Chapter 1

Generic Functions

In this thesis, “generic function” refers specifically to one of the functions defined in this chapter. Each of these functions is able to do its job on any data structure, in fact on any input at all. They are also able to do this without being enumerated for each type, or even each type constructor (i.e. they are not overloaded). The term “generic” has a much wider meaning outside this thesis (which we discuss in section 1.7, in the thesis its meaning is restricted to these six types of functions.

For each of these function we give a general description and then show an encoding of it in our concrete language DGEN. In this way we can introduce *both* the generic functions and our language.

The subject of this thesis is a *compiler*, not a programming language. The concrete langauge we present here (DGEN) is the simplest possible concrete language corresponding to our abstract language SOURCE (see section **??**). It exists only to allow one to experiment with the compiler (www.something.net.au) and to describe algorithims/programs without resorting to abstract syntax.

1.1 dgen

DGEN is the name we have given to our compiler implementation and is also the name of the concrete programming language corresponding to an abstract syntax, SOURCE, that sits at the front of our compiler. SOURCE would be the target of a desugarring phase in, for example, a Haskell or ML compiler based on our technques, so DGEN looks like a desugarred version of those laguages. Rather than describe DGEN in detail, we will take as our starting point the intersection of ML and Haskell98 and describe where DGEN differs from that hypothetical programming langauge. We will start with the abstract syntax of DGEN(which we call SOURCE), to make clearer what we mean by “intersection of Haskell98 and ML” and to separate meaning from syntax. Where appropriate, we will use commonly-understood functional programs to demonstrate DGEN. Figure 1.1 gives our starting point, the abstract syntax of DGEN, SOURCE.

SOURCE has all the usual functional programming features. Data is described as algebraic datatypes (`adt`). Top-level function definitions bind an expression to a name. Anonymous funcitons are defined by lambda abstractions with normal function application for evaluating functions. There are `let` and `letrec` expressions for local function naming. SOURCE also has the usual literal values and an error term ($\varnothing$) which halts execution. Case expressions pull apart data and operate similarly to those

$$
\begin{array}{rcll}
s & ::= & \vec{sd} & \text{(}\textsc{Source}\text{ program)} \\
sd & ::= & \texttt{adt}\ K\ \vec{x}\ \texttt{=}\ \vec{sk} & \text{(top-level definition)} \\
 & | & \texttt{def}\ x\ ot\ \texttt{=}\ se & \\
 & | & \texttt{main}\ meq\ se & \\
sk & ::= & K\ (sk \mid x) & \text{(constructor definition)} \\
se & ::= & x & \text{(expression)} \\
 & | & K\ \vec{se} & \\
 & | & se\ se' & \\
 & | & \lambda x \to se & \\
 & | & \texttt{let}\ x\ \overrightarrow{ot\,\texttt{=}}\ se\ \texttt{in}\ se' & \\
 & | & \texttt{letrec}\ x\ \overrightarrow{ot\,\texttt{=}}\ se\ \texttt{in}\ se' & \\
 & | & \texttt{case}\ \vec{x}\ \texttt{of}\ sp\ \overrightarrow{\to}\ se\ \texttt{otherwise}\ se' & \\
 & | & l & \\
 & | & \varnothing & \\
sp & ::= & x & \text{(pattern)} \\
 & | & l & \\
 & | & K\ \vec{sp} & \\
l & ::= & char & \text{(literal)} \\
 & | & number & \\
 & | & se\ (+ \mid - \mid = \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant)\ se & \\
ot & ::= & \tau \mid \epsilon & \text{(optional type annotation)}
\end{array}
$$

Figure 1.1: Source, the abstract syntax of dgen

$$
\begin{array}{llr}
\tau ::= \alpha & & \text{(type variable)} \\
\quad \mid \; T\ \vec{\tau} & & \text{(paramaterised type)} \\
\quad \mid \; \text{Char} \mid \text{Integer} & & \text{(literal types)} \\
\quad \mid \; \forall \vec{\alpha}.\tau & &
\end{array}
$$

Figure 1.2: Types in Source

in Haskell and ML with the exception that each case expression matches a number of values against an equal number of patterns. A successful match only occurs if *all* values match their corresponding pattern. A pattern is either a variable, literal value (not a literal operation although the above grammar allows that), a constructor with patterns as arguments (nested patterns) or *application pattern matches* of two variables which is one of our extensions for generic functions. You will also notice a primitive operation $\triangleright$ which is our *function extension operation*, also used for generic functions and not usually part of a functional langauge. We will discuss these mechanisms in more detail in chapter **??**.

