A Tutorial on Programming Features in ATS

Hongwei Xi

<*gmhwxiATgmailDOTcom*>

Table of Contents

Preface

This tutorial covers a variety of issues that a programmer typically encounters when programming in ATS. It is written mostly in the style of learn-by-examples. Although it is possible to learn programming in ATS by studying the tutorial (if the reader is familiar with ML and C), I consider the book ***Introduction to Programming in ATS*** a far more approriate source for someone to acquire a view of ATS in a coherent and systematic manner. Of course, there will also be considerable amount of overlapping between these two books. The primary purpose of the tutorial is to bring more insights into a rich set of programming features in ATS and also demonstrate through concrete examples that these features can be made of effective use in the construction of high-quality programs.

I. Basic Tutorial Topics

Table of Contents

Chapter 1. Syntax-Coloring for ATS code

The syntax of ATS is highly involved, which can be a daunting obstacle for beginners trying to read and write ATS code. In order to alleviate this problem, I may employ colors to differentiate various syntactical entities in ATS code. The convention I adopt for coloring ATS syntax is given as follows:

- The keywords in ATS are all colored black (and possibly written in bold face).
- The comments in ATS are all colored gray.
- The code in the statics of ATS is colored blue.
- The code in the dynamics of ATS is colored red unless it represents proofs, for which the color dark green is used.
- The external code (in C) is colored deep brown.

Please find an example of ATS code ***on-line*** that involves all of the syntax-coloring mentioned above.

Chapter 2. Filename Extensions

In ATS, the filename extensions *.sats* and *.dats* are reserved to indicate static and dynamic files, respectively. As these two extensions have some special meaning attached to them, which can be interpreted by the command **atscc**, they should not be replaced arbitrarily.

A static file may contain sort definitions, datasort declarations, static definitions, abstract type declarations, exception declarations, datatype declarations, macro definitions, interfaces for dynamic values and functions, etc. In terms of functionality, a static file in ATS is somewhat similar to a header file (with the extension *.h*) in C or an interface file (with the extension *.mli*) in OCaml.

A dynamic file may contain everything in a static file. In addition, it may also contain definitions for dynamic values and functions. However, interfaces for functions and values in a dynamic file should follow the keyword `extern`, which on the other hand should not be present when such interfaces are declared in a static file. For instance, the following syntax declares interfaces (or types) in a static file for a value named `pi` and a function named `area_of_circle`:

```
val pi : double
fun area_of_circle (radius: double): double
```

When the same interfaces are declared in a *dynamic* file, the following slightly different syntax should be used:

```
extern val pi : double
extern fun area_of_circle (radius: double): double
```

Please note that a static file is essentially a special case of a dynamic file. It is entirely possible to replace a static file with a dynamic one.

As a convention, we often use the filename extension *.cats* to indicate that a file contains some C code that is supposed to be combined with ATS code in certain manner. There is also the filename extension *.hats*, which we occassionally use for a file that should be included in ATS code stored in another file. However, the use of these two filename extensions are not mandatory, that is, they can be replaced if needed or wanted.

Chapter 3. File Inclusion inside ATS Code

As is in C, file inclusion inside ATS code is done through the use of the directive *#include*. For instance, the following line indicates that a file named *foobar* is included, that is, this line is to be replaced with the content of the file *foobar*:

```
#include "foobar.hats"
```

Note that the included file is parsed according to the syntax for statics or dynamics depending on whether the file is included in a static or dynamic file. As a convention, the name of an included file often ends with the extension *.hats*.

A common use of file inclusion is to keep some constants, flags or parameters being defined consistently across a set of files. For instance, the file ***prelude/params.hats*** serves such a purpose. File inclusion can also be used to emulate (in a limited but rather useful manner) functors supported in languages such as SML and OCaml.

Chapter 4. Fixity Declarations

Given a function *f*, the standard syntax for applying *f* to an argument *v* is *f(v)*; for two arguments *v1* and *v2*, the syntax is *f(v1, v2)*. Also, it is allowed in ATS to use infix notation for a binary function application, and prefix/postifix notation for a unary function application.

Each identifier in ATS can be assigned one of the following fixities: *prefix*, *infix* and *postfix*. The fixity declarations for many commonly used identifiers can be found in ***prelude/fixity.ats***. Often, the name *operator* is used to refer to an identifier that is assigned a fixity. For instance, the following syntax declares that `+` and `-` are infix operators of a precedence value equal to 50:

```
infixl 50 + -
```

After this declaration, we can write an expression like `1 + 2 - 3`, which is parsed into `-(+(1, 2), 3)` in terms of the standard syntax for function application.

The keyword `infixl` indicates that the declared infix operators are left-associative. For right-associative and non-associative infix operators, please use the keywords `infixr` and `infix`, respectively. If the precedence value is omitted in a fixity declaration, it is assumed to be equal to 0.

We can also use the following syntax to declare that `iadd`, `fadd`, `padd` and `uadd` are left-associative infix operators with a precedence value equal to that of the operator `+`:

```
infixl (+) iadd fadd padd uadd
```

This is useful as it is difficult in practice to remember the precedence values of (a large collection of) declared operators. Sometimes, we may need to specify that the precedence value of one operator in relation to that of another one. For instance, the following syntax declares that `opr2` is a left-associative infix operator and its precedence value is that of `opr1` plus 10:

```
infixl (opr1 + 10) opr2
```

If the plus sign (+) is changed to the minus sign (-), then the precedence value of `opr2` is that of `opr1` minus 10.

We can also turn an identifier `opr` into a non-associative infix operator (of precedence value 0) by putting the backslash symbol (`\`) in front of it. For instance, the expression `exp1 \opr exp2` stands for `opr (exp1, exp2)`, where `exp1` and `exp2` refer to some expressions, either static or dynamic.

The syntax for declaring (unary) prefix and postfix operators are similar. For instance, the following syntax declares that `~` and `?` are prefix and postfix operators of precedence values 61 and 69, respectively:

```
prefix 61 ~
postfix 69 ?
```

As an example, a postfix operator is involved in the following 3-line program:

```
postfix (imul + 10) !!
extern fun !! (x: int): int
implement !! (x) = if x >= 2 then x * (x - 2)!! else 1
```

For a given occurrence of an operator, we can deprive it of its assigned fixity status by simply putting the keyword `op` in front of it. For instance `1 + 2 - 3` can be writen as `op- (op+ (1, 2), 3)`. It is also possible to (permanently) deprive an operator of its assigned fixity status. For instance, the following syntax does so to the operators `iadd`, `fadd`, `padd` and `uadd`:

```
nonfix iadd fadd padd uadd
```

Note that `nonfix` is a keyword in ATS.

Lastly, please note that each fixity declaration is only effective within its own legal scope.

Chapter 5. Static Load

In ATS, static load (or staload for short) refers to the creation of a namespace populated with the declared names in a loaded package.

