Extending F-lite with generics: G-lite

Nick Gubbels (0710288)
Supervisor: James McKinna

Abstract

In this Bachelor thesis the functional language F-lite is extended with the functionality of generic programming. This extension is called G-lite and it uses a compiler, which will translate the generic code in G-lite to ordinary F-lite code. The basic principles of generics are explained and some existing implementations are shown. A prototype compiler for the new language is developed and the performance of this language is tested in terms of number of lines of code.

Contents

1 Introduction **3**

2 F-lite **4**

3 Generics **8**
3.1 Equality, a typical case for generic programming 8
3.2 Basic data types and combinators 10
3.3 Conversion between data type and generic representation 12
3.4 Making generic functions 14
3.5 Mapping functions 16
3.5.1 Kinds 17
3.5.2 Solution 17
3.6 The advantages of generic programming 19

4 Generics in Clean **20**
4.1 Application with generics in Clean 22
4.1.1 iTasks 22
4.1.2 G∀ST 25

5 Generics in Haskell **26**
5.1 Generic H∀SKELL 26
5.2 PolyP 28
5.3 Template Haskell 29

6 Comparing generic approaches **30**
6.1 Comparison by Hinze et al. [6] 30
6.2 Comparison by Rodriguez et al. [22] 32
6.3 Concluding the comparison 32

7 Using generics in F-lite **33**
7.1 Corresponding G-lite code 35

8 G-lite compiler **36**
8.1 General program 36
8.1.1 File opening/creation 37
8.1.2 Parsing the input file 38
8.1.3 Writing the F-lite output 39
8.2 Function parsing 40
8.2.1 Parsing the function name 40

8.2.2 Parsing the arguments . . . 40
8.2.3 Parsing the function body . . . 42
8.3 Creating conversion functions . . . 42
8.4 Creating new instances . . . 45

9 Code reduction test **47**
9.1 Program 1: equality function . . . 48
9.1.1 Tree . . . 48
9.1.2 Tree and List . . . 48
9.1.3 Tree, List and Rose . . . 49
9.1.4 Tree, List, Rose and Triplet . . . 50
9.2 Increment and print . . . 50
9.2.1 Tree . . . 50
9.2.2 Tree and List . . . 51
9.2.3 Tree, List and Rose . . . 52
9.2.4 Tree, List, Rose and Triplet . . . 53
9.3 Code reduction in Clean . . . 54
9.3.1 Tree . . . 54
9.3.2 Tree and List . . . 55
9.3.3 Tree, List and Rose . . . 55
9.3.4 Tree, List, Rose and Triplet . . . 56

10 Results **58**
10.1 Consequences of adding a new data type . . . 58
10.2 Clean . . . 61

11 Discussion **63**

12 Conclusion **65**
12.1 Future work . . . 65

13 Appendix **69**
13.1 Prototype G-lite Compiler . . . 69
13.2 Comparison programs . . . 71
13.2.1 Program 1: equality . . . 71
13.2.2 Program 2: increment . . . 76

Chapter 1

Introduction

"Generic programming allows you to write a function once, and use it many times at different types." This is stated by Hinze and Jones in [7]. Generic programming makes it possible to write a function for a few data types, such that it will work for all other data types. This is very useful when a certain class of functions is used by many data types and is quite similar for each data type.

The *Reduceron*:"a special-purpose graph reduction machine ... using an FPGA". The design of the Reduceron is explained by Naylor and Runciman in [14] and [15]. In the latter they introduce a language to use with the Reduceron, called F-lite. "F-lite is a core lazy functional language, close to subsets of both Haskell and Clean". The Reduceron is designed for faster evaluation of a functional language, because traditional Von-Neumann architectures are not optimal for functional languages.

To have both the advantages of fast execution of the Reduceron and of generic programming, I did research on the possibilities of generic programming for F-lite and developed a generic version of F-lite, based on the following research question and subquestions:

- **How can generic programming be achieved in F-lite?**
  - What are the consequences of the *untyped* structure of F-lite.
  - What are the results of using generics in F-lite.

In this report I will present the results of this research. First I will explain the Reduceron and F-lite (chapter 2). After that generics will be explained in general (chapter 3) and then some implementations of generics in other languages will be considered: Clean (chapter 4) and Haskell (chapter 5). Thereafter a short comparison of the different approaches will be made (chapter 6). Then G-lite (chapter 7) and the prototype compiler (chapter 8) will be explained. This compiler is then tested (chapter 9) and the results are presented (chapter 10). Finally there is a discussion of the results (chapter 11) and a conclusion with some future work (chapter 12).

Chapter 2

F-lite

Programming in a functional language can have some advantages over programming in an imperative language, e.g. more readable code and the use of pattern matching. However functional programs are not always as fast as imperative ones. Naylor and Runciman state this in [14]: in the traditional Von Neumann architectures memory intensive applications "are limited by the rate that data can travel between the CPU and the memory", but the operational basis of standard lazy functional languages is *graph reduction*: a prime example of a memory intensive application. Therefore the Von Neumann architecture is not optimal for a functional language.

This is the reason why Naylor and Runciman developed the *Reduceron*: "a special-purpose graph reduction machine ... using an FPGA". This machine is special designed for fast execution of functional programs. In [21] Reich, Naylor and Runciman give a method to make the execution of functional programming even more fast, by adding *supercompilation*, "a metaprogramming technique ... with corresponding performance gains at execution time, to the Reduceron.

The design of the Reduceron is explained in [14] and [15]. In the latter they introduce a language to use with the Reduceron, called F-lite. This is a lazy functional language with a syntax close to Clean and Haskell.

In figure 2.1 the syntax of the language is showed as given by Naylor and Runciman [15].

e	::=	$\vec{e}$	(Application)
	|	`case` e `of` $\{\vec{a}\}$	(Case Expression)
	|	`let` $\{\vec{a}\}$ `in` e	(Let Expression)
	|	n	(Integer)
	|	x	(Variable)
	|	p	(Primitive)
	|	f	(Function)
	|	C	(Constructor)
	|	$< \vec{f} >$	(Case Table)
a	::=	$C\vec{e}$ `->` e	(Case Alternative)
b	::=	$x = e$	(Let Binding)
d	::=	$f\vec{x} = e$	

Figure 2.1: Syntax of F-lite [15]

In [15] the authors give the following example. Note that the primitives `(+)`, `(-)` and `(<=)` are predefined in the core language.

```
tri n = case of (<=) n 1 of
    { False -> (+) (tri ((-) n 1)) n ; True -> 1 }
```

A special F-lite compiler, compiles an F-lite program to template code which can be processed by the Reduceron. This happens in the following way [15]:

1. Primitives can only handle integer arguments which are fully evaluated. Therefore statements using primitives need to be translated with two rules. The statement will be translated first with the following rule:

$$pe_0e_1 \rightarrow (e_1(e_0p)) \tag{2.1}$$

When compiled the tri function will first be translated using this rule to:

```
tri n = case 1 (n (<=)) of
    { False -> n (tri (1 (n (-))) (+)) ; True -> 1 }
```

Along with rule (2.1) another rule is used when dealing with primitives:

$$ne \rightarrow en \tag{2.2}$$

The authors explain this rule with the reduction of the statement `(+) (tri 1) (tri 2)`, which will be translated to `tri 2 ((tri 1) (+))` at compile time with rule (1):

```
    tri 2 ((tri 1) (+)) { tri 2 evaluates to 3}
=   3 ((tri 1) (+))     { Rule (2) }
=   (tri 1) (+) 3       { tri 1 evaluates to 1}
=   1 (+) 3             { Rule (2) }
=   (+) 1 3
```

2. After the primitives, the compiler handles `case` statements. These are mainly transformed as follows:

$$\text{case } e \text{ of } \{C_1\vec{x_1} \rightarrow e_1; \ldots; C_m\vec{x_m} \rightarrow e_m\}$$

is translated into:

$$e(alt_1\vec{v_1}\vec{x_1}) \ldots (alt_m\vec{v_m}\vec{x_m}) \tag{2.3}$$

where vector $\vec{v_i}$ contains the free variables occurring in the i^{th} case alternative and each alt_i for i in $1 \ldots m$ has the following definition:

$$alt_i\vec{v_i}\vec{x_i} = e_i \tag{2.4}$$

After this some more refinements are made, but the details are omitted here. For more information on those refinements is referred to [15].

These transformations lead to the following `tri` function:

```
tri n = 1 (n (<=)) <falseCase , trueCase> n
falseCase t n = n (tri (1 (n (-))) (+))
trueCase  t n = 1
```

3. Thereafter the code is transformed to template code. The template code defines *what* is saved *where* in the memory of the Reduceron. The following shows the syntax of the template code:

```
> data Atom =
>    FUN Arity Int -- Function with arity and address
>  | ARG Int       -- Reference to a function argument
>  | PTR Int       -- Pointer to an application
>  | CON Arity Int -- Constructor with arity and index
>  | INT Int       -- Integer literal
>  | PRI String    -- Primitive function name
>  | TAB Int       -- Case table
```

The code of this program [15]:

```
main = tri 5
tri n = let x = n (<=) in 1 x <falseCase , trueCase> n
falseCase t n =
    let {x0 = tri x x1 (+); x1 = 1 x2; x2 = n (-)} in
        n x0
trueCase t n = 1
```

results in the following template code:

```
> tri5 :: Prog
> tri5 = [ (0, [FUN 1 1, INT 5], [])
>        , (1, [INT 1, PTR 0, TAB 2, ARG 0],
>              [[ARG 0, PRI "(<=)"]])
>        , (2, [ARG 1, PTR 0],
>              [[FUN 1 1, PTR 1, PRI "(+)"],
>               [INT 1, PTR 2],
>               [ARG 1, PRI "(-)"]])
>        , (2, [INT 1], []) ]
```

In [14] Naylor and Runciman give a figure of the memory of the Reduceron filled with a factorial function. "The bytecode for `fact`, as it would appear relative to address a in memory":

a	$+1$	$+2$	$+3$
`Start 1 15`	`Ap 7`	`Int 1`	`Ap 5`
$+4$	$+5$	$+6$	$+7$
`End (Int 1)`	`Prim (==)`	`End (Var 0)`	`Ap 12`
$+8$	$+9$	$+10$	$+11$
`Ap 10`	`End (Fun` a`)`	`Ap 14`	`End (Int 1)`
$+12$	$+13$	$+14$	$+15$
`Prim (*)`	`End (Var 0)`	`Prim (-)`	`End (Var 0)`

Note the slightly different syntax:

node	::=	`Start` *ii*	(first node of a function body: arity and size of function)
	|	`Int` *i*	(primitive integer)
	|	`Ap` *i*	(application node: a pointer to a sequence of nodes
	|	`End` *node*	(the final node in a node sequence)
	|	`Prim` *p*	(primitive function name)
	|	`Fun` *i*	(pointer to a function body)
	|	`Var` *i*	(variable representing a function argument)

In this chapter an explanation of the compilation from F-lite to the Reduceron byte code is given. However, in the remainder of this thesis, the focus will lie on F-lite and not the Reduceron itself.

Chapter 3

Generics

Generic programming is writing a function once and for all. As Hinze and Jones state in [7]: "a generic ... function is one that the programmer writes once, but which works over many different data types."

As is stated by Gibbons [4] and Jeuring et al. [9], there are many types of generic programming, which differ on what is made generic [4]:

- "parametrization by value"
- "parametrization by type"
- "parametrization by function"
- "parametrization by structure"
- "parametrization by property"
- "parametrization by stage"
- "parametrization by shape"

The definition of generic programming used in this Bachelor thesis is the notion parametrisation by shape, because generic function are defined over the structure of types.

In the following section generic programming will be explained, using the example of equality functions, this example is also used by Alimarine and Plasmeijer in [1]. In this section, the functional language Clean will be used for the examples.

3.1 Equality, a typical case for generic programming

The equality function is a very common function, which can be useful for any data type. So it can be very useful to have a class of equality functions:

```
class eq a :: a a -> Bool
```

This states that there is a class, `eq`, of functions. Each function in this class is initiated with one type and for each type a the function type is:

```
a a -> Bool
```

Therefore each equality function gets two instances of the same type as a parameter and returns a boolean as answer.

First the equality function for lists is shown. Lists could be defined as follows:

```
:: List a = Nil | Cons a (List a)
```

This declaration states that a list of type a is either an empty list, `Nil` or a node, `Cons`, with an element of type a and another list of type a, where type a can be substituted by any type. In the case of a `Cons` the element of type a is called the *head* of the list and the combined list is called the *tail*.

In general two lists are equal when at each position x the element of one list is equal to the element on position x of the other list. So two lists are equal when:

- the lists are both empty, or
- the head of both lists are equal and the tails are equal too.

In every other case the two lists are not equal.

This results in the following function definition:

```
instance eq (List a) | eq a
where
    eq Nil Nil                  = True
    eq (Cons x xs) (Cons y ys)  = eq x y && eq xs ys
    eq x y                      = False
```

This states that there is an `eq` function for elements of type `List a`, given there is an `eq` function for elements of type a. When, e.g. two lists of integers are compared, there needs to be a function to test if two integers are equal. Using, e.g., the predefined integer equality:

```
instance eq Int
where
    eq x y  = x == y
```

The same can be done for trees. The following is a declaration of a binary tree with elements of type a, which can be substituted by any other type.

```
:: Tree a   = Leaf | Node a (Tree a) (Tree a)
```

Given this declaration two trees are equal, when:

- both trees are just leaves, i.e. empty
- the first element of the trees are equal and the branches of both trees are equal too

Otherwise the trees are not equal.

This results in the following function:

```
instance eq (Tree a) | eq a
where
    eq Leaf Leaf                        = True
    eq (Node x l r) (Node y ll rr)      = eq x y && eq l ll &
       && eq r rr
    eq x y                              = False
```

It is even possible to have another tree definition, which is initiated by two types. One type for elements in the nodes and another one for the elements in the tips:

```
:: Tree2 a b    = Tip a | Bin b (Tree2 a b) (Tree2 a b)
```

Given this definition two trees are equal, when:

- both trees are tips and the elements of the tips are equal
- the node element of the trees are equal and the branches of both trees are equal too

Otherwise the trees are not equal.

This results in the following equality function:

```
instance eq (Tree2 a b) (Tree2 a b) | eq a & eq b
where
    eq (Tip x) (Tip y)              = eq x y
    eq (Bin x l r) (Bin y ll rr)    = eq x y && eq l ll &
       & eq r rr
    eq x y                          = False
```

When comparing the three given functions, it is easy to see they are quite alike. As Alimarine and Plasmeijer state in [1]: "All these instances have one thing in common: they check that the data constructors of both compared objects are the same and that all the arguments of these constructors are equal."

Generic programming makes use of this resemblance. Instead of writing similar functions for each data type, generic programming states that it is sufficient to write the function for some basic types and combinators, since all other data types can be reduced to these basic data types and combinators.

3.2 Basic data types and combinators

Bird, de Moor and Hoogendijk introduce in their article [2] three basic constructors/types, which can along with primitive types, define every other algebraic data type:

- 1, "[which] consists of just one member and serves as the source type for constants"
- $\times$, Cartesian product
- $+$, disjoint sum

Alimarine and Plasmeijer state that a generic representation of a data type can be computed out of the structure of a data type [1]. As given in [1], the following data type:

$$:: Ta_1...a_n = K_1 t_{11}...t_{1l_1} | ... | K_m t_{m1}...t_{ml_m}$$

can be regarded as:

$$T^\circ a_1...a_n = (t_{11} \times ... \times t_{1l_1}) + ... + (t_{m1} \times ... \times t_{ml_m})$$

When using this as a rewriting rule, the following generic representations are obtained for the data types defined above:

- List:

```
:: List a    = Nil | Cons a (List a)
List° a      = 1 + (a × (List a))
```

- Tree:

```
:: Tree a    = Leaf | Node a (Tree a) (Tree a)
Tree° a      = 1 + (a × (Tree a) × (Tree a))
```

- Tree2:

```
:: Tree2 a b     = Tip a | Bin b (Tree2 a b) (Tree2 a
b)
Tree2° a b       = a + (b × (Tree2 a b) × (Tree2 a b)
    )
```

Note the use of the unit constructor, 1. When one of the possible formats (the parts separated by |) of a data type, is just a constructor with no extra terms/arguments, like the first one in:

$$:: Ta = K_1 | K_2\ a$$

it will be empty in the generic representation using the rewriting rule from above. To fill this empty clause the unit constructor is used as a constant.

There is only one problem with this (theoretic) generic representation and that is the use of the symbols 1, + and ×, because those symbols are most likely already reserved in a programming language. In the follow-up of this chapter the representation given by Alimarine and Plasmeijer [1] will be used:

- `UNIT`, as a representation of the 1 type. The `UNIT` constructor is used as a constant as shown above for the 1 constant.
- `PAIR`, as a representation of the Cartesian product. The `PAIR` constructor is used with two parameters, which will be glued together as a *pair*.
- `EITHER`, as a representation of the disjoint sum. The `EITHER` constructor gets two parameters to be distinguished from one another, like the — symbol does in the declaration of the data type.

Using this representation will lead to the following generic representations:

- List:

```
:: List a    = Nil | Cons a (List a)
:: ListG a   = EITHER UNIT (PAIR a (List a))
```

- Tree:

```
:: Tree a    = Leaf | Node a (Tree a) (Tree a)
:: TreeG a   = EITHER UNIT (PAIR a (PAIR (Tree a) (
Tree a)))
```

- Tree2:

```
:: Tree2 a b    = Tip a | Bin b (Tree2 a b) (Tree2 a
b)
:: Tree2G a b   = EITHER a (PAIR b (PAIR (Tree a b) (
Tree a b)))
```

Note the difference in use of these constructors, the original theoretical symbols used infix notation, whereas these are used as constructors with parameters.