Named functions can be annotated with a type (*ot*) which is used to guide type inference for features which can't be inferred. Figure 1.2 describes the langauge of types in Source.

Types are either variables, named types (which can be parameterised by other types), one of the literal types or a quantified type (which are used for rank-2 types and internally in the type inference system).

The concrete syntax (dgen) is the simplest realisation of this abstract syntax that we could create. We will introduce the concrete syntax (and make clearer the semantics) by way of some familiar programming tasks. Listing 1.1 shows dgen code for calculating the extracting the head of a list, including the definition of the list type.

Listing 1.1: head function on lists

```
adt list(a) = Nil() | Cons(a, list(a))

def head(lst)= case [lst] of
                    { [Cons(x,xs)] → x
                    } otherwise    → error "partial definition error in head"
```

The `adt` keyword defines a new datatype which is defined in the normal algebraic datatype fasion, the `def` keyword defines a new function. There is a dinstinguisehd function `main` which defines the function to run on program execution. Errors in dgen carry a string to emit on the console upon the program halting. Listing 1.2 shows a merge sort program which demonstrates more features.

There are no modules, but a c-like preprocessor supports importing definitions from other files. Recusive let bindings are introduced with the `let rec` keyword while `let` introduces non-recursive bindings. Such bindings are separated with the keyword `and`. All functions and constructors take their arguments in parentheses,

Listing 1.2: merge sort

```
#include "dgen_lib/std.dgen"

def cmp_less() = -1
def cmp_greater() = 1
def cmp_equal() = 0

def merge_cmp(cmp,y,z) = case [y,z] of
          { [Nil(), y] → y
          ; [Cons(a,x),Nil()] → Cons(a,x)
          ; [Cons(a,x),Cons(aa,xx)] → if (cmp(a,aa) < cmp_equal | cmp(a,aa) i== cmp_equal)
                                      then Cons(a,merge_cmp(cmp,x,Cons(aa,xx)))
                                      else Cons(aa,merge_cmp(cmp,Cons(a,x),xx))
          } otherwise → error "partial definition error in merge_cmp"

def merge_sort(cmp,x) = letrec n() = length(x)
                        and nn() = n / 2
                        in  if (n < 1 | n i== 1)
                               then x
                               else merge_cmp( cmp
                                             , merge_sort( cmp
                                                         , take(nn,x)
                                                         )
                                             , merge_sort( cmp
                                                         , drop(nn,x)
                                                         )
                                             )
```

separated by commas and empty agrument lists must be given with empty parentheses. Functions and constructors can be "curried" by giving only some of the arguments. Function application to arguments is done with this parameter list syntax but DGEN also allows constructor appliction to arguments when the (partially applied) constructor is the result of some function. In this case the application uses function application syntax except that the @ operator is added to the front of the application. Anonymous functions (lambdas) are introduced with the `fun` keyword. `case` expressions branch on a *list* of expressions (which we call the scruitinees), rather than a single expression. There is thus a corresponding list of patterns in each branch of the case expression. This allows desugaring of the equational style of function definition and smooths the definition of pattern compilation (section **??**). Each `case` expression has a distinguished default branch, identified by the keyword `otherwise` which is the result if none of the branches match. `case` branches are listed in curley braces separated by semi-colons. DGEN has list syntax using square braces and commas, which is the only desuggaring done in the parser (it converts it to a list type defined in the DGEN standard libarary). DGENis passes parameters by value and has no side-effect-causing built-in operations, thus no side-effects at all.