Suppose that a file named `foo.sats` contains the following code:

```
//
datatype
aDatatype =
| A | B of (int, int)
//
val aValue: int
fun aFunction: int -> int
//
```

The following staload-declaration introduces a namespace `FOO` for the names declared in `foo.sats`:

```
staload FOO = "foo.sats"
```

The prefix `$FOO.` needs to be attached to a name in the namespace `FOO` in order for it to be referenced. For instance, the names available in the namespace `FOO` are all referenced in the following code:

```
val a: $FOO.aDatatype = $FOO.A()
val b: $FOO.aDatatype = $FOO.B(0, $FOO.aFunction($FOO.aValue))
```

If the file `foo.sats` is staloaded as follows for the second time:

```
staload FOO2 = "foo.sats"
```

then `foo.sats` is not actually loaded by the compiler. Instead, `FOO2` is simply made to be an alias of `FOO`.

It is also allowed for `foo.sats` to be staloaded as follows:

```
staload "foo.sats"
```

In this case, the namespace for the names declared in `foo.sats` is opened. For instance, the following code shows that these names can now be referenced directly:

```
val a: aDatatype = A()
val b: aDatatype = B(0, aFunction(aValue))
```

Let us suppose that we have the following sequence of declarations:

```
val aValue = 0
staload "foo.sats"
val anotheValue = aValue + 1
```

Does the second occurrence of `aValue` refer to the one introduced in the first declaration, or it refers to the one declared inside `foo.sats`? The answer may be a bit surprising: It refers to the one introduced in the first declaration as a binding for the occurrence of a name is resolved in ATS by searching first through the available names delcared in the same file.

Chapter 6. Dynamic Load

In ATS, dynamic load (or dynload for short) refers to some form of initialization of a loaded package.

Suppose that a file named `foo.dats` contains the following code:

```
//
val x = 1000
val y = x + x // = 2000
val z = y * y // = 4000000
//
extern
fun sum_x_y_z (): int
//
implement sum_x_y_z () = x + y + z
//
```

Clearly, the names x, y, and z must be bound to some values before a call to the function `sum_x_y_z` can be evaluated. In order to create such bindings, some form of initialization is required. Let us further suppose that a file named `foo2.dats` contains the following code:

```
staload "./foo.dats"
dynload "./foo.dats" // for initialization

implement
main0 () =
{
val () = assertloc (4003000 = sum_x_y_z())
} (* end of [main0] *)
```

We can now generate an executable file `mytest` by issuing the following command-line:

```
atscc -o mytest foo.dats foo2.dats
```

Note that **atscc** may need to be changed to **patscc**.

The line starting with the keyword `dynload` is referred to as a dynload-declaration. If it is deleted from the file `foo2.dats`, then executing the above command-line leads to link-time reporting of undefined reference to a variable of certain name ending with the string *__dynloadflag*. The dynload-declaration for `foo.dats` introduces this special variable and then makes a call to a special function associated with `foo.dats` for the purpose of performing some form of initialization. This special function is referred as a dynload-function (for `foo.dats`), which is always idempotent.

There is also a dynload-function generated for `foo2.dats`. As the function `main0`, a variant of the special function `main`, is implemented in `foo2.dats`, the dynload-function for `foo2.dats` is automatically called inside the body of the `main` function.

If there is a reason to suppress the generation of a dynload-function, one can set the value of the flag `ATS_DYNLOADFLAG` to 0. For instance, no dynload-function for `foo.dats` is generated if the following line is added into `foo.dats`:

```
#define ATS_DYNLOADFLAG 0
```

Of course, skipping proper initialization for `foo.dats` means that an erroneous result is expected if the function `sum_x_y_z` is ever called.

If there is a reason to call the dynload-function for `foo2.dats` explicitly, one can introduce an alias for it and then call the alias. For instance, if the following line is added to `foo2.dats`:

```
#define ATS_DYNLOADNAME "foo2_dynload"
```

then the dynload-function for `foo2.dats` is given an alias `foo2_dynload`.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 7. Bracket Overloading

In many programming languages, bracket-notation is commonly associated with array subscripting. For instance, `A[i]` is a left-value in C that refers to array-cell `i` in the array referred to by `A`. There is support in ATS for overloading brackets with multiple function names so that bracket-notation can be employed to call these functions, resulting in code that reads like some form of subscripting. It is expected that this style of calling functions can, sometimes, make the code written in ATS more easily accessible.

Let us now see a simple example of bracket-notation in overloading. In the followng code, a linear abstract type `lock` is introduced together with two functions:

```
//
absvtype lock(a:vt@ype)
//
extern
fun{a:vt0p} lock_acquire(!lock(a)): a
extern
fun{a:vt0p} lock_release(!lock(a), a): void
//
```

As one can imagine, `lock_acquire` is called to obtain the value stored in a given lock while `lock_release` is called to return a value to a given lock.

Suppose that we now introduce the following overloading declarations:

```
//
overload [] with lock_acquire
overload [] with lock_release
//
```

With these declarations, the following code typechecks:

```
//
val
mylock = $extval(lock(int), "mylock")
//
val x0 = mylock[] // = lock_acquire (mylock)
val () = mylock[] := x0 // = lock_release (mylock, x0)
//
```

Note that the bracket-notation in any assigement is only allowed to refer to a function that returns the void-value. In the above example, the function `lock_release` returns the void-value.

In ATS, bracket-notation is already overloaded with functions performing list-subscripting, array-subscripting and matrix-subscripting, and it can also be used to access and update a given reference.

Please find ***on-line*** the entirety of the code presented in this chapter.

Chapter 8. Dot-Symbol Overloading

In many programming languages, the so-called dot-notation is commonly associated with selecting a field from a given tuple-value, record-value or object-value. In ATS, field selection can be done through either pattern matching or the use of dot-notation. For example, the following code constructs a flat tuple and also a boxed one, and then uses dot-notation to select their components:

```
//
val tup_flat = @("a", "b")
val tup_boxed = $tup("a", "b")
//
val-"a" = tup_flat.0 and "b" = tup_flat.1
val-"a" = tup_boxed.0 and "b" = tup_boxed.1
//
```

There is support in ATS for overloading a specified dot-symbol with multiple function names so that dot-notation can be employed to call these functions, resulting in code that reads like field selection from tuples or records. This style of calling functions can, sometimes, make the code written in ATS more easily accessible, and it is especially so when ATS interacts with languages that support object-oriented programming.