The original symbols are close to their actual mathematical relations, but to reason about the relation between a data type and its generic representation is much easier when using the names given by Alimarine and Plasmeijer. Just compare the two generic representations of lists and an explanation of them in words:

- `List° a = 1 + (a × (List a))`

  Hinze and Peyton Jones say the following about this generic representation of lists: "a *List* is a sum (+) of two types: a product (×) of the element type *a* and a *List*, and the unit type (1)."

- `:: ListG a = EITHER UNIT (PAIR a (List a))`

  A list of type *a* is *either* a *unit*, i.e. a `Nil`, or a *pair* of an element of type *a* and a list of type *a*.

The first one is more mathematical than the second one, where the link between original data type and generic representation is clear.

3.3 Conversion between data type and generic representation

So now it is clear how a generic representation is obtained, it is possible to make functions that convert between a data type and its generic representation.

Until now the generic representation was just another representation of the data type. However, a data type by itself is nothing, the instances of the data type are what count. For instance the data type `List Int` cannot be used by itself, it only declares that something is of that type, whereas `Nil`, can be used as an instance of `List Int` and can be used by functions. Therefore to use the generic representations given in the previous section, it is necessary to make instances of them. This can be done by using the generic constructors as the following types:

```
:: UNIT          = UNIT
:: PAIR a b      = PAIR a b
:: EITHER a b    = LEFT a | RIGHT b
```

The use of `UNIT` and `PAIR` is pretty straightforward, only the use of `EITHER` needs some explanation. The type `EITHER` makes a division between two parameters, it is either the one or the other, never both or none. To make this distinction the keywords `LEFT` and `RIGHT` are used.

The next few examples are instances of `List` and the generic representation of the instances. Recall the definition of `List` and the generic representation of the data type:

```
:: List a   = Nil | Cons a (List a)
:: ListG a  = EITHER UNIT (PAIR a (List a))
```

An instance of a list will either be an empty list (`Nil`) or a non-empty list (`Cons ...`), which are respectively `UNIT` and `PAIR...` in the generic representation, i.e. there is a one-to-one mapping between instances of `List` and `ListG`, its generic representation. This is shown in the following examples:

- The empty list []:

  ```
  Nil          // original
  LEFT UNIT    // generic
  ```

  `Nil` is the left option in the definition of list, so `LEFT` is used with its parameter `UNIT`

- The list [1]:

  ```
  Cons 1 Nil           // original
  RIGHT (PAIR 1 Nil)   // generic
  ```

  `Cons` is the right option in the definition of list, so `RIGHT` is used with an instance of its parameter `PAIR a (List a)`: `PAIR 1 Nil`.

- The list [1,2]:

  ```
  Cons 1 (Cons 2 Nil)           // original
  RIGHT (PAIR 1 (Cons 2 Nil)) // generic
  ```

  `Cons` is the right option in the definition of list, so `RIGHT` is used with an instance of its parameter `PAIR a (List a)`: `PAIR 1 (Cons 2 Nil)`.

Note that the list at the end of a `Cons` statement is not converted to its generic representation. The reason for this is lazy evaluation and will be explained later. For now it is enough to know that in the generic representation of a recursive data type, the recursive part can refer to the original data type and does not need to be transformed to a generic representation.

The examples above show how a `List` can be converted to a `ListG`. This results in the following functions:

- From original data type to generic:

  ```
  fromList :: (List a) -> (ListG a)
  fromList Nil             = LEFT UNIT
  fromList (Cons x xs)     = RIGHT (PAIR x xs)
  ```

- From generic to original data type:

  ```
  toList :: (ListG a) -> (List a)
  toList (LEFT UNIT)           = Nil
  toList (RIGHT (PAIR x xs))   = Cons x xs
  ```

The same can be done for `Tree` and `Tree2`. Recall the following definitions and generic representations:

- `Tree`:

```
:: Tree a   = Leaf | Node a (Tree a) (Tree a)
:: TreeG a  = EITHER UNIT (PAIR a (PAIR (Tree a) (
Tree a)))
```

- `Tree2`:

```
:: Tree2 a b    = Tip a | Bin b (Tree2 a b) (Tree2 a
b)
:: Tree2G a b   = EITHER a (PAIR b (PAIR (Tree a b) (
Tree a b)))
```

- `Tree`:
  - From original data type to generic:

```
fromTree :: (Tree a) -> (TreeG a)
fromTree Leaf           = LEFT UNIT
fromTree (Node x l r)   = RIGHT (PAIR x (PAIR l r
    ))
```

  - From generic to original data type:

```
toTree :: (TreeG a) -> (Tree a)
toTree (LEFT UNIT)                    = Leaf
toTree (RIGHT (PAIR x (PAIR l r)))    = Node x l r
```

- `Tree2`:
  - From original data type to generic:

```
fromTree2 :: (Tree2 a b) -> (Tree2G a b)
fromTree2 (Tip x)       = LEFT x
fromTree2 (Bin x l r)   = RIGHT (PAIR x (PAIR l r
    ))
```

  - From generic to original data type:

```
toTree2 :: (Tree2G a b) -> (Tree2 a b)
toTree2 (LEFT x)                      = Tip x
toTree2 (RIGHT (PAIR x (PAIR l r))) = Bin x l r
```

3.4 Making generic functions

In the previous section it is shown how to convert a data type to its generic representation in terms of `UNIT`, `PAIR` and `EITHER`. Because it is possible to convert any non-primitive data type to a generic representation, it is enough to define functions in terms of these generic types. In other words, it suffices to write a function for the types `UNIT`, `PAIR`, `EITHER` and any other primitive data

type that is used, to use that function for any other data type. The following functions are instances for the types `UNIT`, `PAIR` and `EITHER` of the equality function:

- `UNIT`

```
instance eq UNIT | eq a
where
    eq UNIT UNIT = True
```

- `PAIR`

```
instance eq (PAIR a b) | eq a & eq b
where
    eq (PAIR x y) (PAIR xx yy) = eq x xx && eq y yy
```

- `EITHER`

```
instance eq (EITHER a b) | eq a & eq b
where
    eq (LEFT x) (LEFT y)    = x && y
    eq (RIGHT x) (RIGHT y)  = x && y
    eq x y                  = False
```

Instead of writing an equality function for a data type, it is now possible to use the generic equality function. The function call for the generic equality function for the list and trees are as follows:

- `List`:

```
instance eq (List a) | eq a
where
    eq x y  = eq (fromList x) (fromList y)
```

- `Tree`:

```
instance eq (Tree a) | eq a
where
    eq x y  = eq (fromTree x) (fromTree y)
```

- `Tree2`:

```
instance eq (Tree2 a b) | eq a & eq b
where
    eq x y  = eq (fromTree2 x) (fromTree2 y)
```

As can be seen in the definitions, every time a function is called, its arguments are transformed to their generic representations. Therefore it is not necessary to transform recursive parts of a data type immediately to their generic representations. Because when those recursive parts are evaluated, the function called will make sure that they are transformed, so the generic functions will be used.

3.5 Mapping functions

This approach works well for the cases shown above, because all the functions work on types of the same *kind*. Before explaining what these are, the problem will be described with the `map` function as example.

The `map` function can be very useful for many different data types and are quite similar, just like equation functions as shown above:

- `List`:

```
mapList :: (a -> b) (List a) -> List b
mapList f Nil           = Nil
mapList f (Cons x xs)   = Cons (f x) (mapList f xs)
```

- `Tree2`:

```
mapTree2 :: (a -> c) (b -> d) (Tree2 a b) -> Tree2 c
    d
mapTree2 f g (Tip x)        = Tip (g x)
mapTree2 f g (Bin x l r)    = Bin (f x) (mapTree2 f g
     l) (mapTree2 f g r)
```

The examples are quite similar, but the approach described above will not work in this case. The reason for this is the functions have different arities. This is also explained in [24]: "map is datatype-generic in that many different data structures support a mapping operation. ... However, map is also arity-generic because it belongs to a family of related operations that differ in the number of arguments." The functions, like equality as explained above, are what Weirich and Casinghino call *datatype-generic*. But functions like the mapping functions are *arity-generic* as well. This results in a problem, because different instances of the same function cannot have different arities.

The difference between functions like mapping function and functions like equality functions, is that functions like `map` do not focus on the data types itself, but on the structure of the data types. For a mapping function only the structure of a data type, e.g. a list, is important, it does not matter what type of elements it has.

The difference becomes more clear when looking at the function headers of `mapList` and `eq` for lists:

```
eq :: (List a) (List a) -> Bool | eq a
mapList :: (a -> b) (List a) -> List b
```

The equality function works on lists of arbitrary elements, given there is an equality function for the data type of those elements. However this restriction is not present when using the mapping function for lists. This shows that mapping works on lists instead on lists with elements of a certain type. The functions have an extra abstraction, the data type of the elements is hidden for the function. Indeed an extra function to process the elements is given as an argument.

This is where *kinds* come into view. Because of the abstraction mentioned above, the mapping function does not work on types of the same kind. However, before proceeding this discussion, some information about the notion of kinds is needed.

3.5.1 Kinds

First of all there are primitive types, like integers, booleans and strings. There are also types, which use the primitive types to build a new type: algebraic data types, like lists and trees. These data types are nothing on their own, but need some primitives to exist. For example, a list on its own is quite abstract and cannot be used, but a list of integers is concrete and can be used. Therefore to use a list, it needs something to be *complete*. `List Int`, e.g., is a list of integers, but `List` itself is nothing, because it needs a type as an argument before it can be used.

Therefore like functions, algebraic data types, e.g. `List` and `Tree`, can have arguments. Whereas functions have a *type* to show what kind of arguments it has and what the result will be, types have *kinds*.

The details about kinds are omitted and for a more detailed description of kinds and how to derive them from types, will be referred to the article of Hinze [5]. To show the problem with respect to generic programming, the following information will do:

- Predefined data types, like `Int`, `Bool`, `String` ..., have kind `*`. The generic constructor `UNIT` has kind `*` as well.
- `List` and `Tree1` as defined above have kind `* -> *`. Note that, e.g. `(List Int)` has kind `*`, the type constructor `List` which is `* -> *` gets `Int` as an argument with kind `*`, so `(List Int)` has kind `*`.
- The generic constructors `PAIR` and `EITHER` have kind `* -> * -> *`.

3.5.2 Solution

The solution is not that difficult to understand. The problem is that it is not possible to define one function class for types of different kinds. Therefore a type class for every kind is created:

```
class map0 t :: t -> t
class map1 t :: (a -> b) (t a) -> t b
class map2 t :: (a -> b) (c -> d) (t a c) -> t b d
```

The following instances could be used as generic functions:

```
instance map0 UNIT
where
    map0 UNIT = UNIT

instance map2 PAIR
where
    map2 f g (PAIR x y) = PAIR (f x) (g y)

instance map2 EITHER
where
    map2 f g (LEFT x) = LEFT (f x)
    map2 f g (RIGHT x) = RIGHT (g x)
```

However, the function call of the generic functions is not as easy as before. For instance to make an instance for `List a`, because different functions are used:

- processing a `Nil` will result in `UNIT`. Therefore `map0` needs to be used.
- processing a `Cons x xs` will result in `PAIR x xs`. Therefore `map2` is used.

However, the functions that need to be used can be found in the generic representation of `List`:

```
EITHER UNIT (PAIR a (List a))
  |      |       |  |     |
map2 (map0) (map2 f (map1))
```

Therefore the instance for `List` will be as follows:

```
instance map1 List
where
    map1 f x = toList (map2 (map0) (map2 f (map1)) (
        fromList x))
```

This instance works as follows:

- `fromList` results in either `LEFT UNIT` or a `RIGHT (PAIR x xs)`:
  - in the case of `LEFT UNIT` the `EITHER` instance of the function `map2` is called, because it is a `LEFT`, the first function will be used to process the element, so `map0` is called with `UNIT`, which results in a `UNIT`.
  - in the case of `RIGHT (PAIR x xs)` the `RIGHT` case of the `EITHER` instance of `map2` is called, with f as function to process the element x and `map1` is used to process the tail of the list, which results in a recursive call to the `List` instance.

This solution works well for the mapping functions, but the equality function has to be revised as well:

```
class eq0 t :: t t -> Bool
class eq1 t :: (a a -> Bool) (t a) (t a) -> Bool
class eq2 t :: (a a -> Bool) (b b -> Bool) (t a b) (t a b
    ) -> Bool

instance eq0 Int
where
    eq0 x y = x == y

instance eq0 UNIT
where
    eq0 UNIT UNIT = True

instance eq2 PAIR
where
    eq2 eqa eqb (PAIR x1 y1) (PAIR x2 y2) = (eqa x1 x2) &
        & (eqb y1 y2)
```

```
instance eq2 EITHER
where
    eq2 eqa eqb (LEFT x) (LEFT y) = eqa x y
    eq2 eqa eqb (RIGHT x) (RIGHT y) = eqb x y
    eq2 _ _ _ _ = False


instance eq1 List
where
    eq1 eqa l1 l2 = eq2 (eq0) (eq2 eqa eq1) (fromList l1)
        (fromList l2)

instance eq0 (List a) | eq0 a
where
    eq0 l1 l2 = eq1 eq0 l1 l2
```

3.6 The advantages of generic programming

The great advantage of generic programming is that a function only needs to be available for the generic types and any primitive data type that is used. As shown above there has to be a generic representation of the other data types, conversion functions and instances of the needed function for each data type, which calls the conversion functions and the generic functions. However those are all so straightforward and systematic that this can be done by a computer. As is shown in the next two chapter, generic programming is already implemented in Clean and Haskell, where the conversions between representations are automated.

Chapter 4

Generics in Clean

The implementation of generics in Clean is described by Alimarine and Plasmeijer in [1]. The notation used in Clean is already been introduced in the previous chapter. As said in the previous chapter, much work can be done by the compiler when using generics:

- create the generic representation of a given data type
- create the corresponding conversion functions
- create the instance of the given data type for the function needed

The programmer only needs to deliver the following:

- instances of the function for the generic types: `UNIT`, `PAIR` and `EITHER`
- the definition of the new data type
- a request for using generics

Alimarine and Plasmeijer give the following definition of a generic equality function in Clean:

```
generic eq t                                    :: t t ->
    Bool
instance eq Int where
    eq x y                                      = eqInt x y
instance eq UNIT where
    eq x y                                      = True
instance eq PAIR where
    eq eqx eqy (PAIR x1 y1) (PAIR x2 y2)        = eqx x2 &&
        eqy y1 y2
instance eq EITHER where
    eq eql eqr (LEFT x) (LEFT y)                = eql x y
    eq eql eqr (RIGHT x) (RIGHT y)              = eqr x y
    eq eql eqr x y                              = False

instance eq List generic
instance eq Tree generic
```

Note that in the case of generics a class of functions is not created with the `class` keyword (as is shown at the beginning of the previous chapter), but with the keyword `generic` instead. Looking at the example, it is also clear that the implementation supports types of different kinds, because it uses the solution presented in the previous chapter.

The last two statements of the example request the creation of the generic equality function for lists and trees. Such a statement yields the following operations as given in [1]:

1. "Create the class g_k for the kind k of the data type T, if not already created. The instance on T becomes an instance of that class."

2. "Build the generic representation T° for the type T. Also build the conversion functions between T and T°."

3. "Build the specialization of g to the generic representation T°."

4. "Generate an adaptor that converts the function for T° into the function for T."

5. "Build the specialization to the type T. It uses the specialization to T° and the adaptor. The instance on g_k on T is the specialization of the generic function g to the type T."

However the syntax in the paper written by Alimarine and Plasmeijer [1] has been changed a lot. Look at the difference between the generic mapping function in [1]:

```
generic map a1 a2                   :: a1 -> a2
instance map Int where
    map x                           = x
instance map UNIT where
    map x                           = x
instance map PAIR where
    map mapx mapy (PAIR x y)        = PAIR (mapx x) (mapy y)
instance map EITHER where
    map mapl mapr (LEFT x)          = LEFT (mapl x)
    map mapl mapr (RIGHT x)         = RIGHT (mapr x)

instance map List generic
instance map Tree generic
```

and the generic mapping function as defined by Hinze et al. in [6]:

```
generic map a b :: a -> b
map{|UNIT|}      x                          = x
map{|Int|}       i                          = i
map{|Char|}      c                          = c
map{|EITHER|}    mapa mapb (LEFT x )        = LEFT (mapa x )
map{|EITHER|}    mapa mapb (RIGHT y)        = RIGHT (mapb y)
map{|PAIR|}      mapa mapb (PAIR x1 x2)     = PAIR (mapa x1)
    (mapb x2)
map{|CONS|}      mapa       (CONS x )       = CONS (mapa x)
```

```
map{|FIELD|}    mapa    (FIELD x )    = FIELD (mapa x)
map{|OBJECT|}   mapa    (OBJECT x )   = OBJECT (mapa x)
```

This syntax is also the one that is currently implemented in the Clean compiler. The reason the "old" syntax is shown, is that it is more related to the syntax given in the previous chapter, with the generics done by hand.

The new syntax even has more generic types: `CONS`, `FIELD` and `OBJECT`. `CONS` is used to give extra information about the constructors of the original types. In the case of print and parse functions, the functions need more information than just the structure of the types. For example when printing an instance of some type to a file, to save it for later use. When parsing this file, the parser needs to know what kind of data it needs to process, i.e. it needs to know which instance of the parser it has to use. Therefore the `CONS` type is added to show these functions which type it original was, so the compiler can use this information to choose the parses for the right type.

The use of `FIELD` is similar to `CONS`, but instead of information about constructors, it gives information about record types and `OBJECT` gives the necessary information about objects.

The following example shows why `Cons` is necessary in some cases. When printing lists and trees to a file using generic functions, the following can be printed to the file:

```
LEFT UNIT
```

As is shown in the previous chapter, this can either be a `Nil` or a `Leaf`, but given only this information the compiler cannot make a distinction between both, so the compiler needs more information:

```
LEFT (CONS "Tree" UNIT)
```

This gives the compiler the original type in a string, and a parser can use this string with pattern matching to get the right instance of the parse function.