1.2 Generic Update

A generic update function is one that can traverse any data structure and perform a type-preserving transformation at specific nodes in the structure. It can be considered a type-preserving map capable of operating on any structure. Both the structure and the operation are parameters to the generic update function. Throughout this thesis we use the *salary update snippet* to demonstrate this type of function.

The salary update snippet (shown in listing 1.3, which is a translation of an example from [**?**]) defines an algebraic datatype for company structures, and code for updating the salary of all employees of the company. Both the particular company structure to work over, and the updating function are arguments to the generic update function, `generic_update`.

Listing 1.3: The salary update snippet

```
adt company()   = C(list(dept()))
adt dept()      = D(string, manager(), list(sub_unit()))
adt sub_unit()  = PU(employee())
                | DU(dept())
adt employee()  = E(person(), salary())
adt person()    = P(string, string)
adt manager()   = M(employee)
adt salary()    = S(int)

// setup
def gen_com() = let ralf()   = E(P("Ralf", "Amsterdam"), S(8000))
                and joost()  = E(P("Joost", "Amsterdam"), S(1000))
                and marlow() = E(P("Marlow", "Cambridge"), S(2000))
                and blair()  = E(P("Blair", "London"), S(100000))
                in  C([ D("Research", M(ralf), [PU(joost), PU(marlow)])
                      , D("Strategy", M(blair), [])
                      ]
                     )

// logic
def incS(amt, s) = case [s] of
                     { [S(s)]    → S(s + amt)
                     } otherwise → error "partial definition error in incS"
def id(x) = x
def increment(amt) = incS(amt) ▷ id
def generic_update(func, dat) = apply_to_all(func,dat)

//do it
def main_func() = generic_update(increment(237),gen_com)

// > output is:
// > Company[ Department: Research, Ralf<Amsterdam, 8237>,
// >                               [ Joost<Amsterdam, 1237>
// >                               , Marlow<Cambridge, 2237>
// >                               ]
```

`generic_update` actually defers its job to one of our traversal functions, `apply_to_all`, which is discussed when we show our generic traversal functions (in section 1.4).

Given that `apply_to_all` takes a function and applies to each part of the input data, our job is to construct a function to work at each part of the input data. `apply_to_all` has the type `(∀a . a →a) →b →b` meaning this function must have type `∀ a . a →a`. Usually there is only one function with that type, the identity function (`x→x`). We use an *extension* operator (▷) to add specific behaviour to the identify function. the resulting function will give the specific behaviour on input of the specific type, while still deferring to the identity behaviour for all other types. Thus, `increment` is a function which will increment a salary if applied to one and do nothing to its input otherwise. It is this function that we set to be applied to all nodes in the data structure in question. Upon running `generic_update` with `increment(237)` and the example company structure, the result in the same company structure with every salary increase by 237.

1.3 Generic Query

A generic query function is able to traverse any data structure, accumulating a single value which is its result. It can be considered a fold that can operate on any data structure. Both the structure and the accumulation operation are parameters to the function, but the same mechanism can be used to define functions with a set accumulator. Throughout this thesis we use the *name analysis snippet* to demonstrate this type of function.

The name analysis snippet includes the definition of an abstract syntax tree datatype (adapted from a simple imperative language by Reynolds [**?**]), and code to check that every *use* of a name is preceded by the *definition* of that name in values of that datatype. The function of most interest is the `generic_query` function that does the hard work. The `check_it` function is the accumulating operation, and the `a_correct_command` gives some data to test the function on.

We assume the presence of a `generic_query` function, which we describe in section 1.4. It is like a fold over any data structure. Its type signature is `(∀a. r →a →r) →r →b →r` and operates thus.

> This function takes an accumulating (or *folding*) operation and a starting value as its first two parameters. The third parameter is the data to work over and the result is the final result of applying to accumulating function at every node in the input value.