As an example of dot-notation in overloading, let us introduce a non-linear abstract type `point` for points in a 2-dimensional space and also declare some associated functions:

```
//
abstype point = ptr // boxed
//
extern
fun
point_make
  (x: double, y: double): point
//
extern
fun point_get_x (p: point): double
and point_get_y (p: point): double
//
extern
fun point_set_x (p: point, x: double): void
and point_set_y (p: point, x: double): void
//
```

For getting the x-coordinate and y-coordinate of a given point, the functions `point_get_x` and `point_get_y` can be called, respectively. For setting the x-coordinate and y-coordinate of a given point, the functions `point_set_x` and `point_set_y` can be called, respectively. By introducing two dot-symbols `.x` and `.y` and then overloading them with certain function names as follows:

```
symintr .x .y
overload .x with point_get_x
overload .x with point_set_x
overload .y with point_get_y
overload .y with point_set_y
```

we can use dot-notation to call the corresponding get-functions and set-functions as is shown in the following code:

```
val p0 = point_make (1.0, ~1.0)
val x0 = p0.x() // point_get_x (p0)
and y0 = p0.y() // point_get_y (p0)
val () = p0.x := y0 // point_set_x (p0, y0)
and () = p0.y := x0 // point_set_y (p0, x0)
```

Note that writing `p0.x` for `p0.x()` is currently not supported. The dot-notation in any assigement is only allowed to refer to a function that returns the void-value. In the above example, both `point_set_x` and `point_set_y` return the void-value.

Let us introduce a linear abstract type `counter` for counter objects and a few functions associated with it:

```
//
absvtype counter = ptr
//
extern
fun counter_make (): counter
extern
fun counter_free (counter): void
//
extern
fun counter_get (cntr: !counter): int
extern
fun counter_incby (cntr: !counter, n: int): void
//
```

As can be expected, the functions `counter_make` and `counter_free` are for creating and destroying a counter object, respectively. The function `counter_get` returns the current count stored in a give counter, and the function `counter_incby` increase that count by a given integer value.

Let us introduce the following overloading declarations:

```
//
overload .get with counter_get
overload .incby with counter_incby
//
```

As is expected, one can now call `counter_get` as follows:

```
val n0 = c0.get() // = counter_get(c0)
```

Similarly, one can call `counter_incby` as follows to increase the count stored in `c0` by 1:

```
val () = c0.incby(1) // = counter_incby(c0, 1)
```

If we revisit the previous example involving (non-linear) points, then we can see that the following code also typechecks:

```
val p0 = point_make (1.0, ~1.0)
val x0 = p0.x() // point_get_x (p0)
and y0 = p0.y() // point_get_y (p0)
val () = p0.x(y0) // point_set_x (p0, y0)
and () = p0.y(x0) // point_set_y (p0, x0)
```

I may use the phrase *functional dot-notation* to refer to this form of dot-notation so as to differentiate it from the general form of dot-notation.

Please find ***on-line*** the entirety of the code presented in this chapter plus some testing code.

Chapter 9. Recursion

The notion of recursion is ubiquitous in ATS. For instance, there are recursively defined sorts (datasorts) and types (datatypes) in the statics, and there are also recursively defined functions in the dynamics. Literally, the word *recurse* means "go back". When an entity is defined recursively, it means that the entity being defined can appear in its own definition. In the following presentation, I will show several ways to define (or implement) recursive functions and non-recursive functions, where the latter is just a special case of the former.

The keyword `fun` can be used to initiate the definition of a recursive function. For instance, following is the definition of a recursive function:

```
fun
fact(x: int): int =
  if x > 0 then x * fact(x-1) else 1
(* end of [fact] *)
```

A non-recursive function is a special kind of recursive function that does make use of itself in its own definition. So one can certainly use `fun` to initiate the definition of a non-recursive function. However, if there is a need to indicate explicitly that a non-recursive is being defined, then one can use the keyword `fn` to do so. For instance, the definiton of a non-recursive function is given as follows:

```
fn square(x: int): int = x * x
```

which is directly translated by the compiler into the following binding between a name and a lambda-expression:

```
val square = lam (x: int): int => x * x
```

As another example, please note that the two occurrences of the symbol `fact` in the following code refer to two distinct functions:

```
fn
fact(x: int): int =
  if x > 0 then x * fact(x-1) else 1
(* end of [fact] *)
```

While the first `fact` (to the left of the equality symbol) refers to the (non-recursive) function being defined, the second one is supposed to have been declared previously.

A recursive function can also be defined as a recursive value. For instance, the recursive function `fact` defined above can be defined as follows:

```
val
rec
fact : int -> int =
lam (x) =>
  if x > 0 then x * fact(x-1) else 1
(* end of [fact] *)
```

where the keyword `rec` indicates that `fact` is defined recursively, that is, it is allowed to appear in its own definition. In fact, the former definition of `fact` is directly translated into the latter one by the compiler. Of course, one may also use a reference to implement recursion:

```
val
fact = ref<int->int>($UNSAFE.cast(0))
val () =
!fact :=
(
  lam (x:int):int => if x > 0 then x * !fact(x-1) else 1
) (* end of [val] *)
```

But this is definitely not a style I would like to advocate. For the sake of completion, I present yet another way to define `fact` as a fixed-point expression:

```
val
fact =
fix f(x: int): int =>
  if x > 0 then x * f(x-1) else 1
(* end of [fact] *)
```

Of course, if one wants to, then one can always replace a lambda-expression with a fixed-point expression (or simply fix-expression for short). For instance, `lambda(x:int):int => x+x` can be written as `fix _(x:int):int => x+x`.

For defining mutually recursive functions, one can simply use the keyword `and` to concatenate function definitions. For instance, the following code defines two functions `isevn` and `isodd` mutually recursively:

```
fun
```

```
isevn(x: int): bool =
  if x > 0 then isodd(x-1) else true
and
isodd(x: int): bool =
  if x > 0 then isevn(x-1) else false
```

The code, as one may have guessed, is translated by the compiler into the following form (for defining two mutually recursive values):

```
val
rec
isevn : int -> bool =
  lam (x) => if x > 0 then isodd(x-1) else true
and
isodd : int -> bool =
  lam (x) => if x > 0 then isevn(x-1) else false
```

One can certainly use the keyword `and` to concatenate definitions of non-recursive functions, but doing so is probably just a curiosity (instead of a meaningful practice).

Even at this point, I have not presented all of the possible ways to define functions in ATS. For instance, one can also define stack-allocated closure-functions in ATS, which may be either recursive or non-recursive. I plan to introduce such functions elsewhere in this tutorial.

Often, the interface (that is, type) for a function is declared at one place and its definition (or implementation) is given at another place. For instance, one may first introduce the following declaration:

```
extern fun fact (x: int): int
```

Later, one can implement `fact` according to the above declaration:

```
implement
fact (x) =
  if x > 0 then x * fact(x-1) else 1
// end of [fact]
```

When `implement` is used to initiate the definition of a function, any previously declared symbols (including the one that is being defined) can appear in the definition. If it is desirable, one may replace `implement` with `implmnt`.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 10. Datatypes

Datatypes are a form of user-defined types for classifying data (or values). The support for datatypes and pattern matching in ATS is primarily adopted from ML.