4.1 Application with generics in Clean

Functional programming is becoming more popular, but still the main usage of functional programming languages is research. However, generics in Clean go beyond the interest of research. Indeed several applications in Clean use generics.

4.1.1 iTasks

An application that makes use of a lot of generics is the `iTasks` system as described by Plasmeijer, Achten and Koopman in [19] and [20]. `iTasks` is a work-flow system and is build upon `iData`, a domain specific language embedded in Clean. "Work flow systems are automated systems in which tasks are coordinated that have to be executed by humans and computers." [19]. Because it is implemented in a functional language, the `iTasks` system offers more features than other work flow systems, e.g. more combinators, but because it uses Clean it is also strong typed. "Compared with ... commercial systems, the iTask system offers several further advantages: tasks are statically typed, tasks can be higher order, the combinators are fully compositional, dynamic and

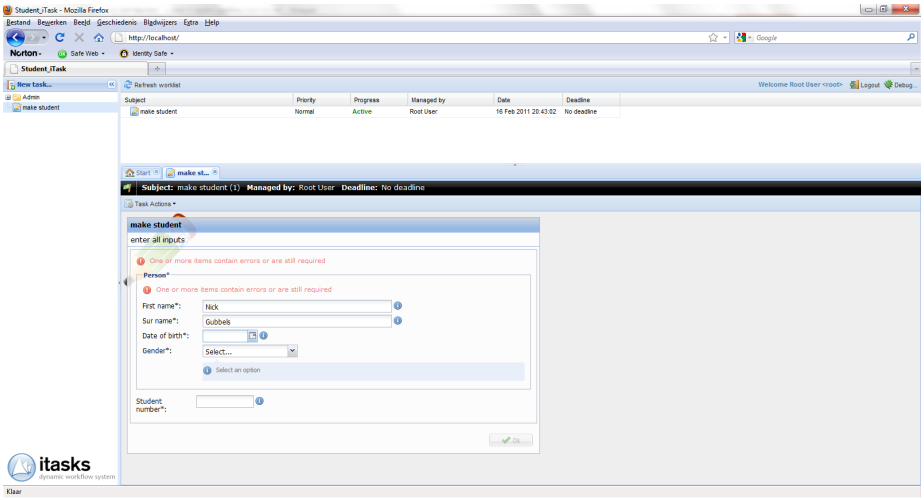
recursive work flows can be specified, and last but not least, the specification is used to generate an executable web-based multi-user work flow application" [20]. As said in [12]:"The iTask combinator language is designed for declarative specification of workflows. This means that the specifications describe what has to be done, not how."

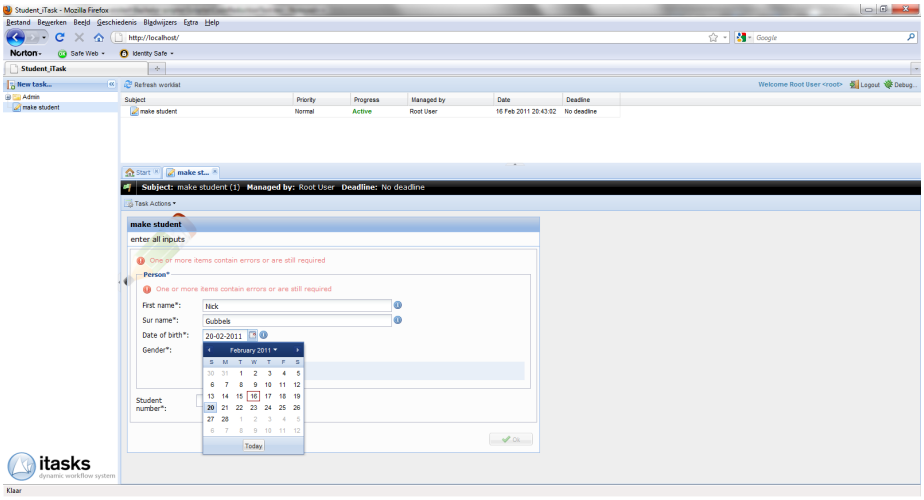
In that article [12] Lijnse and Plasmeijer introduce a new version of `iTasks` where a new user interface has been added as is shown in the figures below. This interface can be used to run the tasks defined in the work flow.

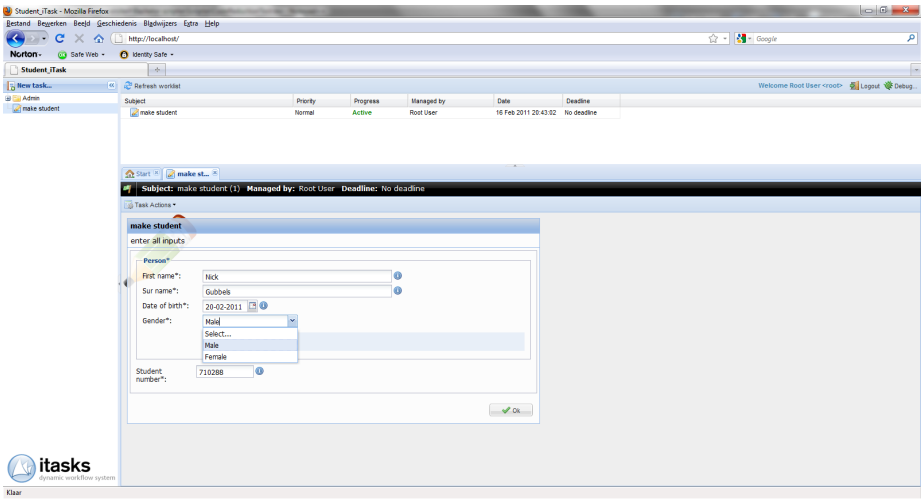
As said before `iTasks` also uses generics, however not just for the implementation of the system itself, but also for setting up a work flow in the system. The following code creates an entry of the type `student` in the system, which is user defined and all the needed function are derived using generics:

```
module Student_iTask

import iTasks

Start :: *World -> *World
Start world = startEngine [workflow "make student"
    studentTask] world

studentTask :: Task Student
studentTask = enterInformation "student" "enter all
    inputs"

:: Student =    { person :: Person
                , student_number :: Int
                }

:: Person = { firstName :: String
            , surName :: String
            , dateOfBirth :: Date
            , gender :: Gender
            }

:: Gender = Male | Female

derive class iTask Student, Person, Gender
derive bimap Maybe, (,)
```

This results in the following input fields in the system:


Student_iTask - Mozilla Firefox
make student
enter all inputs
First name*: Nick
Sur name*: Gubbels
Date of birth*:
Gender*: Select...
Student number*:
itasks


Student_iTask - Mozilla Firefox
make student
enter all inputs
First name*: Nick
Sur name*: Gubbels
Date of birth*: 20-02-2011
Gender*:
Student number*:
itasks


Student_iTask - Mozilla Firefox
make student
enter all inputs
First name*: Nick
Sur name*: Gubbels
Date of birth*: 20-02-2011
Gender*: Male
Select...
Male
Female
Student number*: 710288
itasks

By just entering the type `student` the system knows kind of input values it needs. As can be seen in the screenshots above.

4.1.2 G∀ST

G∀ST is a testing system embedded in Clean as is explained in the article of Koopman et al. [10]. It can be used for *functional testing*, which includes the following:

- "formulation of a property to be obeyed: what has to be tested"
- "generation of test data: the decision for which input values the property should be examined"
- "test execution: running the program with the generated test data"
- "test result analysis: making a verdict based on the results of the test execution"

Three of these operations can be done automatically by the G∀ST system. At the point of generating test data, generics are used. The so called `ggen` function is a generic function and can be derived for any type.

The two applications `iTasks` and G∀ST can even be combined, as is shown by Koopman, Plasmeijer and Achten in their article about the semantics is `iTasks` [11]. The semantics of `iTasks` were implemented in Clean and tested for some properties with the G∀ST system. They even found a counter example for a property of which the authors thought it would hold, but G∀ST proved this was not the case. This shows that the implementation can become so complex that some properties do not hold although it may seem that they do.

Chapter 5

Generics in Haskell

As there are many different Haskell compilers, there are also many different approaches to generic programming in Haskell. In the following section some of those approaches will be discussed:

- Generic H∀SKELL, as described by Clarke and Löh in [3]
- PolyP and PolyLib, as described respectively by Jansson and Jeuring in [8] and Norell and Jansson in [18]
- Template Haskell, as described by Sheard and Peyton Jones in [23] and Norell and Jansson [17]

5.1 Generic H∀SKELL

Generic H∀SKELL is an extension of Haskell, which provides Haskell with generic programming and is described by Clarke and Löh in [3]. It uses the following types to create the generic representations:

```
data a :+: b    =   Inl a | Inl b
data a :*: b    =   a :*: b
data Unit       =   Unit
data Con a      =   Con ConDescr a
data Lab a      =   Lab LabDescr a
```

These types include the three types given by Bird, de Moor and Hoogendijk in their article [2]:

- `:+:` is the sum of two types, which is + in [2]
- `:*:` is the Cartesian product of two types, which is represented as × in [2]
- `Unit` is the constant type, which is 1 in [2]

However, in Generic H∀SKELL there are two extra generic types: `Con` and `Lab`.

`Con` gives information about the constructors of the type. This is needed,e.g. for printing/parsing functions as is explained before in the case of the `Cons` type of Clean.

`Lab` is a label for elements inside the type. Whereas `Con` specifies information about the constructors of the type, `Lab` specifies information about the elements inside the type.

The authors give in [3] a `Tree` definition as an example to show how their generic representation is used:

```
data Tree a = Leaf
            | Node{ref :: a, left :: Tree a, right ::
              Tree a}
```

This results in the following generic representation:

```
type Tree' a =  Con Unit
             :+: Con (Lab a :*: Lab(Tree a) :*: Lab (Tree
                 a))
```

The following code represents a generic mapping function [3]:

```
gmap<Unit>                          = id
gmap<:+:> gmapA gmapB (Inl a)       = Inl (gmapA a)
gmap<:+:> gmapA gmapB (Inr b)       = Inr (gmapB b)
gmap<:*:> gmapA gmapB (a :*: b)     = (gmapA a) :*: (gmapB b)
gmap<Con c> gmapA (Con _ a)         = Con c (gmapA a)
gmap<Lab l> gmapA (Lab _ a)         = Lab l (gmapA a)
```

Unlike Clean, the creation of instances for other types is implicit. Whereas Clean uses a `derive` statement, Generic H∀SKELL creates instances for the other types, when the instance is used:

```
gmap<Tree> (+1) t
```

In this case (given in [3] but slightly adapted), the instance for `Tree` is created and gmap is used with an increment function over `t`.

The Generic H∀SKELL compiler does the following, as is described by Clarke and Löh in [3]:

1. "translating generic definition into Haskell code". The authors give the following example as a translation of the `:+:` ans `Con` cases of the `gmap` function.

   ```
   gmap__Sum :: (a -> c) -> (b -> d) -> a :+: b -> c :+:
       d
   gmap__Sum gmapA gmapB (Inl a)    = Inl (gmapA a)
   gmap__Sum gmapA gmapB (Inr b)    = Inl (gmapB b)
   gmap__Con :: ConDescr -> (a -> b) -> Con a -> Con b
   gmap__Con c gmapA (Con _ a)      = Con c (gmapA a)
   ```

2. "translating calls to generic functions into an appropriate Haskell expression". The statements are as followed rewritten:

   ```
   poly<F A> = poly<F> (poly<A>)
   ```

   where `poly` is a generic defined function. The following example is given by the authors:

   ```
   gmap<Either [String]>
   ```

is transformed into

gmap<**Either**> (gmap<[]> (gmap<**String**>))

After this these functions are renamed to for example `gmap__Either`.

3. "specialising generic entities to the types at which they are applied" This process is split into two parts:

   (a) "the generic function is specialised to the named type's structure type" For example when the `gmap` function for `Tree` needs to be generated, first the function for its generic representation, `Tree‘` is created. As given by the authors:

   ```
   gmap__Tree‘ :: (a -> b) -> Tree‘ a -> Tree‘ b
   gmap__Tree‘ a =
       gmap__Con a
           ‘gmap__:+:’
       gmap__Con (gmap__Lab a ‘gmap__:*:’ gmap__Lab
           (gmap__Tree a)
                       ‘gmap__:*:’ gmap__Lab (
                           gmap__Tree a))
   ```

   (b) "the resulting specialised function is converted using a generic wrapping to a specialisation for the original named type" After the case for the generic representation, the instance of the original type is created. Like the following given by Clarke and Löh:

   ```
   gmap__Tree :: (a -> b) -> Tree a -> Tree b
   gmap__Tree a = bimap__Expr (gmap__Tree‘ a)
   ```

   The `bimap` function takes care of the the conversion between `Tree‘` and `Tree`.

5.2 PolyP

In the Haskell extension *PolyP*, designed by Jansson and Jeuring [8], polytypic, i.e. generic, functions can be created. Using the `polytypic` construct the PolyP compiler will generate code and translate PolyP to Haskell, i.e. to run PolyP code it has to be compiled/translated to Haskell and then using a Haskell compiler to run the code. It uses the following grammar for functors, which are similar to generic representations as shown before:

$$F ::= f|F + F|F \times F|()|Par|Rec|D@F|Con\ \tau$$

The sum and the Cartesian product are the same as described before for the generic representations. () is the empty statement, which is like the 1 or `UNIT` in the generic representations as shown before. *Par* returns the data type, which was given as a parameter, e.g. when using a list of integers, `par` will return the type `Int`. *Rec* is a recursion of the type. $D@F$ is used to define a data type in terms of another user-defined type and $Con\tau$ is used for constants.

List is represented in PolyP as follows [8]

```
FList = () + Par + Rec
```

The following is an example of generics in PolyP [8]:

```
polytypic fl :: f a [a] -> [a] =
  case f of
    g + h -> either fl fl
    g * h -> \(x,y) -> fl x ++ fl y
    ()    -> \x -> []
    Par   -> \x -> [x]
    Rec   -> \x -> x
    d @ g -> concat . flatten . pmap fl
    Con t -> \x -> []
```

However according to [18] PolyP has a new version called PolyLib with the following notations of the generic types:

```
data (g :+: h) p r      = InL (g p r)
                        | InR (h p r)
data (g :*: h) p r      = g p r :*: h p r
data Empty p r          = Empty
newtype Par p r         = Par {unPar :: p}
newtype Rec p r         = Rec {unRec :: r}
newtype (d :@: g) p r   = Comp {unComp :: d (g p r)}
newtype Const t p r     = Const {unConst :: t}
```

Unlike PolyP, PolyLib is a library extension of Haskell instead of a compiler, so it can be used with ordinary Haskell.

As stated by Norell and Jansson in [18] the compilation is as follows: "the compilation of a PolyP program consists of three phases. ... In the first phase ... the pattern functor [i.e. the generic representation] of each regular data type is computed and an instance of the class `FunctorOf` is generated, relating the datatype to its functor. The second phase ... deals with the polytypic definitions. For every polytypic function a type class is generated and each branch in the type class is translated to an instance of this class. The third phase ... consists of inferring the class constraints introduced by our new classes."

5.3 Template Haskell

Template Haskell as described by Sheard and Peyton Jones in [23] is an extension of Haskell in the form of a library and it extents Haskell with "compile-time meta-programming" and it is implemented in the Glasgow Haskell Compiler. It adds many new features to Haskell.

Norell and Jansson describe in their article [17] how they use the Template Haskell libraries to implement both Generic H∀SKELL and PolyP. The difference with their original implementations, except for PolyLib, is that Template Haskell works with libraries instead of new compilers. Indeed "with this approach, generic functions are written in Haskell (with the Template Haskell extension), so there is no need for an external tool."

Chapter 6

Comparing generic approaches

Using the comparisons by Hinze et al. in [6] and Rodriguez et al. in [22], a comparison is made between the described implementations of generic programming:

- Clean
- Generic H∀SKELL
- PolyP/PolyLib

Template Haskell is also described in the previous chapter. However, Template Haskell itself is not a generic programming approach, so this will not be considered in the comparison. However, Hinze et al. use Template Haskell in their comparison, but this is DrIFT (a generic programming implementation for Haskell) implemented with Template Haskell.

6.1 Comparison by Hinze et al. [6]

Hinze et al. compare a set of implementations of generic programming in Haskell, however the implementation of generics in Clean is also considered, because "Clean is not Haskell, but it is sufficiently close to be listed here". The following implementations are considered:

- Generic H∀SKELL
- Clean
- PolyP
- Scrap Your Boilerplate
- DrIFT
- Template Haskell as implementation of DrIFT
- Lightweight Implementation of Generics and Dynamics

- Derivable Type Classes
- Generics for the Masses

The authors use the following criteria to test the different implementations [6]:

- "*structure*", e.g. are polymorphic functions possible.
- "*type completeness principle*": "the Type Completeness Principle says that no programming-language operation should be arbitrarily restricted in the types of its operands, or, equivalently, all programming-language operation should be applicable to all operands for which they make sense." This leads to criteria like *full reflexivity*: "a generic function can be used on any type that is definable in the language".
- "*well-typed expressions do not go wrong*", i.e. is the generic system type-safe.
- "*information in types*", e.g. do the types of the generic functions corresponds to intuition.
- "*integration with the underlying programming language*"
- "*tools*", are there any tools available for the implementation, e.g. a compiler or library? Is there code optimisation?

The following table shows the results of the comparison of Hinze et al. with respect to the implementations mentioned in previous chapters.

	Structure	Completeness	Safe	Info	Integration	Tools
GH	++	+	++	++	++	+
Clean	o	+	++	++	++	+
PolyP	o	-	+	+	+	-

According to this comparison Generic H∀SKELL and Clean have similar results. Generic H∀SKELL scores better for the structure criteria. This is due to the following: "Clean supports the definition of generic functions in the style of Generic Haskell. [However] it does not support type-indexed data types."