With this function present, we can then define an actual generic update example by defining the accumulating function only. As in generic traversal, this function needs a very polymorphic type since it is applied at every node, and again we build it up with the extension mechanism. We define separate functions for each *interesting* data type (i.e for `comm` and `int_exp`, but not for `bool_exp` since it plays no part in the computation) and combine them with a generic failover[1] case (`fun(a) = strbool` on line 37). The resulting function, `check_it` will use `check_comm` when it encounters a command node, `check_intexp` when it encounters an integer expression and the failover case for all other nodes.

1.4 Generic Traversal

A generic traversal function is either a generic query or a generic update where the traversal strategy is defined. For example, if we have only one generic update function, we can only traverse a tree in one way. If we are able to define different generic update functions, such as `generic_top_down` and `generic_bottom_up`, we can customise our traversal to suit our purposes. To demonstrate generic traversal we will encode three traversal strategies for both queries and updates; top-down,

[1] I like the term "failover" for this function since it is a function that protects us from unsafe generic traversal by stepping in when all other options fail.

bottom-up and once. The same techniques used to define them can be used to define other traversal strategies.

It is possible to define a set of generic traversal operations up-front and to require your programmer to use one of these for all generic updates and generic queries. This can work quite well in practice, but is not what we mean when we say "generic traversal". When we use this term we are referring to programmer-generated generic traversals, and the ability for the programmer to create custom traversals that suit their purpose. For example, our name analysis snippet can only work if the `generic_query` operation goes top-down, applying the accumulating function to nodes higher in the value before passing the result to those lower in the value. This is not a universal requirement and it may be necessary at some time to have a similar function that works from the bottom up. With generic traversal, the programmer is free to create whatever traversal they need.

1.4.1 Bottom-Up Update

`apply_to_all`, as used in the salary update snippet, is actually a bottom-up traversal, its full definition is shows in listing 1.5

`apply_to_all` has two arguments, a function to apply at every node (`f`), and a value to traverse (`g`). It works by inspecting its argument (`case [g]`) and branching based on whether it is a *compound* (`c(a)`) or a *atom* (`o`). This relies on the ability to see all values in the language as binary tree, compounds are the internal nodes and atoms are the leaves. We describe how to do this in section ??, but for now we must take as an assumption that it works.

With this in place, we can apply `f` to all nodes by applying it to the current node and recursively applying `apply_to_all` to each child (`c` is the left child and a is the right). We then stitch the two transformed halves back together with the @ annotation, which tell us that the following expression is a data reconstruction, not a function application.

`apply_to_all` is bottom-up, which is encoded by applying the function `f` to the result of stitching together the transformed children. We show how to encode a top-down traversal in section 1.4.2.

1.4.2 Top-Down Update

It is quite straightforward to write a version of `generic_update` which processes its second argument from the top-down instead of from the bottom up. Listing 1.6 shows just such a function.

The general mechanism is the same but we first call the transformation function (`f`) on the current node and then recursively call `apply_to_all_td` on the result of that function call. We must be careful *not* to re-apply `f` when the result of calling it on the current node is an atom.

1.4.3 Top Down Query

`generic_query` is actually a *top-down* generic query function. The accumulation function will first be applied to the node being inspected, then the result is threaded to the right-hand argument, finally to the left-hand argument. So in fact, it is top-down, right-to-left. In section **??** we show a bottom-up, left-to-right version.

As with `apply_to_all`, we rely on the ability to see any value as a tree, meaning our job is to apply the accumulator at the right places and to thread the output of the accumulator to the right recursive calls in the right order. We first apply to accumulator to the whole value at the current node, then use the result of this as the start value for a recursive call on the right sub-tree, finally passing that result to the start parameter of a recursive call for the left sub-tree.

1.4.4 Bottom Up Query

We can easily create a bottom-up version of `generic_query` by changing the places at which we call the accumulator function. Listing 1.8 shows just such a function. Instead of first calling the accumulator on the current node, we defer that job until the left and right subtrees are processed, passing the final result of the right subtree as the start value for the current node.