The following code declares a datatype of the name `weekday` for values representing weekdays:

```
datatype weekday =
  | Monday | Tuesday | Wednesday | Thursday | Friday
```

There are five data constructors associated with `weekday`, which are `Monday`, `Tuesday`, `Wednesday`, `Thursday`, and `Friday`. All of these data constructors are nullary, that is, they take no arguments to form values (of the type `weekday`).

Each nullary constructor is represented as a small integer (e.g., one that is less than 1024). One can use the following function `weekday2int` to find out the integers reprsenting the constructors associated with `weekday`:

```
//
staload UN = "prelude/SATS/unsafe.sats"
//
fun
weekday2int
  (wd: weekday): int = $UN.cast{int}($UN.cast{intptr}(wd))
//
```

The small integer representing a nullary constructor is often referred to as the tag of the constructor. In this case, the tags for `Monday`, `Tuesday`, `Wednesday`, `Thursday`, and `Friday` are 0, 1, 2, 3, and 4, respectively.

Given a nullary constructor `foo`, both `foo` and `foo()` can be used to refer the value constructed by `foo`. However, only `foo()` can be used as the pattern that matches this value. For instance, the following function tests whether a given value of the type `weekday` is formed with the constructor `Friday`:

```
fun
isFriday(x: weekday): bool =
  case+ x of Friday() => true | _ => false
```

Note that the pattern `Friday()` cannot be replaced with `Friday` as the latter is treated as a variable when used as a pattern. On the other hand, both of the following assertions hold at run-time as `Friday` and `Friday()` refer to the same value:

```
val () = assertloc (isFriday(Friday))
val () = assertloc (isFriday(Friday()))
```

If there is only one constructor associated with a datatype, then there is no tag in the representation for values of this datatype.

A datatype is list-like if there are two data constructors associated with it such that one is nullary (nil-like) and the other is non-nullary (cons-like). For instance, the datatype `ab` declared as follows is list-like:

```
datatype ab = A of () | B of (bool)
```

The values of a list-like datatype are represented in a special way. Given a value of the datatype; if it is constructed with the nil-like constructor, then it is represented as the null-pointer; if it is constructed with the cons-like constructor, then it is reprenstend as a heap-allocated tuple containing all of the arguments passed to the constructor. In the above case, the value `A()` is represented as the null pointer, and the value `B(b)` for each boolean `b` is represented as a heap-allocated singleton tuple containing the only component `b`. This description can be readily verified with the following code:

```
val () = assertloc (iseqz($UN.cast{ptr}(A())))
val () = assertloc (true = $UN.ptr0_get<bool>($UN.cast{ptr}(B(true))))
val () = assertloc (false = $UN.ptr0_get<bool>($UN.cast{ptr}(B(false))))
```

The following declaration introduces a datatype of the name `abc`:

```
datatype abc =
  | A of () | B of (bool) | C of (int, double)
```

The three constructors associated with `abc` are `A`, `B`, and `C`; `A` is nullary; `B` is unary, taking a boolean to form a value (of the type `abc`); `C` is binary, taking an integer and a float (of double precision) to form a value (of the type `abc`).

In a general case, if a data constructor is n-ary for some positive n, then each value it constructs is a heap-allocated tuple of n+1 components, where the first one is the tag assigned to the constructor and the rest are the arguments passed to the constructor. For instance, the following function can be called to find out the tags assigned to the constructors associated with `abc`:

```
fun
abc2tag
(x: abc): int = let
  val p = $UN.cast{intptr}(x)
in
//
case+ 0 of
| _ when p < 1024 => $UN.cast{int}(p)
| _ (*heap-allocated*) => $UN.ptr0_get<int>($UN.cast{ptr}(p))
//
end // end of [abc2tag]
```

In this case, the tags assigned to `A`, `B`, and `C` are 0, 1, and 2, respectively.

Datatypes can be defined recursively. As an example, the following declaration introduces a recursively defined datatype `intexp` (for representing arithemetic integer expressions):

```
datatype
intexp =
| Int of int
| Neg of (intexp)
| Add of (intexp, intexp)
| Sub of (intexp, intexp)
```

For instance, `(1+2)-3` can be represented as `Sub(Add(Int(1), Int(2)), Int(3))`. As another example, the following code introduces two mutually recursively defined datatypes `intexp` and `boolexp` (for integer expressions and boolean expressions, respectively):

```
datatype
intexp =
| Int of int
| Neg of (intexp)
| Add of (intexp, intexp)
| Sub of (intexp, intexp)
| IfThenElse of (boolexp, intexp, intexp)

and
boolexp =
| Bool of bool // constant
| Not of (boolexp) // negation
| Less of (intexp, intexp) // Less(x, y): x < y
| LessEq of (intexp, intexp) // LessEq(x, y): x <= y
| Conj of (boolexp, boolexp) // Conj(x, y): x /\ y
| Disj of (boolexp, boolexp) // Disj(x, y): x \/ y
```

The code below implements two mutually recursive functions `eval_intexp` and `eval_boolexp` for evaluating integer expressions and boolean expressions, respectively:

```
//
symintr eval
//
extern
fun eval_intexp : intexp -> int
extern
fun eval_boolexp : boolexp -> bool
//
overload eval with eval_intexp
overload eval with eval_boolexp
//
(* ****** ****** *)
//
implement
eval_intexp
  (e0) = (
//
case+ e0 of
| Int (i) => i
| Neg (e) => ~eval(e)
| Add (e1, e2) => eval(e1) + eval(e2)
| Sub (e1, e2) => eval(e1) - eval(e2)
| IfThenElse
    (e_test, e_then, e_else) =>
    if eval(e_test) then eval(e_then) else eval(e_else)
//
) (* end of [eval_intexp] *)
//
implement
eval_boolexp
  (e0) = (
//
case+ e0 of
| Bool (b) => b
| Not (e) => ~eval(e)
| Less (e1, e2) => eval(e1) < eval(e2)
| LessEq (e1, e2) => eval(e1) <= eval(e2)
| Conj (e1, e2) => eval(e1) && eval(e2)
| Disj (e1, e2) => eval(e1) || eval(e2)
//
) (* end of [eval_boolexp] *)
//
(* ****** ****** *)
```

The datatypes presented in this chapter are simple datatypes. Other more advanced forms of datatypes include polymorphic datatypes, dependent datatypes, and linear datatypes, which will be covered elsewhere.

Please find ***on-line*** the entirety of the code used in this chapter plus some code for testing.

Chapter 11. Functional Lists

A functional list is just a singly-linked list that is immutable after its construction. The following datatype declaration introduces a type `list` for functional lists:

```
//
datatype
list(a:t@ype, int) =
| list_nil(a, 0) of ()
| {n:nat}
  list_cons(a, n+1) of (a, list(a, n))
//
```

Given a type T and an integer N, the type `list(T,N)` is for a list of length N that contains elements of type T. The interfaces for various functions on functional lists can be found in the SATS file ***prelude/SATS/list.sats***, which is automatically loaded by **atsopt**.