However, PolyP scores a lot lower than Generic H∀SKELL and Clean. Some of the reasons are:

- "PolyP is not fully reflexive: polytypic [i.e. generic] functions can only be used on regular data types of kind $* \to *$." (Completeness)
- The integration with ordinary Haskell is not without consequences. "PolyP1 and the optional compiler of PolyP2 do not know about classes, or types of kind other than $* \to *$, and lack several syntactic constructions that are common in Haskell, such as `where` clauses and operator sections." However "the Haskell library part of PolyP2 integrates seamlessly with Haskell." This library part is also known as PolyLib.

6.2 Comparison by Rodriguez et al. [22]

Rodriguez et al. compare many generic approaches of Haskell, but they only consider one of the implementations of the previous chapters: PolyLib. Therefore the details about the testing/comparing criteria are omitted and for more information is referred to [22].

However the authors give a conclusion about PolyLib which is useful for this comparison. "The library is limited to regular datatypes (with one parameter) so the supported universe is relatively small." Their overall conclusion is that PolyLib is not very suitable as a generic programming extension of Haskell, but was included as a "classic reference" and because of its expressiveness.

6.3 Concluding the comparison

Both of the comparisons made by Hinze et al. [6] and by Rodriguez et al. [22] conclude that PolyP/PolyLib is quite limited because it can handle only a small set of data types. However, Generic H∀SKELL and Clean score a lot better and score quite similar.

Chapter 7

Using generics in F-lite

As shown in many papers (like [2], [7], [1] and many others) non-basic data types in a functional language can be represented by the basic data types and some generic types:

- 1 or `UNIT`, which represents types with no value, like constructors.
- $\times$ or `PAIR`, which represents a pair of two data types.
- $+$ or `EITHER`, which is used to show two different cases. It gets two other data types as parameter and in every case exactly one of them holds: it is either data type a or data type b. To distinguish between the two types, as instantiation of this type either a `LEFT` or `RIGHT` is used with one parameter. So `EITHER INT CHAR` can be instantiated by either a `LEFT INT` or a `RIGHT CHAR`.

As a starting point for making a generic approach for F-lite, some F-lite code is provided in which an equality function for Trees is defined based on the generic types.

To do this it is useful to take note of the type of the tree that are going to be used. In this case `Tree a` (a tree with elements of type a):

```
Leaf | Bin a (Tree a) (Tree a)
```

This definition shows that a `Tree` of type a is either a `Leaf` or a `Bin` with an element of type a and two trees as children. When looking at the definition you can see some correspondences with the generic types. The `|` distinguishes two cases, a leaf of a bin, so this indicates the use of the `EITHER` type. Which yields the following semi-generic representation:

```
EITHER Leaf (Bin a (Tree a) (Tree a))
```

The `Leaf` element is just a constructor, so it has the type `UNIT`:

```
EITHER UNIT (Bin a (Tree a) (Tree a))
```

In the case of `Bin` you can see three types together, so we use two `PAIR`'s to glue those together, this can be done in two ways:

```
EITHER UNIT (PAIR a (PAIR (Tree a) (Tree a)))
EITHER UNIT (PAIR (PAIR a (Tree a)) (Tree a))
```

The choice is not important, it is just a matter of notation, so we will use the first one. Note that in this case it is not necessary to make a `UNIT` type for the `Bin` constructor, because constructors are used to distinguish the different cases and this is already done by the `EITHER`. In the case of the `Leaf` it was necessary because it needs an argument for each case and there was nothing else besides the constructor, but in the case of the `Bin` there were an element and two recursions, so those can represent this case without the use of the `UNIT` type.

All that is left are the elements of type a and the two recursions. If the element is a basic data type or a generic representation of another data type, it can stay there unchanged, if it is not it is necessary to make a generic representation of that as well. In the case of the recursions, it has to be changed to be a recursion of the generic representation, say `GenTree`:

```
EITHER UNIT (PAIR a (PAIR (GenTree a) (GenTree a)))
```

This is only the case when a tree expression is fully evaluated, but when using lazy functions the following is the case:

```
EITHER UNIT (PAIR a (PAIR (Tree a) (Tree a)))
```

The recursions will be transformed when they are evaluated.

To use this generic representation in the program we need a function which rewrites a tree to its generic representation and a function which rewrites the generic representation to a tree. First we focus on the function from a tree to the generic representation. As we already have seen, a tree has two cases: either a leaf or a bin, so the function has two cases as well:

```
fromTree Leaf = LEFT UNIT
fromTree (Bin x l r) = RIGHT (PAIR x (PAIR l r))
```

Because the output of the function is an instance of the generic representation, `LEFT` and `RIGHT` are used instead of `EITHER`. As you can see this function uses the lazy representation.

The function from the generic representation to a tree, is quite alike, it is the same but the other way round:

```
toTree (LEFT UNIT) = Leaf;
toTree (RIGHT(PAIR x (PAIR l r))) = Bin x l r;
```

In the following example a generic instance of an equality function is shown. In order to be able to use the generic representation of a tree in the equality function, there have to be at least the instances for the generic types:

- Two `UNIT`s are always equal.
- Two `PAIR`s are equal if the elements of the one are equal to the elements of the other (in the same order).
- Two `LEFT`s (or two `RIGHT`s) are equal if there element is equal and a `LEFT` is always different from a `RIGHT`.

Therefore:

```
eq UNIT UNIT = True
eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b)
eq (LEFT x) (LEFT y) = eq x y
eq (RIGHT x) (RIGHT y) = eq x y
eq (LEFT x) (RIGHT y) = False
eq (RIGHT x) (LEFT y) = False
```

Besides that it is necessary to have an instance for every basic type that is used in the generic representation. In the example shown below a tree of integers is used, so there has to be an instance for integers as well (using the already defined (==) for integers):

```
eq (Int x) (Int y) = (==) x y
```

Then the only thing left to do is the equality instance for trees using the equality functions for the generic types.

```
eq (Tree x) (Tree y) = eq (fromTree x) (fromTree y)
```

Using the following as example tree:

```
Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree Leaf))) (Tree Leaf))
```

7.1 Corresponding G-lite code

Because the creating of the new instances for a function and the function from and to the generic representation is very straightforward, so it can be done by a compiler.

But the compiler will need some information:

- For which data type?
- For which function?
- What is the type of that function?

This might seem a little strange, because there are no explicit types in F-lite, but they are there implicitly through the use of constructors. However, for the creation of the generic representation of some type it is necessary to know the type and not just its constructors.

So for our example we need the following statements:

```
Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic eq -> (Tree a) (Tree a) -> Bool
```

Chapter 8

G-lite compiler

As said before many operations in generic programming can be done automatically by a computer. As shown in the previous chapter F-lite can be extended with generic programming: G-lite. A prototype G-lite compiler has been written, which does all the automatic operations and translates the G-lite code to F-lite code, so it can be used on the Reduceron or in an F-lite compiler or interpreter.

The G-lite compiler is written in Python. The choice for this language is based on a few reasons:

- Python has easy read and write operations to files, which is more difficult in functional programming languages, like Clean.
- Python comes with very useful functions to parse and rewrite a string. For example the *split* function is very useful, which is explained later in this chapter, along with the other used functions.
- Python is generally imperative but makes use of functional language elements like lists, list comprehensions and the slicing of lists. The combination of both can be very useful, like looping over a list comprehension.

In the next few section the different parts of the compiler are explained.

8.1 General program

In this section the main function of the program is explained. The main program can be split into a few different parts:

1. File opening/creation: in this part the user is asked to give the file names of the input and output files. The input file will be the G-lite program and the output will be the corresponding F-lite program. The given files are then opened for reading/writing.
2. Parsing the input file: in this part the input file is parsed and corresponding functions are called.
3. Writing the output file: in this part the output file is written, based on the information gained by the parser.

8.1.1 File opening/creation

In this part the G-lite file will be opened and the F-lite file will be created. To be able to do this, the user has to give the program the file names for the input and output files. This can be done in two ways:

1. giving the file names as an argument to the program.

   In Python this can be done, by using the `sys` environment:

   ```
   import sys
   ```

   The `sys` environment provides a list, which contains the system arguments, in other words the arguments passed on to the program. The first element of the list contains some information on how the Python interpreter is used in this program call, but does not provide any useful information for us. However, the other elements of the list do. They contain the arguments passed on to the program call. Therefore if the first argument is the file name of the input file and the second argument is the file name of the output file, the following will store those file names in the variable `input`:

   ```
   input = sys.argv[1]
   output = sys.argv[2]
   ```

2. start the program and give the file names when the program asks for it.

   This method uses the console screen to interact with the user and asks the user for keyboard input. In Python this can be done with the `raw_input` function. This is some sort of a 2-in-1 function, whereas is prints the given string as text on the console and waits for some keyboard input from the user, so it prints output and gets input. To get both input and output file names, the following statements will suffice:

   ```
   input = raw_input('Please give input file: ')
   output = raw_input('Please give output file: ')
   ```

Because there are two possible ways of providing the needed information to the program, the program needs to distinguish two different possible actions. This is done by first testing if there are system arguments present, if they are use them as file names, if there is no system argument, then prompt for the file names. To realise this, the exceptions of Python are used:

```
try:
    input = sys.argv[1]
except IndexError:
    input = raw_input('Please give input file: ')
try:
    output = sys.argv[2]
except IndexError:
    output = raw_input('Please give output file: ')
```

The idea behind this is, try to access the `sys.argv`, if there is no element on that index of the list, the Python interpreter will raise an `IndexError`. The

meaning of this error is that the program tried to access something out of the bounds of the list, so there is no element present. The program then catches this error, and prompts the user for the right information.

Then both input and output files need to be opened. In Python this can be done by using the `open` function. The first argument of this function is the name/path of the file to be opened, whereas the second argument is the mode, e.g. read (`r`) and write (`w`).

```
input = open(input, 'r')
output = open(output, 'w')
```

8.1.2 Parsing the input file

An F-lite program is simply just a set of functions. So when parsing the G-lite the result has to be a set of functions, so there has to be some data type to store the functions. This will be done with a list.

```
function = []
```

This definition is not very detailed, but Python does not need to know in advance what the elements of a list will be, it only needs to know this variable is a list. However, the list will be used in the following way: it will be a list of function definitions. Those function definitions will be a list of:

- one header, i.e. the name of the function
- a list of arguments
- a list of body statements

For example the following function definition:

```
eq (Int x) (Int y) = (==) x y;
```

will be as follows in the list of functions:

```
[['eq',['(Int x)','(Int y)'],['(==) x y']]]
```

Looking at a G-lite program, there are three possible kinds of statements:

- a function, e.g.:

  ```
  eq (Int x) (Int y) = (==) x y;
  ```

- a type definition, e.g.:

  ```
  Tree a :: Leaf | Bin a (Tree a) (Tree a)
  ```

- a generic statement, e.g.:

  ```
  Generic eq (Tree a) (Tree a) -> Bool
  ```

Those three kinds of statements have all one thing that can be used to discriminate one from another:

- Function: `=`

- Type definition: `::`
- Generic: `Generic`

For each type of statement there is a corresponding function, which returns a (list of) function(s) and will be added to the list of functions which are already stored. These corresponding functions will be called after one of the keywords shown above is in one of the lines of the input:

```
for line in input:
    line = line.strip()
    if line > '':
        if '=' in line:
            functions.append(function(line))
        if '::' in line:
            functions.extend(typedef(line))
        if 'Generic' in line:
            functions.append(createInstance(line.replace(
                'Generic ','')))
```

This code states the following: for each line in the input file:

1. remove the whitespace at the beginning and end of the line, i.e. strip the line
2. check if `=`, `::` or `Generic` can be found in the line
3. if so pass the line on to the corresponding function. This function returns the function(s) to be added to the list of functions. This is either done with the `append` or the `extend` function, which are primitive Python functions. The difference between both is that append puts its argument at the end of the given list, and extend puts the elements of the argument, which is also a list, in the list of functions. Like in the following examples:

```
[1].append(2) = [1,2]
[1].append([2,3]) = [1,[2,3]]
[1].extend([2,3]) = [1,2,3]
```

After all the lines of the input files are read, the created list of functions will be sorted, because all the instances of a function has to follow each other in F-lite.

```
functions.sort()
```

8.1.3 Writing the F-lite output

Writing the F-lite file is not that difficult when all the functions are available in a list. Just start the file with `{`, then write every entry of the list on a new line and end the file with `}`. The list of functions, is actually a list of lists, where the latter consists of a header, a list of arguments and a list of body statements. Very useful in Python is that in a for-loop the inner list can be unpacked into three variables.

```
print >> output, '{'
for (fname, farg, fbody) in functions:
    line = fname
    for a in farg:
        line += ' ' + a
    line += ' ='
    for b in fbody:
        line += ' ' + b
    line += ';'
    print >> output, line
print >> output, '}'
```

8.2 Function parsing

In this section the parsing of functions will be explained. As shown in the section above, this function will be called if the = symbol is found in the current line. The only argument of the function is the line to be parsed. The parsing is split in three parts:

- parsing the function name
- parsing the arguments
- parsing the body

8.2.1 Parsing the function name

The function name is the first *word* in the function definition and can be found before the first space (' ') in the line. So the line is split into several parts, which are separated from each other by spaces. To do this, the primitive Python function split is used. It is an operation on strings and returns a list of elements in the string which are *split* from the delimiter. For example:

```
'eq (Int x) (Int y) = (==) x y;'.split(' ') =
    ['eq', '(Int x)', '(Int y)', '=', '(==)', 'x', 'y;']
```

Therefore to get the function name, the split function is used and the first element of the list will be the function name:

```
functionName = line.split(' ')[0]
```

where line is the current line to be parsed.

8.2.2 Parsing the arguments

The arguments of a function are all elements before the = symbol, except for the very first element, because that is the function name:

```
line.split('=')[0].split(' ')[1:]
```

This states that `line` is split with = as the delimiter. This results in a list of two items, the left and right parts beside the = symbol. The left one is needed, so the first element is selected with `[0]`. This element is then split with spaces as the

delimiter, to remove the function name. Out of the result all the elements are selected except the very first, i.e. the second element until the end of the list: `[1:]`. The result of this statement is a list with the arguments of the function as elements.

However it is not totally correct. It works fine in the following case:

```
eq x y = False; # line
['eq x y']      # line.split('=')[0]
['x','y']       # line.split('=')[0].split(' ')[1:]
```

However when arguments are grouped together by parentheses it will not work, because the grouped arguments need to stay grouped together:

```
eq (Int x) y = False;    # line
['eq (Int x) y']         # line.split('=')[0]
['(Int','x)','y']        # line.split('=')[0].split(' ')
  [1:]
```

Therefore it is necessary to process the arguments a bit more, so a function is written to group these arguments together:

```
def parseArg(line):
    arg = []
    parentheses = 0
    current = ''
    for char in line:
        if char == '(':
            parentheses += 1
            current += char
        elif char == ')':
            parentheses -= 1
            current += char
        elif char == ' ':
            if parentheses == 0:
                arg.append(current)
                current = ''
            else:
                current += char
        else:
            current += char
    if current > '':
        arg.append(current)
    return arg
```

A string is parsed into arguments by processing each character separately. The following happens:

- when the character is an opening parenthesis `(`, the parenthesis counter is incremented and the character is saved in the current argument
- when the character is a closing parenthesis `)`, the parenthesis counter is decremented and the character is saved in the current argument
- when the character is a space, there are two possibilities:

  - the parenthesis counter is 0, then the current argument is added to the list of arguments
  - the parenthesis counter is greater than 0, then the space is added to the current argument
- any other character will be added to the current character

Because this `parseArg` needs a string as input and the non-grouped arguments were a list, this list has to be combined to a string:

```
def combine(list):
    string = ''
    for s in list:
        string += ' '
        string += s
    return string.strip()
```

Therefore:

```
functionArguments = parseArg(combine(line.split('=')[0].
    split(' ')[1:]))
```

8.2.3 Parsing the function body

The function body is just the part right to the `=` symbol:

```
line.split('=')[1].strip().strip(';')
```

with whitespace and the semicolon at the end of the body removed.

However, there is an example where this won't work. If the function `(==)` is in the function body, as in one of the examples above, the `=`'s in the function will also be split as well. Therefore to get these back the following function will combine the split parts:

```
def combineIs(list):
    string = ''
    for s in list:
        string += '='
        string += s
    return string.strip('=')
```

8.3 Creating conversion functions

The type definition consists of two parts split by `::` the type name and the definition.

```
type = line.split('::')[0].strip()
definition = line.split('::')[1].strip()
```

As shown in previous chapters the generic representation given a type can be found by using the following rewriting rules as shown in [1]:

$$
\begin{aligned}
:: Ta_1...a_n &= K_1 t_{11}...t_{1l_1}|...|K_m t_{m1}...t_{ml_m} \\
T^{\circ} a_1...a_n &= (t_{11} \times ... \times t_{1l_1}) + ... + (t_{m1} \times ... \times t_{ml_m})
\end{aligned}
$$

The G-lite syntax is a bit different, but it is quite similar:

$$Ta_1...a_n :: K_1t_{11}...t_{1l_1}|...|K_mt_{m1}...t_{ml_m}$$
$$T^\circ a_1...a_n = (t_{11} \times ... \times t_{1l_1}) + ... + (t_{m1} \times ... \times t_{ml_m})$$

However, these rewriting rules are to create the generic representation, but not to create the conversion functions, the functions from and to the generic types, which is what is needed. F-lite does not use types, so only the conversion functions are needed and not the generic representation/type. However, these rewriting rules can be very useful to create the conversion functions:

- the parts split by `|`'s in the original type and the corresponding parts in the generic representation split by `+` are separate instances in the conversion functions, because an instance of the type can be just one of them at a time. For example, a list is either a `Nil` or a `Cons`, but never both.
- all the other parts belong together and have to be paired together, so every $\times$ will result in a `PAIR`.

