat arg)) False
natRingModel "Mult" arg = NatDom (foldl (*) 1 (map unpackNat arg)) False
natRingModel "Neg" [x] = NatDom (negate (unpackNat x)) $ getBool x
natRingModel "Eq" [x, y] = BoolDom (x==y)
natRingModel "LEq" [x, y] =BoolDom (x<=y)

--Semantics of a ring(when carrier set is a real)
realRingModel ::FunModel Ring
```

```
105 realRingModel "Add" arg = RealDom (foldl (+) 0 ((map unpackReal arg))) False
106 realRingModel "Mult" arg = RealDom (foldl (*) 1 ((map unpackReal arg))) False
107 realRingModel "Neg" [x] = RealDom (negate (unpackReal x)) $ getBool x
108 realRingModel "Eq" [x, y] = BoolDom (x==y)
109 realRingModel "LEq" [x, y] =BoolDom (x<=y)
110
111 --Semantics of a ring(when carrier set is bool)
112 boolRingModel ::FunModel Ring
113 boolRingModel "Add" arg = BoolDom $ foldl (/=) False (map unpackBool arg)
114 boolRingModel "Mult" arg = BoolDom $ foldl (&&) True (map unpackBool arg)
115 boolRingModel "Neg" [x] = BoolDom (not (unpackBool x))
116 boolRingModel "Eq" [x, y] = BoolDom (x==y)
117 boolRingModel "LEq" [x, y] =BoolDom (x<=y)
118
119
120 -- | Eval
121 eval ::FunModel Ring->Signature->Enviroment Ring->ExprAST Ring->Ring
122 eval mod sig env (Lit i _) = i
123 eval mod sig env (Var name) = case lookup name env of
124                         Just x -> eval mod sig env x
125                         Nothing -> error "No such var in env"
126
127 eval mod (types, funcs) env (Fun name args) | elem name (map (\(x,y,z)->x) funcs) =mod
        name (map (eval mod (types, funcs) env) args)
128                                       | otherwise= error "Can't find function in
                                              signature"
129
130 eval mod (sig) env (Assert expr) = let res@(BoolDom b) = eval mod sig env expr in if b
        then res else error $"Assert did not hold"++ (show expr)
131
132
133 -- | Tests that carrier set is abelian group
134 testAssoc :: ExprAST valuedomain--We use Integer to represent the set
135 testAssoc =
136    Assert (
137       Fun "Eq" [
138           Fun "Add" [
139               Fun "Add" [
140                   Var"a",
141                   Var"b"
142                   ],
143               Var"c"
144           ],
145           Fun "Add" [
146               Var"a",
147               Fun "Add"
148               [
149                   Var"b",
150                   Var"c"
151               ]
```

```
            ]
        ]
    )

testAddComm ::ExprAST valuedomain
testAddComm =
    Assert (
        Fun "Eq" [
            Fun "Add" [
                Var "a",
                Var "b"
            ],
            Fun "Add" [
                Var "b",
                Var"a"
            ]
        ]
    )

testAddID ::ExprAST valuedomain
testAddID =
    Assert (
        Fun "Eq" [
            Fun "Add" [
                Var "a",
                Var "addConst"],
            Var"a"
        ]
    )

testAddInv ::ExprAST valuedomain
testAddInv =
    Assert (
        Fun "Eq" [
            Fun "Add" [
                Var"a",
                Fun "Neg" [
                    Var"a"
                ]
            ],
            Var "addConst"
        ]
    )

--Tests that carrier set is a monoid under multiplication
testMultAssoc::ExprAST valuedomain
testMultAssoc =
    Assert (
        Fun "Eq" [
            Fun "Mult" [
```

```
202                 Fun "Mult" [
203                     Var"a",
204                     Var"b"
205                 ],
206                 Var "c"
207             ],
208             Fun "Mult" [
209                 Var"a",
210                 Fun "Mult" [
211                     Var"b",
212                     Var"c"
213                 ]
214             ]
215         ]
216     )
217
218 testMultID ::ExprAST valuedomain
219 testMultID =
220     Assert (
221         Fun "Eq" [
222             Fun "Mult" [
223                     Var"a",
224                     Var "multConst"
225                 ],
226             Var "a"
227         ]
228     )
229
230 --Test that multiplication is distributive
231 testLDist :: ExprAST valuedomain
232 testLDist =
233     Assert (
234         Fun "Eq" [
235             Fun "Mult" [
236                 Var "a",
237                 Fun "Add" [Var "b",Var "c"]
238             ],
239             Fun "Add" [
240                 (Fun "Mult" [Var"a",Var "b"]),
241                 (Fun "Mult" [Var"a",Var "c"])
242             ]
243         ]
244     )
245
246 testRDist :: ExprAST valuedomain
247 testRDist =
248     Assert (
249         Fun "Eq" [
250             Fun "Mult" [
251                 Fun "Add" [Var "b", Var"c"],
```

```
252                 Var"a"
253                 ],
254             Fun "Add" [
255                 (Fun "Mult" [Var "b",Var "a"]),
256                 (Fun "Mult" [Var "c",Var "a"])
257                 ]
258         ]
259     )
260
261
262
263 tests = [testAssoc, testAddComm, testAddID, testAddInv, testMultAssoc, testMultID,
        testLDist, testRDist]
264
265 -- | main
266 test tpname (sig, model) dta = let env = [("a", Lit (dta!!0) tpname), ("b", Lit (dta!!1)
        tpname), ("c", Lit (dta!!2) tpname), ("multConst", Lit (dta!!3) tpname), ("addConst",
        Lit (dta!!2) tpname)] in
267     map (eval model sig env) tests
```