There are various functions in ATSLIB for list construction. In practice, a list is usually built by making direct use of the constructors `list_nil` and `list_cons`. For instance, the following function `fromto` builds a list of integers between two given ones:

```
//
fun
fromto
{m,n:int | m <= n}
(
  m: int(m), n: int(n)
) : list(int, n-m) =
  if m < n then list_cons(m, fromto(m+1, n)) else list_nil()
//
```

Traversing a list is commonly done by making use of pattern matching. For instance, the following code implements a function for computing the length of a given list:

```
//
fun
{a:t@ype}
list_length
  {n:nat}
(
  xs: list(a, n)
) : int(n) = let
```

```
//
fun
loop
{i,j:nat}
(
  xs: list(a, i), j: int(j)
) : int(i+j) =
(
case+ xs of
| list_nil () => j
| list_cons (_, xs) => loop(xs, j+1)
)
//
in
  loop (xs, 0)
end // end of [list_length]
//
```

Given a non-empty list xs, `xs.head()` and `xs.tail()` refer to the head and tail of xs, respectively, where `.head()` and `.tail()` are overloaded dot-symbols. For instance, the function `list_foldleft` for folding a given list from the left can be implemented as follows:

```
//
fun
{a,b:t@ype}
list_foldleft
  {n:nat}
(
  f: (a, b) -> a, ini: a, xs: list(b, n)
) : a =
  if iseqz(xs)
    then ini else list_foldleft(f, f(ini, xs.head()), xs.tail())
  // end of [if]
//
```

And the function `list_foldright` for folding a given list from the right can be implemented as follows:

```
//
fun
{a,b:t@ype}
list_foldright
  {n:nat}
(
  f: (a, b) -> b, xs: list(a, n), snk: b
) : b =
  if iseqz(xs)
    then snk else f(xs.head(), list_foldright(f, xs.tail(), snk))
  // end of [if]
//
```

Note that `list_foldleft` is tail-recursive but `list_foldright` is not.

As an application of `list_foldleft`, the following code implements a function for reversing a given list:

```
fun
{a:t@ype}
list_reverse
(
  xs: List0(a)
) : List0(a) =
(
list_foldleft<List0(a),a>
  (lam (xs, x) => list_cons(x, xs), list_nil, xs)
)
```

where the type constructor `List0` is for lists of unspecified length:

```
typedef List0(a:t@ype) = [n:nat] list (a, n)
```

Clearly, `list_reverse` is length-preserving, that is, it always returns a list of the same length as its input. Unfortunately, this invariant is not captured in the above implementation of `list_reverse` based on `list_foldleft`. For the purpose of comparison, another implementation of `list_reverse` is given as follows that captures the invariant of `list_reverse` being length-preserving:

```
fun
{a:t@ype}
list_reverse
  {n:nat}
(
  xs: list(a, n)
) : list(a, n) = let
//
fun
loop{i,j:nat}
(
  xs: list(a, i), ys: list(a, j)
) : list(a, i+j) =
  case+ xs of
  | list_nil () => ys
  | list_cons (x, xs) => loop (xs, list_cons (x, ys))
//
in
  loop (xs, list_nil)
end // end of [list_reverse]
```

As an application of `list_foldright`, the following code implements a function for concatenating two given lists:

```
//
fun
{a:t@ype}
list_append
(
  xs: List0(a), ys: List0(a)
) : List0(a) =
  list_foldright<a, List0(a)>(lam (x, xs) => list_cons(x, xs), ys, xs)
//
```

The type assigned to `list_append` states that this function takes two lists as its arguments and returns one of unspecified length. For the purpose of comparison, another implementation of `list_append` is given as follows:

```
//
fun
{a:t@ype}
list_append
  {m,n:nat}
(
  xs: list(a,m), ys: list(a,n)
) : list(a,m+n) =
(
case+ xs of
| list_nil () => ys
| list_cons (x, xs) => list_cons (x, list_append (xs, ys))
)
//
```

where the type assigned to `list_append` states that this function takes two lists of length m and n, respectively, and returns another list of length m+n.

One may think of a functional list as a representation for the finite mapping that maps each natural number i less than the length of the list to element i in the list. The following function `list_get_at` is for accessing a list element at a given position:

```
//
fun
{a:t@ype}
list_get_at
  {n:nat}
(
  xs: list(a, n), i: natLt(n)
) : a =
  if i > 0 then list_get_at(xs.tail(), i-1) else xs.head()
//
```

This function can be called through the use of the bracket notation. In other words, `xs[i]` is automatically interpreted as `list_get_at(xs, i)` whenever xs and i are a list and an integer, respectively. Note that the time-complexity of `list_get_at(xs, i)` is O(i). If one uses `list_get_at` frequently when handling lists, then it is almost always a sure sign of poor programming style.

There is no destructive update on a functional list as it is immutable. The following function `list_set_at` can be called to construct a list that differs from a given one only at a given position:

```
//
fun
{a:t@ype}
list_set_at
  {n:nat}
(
  xs: list(a, n), i: natLt(n), x0: a
) : list(a, n) =
  if i > 0
    then list_cons(xs.head(), list_set_at(xs.tail(), i-1, x0))
    else list_cons(x0, xs.tail())
  // end of [if]
//
```

While it is fine to call `list_set_at` occasionally, a need to do so repeatedly often indicates that another data structure should probably be chosen in place of functional list.

Functional lists are by far the most widely used data structure in functional programming. However, one should not attempt to use a functional list like an array as doing so is inefficient both time-wise and memory-wise.

Please find ***on-line*** the entirety of the code used in this chapter plus some testing code.

Chapter 12. Functional Sets and Maps

Both (finite) sets and (finite) maps are commonly used data structures. Functional sets and maps are immutable after their construction. Insertion into or removal from a functional set/map results in the construction of a new set/map while the original is kept intact. Usually the newly constructed set/map and the original one share a lot of underlying representation. Note that a functional set/map cannot be safely freed explicitly and the memory for representing it can only be reclaimed through garbage collection (GC).

Functional Sets

Suppose that a set is needed for collecting values of type `elt_t`. The following code essentially sets up an interface for creating and operating on such a set based on a balanced-tree implementation in ATSLIB/libats:

```
local
//
typedef elt = elt_t
//
staload
FS = "libats/ML/SATS/funset.sats"
implement
$FS.compare_elt_elt<elt>(x, y) = compare(x, y)
//
in (* in-of-local *)

#include "libats/ML/HATS/myfunset.hats"

end // end of [local]
```

Please find ***on-line*** the HATS file mentioned in the code, which is just a convenience wrapper made to simplify programming with functional sets. Note that it is assumed here that there is a comparison function on values of the type `elt_t` that overloads the symbol `compare`. If this is not the case, one needs to implement such a function.