On the left side of the `::` is the type name and its type arguments and on the right side is the type definition, so:

```
type = line.split('::')[0].strip()
definition = line.split('::')[1].strip()
```

As said before every part in the definition can be handled apart, because they will result in separate instances of the conversion functions. For every part the following happens:

1. parse the definition, with parentheses grouped together with the `parseArg` function.
2. the result of the parsing gives two kinds of information:
   (a) constructor
   (b) number of arguments after the constructor

   For example, when using the following type:

   ```
   List a :: Nil | Cons a (List a)
   ```

   the result of the parsing of these definitions will be:

   ```
   ['Nil']
   ['Cons', 'a', '(List a)']
   ```

   This shows that the constructor is `Cons` and the number of arguments is two.
3. using this information the original part of the conversion is created by filling random variables in the places of the arguments. The previous example will result in:

   ```
   Nil
   Cons a b
   ```

4. using the same information the generic part of the conversion is created. For this the only information needed is the number of arguments:

   - if it is greater than zero, then the result will be random variables, as much as there were arguments, paired together with `PAIR` statements. The `Cons` definition in the example will be:

     PAIR a b

   - else the result will be `UNIT`. So `Nil` in the example will result in `UNIT`

   Because those definitions are separated by `EITHER` types, the `LEFT` and `RIGHT` tags need to be added.

   LEFT UNIT
   RIGHT (PAIR a b)

5. finally the conversions are added to a list, in the same way as functions are added to the saved list of functions.

   This results in the following code:

```
def typedef(line):
    type = line.split('::')[0].strip()
    definition = line.split('::')[1].strip()
    conversions = []
    current = 1
    max = len(definition.split('|'))
    for i in definition.split('|'):
        vars = ['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']
        defs = parseArg(i.strip())
        numberArguments = len(defs[1:])
        if numberArguments > 0:
            original = '(' + defs[0]
            for j in range(numberArguments):
                original += ' ' + vars[j]
            original += ')'
            generic = leftRightTags(current, max) +
                pair(vars[:numberArguments])
        else:
            original = defs[0]
            generic = leftRightTags(current, max) + '
                UNIT'
        conversions.append(['to'+type.split(' ')[0].
            strip(),
            ['('+generic+')'],[original.strip('()')
                ]])
        conversions.append(['from'+type.split(' ')
            [0].strip(),
            [original],[generic]])
        current += 1
    return conversions
```

Where the function `pair` is used to pair variables together:

```
def pair(defs):
    if len(defs) == 1:
        return defs[0]
    else:
        return '(PAIR ' + defs[0] + ' ' + pair(defs
            [1:]) + ')'
```

and the function `leftRightTags` adds the `LEFT` and `RIGHT` tags, given the number of definitions and the current number of the definition:

```
def leftRightTags(i,max):
    tags = ''
    for j in range (i-1):
        tags += 'RIGHT '
    if i < max:
        tags += 'LEFT '
    return tags
```

Giving the definition of a list

```
List a :: Nil | Cons a (List a)
```

the conversion creation function will result in the following conversions:

```
fromList Nil = LEFT UNIT
fromList (Cons a b) = RIGHT (PAIR a b)
toList (LEFT UNIT) = Nil
toList (RIGHT (PAIR a b)) = Cons a b
```

8.4 Creating new instances

The `Generic` statement consists of the name of the function and the function type. The new instance to be created consists of:

- that function name
- some arguments, as presented by the statement, but with random variable names
- the function body, which calls the same function, but makes use of generic conversions. In other words converting to the generic representation and use the generic function. It is also possible that the function returns something of the original data type, in that case the result of the generic function needs to be converted back to the original data type.

The following statement

```
Generic eq (List a) (List a) -> Bool
```

will result in

```
eq (List a) (List b) = eq (fromList a) (fromList b)
```

The function is defined as follows:

```
def createInstance(line):
    funcName = line.split(' ')[0].strip()
    args = parseArg(line.split('->')[0])[1:]
    result = parseArg(line.split('->')[1].strip())[0].
        strip('()').split(' ')[0].strip()
    funcArgs = []
    funcBody = [funcName]
    currentvar = 0
    for arg in args:
        vars = ['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']
        type = arg.strip('()').split(' ')[0].strip()
        if type in genericTypes:
            funcArgs.append('(' + type + ' ' + vars[
                currentvar] + ')')
            funcBody.append('(from' + type + ' ' + vars[
                currentvar] + ')')
        else:
            funcArgs.append(vars[currentvar])
            funcBody.append(vars[currentvar])
        currentvar += 1
    if result in genericTypes:
        funcBody.insert(0,'to' + result + ' ' + '(')
        funcBody.append(')')
    return [funcName,funcArgs,funcBody]
```

Chapter 9

Code reduction test

In order to test the gain of F-lite in combination with generics, a comparison between the code of ordinary F-lite code and G-lite code is done. This test is done over two main programs. The first program implements the equality function as described in chapter 3 and the second program implements an increment function and a print function. In the F-lite programs all the functions are ordinary F-lite functions, whereas in the G-lite programs the functions are implemented with Generics and makes use of the compiler to generate the conversions between ordinary constructors, i.e. data types, automatically.

The comparison made between F-lite and G-lite is a code reduction test to check whether using the G-lite compiler reduces the number of lines of code, when programming in G-lite instead of F-lite. However, the test will not show only whether G-lite is more efficient in terms of lines of code, but also the degree in which G-lite code increases when a new data type of function is added and the degree in which F-lite code increases in lines of code. This is done as follows: as said above there are two programs in both F-lite and G-lite. The first program has one generic function in G-lite and the second program has two generic functions. This is to show how the increase in lines of code, when adding a new function, depends on the number of functions. To show the increase in lines of code when adding a new data type, each program implements first a `Tree`, then a `List`, then a `Rose` and finally a `Triplet`, which have the following definition:

- Tree:

```
1 Tree a :: Leaf | Bin a (Tree a) (Tree a)
```

- List:

```
1 List a :: Nil | Cons a (List a)
```

- Rose:

```
1 Rose a :: Ros a (List (Rose a))
```

- Triplet:

```
1 Triplet a b c :: First a | Second b | Third c
```

These data types are very different from each other. The `Lists` and `Trees` have both two constructors, e.g. `List` is either a `Nil` or a `Cons`, whereas the `Roses` have one and the `Triplet` has three. The reason for this selection is that it is quite likely that the increase of code is dependent on the data type and especially the number of constructors is has.

Every program has some other functions as well, for example the `emitStr` function, which is to print a string on the screen. These functions are in the F-lite and G-lite version exactly the same and will not be considered during the analysis.

In the following sections the important parts of the programs are listed, i.e. the parts that are used for analysis. The full programs are listed in the appendix.

9.1 Program 1: equality function

9.1.1 Tree

F-lite

```
1 eq (Int x) (Int y) = (==) x y;
2 eq (Tree Leaf) (Tree Leaf) = True;
3 eq (Tree (Bin x l r)) (Tree (Bin y ll rr)) = and (eq x y)
      (and (eq l ll) (eq r rr));
4 eq (Tree Leaf) (Tree (Bin x l r)) = False;
5 eq (Tree (Bin x l r)) (Tree Leaf) = False;
```

G-lite

```
1 eq (Int x) (Int y) = (==) x y;
2 eq UNIT UNIT = True;
3 eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b);
4 eq (LEFT x) (LEFT y) = eq x y;
5 eq (RIGHT x) (RIGHT y) = eq x y;
6 eq (LEFT x) (RIGHT y) = False;
7 eq (RIGHT x) (LEFT y) = False;
8 Tree a :: Leaf | Bin a (Tree a) (Tree a)
9 Generic eq (Tree a) (Tree a) -> Bool
```

9.1.2 Tree and List

F-lite

```
1 eq (Int x) (Int y) = (==) x y;
2 eq (Tree Leaf) (Tree Leaf) = True;
3 eq (Tree (Bin x l r)) (Tree (Bin y ll rr)) = and (eq x y)
      (and (eq l ll) (eq r rr));
4 eq (Tree Leaf) (Tree (Bin x l r)) = False;
5 eq (Tree (Bin x l r)) (Tree Leaf) = False;
6 eq (List Nil) (List Nil) = True;
7 eq (List (Cons x xs)) (List (Cons y ys)) = and (eq x y) (
    eq xs ys);
```

```
8  eq (List Nil) (List (Cons x xs)) = False;
9  eq (List (Cons x xs)) (List Nil) = False;
```

G-lite

```
1  eq (Int x) (Int y) = (==) x y;
2  eq UNIT UNIT = True;
3  eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b);
4  eq (LEFT x) (LEFT y) = eq x y;
5  eq (RIGHT x) (RIGHT y) = eq x y;
6  eq (LEFT x) (RIGHT y) = False;
7  eq (RIGHT x) (LEFT y) = False;
8  Tree a :: Leaf | Bin a (Tree a) (Tree a)
9  Generic eq (Tree a) (Tree a) -> Bool
10 List a :: Nil | Cons a (List a)
11 Generic eq (List a) (List a) -> Bool
```

9.1.3 Tree, List and Rose

F-lite

```
1  eq (Int x) (Int y) = (==) x y;
2  eq (Tree Leaf) (Tree Leaf) = True;
3  eq (Tree (Bin x l r)) (Tree (Bin y ll rr)) = and (eq x y)
       (and (eq l ll) (eq r rr));
4  eq (Tree Leaf) (Tree (Bin x l r)) = False;
5  eq (Tree (Bin x l r)) (Tree Leaf) = False;
6  eq (List Nil) (List Nil) = True;
7  eq (List (Cons x xs)) (List (Cons y ys)) = and (eq x y) (
       eq xs ys);
8  eq (List Nil) (List (Cons x xs)) = False;
9  eq (List (Cons x xs)) (List Nil) = False;
10 eq (Rose (Ros x r)) (Rose (Ros y rr)) = and (eq x y) (eq
       r rr);
```

G-lite

```
1  eq (Int x) (Int y) = (==) x y;
2  eq UNIT UNIT = True;
3  eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b);
4  eq (LEFT x) (LEFT y) = eq x y;
5  eq (RIGHT x) (RIGHT y) = eq x y;
6  eq (LEFT x) (RIGHT y) = False;
7  eq (RIGHT x) (LEFT y) = False;
8  Tree a :: Leaf | Bin a (Tree a) (Tree a)
9  Generic eq (Tree a) (Tree a) -> Bool
10 List a :: Nil | Cons a (List a)
11 Generic eq (List a) (List a) -> Bool
12 Rose a :: Ros a (List (Rose a))
13 Generic eq (Rose a) (Rose a) -> Bool
```

9.1.4 Tree, List, Rose and Triplet

F-lite

```
1  eq (Int x) (Int y) = (==) x y;
2  eq (Tree Leaf) (Tree Leaf) = True;
3  eq (Tree (Bin x l r)) (Tree (Bin y ll rr)) = and (eq x y)
       (and (eq l ll) (eq r rr));
4  eq (Tree Leaf) (Tree (Bin x l r)) = False;
5  eq (Tree (Bin x l r)) (Tree Leaf) = False;
6  eq (List Nil) (List Nil) = True;
7  eq (List (Cons x xs)) (List (Cons y ys)) = and (eq x y) (
       eq xs ys);
8  eq (List Nil) (List (Cons x xs)) = False;
9  eq (List (Cons x xs)) (List Nil) = False;
10 eq (Rose (Ros x r)) (Rose (Ros y rr)) = and (eq x y) (eq
       r rr);
11 eq (Triplet (First x)) (Triplet (First y)) = eq x y;
12 eq (Triplet (First x)) (Triplet (Second y)) = False;
13 eq (Triplet (First x)) (Triplet (Third y)) = False;
14 eq (Triplet (Second x)) (Triplet (First y)) = False;
15 eq (Triplet (Second x)) (Triplet (Second y)) = eq x y;
16 eq (Triplet (Second x)) (Triplet (Third y)) = False;
17 eq (Triplet (Third x)) (Triplet (First y)) = False;
18 eq (Triplet (Third x)) (Triplet (Second y)) = False;
19 eq (Triplet (Third x)) (Triplet (Third y)) = eq x y;
```

G-lite

```
1  eq (Int x) (Int y) = (==) x y;
2  eq UNIT UNIT = True;
3  eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b);
4  eq (LEFT x) (LEFT y) = eq x y;
5  eq (RIGHT x) (RIGHT y) = eq x y;
6  eq (LEFT x) (RIGHT y) = False;
7  eq (RIGHT x) (LEFT y) = False;
8  Tree a :: Leaf | Bin a (Tree a) (Tree a)
9  Generic eq (Tree a) (Tree a) -> Bool
10 List a :: Nil | Cons a (List a)
11 Generic eq (List a) (List a) -> Bool
12 Rose a :: Ros a (List (Rose a))
13 Generic eq (Rose a) (Rose a) -> Bool
14 Triplet a b c :: First a | Second b | Third c
15 Generic eq (Triplet a b c) (Triplet a b c) -> Bool
```

9.2 Increment and print

9.2.1 Tree

F-lite

```
inc (Int x) = (Int ((+) x 1));
inc (Tree Leaf) = Tree Leaf;
inc (Tree (Bin x l r)) = Tree (Bin (inc x) (inc l) (inc r
    ));
print (Int x) k = emitInt x k;
print (Tree Leaf) k = k;
print (Tree (Bin x l r)) k = print x (emitStr " " (print
    l (emitStr " " (print r k))));
```

G-lite

```
inc (Int x) = (Int ((+) x 1));
inc UNIT = UNIT;
inc (PAIR x y) = PAIR (inc x) (inc y);
inc (LEFT x) = LEFT (inc x);
inc (RIGHT x) = RIGHT (inc x);
print (Int x) k = emitInt x k;
print UNIT k = k;
print (PAIR x y) k = print x (emitStr " " (print y k));
print (LEFT x) k = print x k;
print (RIGHT x) k = print x k;
Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic inc (Tree a) -> (Tree a)
Generic print (Tree a) Int -> Int
```

9.2.2 Tree and List

F-lite

```
inc (Int x) = (Int ((+) x 1));
inc (List Nil) = List Nil;
inc (List (Cons x xs)) = List (Cons (inc x) (inc xs));
inc (Tree Leaf) = Tree Leaf;
inc (Tree (Bin x l r)) = Tree (Bin (inc x) (inc l) (inc r
    ));
print (Int x) k = emitInt x k;
print (List Nil) k = k;
print (List (Cons x xs)) k = print x (emitStr " " (print
    xs));
print (Tree Leaf) k = k;
print (Tree (Bin x l r)) k = print x (emitStr " " (print
    l (emitStr " " (print r k))));
```

G-lite

```
inc (Int x) = (Int ((+) x 1));
inc UNIT = UNIT;
inc (PAIR x y) = PAIR (inc x) (inc y);
inc (LEFT x) = LEFT (inc x);
inc (RIGHT x) = RIGHT (inc x);
print (Int x) k = emitInt x k;
print UNIT k = k;
```

```
8  print (PAIR x y) k = print x (emitStr " " (print y k));
9  print (LEFT x) k = print x k;
10 print (RIGHT x) k = print x k;
11 Tree a :: Leaf | Bin a (Tree a) (Tree a)
12 Generic inc (Tree a) -> (Tree a)
13 Generic print (Tree a) Int -> Int
14 List a :: Nil | Cons a (List a)
15 Generic inc (List a) -> (List a)
16 Generic print (List a) Int -> Int
```

9.2.3 Tree, List and Rose

F-lite

```
1  inc (Int x) = (Int ((+) x 1));
2  inc (List Nil) = List Nil;
3  inc (List (Cons x xs)) = List (Cons (inc x) (inc xs));
4  inc (Tree Leaf) = Tree Leaf;
5  inc (Tree (Bin x l r)) = Tree (Bin (inc x) (inc l) (inc r
       ));
6  inc (Rose (Ros x r)) = Rose (Ros (inc x) (inc r));
7  print (Int x) k = emitInt x k;
8  print (List Nil) k = k;
9  print (List (Cons x xs)) k = print x (emitStr " " (print
       xs k));
10 print (Tree Leaf) k = k;
11 print (Tree (Bin x l r)) k = print x (emitStr " " (print
       l (emitStr " " (print r k))));
12 print (Rose (Ros x r)) k = print x (emitStr " " (print r
       k));
```

G-lite

```
1  inc (Int x) = (Int ((+) x 1));
2  inc UNIT = UNIT;
3  inc (PAIR x y) = PAIR (inc x) (inc y);
4  inc (LEFT x) = LEFT (inc x);
5  inc (RIGHT x) = RIGHT (inc x);
6  print (Int x) k = emitInt x k;
7  print UNIT k = k;
8  print (PAIR x y) k = print x (emitStr " " (print y k));
9  print (LEFT x) k = print x k;
10 print (RIGHT x) k = print x k;
11 Tree a :: Leaf | Bin a (Tree a) (Tree a)
12 Generic inc (Tree a) -> (Tree a)
13 Generic print (Tree a) Int -> Int
14 List a :: Nil | Cons a (List a)
15 Generic inc (List a) -> (List a)
16 Generic print (List a) Int -> Int
17 Rose a :: Ros a (List (Rose a))
18 Generic inc (Rose a) -> (Rose a)
```

```
19 Generic print (Rose a) Int -> Int
```

9.2.4 Tree, List, Rose and Triplet

F-lite

```
1  inc (Int x) = (Int ((+) x 1));
2  inc (List Nil) = List Nil;
3  inc (List (Cons x xs)) = List (Cons (inc x) (inc xs));
4  inc (Tree Leaf) = Tree Leaf;
5  inc (Tree (Bin x l r)) = Tree (Bin (inc x) (inc l) (inc r
       ));
6  inc (Rose (Ros x r)) = Rose (Ros (inc x) (inc r));
7  inc (Triplet (First x)) = First (inc x);
8  inc (Triplet (Second x)) = Second (inc x);
9  inc (Triplet (Third x)) = Third (inc x);
10 print (Int x) k = emitInt x k;
11 print (List Nil) k = k;
12 print (List (Cons x xs)) k = print x (emitStr " " (print
       xs k));
13 print (Tree Leaf) k = k;
14 print (Tree (Bin x l r)) k = print x (emitStr " " (print
       l (emitStr " " (print r k))));
15 print (Rose (Ros x r)) k = print x (emitStr " " (print r
       k));
16 print (First x) k = print x k;
17 print (Second x) k = print x k;
18 print (Third x) k = print x k;
```

G-lite

```
1  inc (Int x) = (Int ((+) x 1));
2  inc UNIT = UNIT;
3  inc (PAIR x y) = PAIR (inc x) (inc y);
4  inc (LEFT x) = LEFT (inc x);
5  inc (RIGHT x) = RIGHT (inc x);
6  print (Int x) k = emitInt x k;
7  print UNIT k = k;
8  print (PAIR x y) k = print x (emitStr " " (print y k));
9  print (LEFT x) k = print x k;
10 print (RIGHT x) k = print x k;
11 Tree a :: Leaf | Bin a (Tree a) (Tree a)
12 Generic inc (Tree a) -> (Tree a)
13 Generic print (Tree a) Int -> Int
14 List a :: Nil | Cons a (List a)
15 Generic inc (List a) -> (List a)
16 Generic print (List a) Int -> Int
17 Rose a :: Ros a (List (Rose a))
18 Generic inc (Rose a) -> (Rose a)
19 Generic print (Rose a) Int -> Int
20 Triplet a b c :: First a | Second b | Third c
```

```
21 Generic inc (Triplet a b c) -> (Triplet a b c)
22 Generic print (Triplet a b c) Int -> Int
```

9.3 Code reduction in Clean

To compare the results of G-lite with respect to the lines of code it reduces by introducing generic programming, a similar comparison is made with Clean code. In this case an increment function is written, like in G-lite and F-lite. First in ordinary Clean code and second in generic Clean code.