1.5 Generic Equality

Generic equality is a single function which can determine the equality of two values of any type. It must be restricted to take arguments of the same type and it must return a boolean indicating if they are equal or not. The `geq` function in listing 1.9 is a generic equality function.

Unfortunately, this snippet suffers from noise caused by our unsophisticated parser, but the verbose encoding does not infringe on clarity. First, we must define a sequence of generic versions of the built-in equality functions, each able to deal with one more of the built-in types. The first of these, `g_str_equals`, can deal with either two strings (by the left argument to extension, or two constructors (by the right argument). This *constructor* equality (===) is the secret to writing this function. The pulling apart of data which we did in generic traversal means that constructors can be exposed without their arguments. If we add some basic built-in functions that work on these "lonely" constructors, we can write functions like equality (and the next two, show and bitstring). In this process, the constructor equality function is used as the right hand argument to an extension operator, which means it must have type `(a,a)` $\rightarrow$`bool`. It will give a run-time error for non-constructor input. We then build this up through the next three definitions until we have a function with extensions for all built-in types plus constructors. This function (`bi_eq`) is used to check the equality of atoms, and all other equalilty checks are done by the main `geq` function. `geq` inspects its two arguments and, if they are the same strucure, either recurses on the branches for compound inputs, calls the atomic equality for atom inputs. It gives fail for inputs of different structure.

This one function is capable of testing the equality of any two values (of the same type, notice the type signature is `(a,a)` $\rightarrow$`bool)` in the langauge.

1.6 Generic Show

Generic show is a single function which can encode, as a string, any value of any type. Listing 1.10 is a code snippet demonstrating this function.

We use exactly the same mechanism in `gshow` as we did in `geq`. The main difference is that we need a *different* built-in function working on constructors. In this case we use `show_const` which will convert any "lonely" constructor to a string representation, and will give a run-time error otherwise. Thus it, as with `===`, has a very polymorphic type (`(a)` $\rightarrow$`String`), which means it can be used in the right hand side of the extension operator ($\triangleright$).

1.7 Other "Generic" Functions

"Generic" is a heavily overloaded term in computing, in this section we further clarify how we use it in this thesis by explaining what it is *not* in this context.

1.7.1 Object Oriented Generics

The generic functions of Object Oriented (OO) languages like Java and C# bear no relation to the generic functions we have listed above. This thesis starts from the assumption that the parametric polymorphism that these mechanisms implement is already available as a starting point. The functions above are all from a higher-level form of generics.

1.7.2 Term-Rewriting

Term-rewriting libraries for functional programming languages are sometimes referred to as generics. Generic update and generic query with generic traversal gives up term-rewriting ability. So our use of generics is compatible with that used in these libraries.

1.7.3 Datatype Generics

Datatype generics is a more specifically defined term, characterised by a set of canonical examples. All our six of our functions are included in this definition. There are also other functions in this set, such as generic map and generic read, which we do not include. We discuss *these* functions in section **??**.

Listing 1.4: The name analysis snippet

```
adt comm() = CAssign(string, int_exp())
           | CDecl(string, int_exp(), comm())
           | CSkip()
           | CSeq(comm(), comm())
           | CWhile(bool_exp(), comm())
           | CPut(int_exp())

adt int_exp() = IUse(string)
              | ILit(int)
              | IPlus(int_exp(), int_exp())

adt bool_exp() = BTrue()
               | BFalse()
               | BEq(int_exp(), int_exp())
               | BNEq(int_exp(), int_exp())
               | BAnd(bool_exp(), bool_exp())
               | BOr(bool_exp(), bool_exp())

def check_comm(strbool,comm)
  :: (pair(list(string),bool),comm()) → pair(list(string),bool)
  = case [strbool,comm] of
         { [Pair(lst,b), CAssign(s, ie)] → Pair(lst,elem(fun(p,q) = p s== q, s,lst) & b)
         ; [Pair(lst,b), CDecl(s,ie,c)]  → Pair(Cons(s,lst), b)
         ; [Pair(lst,b), z]              → strbool
         } otherwise        → error "partial definition error in check_comm"

def check_intexp(strbool,comm)
  :: (pair(list(string),bool),int_exp()) → pair(list(string),bool)
  = case [strbool,comm] of
         { [Pair(lst,b), IUse(s)] → Pair(lst,elem(fun(p,q) = p s==q, s,lst) & b)
         ; [Pair(lst,b), z]       → strbool
         } otherwise              → error "partial definition error in idbu"

def check_it(strbool)  :: (pair(list(string),bool), a) → pair(list(string),bool)
                       = check_comm(strbool) ▷ check_intexp(strbool) ▷
fun(a) = strbool

def decl_before_use(comm) = snd(generic_query(check_it,Pair([],true),comm))

def a_correct_command() = CDecl( "v"
                               , ILit(1)
                               , CWhile( BNEq(IUse("v"), ILit(3))
                                       , CSeq( CAssign("v",
IPlus(IUse("v"),ILit(1)))
                                             , CPut(IUse("v"))
                                             )
                                       )
                               )

def main_func() = (show_bool(decl_before_use(a_correct_command)))

// > output is:
// > true
```