Assume that `elt_t` is `int`. The following line of code creates a functional set (of integers) containing no elements:

```
val myset = myfunset_nil()
```

The function for inserting an element into a given set is assigned the following type:

```
//
fun myfunset_insert(xs: &myset >> _, x0: elt): bool
//
```

The dot-symbol `.insert` is overloaded with the function `myfunset_insert`. Note that the first argument of `myfunset_insert` is call-by-reference. If the given element is inserted into the given set, then the newly created set is stored into the call-by-reference argument and `false` is returned (to indicate no error). Otherwise, `true` is returned (to indicate a failure). The following few lines of code shows how insertion can be operated on a functional set:

```
//
var myset = myset
//
val-false = myset.insert(0) // inserted
val-(true) = myset.insert(0) // not actually inserted
val-false = myset.insert(1) // inserted
val-(true) = myset.insert(1) // not actually inserted
//
```

The first line in the above code may seem puzzling: Its sole purpose is to create a left-value to be passed as the first argument to `myfunset_insert`. During the course of debugging, one may want to print out the values contained in a given set:

```
//
val () = fprintln! (stdout_ref, "myset = ", myset)
//
```

where the symbol `fprint` is overloaded with `fprint_myset`. The function for removing an element from a given set is assigned the following type:

```
//
fun myfunset_remove(xs: &myset >> _, x0: elt): bool
//
```

The dot-symbol `.remove` is overloaded with the function `myfunset_remove`. Note that the first argument of `myfunset_remove` is call-by-reference. If the given element is removed from the given set, then the newly created set is stored into the call-by-reference argument and `true` is returned. Otherwise, `false` is returned. The following few lines of code shows how removal can be operated on a functional set:

```
val-true = myset.remove(0) // removed
val-false = myset.remove(0) // not actually removed
val-true = myset.remove(1) // removed
val-false = myset.remove(1) // not actually removed
```

Various common set operations can be found in ***libats/ML/HATS/myfunset.hats***. By following the types assigned to these operations, one should have no difficulty in figuring out how they are supposed to be called. Please find the entirety of the code used in this section ***on-line***.

Functional Maps

Suppose that a map is needed for mapping keys of type `key_t` to items of type `itm_t`. The following code essentially sets up an interface for creating and operating on such a map based on a balanced-tree implementation in ATSLIB/libats:

```
local
//
typedef
key = key_t and itm = itm_t
//
staload
FM = "libats/ML/SATS/funmap.sats"
implement
$FM.compare_key_key<key>(x, y) = compare(x, y)
//
in (* in-of-local *)

#include "libats/ML/HATS/myfunmap.hats"

end // end of [local]
```

Please find ***on-line*** the HATS file mentioned in the code, which is just a convenience wrapper made to simplify programming with functional maps. Note that it is assumed here that there is a comparison function on values of the type `key_t` that overloads the symbol `compare`. If this is not the case, one needs to implement such a function.

Assume that `key_t` is `string` and `itm_t` is `int`. The following line of code creates an empty functional map:

```
val mymap = myfunmap_nil()
```

The following few lines insert some key/item pairs into `mymap`:

```
//
var mymap = mymap
//
val-~None_vt() = mymap.insert("a", 0)
val-~Some_vt(0) = mymap.insert("a", 1)
//
val-~None_vt() = mymap.insert("b", 1)
val-~Some_vt(1) = mymap.insert("b", 2)
//
```

```
val-~None_vt() = mymap.insert("c", 2)
val-~Some_vt(2) = mymap.insert("c", 3)
//
```

The dot-symbol `.insert` is overloaded with a function of the name `myfunmap_insert`. The first line in the above code may seem puzzling: Its sole purpose is to create a left-value to be passed as the first argument to `myfunmap_insert`. Given a key and an item, `mymap.insert` inserts the key/item pair into `mymap`. If the key is in the domain of the map represented by `mymap` before insertion, then the original item associated with the key is returned. Otherwise, no item is returned. As can be expected, the size of `mymap` is 3 at this point:

```
val () = assertloc (mymap.size() = 3)
```

The dot-symbol `.size` is overloaded with a function of the name `myfunmap_size`, which returns the number of key/item pairs stored in a given map. During the course of debugging, one may want to print out the key/item pairs in a given map:

```
//
val () = fprintln! (stdout_ref, "mymap = ", mymap)
//
```

where the symbol `fprint` is overloaded with `fprint_mymap`. The next two lines of code show how search with a given key operates on a map:

```
val-~None_vt() = mymap.search("")
val-~Some_vt(1) = mymap.search("a")
```

The dot-symbol `.search` is overloaded with a function of the name `myfunmap_search`, which returns the item associated with a given key if it is found. The next few lines of code remove some key/item pairs from `mymap`:

```
//
val-true = mymap.remove("a")
val-false = mymap.remove("a")
//
val-~Some_vt(2) = mymap.takeout("b")
val-~Some_vt(3) = mymap.takeout("c")
//
```

The dot-symbol `.remove` is overloaded with a function of the name `myfunmap_remove` for removing a key/item pair of a given key. If a key/item pair is removed, then the function returns true. Otherwise, it returns false to indicates that no key/item pair of the given key is stored in the map being operated on. The dot-symbol `.takeout` is overloaded with a function of the name `myfunmap_takeout`, which is similar to `myfunmap_remove` excepting for returning the removed item.

Various common map operations can be found in ***libats/ML/HATS/myfunmap.hats***. By following the types assigned to these operations, one should have no difficulty in figuring out how they are supposed to be called. Please find the entirety of the code used in this section ***on-line***.

Chapter 13. Exceptions

While exceptions can be very useful in practice, it is also very common to see code that misuses exceptions.

Generally speaking, there are exceptions that are meant to be raised but not captured for the purpose of aborting program execution, and there are also exceptions (often declared locally) that are meant to be raised and then captured so as to change the flow of program execution. For instance, the exception `ArraySubscriptExn` is raised when out-of-bounds array subscripting is detected at run-time. Once it is raised, `ArraySubscriptExn` is usually not meant to be captured. While there is certainly nothing preventing a programer from writing code that captures a raised `ArraySubscriptExn`, a major concern is that reasoning can become greatly complicated on code that does so. In the following presentation, I will soley focus on exceptions that are meant to be raised and then captured.