For the Clean implementations of the increment function, the same data types are used and are added in the same order. Note that in the ordinary Clean code the functions are instances of a class of functions, instead of functions by themselves. The reason is that for generics also some sort of class is created. However the use of classes does not change the number of lines for the functions:

```
1 instance inc` Int where
2     inc` x = x + 1
3
4 incInt :: Int -> Int
5 incInt x = x + 1
```

It is just one line of code extra for the `class` definition, but the same happens when using the `generic` clause. Therefore both statements cancel each other when using them in the comparison between ordinary and generic Clean code.

9.3.1 Tree

Using ordinary Clean

```
1 class inc` a :: a -> a
2 instance inc` Int where
3     inc` x = x + 1
4 instance inc` (Tree a) | inc` a where
5     inc` Leaf = Leaf
6     inc` (Bin x l r) = Bin (inc` x) (inc` l) (inc` r)
7 :: Tree a = Leaf | Bin a (Tree a) (Tree a)
```

Using generics in Clean

```
1 generic gInc a :: a -> a
2 gInc{|UNIT|} UNIT = UNIT
3 gInc{|PAIR|} inca incb (PAIR x y) = PAIR (inca x) (incb y
    )
4 gInc{|EITHER|} inca incb (LEFT x) = LEFT (inca x)
5 gInc{|EITHER|} inca incb (RIGHT x) = RIGHT (incb x)
6 gInc{|CONS|} inca (CONS x) = CONS (inca x)
7 gInc{|OBJECT|} inca (OBJECT x) = OBJECT (inca x)
8 gInc{|Int|} x = x + 1
9 derive gInc Tree
10 :: Tree a = Leaf | Bin a (Tree a) (Tree a)
```

9.3.2 Tree and List

Using ordinary Clean

```
class inc` a :: a -> a
instance inc` Int where
    inc` x = x + 1
instance inc` (Tree a) | inc` a where
    inc` Leaf = Leaf
    inc` (Bin x l r) = Bin (inc` x) (inc` l) (inc` r)
instance inc` (List a) | inc` a where
    inc` Nil = Nil
    inc` (Cons x xs) = Cons (inc` x) (inc` xs)
:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)
```

Using generics in Clean

```
generic gInc a :: a -> a
gInc{|UNIT|} UNIT = UNIT
gInc{|PAIR|} inca incb (PAIR x y) = PAIR (inca x) (incb y
    )
gInc{|EITHER|} inca incb (LEFT x) = LEFT (inca x)
gInc{|EITHER|} inca incb (RIGHT x) = RIGHT (incb x)
gInc{|CONS|} inca (CONS x) = CONS (inca x)
gInc{|OBJECT|} inca (OBJECT x) = OBJECT (inca x)
gInc{|Int|} x = x + 1
derive gInc Tree, List
:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)
```

9.3.3 Tree, List and Rose

Using ordinary Clean

```
class inc` a :: a -> a
instance inc` Int where
    inc` x = x + 1
instance inc` (Tree a) | inc` a where
    inc` Leaf = Leaf
    inc` (Bin x l r) = Bin (inc` x) (inc` l) (inc` r)
instance inc` (List a) | inc` a where
    inc` Nil = Nil
    inc` (Cons x xs) = Cons (inc` x) (inc` xs)
instance inc` (Rose a) | inc` a where
    inc` (Rose x xs) = Rose (inc` x) (inc` xs)
:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)
:: Rose a = Rose a (List (Rose a))
```

Using generics in Clean

```
1  generic gInc a :: a -> a
2  gInc{|UNIT|} UNIT = UNIT
3  gInc{|PAIR|} inca incb (PAIR x y) = PAIR (inca x) (incb y
       )
4  gInc{|EITHER|} inca incb (LEFT x) = LEFT (inca x)
5  gInc{|EITHER|} inca incb (RIGHT x) = RIGHT (incb x)
6  gInc{|CONS|} inca (CONS x) = CONS (inca x)
7  gInc{|OBJECT|} inca (OBJECT x) = OBJECT (inca x)
8  gInc{|Int|} x = x + 1
9  derive gInc Tree, List, Rose
10 :: Tree a = Leaf | Bin a (Tree a) (Tree a)
11 :: List a = Nil | Cons a (List a)
12 :: Rose a = Rose a (List (Rose a))
```

9.3.4 Tree, List, Rose and Triplet

Using ordinary Clean

```
1  class inc` a :: a -> a
2  instance inc` Int where
3      inc` x = x + 1
4  instance inc` (Tree a) | inc` a where
5      inc` Leaf = Leaf
6      inc` (Bin x l r) = Bin (inc` x) (inc` l) (inc` r)
7  instance inc` (List a) | inc` a where
8      inc` Nil = Nil
9      inc` (Cons x xs) = Cons (inc` x) (inc` xs)
10 instance inc` (Rose a) | inc` a where
11     inc` (Rose x xs) = Rose (inc` x) (inc` xs)
12 instance inc` (Triplet a b c) | inc` a & inc` b & inc` c
       where
13     inc` (First x) = First (inc` x)
14     inc` (Second x) = Second (inc` x)
15     inc` (Third x) = Third (inc` x)
16 :: Tree a = Leaf | Bin a (Tree a) (Tree a)
17 :: List a = Nil | Cons a (List a)
18 :: Rose a = Rose a (List (Rose a))
19 :: Triplet a b c = First a | Second b | Third c
```

Using generics in Clean

```
1  generic gInc a :: a -> a
2  gInc{|UNIT|} UNIT = UNIT
3  gInc{|PAIR|} inca incb (PAIR x y) = PAIR (inca x) (incb y
       )
4  gInc{|EITHER|} inca incb (LEFT x) = LEFT (inca x)
5  gInc{|EITHER|} inca incb (RIGHT x) = RIGHT (incb x)
6  gInc{|CONS|} inca (CONS x) = CONS (inca x)
7  gInc{|OBJECT|} inca (OBJECT x) = OBJECT (inca x)
8  gInc{|Int|} x = x + 1
9  derive gInc Tree, List, Rose, Triplet
```

```
10 :: Tree a = Leaf | Bin a (Tree a) (Tree a)
11 :: List a = Nil | Cons a (List a)
12 :: Rose a = Rose a (List (Rose a))
13 :: Triplet a b c = First a | Second b | Third c
```

Chapter 10

Results

In the previous chapter the code used for the analysis is shown. In this chapter the results are discussed.

Out of the collected data the following table is created:

	Program 1		Program 2	
	F-lite	G-lite	F-lite	G-lite
Tree	5	9	6	13
Tree & List	9	11	10	16
Tree, List & Rose	10	13	12	19
Tree, List, Rose & Triplet	19	15	18	22

In the following section the consequences of adding an extra data type to the program with respect to the number of lines of code will be discussed.

10.1 Consequences of adding a new data type

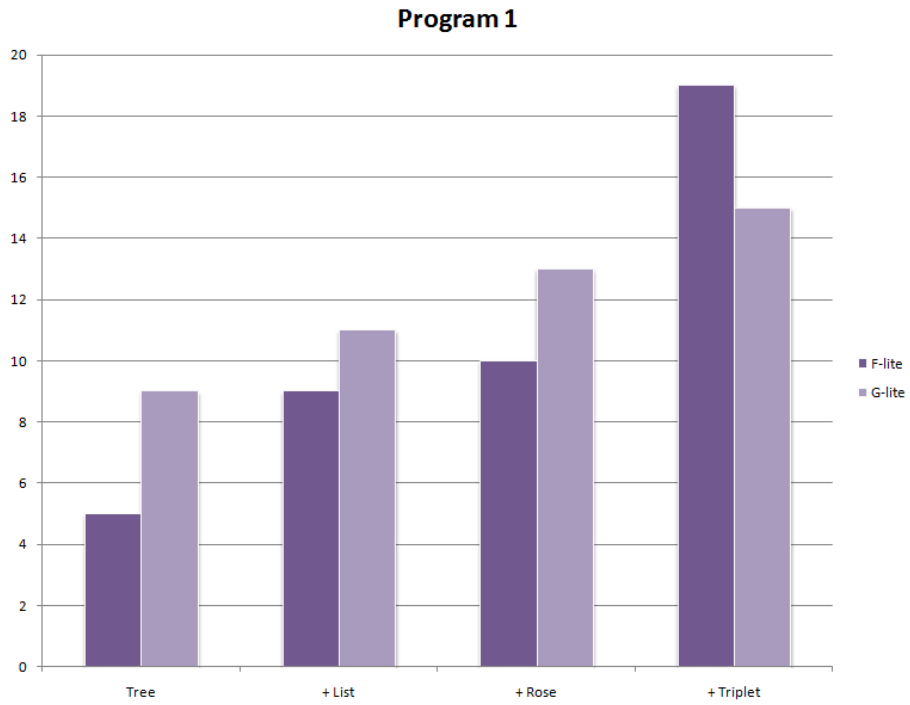
In the table above it is shown what happens when an extra data type is added to the program. The programs with only trees are used as a basic form. Then a list, a rose and a triplet are added to the program, which have the following data types:

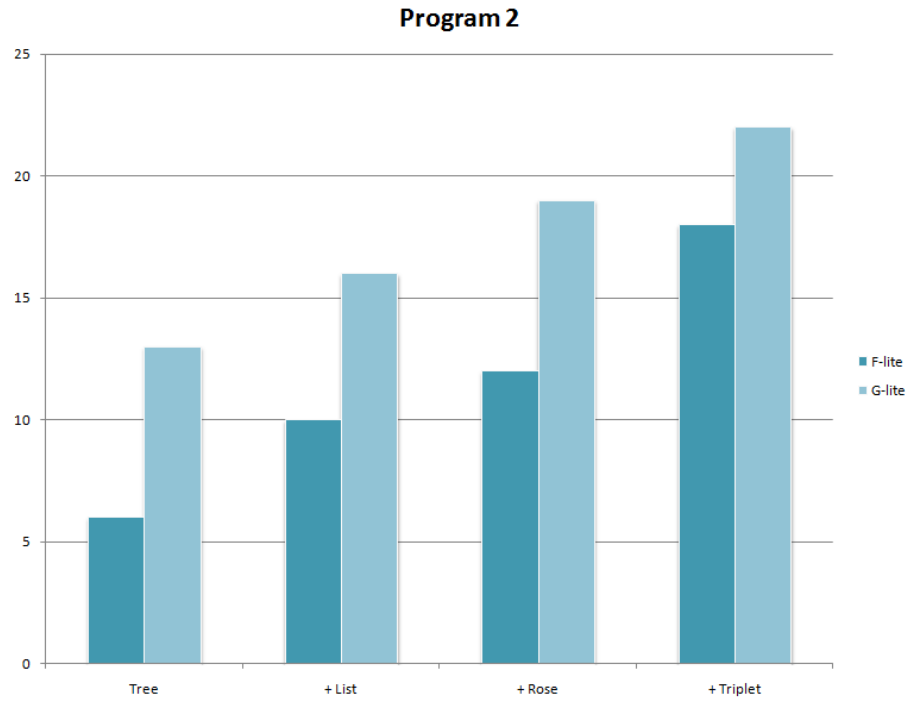
```
Tree a :: Leaf | Bin a (Tree a) (Tree a)

List a :: Nil | Cons a (List a)

Rose a :: Ros a (List (Rose a))

Triplet a b c :: First a | Second b | Third c
```

This shows that a tree has two constructors (`Leaf` and `Bin`), a lists has two constructors as well (`Nil` and `Cons`), a Rose has one constructor (`Ros`) and a Triplet has three constructors (`First`, `Second` and `Third`). When looking at the two F-lite programs, the following increases in lines of code can be found:

The functions in the two programs had the following types:

```
eq :: a a -> Bool
```


Program 1
20
18
16
14
12
10
8
6
4
2
0
Tree
+ List
+ Rose
+ Triplet
F-lite
G-lite


Program 2
25
20
15
10
5
0
Tree
+ List
+ Rose
+ Triplet
F-lite
G-lite

Added data type	Program 1	Program 2
List	4	4
Rose	1	2
Triplet	9	6

Figure 10.1: Table 2: increases in number of lines of code

```
inc :: a -> a
print :: a -> Int
```

The first program had one function with two arguments. When adding a new data type, every argument has to get every possible constructor of that data type, like for lists:

```
eq (List Nil) (List Nil) = ...
eq (List Nil) (List Cons ...) = ...
eq (List Cons...) (List Nil) = ...
eq (List Cons ...) (List Cons ...) = ...
```

Therefore in this case the number of new lines in the code can be calculated as follows: two constructors and two arguments = $2^2 = 4$. This also holds for the rose and triplet:

$$\text{Rose: one constructor and two arguments} = 1^2 = 1$$
$$\text{Triplet: three constructors and two arguments} = 3^2 = 9$$

For the second program, it holds as well:

- List:

$$\text{inc: two constructors and one argument} = 2^1 = 2$$
$$\text{print: two constructors and one argument} = 2^1 = 2$$
$$\text{inc + print: } 2 + 2 = 4$$

- Rose:

$$\text{inc: one constructor and one argument} = 1^1 = 1$$
$$\text{print: one constructors and one argument} = 1^1 = 1$$
$$\text{inc + print: } 1 + 1 = 2$$

- Triplet:

$$\text{inc: three constructors and one argument} = 3^1 = 2$$
$$\text{print: three constructors and one argument} = 3^1 = 2$$
$$\text{inc + print: } 3 + 3 = 6$$

Therefore when using F-lite the following function can be used, to calculate the increase of line of code, when adding a new data type:

$$\sum_{f}^{Functions} cons^{arg(f)} \tag{10.1}$$

For G-lite the same can be done. However, it is not that difficult, because for each new data type there has to be the following:

- type definition
- for each *generic* function a `Generic` statement

Therefore the increases in terms of lines of code in G-lite can be defined as follows:

$$\text{number of functions} + 1 \tag{10.2}$$

Of course this function holds for both programs. The first program has one *generic* function, so it increases every time with 2, and the second program has two *generic* functions, so it increases with 3 each time.

10.2 Clean

A similar test has been done with an increment function in Clean to compare the efficiency of Clean and G-lite with respect to the reduction of lines of code. This leaded to the following results:

	Ordinary functions	Generic functions
Tree	7	10
Tree & List	11	11
Tree, List & Rose	14	12
Tree, List, Rose & Triplet	19	13

The increases when adding the new data types are as follows:

	Generic	Non-generic
List	1	4
Rose	1	3
Triplet	1	5

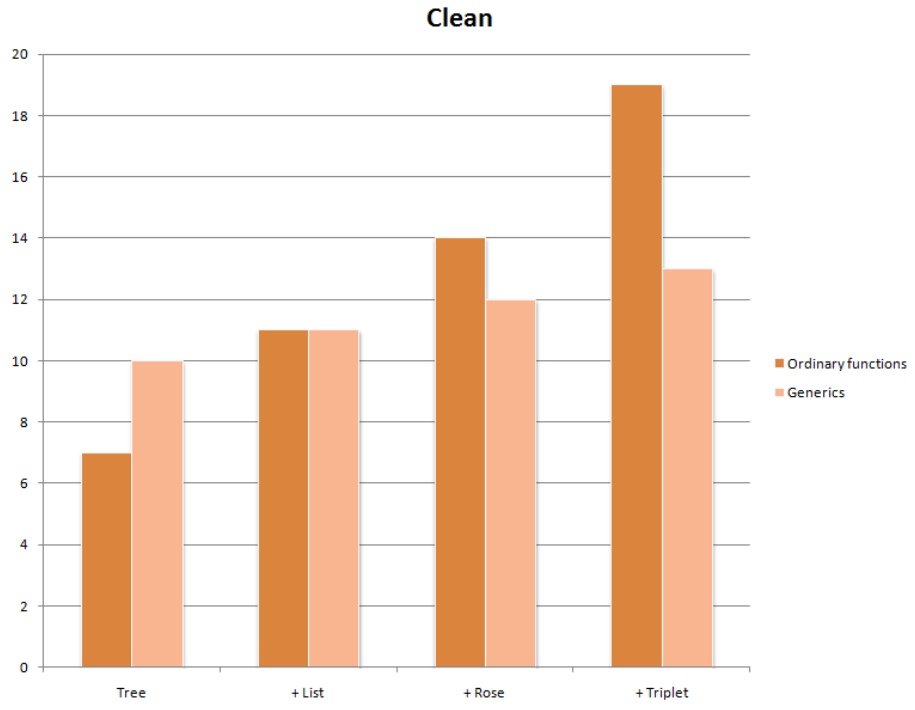
The increase of the number of lines of code in Clean can also be expressed in a function. With ordinary Clean functions the growth is as follows:

$$number\ of\ constructors + 2 \tag{10.3}$$

There has to one case for each constructor and assuming every case makes use of just one line, it grows with the number of constructors the type has plus two extra lines, namely for the instance declaration and one line for the definition of the type.