Listing 1.5: A bottom-up generic traversal

```
                      {[c(a)]     → false
                      } otherwise → true

/**
 * Type preserving at every node
```

Listing 1.6: Top-down update

```
def apply_to_all(f,g) :: (∀ a . (a) → a, b) → b =
                         case [g] of
                              { [c(a)]    → f(@apply_to_all(f,c)(apply_to_all(f,a)))
                              ; [o]       → f(o)
                              } otherwise → error "partial definition error in apply_to_all"
```

Listing 1.7: A top-down (right-to-left) generic query

```
                             { [c(a)]    → generic_all(f,c) & f(a)
                             ; [o]       → true
                             } otherwise → error "partial definition error in generic_all"

def generic_some_argsonly(f,g) :: (∀ a . (a) → bool, b) → bool =
                         case [g] of
```

Listing 1.8: A bottom-up (left-to-right) generic query

```
                                ; [o]       → false
                                } otherwise → error "partial definition error in generic_all"

/**
 * Generic queries (topdown and bottom up)
 **/
```

Listing 1.9: Generic Equality

```
def g_str_equals(b) :: (a,a) → Bool
                    = let str_equals(a,b)  = a s== b
                      in  str_equals(b) ▷ fun(g) = b === g

def g_int_equals(b) :: (a,a) → Bool
                     = let int_equals(a,b)  = a i== b
                       in  int_equals(b) ▷ g_str_equals(b)

def g_char_equals(b) :: (a,a) → Bool
                      = let char_equals(a,b)  = a c== b
                        in  char_equals(b) ▷ g_int_equals(b)

def g_bool_equals(b) :: (a,a) → Bool
                      = let bool_equals(a,b)  = a b== b
                        in  bool_equals(b) ▷ g_char_equals(b)

def bi_eq(x,y) :: (a,a) → Bool
               = let fst_part() = g_bool_equals(x)
                 in  fst_part(y)

def geq(a,b) :: (a,a) → bool = case [a,b] of
                                     { [c1(a1),c2(a2)] → geq(c1,c2) & geq(a1,a2)
                                     ; [c1(a2),z2]     → false
                                     ; [z1,c2(a2)]     → false
                                     ; [z1,z2]         → bi_eq(z1, z2)
                                     } otherwise       →
error "partial definition error in geq"

def main_func() = geq([One(),Zero()],[One(),One()])
```

Listing 1.10: Generic Show

```
def gshow(a) :: (a) → String
             = case [a] of
                    { [c(p)] → gshow(c) ++ "(" ++ gshow(p) ++ ")"
                    ; [z]    → bishow(z)
                    } otherwise → error "partial definition error in gshow"

def bishow() :: (a) → String
             = let si(x) = show_int(x)
               and sc(x) = show_char(x)
               and sb(x) = show_bool(x)
               and ss(x) = x
               and ds(x) = show_constr(x)
               in  si ▷ sc ▷ sb ▷ ss ▷ ds
```