Let us now take a look at the following code that implements a function for finding the rightmost element in a list that satisfies a given predicate:

```
extern
fun{a:t@ype}
list_find_rightmost
  (List (a), (a) -<cloref1> bool): Option_vt (a)
//
implement{a}
list_find_rightmost
  (xs, pred) = let
//
fun aux
(
  xs: List(a)
) : Option_vt (a) =
  case+ xs of
  | nil () => None_vt ()
  | cons (x, xs) => let
      val res = aux (xs)
    in
      case+ res of
      | Some_vt _ => res
      | ~None_vt () =>
          if pred (x) then Some_vt (x) else None_vt ()
        // end of [None]
    end (* end of [cons] *)
//
```

```
in
  aux (xs)
end // end of [list_find_rightmost]
```

Suppose that `list_find_rightmost` is called on a list xs of length N (for some large natural number N) and a predicate pred. The evaluation of this call leads to a call to the inner function `aux`, which in turn generates N additional recursive calls to `aux`. Assume that only the last element of xs satisfies the predicate pred. Then there are still N-1 call frames for `aux` on the call stack when the rightmost element satisfying the given predicate is found, and these frames need to be unwinded *one-by-one* before the found element can be returned to the original call to `list_find_rightmost`. This form of inefficiency is eliminated in the following exception-based implementation of `list_find_rightmost`:

```
implement{a}
list_find_rightmost
  (xs, pred) = let
//
exception Found of (a)
//
fun aux
(
  xs: List(a)
) : void =
  case+ xs of
  | nil () => ()
  | cons (x, xs) => let
      val () = aux (xs)
    in
      if pred (x) then $raise Found(x) else ()
    end (* end of [cons] *)
//
in
//
try let
  val () = aux (xs)
in
  None_vt ()
end with
  | ~Found(x) => Some_vt (x)
//
end // end of [list_find_rightmost]
```

When a try-with-expression is evaluated, a label is created for the portion of the call stack needed to evaluate the clauses (often referred to as exception-handlers) following the keyword `with`, and this label is then pushed onto a designated global stack. When an exception is raised, the labels on the global stack are tried one-by-one until the raised exception is captured by an exception-handler (that is, the value representing the exception matches the pattern guard of the exception-handler) or the current program evaluation aborts. The above exception-based implementation of `list_find_rightmost` uses a raised exception to carry the element found during a recursive call to `aux` so that this element can be returned in a single jump to the original call to `list_find_rightmost`, bypassing all the intermediate call frames (for recursive calls to `aux`) on the call stack. In general, the range between the point where an exception is raised and the point where the raised exception is captured should span multiple call frames. If not, then the use of exception may be questionable.

The implementation of the run-time support for exceptions in ATS makes use of the function `alloca` declared in `alloca.h` and the functions `setjmp` and `longjmp` declared in `setjmp.h`. If **gcc** or **clang** is used to compile the C code generated from ATS source, one can pass the flag -D_GNU_SOURCE so as to make sure that the header file `alloca.h` is properly included.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 14. References

A reference is a singleton array, that is, an array of size 1. It is persistent in the sense that the (heap-allocated) memory for storing the content of a reference cannot be freed manually in a type-safe manner. Instead, it can only be reclaimed through garbage collection (GC).

Given a viewtype VT, the type for references to values of viewtype VT is `ref`(VT). For convenience, the type constructor `ref` is declared to be abstract in ATS. However, it can be defined as follows:

```
typedef ref (a:vt@ype) = [l:addr] (vbox (a @ l) | ptr l)
```

The interfaces for various functions on references can be found in ***prelude/SATS/reference.sats***.

For creating a reference, the function template `ref_make_elt` of the following interface can be called:

```
fun{a:vt@ype} ref_make_elt (x: a):<!wrt> ref a
```

It is also allowed to use the shorthand `ref` for `ref_make_elt`. Note that the symbol `!wrt` indicates that the so-called `wrt`-effect may occur when `ref_make_elt` is called.

For reading from and writing through a reference, the function templates `ref_get_elt` and `ref_set_elt` can be used, respectively, which are assigned the following types:

```
fun{a:t@ype} ref_get_elt (r: ref a):<!ref> a
fun{a:t@ype} ref_set_elt (r: ref a, x: a):<!refwrt> void
```

Note that the symbol `!ref` indicates that the so-called ref-effect may occur when `ref_get_elt` is evaluated. Similarly, `!refwrt` means both ref-effect and wrt-effect may occur when `ref_set_elt`. Given a reference `r` and a value `v`, `ref_get_elt(r)` and `ref_set_elt(r, v)` can be written as `!r` and `!r := v`, respectively, and can also be written as `r[]` and `r[] := v`, respectively, in terms of bracket-notation.

A reference is typically employed to record some form of persistent state. For instance, following is such an example:

```
local
//
#define BUFSZ 128
//
val count = ref<int> (0)
//
```

```
in (* in of [local] *)

fun genNewName
  (prfx: string): string = let
  val n = !count
  val () = !count := n + 1
  var res = @[byte][BUFSZ]((*void*))
  val err =
  $extfcall (
    int, "snprintf", addr@res, BUFSZ, "%s%i", prfx, n
  ) (* end of [$extfcall] *)
in
  strptr2string(string0_copy($UNSAFE.cast{string}(addr@res)))
end // end of [genNewName]

end // end of [local]
```

The function `genNewName` is called to generate fresh names. As the integer content of the reference `count` is updated whenever a call to `genNewName` is made, each name returned by `genNewName` is guaranteed to have not been generated before. Note that the use of `$extfcall` is for making a direct call to the function `snprintf` in C.

Misuse of References References are commonly misused in practice. The following program is often written by a beginner of functional programming who has already learned (some) imperative programming:

```
fun fact
  (n: int): int = let
  val res = ref<int> (1)
  fun loop (n: int):<cloref1> void =
    if n > 0 then (!res := n * !res; loop(n-1)) else ()
  val () = loop (n)
in
  !res
end // end of [fact]
```

The function `fact` is written in such a style as somewhat a direct translation of the following C code:

```
int fact (int n) {
  int res = 1 ;
  while (n > 0) { res = n * res; n = n - 1; } ;
  return res ;
}
```

In the ATS implementation of `fact`, `res` is a heap-allocated reference and it becomes garbage (waiting to be reclaimed by the GC) after a call to `fact` returns. On the other hand, the variable `res` in the C implementation of `fact` is stack-allocated (or it can even be mapped to a machine register), and there is no generated garbage after a call to `fact` returns. A proper translation of the C implementation in ATS can actually be given as follows:

```
fun fact
  (n: int): int = let
  fun loop (n: int, res: int): int =
    if n > 0 then loop (n-1, n * res) else res
  // end of [loop]
in
  loop (n, 1)
end // end of [fact]
```

which makes no use of references at all.

Unless strong justification can be given, making extensive use of (dynamically created) references is often a sure sign of poor coding style.