The same can be done for the generic case. In this case the growth is just 1. This is the line used for the definition of the type. For the derivation of this new type, no new line is needed, because derivations can be stated on one line, which makes the notation much more compact.

It is clear that Clean reduces more code and in more cases than G-lite. This is due for some nice one liners in Clean, which are not available in G-lite.


Clean
20
18
16
14
12
10
8
6
4
2
0
Tree
+ List
+ Rose
+ Triplet
Ordinary functions
Generics

Chapter 11

Discussion

The method used to implement G-lite as explained before has some disadvantages.

First of all the compiler itself is based on the syntactic structure of functions and new statements designed for G-lite. It is implemented using some basic string operations, like splitting. For a more general system Abstract Syntax Trees could have been used. However, the creation and usage of those where not known to the author at that time and therefore were not used as an implementation. Even though this would have been more general and easier to extend, that would have had some disadvantages as well. For example in the current implementation when processing functions, those are just copied into a data structure and not processed any more. The G-lite compiler simply does not need more information about normal functions, because that is F-lite code and that has to be processed by the F-lite compiler, so it has no use to parse them into detailed syntax trees.

However, some problems could occur using this prototype compiler. An example is generic functions with an function as an argument, e.g. `Int -> Char`, which would lead to multiple `->` in the type of the function specified in the `generic` statement. Which would lead to problems when compiling, because the compiler uses the `->` to split the arguments from the result. However, this problem could be solved by not mentioning the type of the argument, but just state a variable, e.g. `f`. This can be done, because the type of the function is not used strictly. The type is only used by the compiler to know which arguments need to be converted to a generic representation and if the result has to be converted back to an original type. Therefore all types which are not used generic can be declared by just one character. However, for completeness all types given as examples show their full type. Therefore

```
Generic eq (List a) (List a) -> Bool
```

could be rewritten to:

```
Generic eq (List a) (List a) -> b
```

or even:

```
Generic eq (List a) (List a) -> a
```

This type is not “correct”, but G-lite does not check the types, it only uses the information it needs.

A second point that needs to be mentioned is the fact that it is not explicitly clear from the G-lite syntax, that generic functions are depending of the function definitions of the primitive types. In Clean the following function header is possible:

```
func a :: a -> Int | otherFunc a
```

This states that `func` can transform something of type a to something of type b, **given there is an instance of** `otherFunc` **of** a. In Haskell there is a similar notation. This states explicitly that one function is dependent on another function. This is also the case with generic programming. If there are no instances for the primitives used, the generic function will not work at all. Therefore although this relation is not mentioned explicitly, it is there implicitly. However in Clean this is only used for normal function, e.g. defined as a `Class` of function, and not for generics as well.

Finally generics are easy to use as mentioned before. However this is not for free. Generic programming is less efficient as ordinary programming. This is because of the conversions to and from the generic representation. In other words, before something can be used it has to be transformed into something else and if the work is done it has to be transformed back. It is clear why this had disadvantages in performance in comparison with functions that handle the data types by themselves. Van Noort et al. [16] and Magalhães [13] show performance problems with generics in larger programs and give some optimisations to improve performance.

However G-lite is designed to run on the Reduceron, which is special designed for fast execution of functional programming languages, which could result in a less drastic decrease in performance, but this is not tested in the current research project, as G-lite only is tested in combination with the F-lite interpreter given by the Naylor and Runciman [15].

Chapter 12

Conclusion

In this Bachelor thesis F-lite, the core functional language of the Reduceron is extended with G-lite, which happens to be untyped. In this thesis a description of generics and G-lite is presented and a compiler has been created for G-lite, which could be used alongside the F-lite interpreter. It is shown that in some cases G-lite can be more efficient in terms of number of lines of code in some cases. As is shown, some cases were less efficient, however, the testing programs were small examples and generics are more useful when dealing with much larger programs. However, Clean was shown to be more efficient in decreasing the number of lines of code when using generics instead of ordinary Clean functions, but this is due to the short syntax used in Clean.

12.1 Future work

This research has shown that it is possible to have generics in an untyped language and that generics could be run on the Reduceron, since F-lite is designed to run on the Reduceron and G-lite compiles into F-lite. However, there is still a lot of work to be done in the subject of this research

G-lite has some limitations, for example kinds are not implemented yet, however this is needed in order to make generic version of for example mapping and zipping functions.

The performance of G-lite versus F-lite is not researched as well. Is G-lite less efficient than F-lite? What are the consequences of generics on the Reduceron? G-lite can be optimised as well. G-lite uses basic generics and no optimisation at all. Extending the compiler with some optimisations will boost the performance.

Bibliography

[1] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for clean. In T Arts and M Mohnen, editors, *Implementation of functional languages*, volume 2312 of *Lecture notes in computer science*, pages 168–185. Springer-Verlag Berlin, Heidelberger Platz 3, D-14197 Berlin, Germany, 2002.

[2] Richard Bird, Oege De Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6:1–28, 1996.

[3] Dave Clarke and Andres Löh. Generic haskell, specifically. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 21–47, Deventer, The Netherlands, The Netherlands, 2003. Kluwer, B.V.

[4] Jeremy Gibbons. Datatype-generic programming. In *Proceedings of the 2006 international conference on Datatype-generic programming*, SSDGP'06, pages 1–71, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] Ralf Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction: 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3-5, 2000 Proceedings*, 2000.

[6] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Lecture notes of the Spring School on Datatype-Generic Programming 2006*, volume LNCS 4719, pages 72 – 149. Springer-Verlag, 2007.

[7] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Proceedings of the Fourth Haskell Workshop*, pages 227–236. Elsevier Science, 2000.

[8] Patrik Jansson and Johan Jeuring. Polyp - a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 470–482, New York, NY, USA, 1997. ACM.

[9] Johan Jeuring, Sean Leather, José Pedro Magalhães, and Alexey Rodriguez Yakushev. Libraries for generic programming in haskell. In *Proceedings of the 6th international conference on Advanced functional programming*, AFP'08, pages 165–229, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In Ricardo Pea and Thomas Arts, editors, *Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 991–991. Springer Berlin / Heidelberg, 2003.

[11] Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An effective methodology for defining consistent semantics of complex systems. In Zoltn Horvth, Rinus Plasmeijer, and Viktria Zsk, editors, *Central European Functional Programming School*, volume 6299 of *Lecture Notes in Computer Science*, pages 224–267. Springer Berlin / Heidelberg, 2010.

[12] Bas Lijnse and Rinus Plasmeijer. itasks 2: itasks for end-users. In Marco Morazn and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages*, volume 6041 of *Lecture Notes in Computer Science*, pages 36–54. Springer Berlin / Heidelberg, 2011.

[13] José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing generics is easy! In *PEPM '10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 33–42, New York, NY, USA, 2010. ACM.

[14] Matthew Naylor and Colin Runciman. The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 129–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Matthew Naylor and Colin Runciman. The reduceron reconfigured. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 75–86, New York, NY, USA, 2010. ACM.

[16] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 13–24, New York, NY, USA, 2008. ACM.

[17] Ulf Norell and Patrik Jansson. Prototyping generic programming in template haskell. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 314–333. Springer Berlin / Heidelberg, 2004.

[18] Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In *Implementation of Functional Languages, Lecture Notes in Computer Science*, volume 3145/2005, pages 168–184, 2005.

[19] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. itasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 141–152, New York, NY, USA, 2007. ACM.

[20] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. An introduction to itasks: Defining interactive work flows for the web. In Zoltn Horvth, Rinus Plasmeijer, Anna Sos, and Viktria Zsk, editors, *Central European Functional Programming School*, volume 5161 of *Lecture Notes in Computer Science*, pages 1–40. Springer Berlin / Heidelberg, 2008.

[21] Jason S. Reich, Matthew Naylor, and Colin Runciman. Supercompilation and the Reduceron. In *Proceedings of the Second International Workshop on Metacomputation in Russia*, 2010.

[22] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 111–122, New York, NY, USA, 2008. ACM.

[23] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37:60–75, December 2002.

[24] Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 15–26, New York, NY, USA, 2010. ACM.

Chapter 13

Appendix

13.1 Prototype G-lite Compiler

```
import sys

def combine(list):
    string = ''
    for s in list:
        string += ' '
        string += s
    return string.strip()

def combineIs(list):
    string = ''
    for s in list:
        string += '='
        string += s
    return string.strip('=')

def pair(defs):
    if len(defs) == 1:
        return defs[0]
    else:
        return '(PAIR ' + defs[0] + ' ' + pair(defs[1:]) + ')'

def parseArg(line):
    arg = []
    parentheses = 0
    current = ''
    for char in line:
        if char == '(':
            parentheses += 1
            current += char
        elif char == ')':
            parentheses -= 1
            current += char
        elif char == ' ':
            if parentheses == 0:
                arg.append(current)
                current = ''
            else:
                current += char
        else:
            current += char
    if current > '':
        arg.append(current)
    return arg

def leftRightTags(line, l, r):
    if l > 0:
        return leftRightTags('LEFT ' + line, l-1, r)
    elif r > 0:
```

```
        if line[0] == '(' or not ' ' in line:
            return leftRightTags('RIGHT ' + line, l, r-1)
        else:
            return leftRightTags('RIGHT (' + line + ')', l, r-1)
    else:
        return line

def typedef(line):
    type = line.split('::')[0].strip()
    typename = type.split(' ')[0].strip()
    genericTypes.append(typename)
    definition = line.split('::')[1].strip()
    conversions = []
    current = 1
    max = len(definition.split('|'))
    for i in definition.split('|'):
        vars = ['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']
        defs = parseArg(i.strip())
        numberArguments = len(defs[1:])
        if numberArguments > 0:
            original = '(' + defs[0]
            for j in range(numberArguments):
                original += ' ' + vars[j]
            original += ')'
            if current < max:
                generic = leftRightTags(pair(vars[:numberArguments]),1,
                    current -1)
            else:
                generic = leftRightTags(pair(vars[:numberArguments]),0,
                    current -1)
        else:
            original = defs[0]
            if current < max:
                generic = leftRightTags('UNIT', 1, current -1)
            else:
                generic = leftRightTags('UNIT', 0, current -1)
        if max > 1:
            conversions.append(['to'+typename,['('+generic+')'],[
                typename + ' ' + original]])
        else:
            conversions.append(['to'+typename,[generic],[typename + ' '
                + original]])
        conversions.append(['from'+typename,[original],[generic]])
        current += 1
    return conversions


def function(line):
    functionName = line.split(' ')[0]
    functionArguments = parseArg(combine(line.split('=')[0].split(' ')[1
        :]))
    functionBody = [combineIs(line.split('=')[1:]).strip().strip(';')]
    return [functionName, functionArguments, functionBody]

def createInstance(line):
    funcName = line.split(' ')[0].strip()
    args = parseArg(line.split('->')[0])[1:]
    result = parseArg(line.split('->')[1].strip())[0].strip('()').split(
        ' ')[0].strip()
    funcArgs = []
    funcBody = [funcName]
    currentvar = 0
    for arg in args:
        vars = ['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']
        type = arg.strip('()').split(' ')[0].strip()
        if type in genericTypes:
            funcArgs.append('(' + type + ' ' + vars[currentvar] + ')')
            funcBody.append('(from' + type + ' ' + vars[currentvar] + ')
                ')
        else:
            funcArgs.append(vars[currentvar])
            funcBody.append(vars[currentvar])
```

```
            currentvar += 1
        if result in genericTypes:
            funcBody.insert(0,'to' + result + ' ' + '(')
            funcBody.append(')')
        return [funcName,funcArgs,funcBody]

try:
    input = sys.argv[1]
except IndexError:
    input = raw_input('Please give input file: ')
try:
    output = sys.argv[2]
except IndexError:
    output = raw_input('Please give output file: ')
input = open(input, 'r')
output = open(output, 'w')
functions = []
genericTypes = []
for line in input:
    line = line.strip()
    if line > '':
        if '=' in line:
            functions.append(function(line))
        if '::' in line:
            functions.extend(typedef(line))
        if 'Generic' in line:
            functions.append(createInstance(line.replace('Generic ','''))
                )
functions.sort()
print >> output, '{'
for (fname, farg, fbody) in functions:
    line = fname
    for a in farg:
        line += ' ' + a
    line += ' ='
    for b in fbody:
        line += ' ' + b
    line += ';'
    print >> output, line
print >> output, '}'
```

13.2 Comparison programs

13.2.1 Program 1: equality

F-lite

- Tree:

```
{

and False x = False;
and True x = x;

eq (Int x) (Int y) = (==) x y;
eq (Tree Leaf) (Tree Leaf) = True;
eq (Tree (Bin x l r)) (Tree (Bin y ll rr)) = and (eq x y) (and (eq
    l ll) (eq r rr));
eq (Tree Leaf) (Tree (Bin x l r)) = False;
eq (Tree (Bin x l r)) (Tree Leaf) = False;

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print True = emitStr "True" 0;
print False = emitStr "False" 0;

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));
```

```
main = print(eq tree1 tree2);

}
```

- `Tree` and `List`:

```
{

and False x = False;
and True x = x;

eq (Int x) (Int y) = (==) x y;
eq (Tree Leaf) (Tree Leaf) = True;
eq (Tree (Bin x l r)) (Tree (Bin y ll rr)) = and (eq x y) (and (eq
     l ll) (eq r rr));
eq (Tree Leaf) (Tree (Bin x l r)) = False;
eq (Tree (Bin x l r)) (Tree Leaf) = False;
eq (List Nil) (List Nil) = True;
eq (List (Cons x xs)) (List (Cons y ys)) = and (eq x y) (eq xs ys)
    ;
eq (List Nil) (List (Cons x xs)) = False;
eq (List (Cons x xs)) (List Nil) = False;

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print True = emitStr "True" 0;
print False = emitStr "False" 0;

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

main = print(eq list1 list1);

}
```

- `Tree`, `List` and `Rose`:

```
{

and False x = False;
and True x = x;

eq (Int x) (Int y) = (==) x y;
eq (Tree Leaf) (Tree Leaf) = True;
eq (Tree (Bin x l r)) (Tree (Bin y ll rr)) = and (eq x y) (and (eq
     l ll) (eq r rr));
eq (Tree Leaf) (Tree (Bin x l r)) = False;
eq (Tree (Bin x l r)) (Tree Leaf) = False;
eq (List Nil) (List Nil) = True;
eq (List (Cons x xs)) (List (Cons y ys)) = and (eq x y) (eq xs ys)
    ;
eq (List Nil) (List (Cons x xs)) = False;
eq (List (Cons x xs)) (List Nil) = False;
eq (Rose (Ros x r)) (Rose (Ros y rr)) = and (eq x y) (eq r rr);

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print True = emitStr "True" 0;
print False = emitStr "False" 0;

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));
```

```
list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

rose1 = Rose (Ros list1 (List (Cons (Rose (Ros list2 (List Nil)))
    (List Nil))));

main = print (eq rose1 rose1);

}
```

- `Tree`, `List`, `Rose` and `Triplet`:

```
{

and False x = False;
and True x = x;

eq (Int x) (Int y) = (==) x y;
eq (Tree Leaf) (Tree Leaf) = True;
eq (Tree (Bin x l r)) (Tree (Bin y ll rr)) = and (eq x y) (and (eq
    l ll) (eq r rr));
eq (Tree Leaf) (Tree (Bin x l r)) = False;
eq (Tree (Bin x l r)) (Tree Leaf) = False;
eq (List Nil) (List Nil) = True;
eq (List (Cons x xs)) (List (Cons y ys)) = and (eq x y) (eq xs ys)
    ;
eq (List Nil) (List (Cons x xs)) = False;
eq (List (Cons x xs)) (List Nil) = False;
eq (Rose (Ros x r)) (Rose (Ros y rr)) = and (eq x y) (eq r rr);
eq (Triplet (First x)) (Triplet (First y)) = eq x y;
eq (Triplet (First x)) (Triplet (Second y)) = False;
eq (Triplet (First x)) (Triplet (Third y)) = False;
eq (Triplet (Second x)) (Triplet (First y)) = False;
eq (Triplet (Second x)) (Triplet (Second y)) = eq x y;
eq (Triplet (Second x)) (Triplet (Third y)) = False;
eq (Triplet (Third x)) (Triplet (First y)) = False;
eq (Triplet (Third x)) (Triplet (Second y)) = False;
eq (Triplet (Third x)) (Triplet (Third y)) = eq x y;

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print True = emitStr "True" 0;
print False = emitStr "False" 0;

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

rose1 = Rose (Ros list1 (List (Cons (Rose (Ros list2 (List Nil)))
    (List Nil))));

triplet1 = Triplet (First rose1);
triplet2 = Triplet (Second list1);
triplet3 = Triplet (Third tree1);

main = print (eq triplet1 triplet1);

}
```

G-lite

- `Tree`:

```
{

and False x = False;
```

```
and True x = x;

eq (Int x) (Int y) = (==) x y;
eq UNIT UNIT = True;
eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b);
eq (LEFT x) (LEFT y) = eq x y;
eq (RIGHT x) (RIGHT y) = eq x y;
eq (LEFT x) (RIGHT y) = False;
eq (RIGHT x) (LEFT y) = False;

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print True = emitStr "True" 0;
print False = emitStr "False" 0;

Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic eq (Tree a) (Tree a) -> Bool

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

main = print(eq tree1 tree1);

}
```

- `Tree` and `List`:

```
{

and False x = False;
and True x = x;

eq (Int x) (Int y) = (==) x y;
eq UNIT UNIT = True;
eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b);
eq (LEFT x) (LEFT y) = eq x y;
eq (RIGHT x) (RIGHT y) = eq x y;
eq (LEFT x) (RIGHT y) = False;
eq (RIGHT x) (LEFT y) = False;

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print True = emitStr "True" 0;
print False = emitStr "False" 0;

Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic eq (Tree a) (Tree a) -> Bool

List a :: Nil | Cons a (List a)
Generic eq (List a) (List a) -> Bool

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

main = print(eq list1 list1);

}
```

- `Tree`, `List` and `Rose`:

```
{
```

```
and False x = False;
and True x = x;

eq (Int x) (Int y) = (==) x y;
eq UNIT UNIT = True;
eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b);
eq (LEFT x) (LEFT y) = eq x y;
eq (RIGHT x) (RIGHT y) = eq x y;
eq (LEFT x) (RIGHT y) = False;
eq (RIGHT x) (LEFT y) = False;

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print True = emitStr "True" 0;
print False = emitStr "False" 0;

Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic eq (Tree a) (Tree a) -> Bool

List a :: Nil | Cons a (List a)
Generic eq (List a) (List a) -> Bool

Rose a :: Ros a (List (Rose a))
Generic eq (Rose a) (Rose a) -> Bool

tree = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree Leaf
    ))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (Tree
    Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree (List Nil));
list2 = List (Cons tree2 (List Nil));

rose1 = Rose (Ros list1 (List (Cons (Rose (Ros list2 (List Nil)))
    (List Nil))));

main = print(eq rose1 rose1);

}
```

- `Tree`, `List`, `Rose` and `Triplet`:

```
{

and False x = False;
and True x = x;

eq (Int x) (Int y) = (==) x y;
eq UNIT UNIT = True;
eq (PAIR x y) (PAIR a b) = and (eq x a) (eq y b);
eq (LEFT x) (LEFT y) = eq x y;
eq (RIGHT x) (RIGHT y) = eq x y;
eq (LEFT x) (RIGHT y) = False;
eq (RIGHT x) (LEFT y) = False;

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print True = emitStr "True" 0;
print False = emitStr "False" 0;

Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic eq (Tree a) (Tree a) -> Bool

List a :: Nil | Cons a (List a)
Generic eq (List a) (List a) -> Bool

Rose a :: Ros a (List (Rose a))
Generic eq (Rose a) (Rose a) -> Bool

Triplet a b c :: First a | Second b | Third c
Generic eq (Triplet a b c) (Triplet a b c) -> Bool
```

```
tree = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree Leaf
    ))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (Tree
    Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree (List Nil));
list2 = List (Cons tree2 (List Nil));

rose1 = Rose (Ros list1 (List (Cons (Rose (Ros list2 (List Nil)))
    (List Nil))));

triplet1 = Triplet (First rose1)
triplet2 = Triplet (Second list1)
triplet3 = Triplet (Third tree1)

main = print(eq triplet1 triplet1);

}
```

13.2.2 Program 2: increment

F-lite

- `Tree`:

```
{

inc (Int x) = (Int ((+) x 1));
inc (Tree Leaf) = Tree Leaf;
inc (Tree (Bin x l r)) = Tree (Bin (inc x) (inc l) (inc r));

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print (Int x) k = emitInt x k;
print (Tree Leaf) k = k;
print (Tree (Bin x l r)) k = print x (emitStr " " (print l (
    emitStr " " (print r k))));

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

main = print (inc tree1) 0;

}
```

- `Tree` and `List`:

```
{

inc (Int x) = (Int ((+) x 1));
inc (List Nil) = List Nil;
inc (List (Cons x xs)) = List (Cons (inc x) (inc xs));
inc (Tree Leaf) = Tree Leaf;
inc (Tree (Bin x l r)) = Tree (Bin (inc x) (inc l) (inc r));

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print (Int x) k = emitInt x k;
print (List Nil) k = k;
print (List (Cons x xs)) k = print x (emitStr " " (print xs));
print (Tree Leaf) k = k;
print (Tree (Bin x l r)) k = print x (emitStr " " (print l (
    emitStr " " (print r k))));

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
```

```
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

main = print (inc list1) 0;

}
```

- `Tree`, `List` and `Rose`:

```
{

inc (Int x) = (Int ((+) x 1));
inc (List Nil) = List Nil;
inc (List (Cons x xs)) = List (Cons (inc x) (inc xs));
inc (Tree Leaf) = Tree Leaf;
inc (Tree (Bin x l r)) = Tree (Bin (inc x) (inc l) (inc r));
inc (Rose (Ros x r)) = Rose (Ros (inc x) (inc r));

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print (Int x) k = emitInt x k;
print (List Nil) k = k;
print (List (Cons x xs)) k = print x (emitStr " " (print xs k));
print (Tree Leaf) k = k;
print (Tree (Bin x l r)) k = print x (emitStr " " (print l (
    emitStr " " (print r k))));
print (Rose (Ros x r)) k = print x (emitStr " " (print r k));

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

rose1 = Rose (Ros list1 (List (Cons (Rose (Ros list2 (List Nil)))
    (List Nil))));

main = print (inc rose1) 0;

}
```

- `Tree`, `List`, `Rose` and `Triplet`:

```
{

inc (Int x) = (Int ((+) x 1));
inc (List Nil) = List Nil;
inc (List (Cons x xs)) = List (Cons (inc x) (inc xs));
inc (Tree Leaf) = Tree Leaf;
inc (Tree (Bin x l r)) = Tree (Bin (inc x) (inc l) (inc r));
inc (Rose (Ros x r)) = Rose (Ros (inc x) (inc r));
inc (Triplet (First x)) = First (inc x);
inc (Triplet (Second x)) = Second (inc x);
inc (Triplet (Third x)) = Third (inc x);

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print (Int x) k = emitInt x k;
print (List Nil) k = k;
print (List (Cons x xs)) k = print x (emitStr " " (print xs k));
print (Tree Leaf) k = k;
print (Tree (Bin x l r)) k = print x (emitStr " " (print l (
    emitStr " " (print r k))));
print (Rose (Ros x r)) k = print x (emitStr " " (print r k));
print (First x) k = print x k;
```

```
print (Second x) k = print x k;
print (Third x) k = print x k;

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

rose1 = Rose (Ros list1 (List (Cons (Rose (Ros list2 (List Nil)))
    (List Nil))));

triplet1 = Triplet (First rose1);
triplet2 = Triplet (Second list1);
triplet3 = Triplet (Third tree1);

main = print (inc triplet1) 0;

}
```

G-lite

- `Tree`:

```
{

inc (Int x) = (Int ((+) x 1));
inc UNIT = UNIT;
inc (PAIR x y) = PAIR (inc x) (inc y);
inc (LEFT x) = LEFT (inc x);
inc (RIGHT x) = RIGHT (inc x);

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print (Int x) k = emitInt x k;
print UNIT k = k;
print (PAIR x y) k = print x (emitStr " " (print y k));
print (LEFT x) k = print x k;
print (RIGHT x) k = print x k;

Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic inc (Tree a) -> (Tree a)
Generic print (Tree a) Int -> Int

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

main = print (inc tree1) 0;

}
```

- `Tree` and `List`:

```
{

inc (Int x) = (Int ((+) x 1));
inc UNIT = UNIT;
inc (PAIR x y) = PAIR (inc x) (inc y);
inc (LEFT x) = LEFT (inc x);
inc (RIGHT x) = RIGHT (inc x);

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print (Int x) k = emitInt x k;
print UNIT k = k;
```

```
print (PAIR x y) k = print x (emitStr " " (print y k));
print (LEFT x) k = print x k;
print (RIGHT x) k = print x k;

Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic inc (Tree a) -> (Tree a)
Generic print (Tree a) Int -> Int

List a :: Nil | Cons a (List a)
Generic inc (List a) -> (List a)
Generic print (List a) Int -> Int

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

main = print (inc list1) 0;

}
```

- `Tree`, `List` and `Rose`:

```
{

inc (Int x) = (Int ((+) x 1));
inc UNIT = UNIT;
inc (PAIR x y) = PAIR (inc x) (inc y);
inc (LEFT x) = LEFT (inc x);
inc (RIGHT x) = RIGHT (inc x);

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print (Int x) k = emitInt x k;
print UNIT k = k;
print (PAIR x y) k = print x (emitStr " " (print y k));
print (LEFT x) k = print x k;
print (RIGHT x) k = print x k;

Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic inc (Tree a) -> (Tree a)
Generic print (Tree a) Int -> Int

List a :: Nil | Cons a (List a)
Generic inc (List a) -> (List a)
Generic print (List a) Int -> Int

Rose a :: Ros a (List (Rose a))
Generic inc (Rose a) -> (Rose a)
Generic print (Rose a) Int -> Int

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (
    Tree Leaf)(Tree Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree
    Leaf))) (Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

rose1 = Rose (Ros list1 (List (Cons (Rose (Ros list2 (List Nil)))
    (List Nil))));

main = print (inc rose1) 0;

}
```

- `Tree`, `List`, `Rose` and `Triplet`:

```
{

inc (Int x) = (Int ((+) x 1));
inc UNIT = UNIT;
inc (PAIR x y) = PAIR (inc x) (inc y);
inc (LEFT x) = LEFT (inc x);
inc (RIGHT x) = RIGHT (inc x);

emitStr Nil k = k;
emitStr (Cons x xs) k = emit x (emitStr xs k);

print (Int x) k = emitInt x k;
print UNIT k = k;
print (PAIR x y) k = print x (emitStr " " (print y k));
print (LEFT x) k = print x k;
print (RIGHT x) k = print x k;

Tree a :: Leaf | Bin a (Tree a) (Tree a)
Generic inc (Tree a) -> (Tree a)
Generic print (Tree a) Int -> Int

List a :: Nil | Cons a (List a)
Generic inc (List a) -> (List a)
Generic print (List a) Int -> Int

Rose a :: Ros a (List (Rose a))
Generic inc (Rose a) -> (Rose a)
Generic print (Rose a) Int -> Int

Triplet a b c :: First a | Second b | Third c
Generic inc (Triplet a b c) -> (Triplet a b c)
Generic print (Triplet a b c) Int -> Int

tree1 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree Leaf))) (
    Tree (Bin (Int 6) (Tree Leaf) (Tree (Bin (Int 7) (Tree Leaf)(Tree
    Leaf))))));
tree2 = Tree (Bin (Int 5) (Tree (Bin (Int 4) (Tree Leaf) (Tree Leaf))) (
    Tree Leaf));

list1 = List (Cons tree1 (List Nil));
list2 = List (Cons tree2 (List Nil));

rose1 = Rose (Ros list1 (List (Cons (Rose (Ros list2 (List Nil))) (List
    Nil))));

triplet1 = Triplet (First rose1);
triplet2 = Triplet (Second list1);
triplet3 = Triplet (Third tree1);

main = print (inc triplet1) 0;

}
```

Clean

- Tree:

```
module IncTreeListRoseTriplet

import StdEnv

class inc` a :: a -> a
instance inc` Int
where
    inc` x = x + 1
instance inc` (Tree a) | inc` a
where
    inc` Leaf = Leaf
    inc` (Bin x l r) = Bin (inc` x) (inc` l) (inc` r)

:: Tree a = Leaf | Bin a (Tree a) (Tree a)

Start :: Tree Int
Start = inc` (Bin 1 (Bin 2 Leaf Leaf) Leaf)
```

- `Tree` and `List`:

```
module IncTreeListRoseTriplet

import StdEnv

class inc` a :: a -> a
instance inc` Int
where
    inc` x = x + 1
instance inc` (Tree a) | inc` a
where
    inc` Leaf = Leaf
    inc` (Bin x l r) = Bin (inc` x) (inc` l) (inc` r)
instance inc` (List a) | inc` a
where
    inc` Nil = Nil
    inc` (Cons x xs) = Cons (inc` x) (inc` xs)

:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)

Start :: List Int
Start = inc` (Cons 1 (Cons 2 (Cons 3 Nil)))
```

- `Tree`, `List` and `Rose`:

```
module IncTreeListRoseTriplet

import StdEnv

class inc` a :: a -> a
instance inc` Int
where
    inc` x = x + 1
instance inc` (Tree a) | inc` a
where
    inc` Leaf = Leaf
    inc` (Bin x l r) = Bin (inc` x) (inc` l) (inc` r)
instance inc` (List a) | inc` a
where
    inc` Nil = Nil
    inc` (Cons x xs) = Cons (inc` x) (inc` xs)
instance inc` (Rose a) | inc` a
where
    inc` (Rose x xs) = Rose (inc` x) (inc` xs)

:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)
:: Rose a = Rose a (List (Rose a))

Start :: List Int
Start = inc` (Cons 1 (Cons 2 (Cons 3 Nil)))
```

- `Tree`, `List`, `Rose` and `Triplet`:

```
module IncTreeListRoseTriplet

import StdEnv

class inc` a :: a -> a
instance inc` Int
where
    inc` x = x + 1
instance inc` (Tree a) | inc` a
where
    inc` Leaf = Leaf
    inc` (Bin x l r) = Bin (inc` x) (inc` l) (inc` r)
instance inc` (List a) | inc` a
where
    inc` Nil = Nil
    inc` (Cons x xs) = Cons (inc` x) (inc` xs)
instance inc` (Rose a) | inc` a
where
```

```
    inc` (Rose x xs) = Rose (inc` x) (inc` xs)
instance inc` (Triplet a b c) | inc` a & inc` b & inc` c
where
    inc` (First x) = First (inc` x)
    inc` (Second x) = Second (inc` x)
    inc` (Third x) = Third (inc` x)

:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)
:: Rose a = Rose a (List (Rose a))
:: Triplet a b c = First a | Second b | Third c

Start :: List Int
Start = inc` (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Clean with generics

- `Tree`:

```
module IncTreeListRoseTripletGeneric

import StdEnv
import StdGeneric

generic gInc a :: a -> a

gInc{|UNIT|} UNIT = UNIT
gInc{|PAIR|} inca incb (PAIR x y) = PAIR (inca x) (incb y)
gInc{|EITHER|} inca incb (LEFT x) = LEFT (inca x)
gInc{|EITHER|} inca incb (RIGHT x) = RIGHT (incb x)
gInc{|CONS|} inca (CONS x) = CONS (inca x)
gInc{|OBJECT|} inca (OBJECT x) = OBJECT (inca x)
gInc{|Int|} x = x + 1

derive gInc Tree

:: Tree a = Leaf | Bin a (Tree a) (Tree a)
Start :: List Int
Start = gInc{|*|} (Cons 1 (Cons 2 (Cons 3 Nil)))
```

- `Tree` and `List`:

```
module IncTreeListRoseTripletGeneric

import StdEnv
import StdGeneric

generic gInc a :: a -> a

gInc{|UNIT|} UNIT = UNIT
gInc{|PAIR|} inca incb (PAIR x y) = PAIR (inca x) (incb y)
gInc{|EITHER|} inca incb (LEFT x) = LEFT (inca x)
gInc{|EITHER|} inca incb (RIGHT x) = RIGHT (incb x)
gInc{|CONS|} inca (CONS x) = CONS (inca x)
gInc{|OBJECT|} inca (OBJECT x) = OBJECT (inca x)
gInc{|Int|} x = x + 1

derive gInc Tree, List

:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)

Start :: List Int
Start = gInc{|*|} (Cons 1 (Cons 2 (Cons 3 Nil)))
```

- `Tree`, `List` and `Rose`:

```
module IncTreeListRoseTripletGeneric

import StdEnv
import StdGeneric

generic gInc a :: a -> a
```

```
gInc{|UNIT|} UNIT = UNIT
gInc{|PAIR|} inca incb (PAIR x y) = PAIR (inca x) (incb y)
gInc{|EITHER|} inca incb (LEFT x) = LEFT (inca x)
gInc{|EITHER|} inca incb (RIGHT x) = RIGHT (incb x)
gInc{|CONS|} inca (CONS x) = CONS (inca x)
gInc{|OBJECT|} inca (OBJECT x) = OBJECT (inca x)
gInc{|Int|} x = x + 1

derive gInc Tree, List, Rose

:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)
:: Rose a = Rose a (List (Rose a))

Start :: List Int
Start = gInc{|*|} (Cons 1 (Cons 2 (Cons 3 Nil)))
```

- `Tree`, `List`, `Rose` and `Triplet`:

```
module IncTreeListRoseTripletGeneric

import StdEnv
import StdGeneric

generic gInc a :: a -> a

gInc{|UNIT|} UNIT = UNIT
gInc{|PAIR|} inca incb (PAIR x y) = PAIR (inca x) (incb y)
gInc{|EITHER|} inca incb (LEFT x) = LEFT (inca x)
gInc{|EITHER|} inca incb (RIGHT x) = RIGHT (incb x)
gInc{|CONS|} inca (CONS x) = CONS (inca x)
gInc{|OBJECT|} inca (OBJECT x) = OBJECT (inca x)
gInc{|Int|} x = x + 1

derive gInc Tree, List, Rose, Triplet

:: Tree a = Leaf | Bin a (Tree a) (Tree a)
:: List a = Nil | Cons a (List a)
:: Rose a = Rose a (List (Rose a))
:: Triplet a b c = First a | Second b | Third c

Start :: List Int
Start = gInc{|*|} (Cons 1 (Cons 2 (Cons 3 Nil)))
```