Statically Allocated References Creating a reference by calling `ref_make_elt` involves dynamic memory allocation. If this is not desirable or even acceptable, it is possible to only employ statically allocated memory in a reference creation as is shown below:

```
var myvar: int = 0
val myref = ref_make_viewptr (view@(myvar) | addr@(myvar))
```

The function `ref_make_viewptr` takes a pointer and a proof of some at-view associated with the pointer and returns a reference after consuming the proof. As `ref_make_viewptr` is a cast-function, it causes no run-time overhead. In the above code, `myvar` is statically allocated and it is no longer available after its at-view proof is consumed by `ref_make_viewptr`. It should be interesting to observe that both `myvar` and `myref` are just the same pointer in C but they are the reification of fundamentally different concepts in ATS: the former is a linear variable while the latter is a non-linear reference.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 15. Persistent Arrays

A persistent array of size n is just n heap-allocated cells (or references) in a row. It is persistent in the sense that the memory allocated for the array cannot be freed manually. Instead, it can only be safely reclaimed through garbage collection (GC).

Given a viewtype VT, the type for persistent arrays containing N values of viewtype VT is `arrayref(VT, N)`. Note that arrays in ATS are the same as those in C: There is no size information attached to them. The interfaces for various functions on persistent arrays can be found in the SATS file ***prelude/SATS/arrayref.sats***, which is automatically loaded by **atsopt**.

There are various functions in ATSLIB for array creation. For instance, the following two are commonly used:

```
fun{a:t@ype}
arrayref_make_elt
  {n:nat} (asz: size_t n, elt: a):<!wrt> arrayref (a, n)
// end of [arrayref_make_elt]

fun{a:t@ype}
arrayref_make_listlen
  {n:int} (xs: list (a, n), n: int n):<!wrt> arrayref (a, n)
// end of [arrayref_make_listlen]
```

Applied to a size and an element, `arrayref_make_elt` returns an array of the given size in which each cell is initialized with the given element. Applied to a list of elements and the length of the list, `arrayref_make_listlen` returns an array of size equal to the given length in which each cell is initialized with the corresponding element in the given list.

For reading from and writing to an array, the function templates `arrayref_get_at` and `arrayref_set_at` can be used, respectively, which are assigned the following interfaces:

```
fun{a:t@ype}
arrayref_get_at
  {n:int} (A: arrayref (a, n), i: sizeLt (n)):<!ref> a

fun{a:t@ype}
arrayref_set_at
  {n:int} (A: arrayref (a, n), i: sizeLt (n), x: a):<!ref> void
```

Given an array `A`, an index `i` and a value `v`, `arrayref_get_at(A, i)` and `arrayref_set_at(A, i, v)` can be written as `A[i]` and `A[i] := v`, respectively.

As an example, the following function template reverses the content of a given array:

```
fun{a:t@ype}
arrayref_reverse{n:nat}
(
  A: arrayref (a, n), n: size_t (n)
) : void = let
//
fun loop
  {i: nat | i <= n} .<n-i>.
(
  A: arrayref (a, n), n: size_t n, i: size_t i
) : void = let
  val n2 = half (n)
in
  if i < n2 then let
    val tmp = A[i]
    val ni = pred(n)-i
  in
    A[i] := A[ni]; A[ni] := tmp; loop (A, n, succ(i))
  end else () // end of [if]
end // end of [loop]
//
in
  loop (A, n, i2sz(0))
end // end of [arrayref_reverse]
```

If the test `i < n2` is changed to `i <= n2`, a type-error is to be reported. Why? The reason is that `A[n-1-i]` becomes out-of-bounds array subscripting in the case where `n` and `i` both equal zero. Given that it is very unlikely to encounter a case where an array of size 0 is involved, a bug like this, if not detected early, can be buried so scarily deep!

The careful reader may have already noticed that the sort `t@ype` is assigned to the template parameter `a`. In other words, the above implementation of `arrayref_reverse` cannot handle a case where the values stored in a given array are of some linear type. The reason for choosing the sort `t@ype` is that both `arrayref_get_at` and `arrayref_set_at` can only be applied to an array containing values of a nonlinear type. In the following implementation, the template parameter is given the sort `vt@ype` so that an array containing linear values, that is, values of some linear type can be handled:

```
fun{a:vt@ype}
arrayref_reverse{n:nat}
(
  A: arrayref (a, n), n: size_t (n)
) : void = let
//
fun loop
  {i: nat | i <= n} .<n-i>.
(
  A: arrayref (a, n), n: size_t n, i: size_t i
) : void = let
  val n2 = half (n)
in
  if i < n2 then let
    val () = arrayref_interchange (A, i, pred(n)-i) in loop (A, n, succ(i))
  end else () // end of [if]
end // end of [loop]
//
in
  loop (A, n, i2sz(0))
end // end of [arrayref_reverse]
```

The interface for the function template `arrayref_interchange` is given below:

```
fun{a:vt@ype}
arrayref_interchange{n:int}
  (A: arrayref (a, n), i: sizeLt n, j: sizeLt n):<!ref> void
// end of [arrayref_interchange]
```

Note that `arrayref_interchange` can not be implemented in terms of `arrayref_get_at` and `arrayref_set_at` (unless some form of type-unsafe code is employed).

There are various functions available for traversing an array from left to right or from right to left. Also, the following two functions can be conveniently called to traverse an array from left to right:

```
//
fun{a:t0p}
arrayref_head{n:pos} (A: arrayref (a, n)): (a) // A[0]
fun{a:t0p}
arrayref_tail{n:pos} (A: arrayref (a, n)): arrayref (a, n-1)
//
overload .head with arrayref_head
overload .tail with arrayref_tail
//
```

For instance, the fold-left function for arrays can be implemented as follows:

```
fun{a,b:t@ype}
arrayref_foldleft{n:int}
(
  f: (a, b) -> a, x: a, A: arrayref (b, n), n: size_t(n)
) : a =
(
if n > 0
  then arrayref_foldleft<a,b> (f, f (x, A.head()), A.tail(), pred(n))
  else x
// end of [if]
) (* end of [arrayref_foldleft] *)
```

As can be expected, `A.head` and `A.tail` translate into `A[0]` and `ptr_succ<T>(p0)`, respectively, where T is the type for the elements stored in A and p0 is the starting address of A.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 16. Persistent Arrays-with-size

I use the name *array-with-size* to refer to a persistent array with attached size information. Given a viewtype VT, the type for an array-with-size that contains N values of viewtype VT is `arrszref(VT, N)`. Essentially, such a value is a boxed pair of two components of types `arrayref(VT, N)` and `size_t(N)`. The interfaces for various functions on persistent arrays-with-size can be found in ***prelude/SATS/arrayref.sats***.

For creating an array-with-size, the following functions `arrszref_make_arrpsz` and `arrszref_make_arrayref` can be called:

```
fun{}
arrszref_make_arrpsz
  {a:vt0p}{n:int} (arrpsz (INV(a), n)): arrszref(a)
fun{}
arrszref_make_arrayref
  {a:vt0p}{n:int} (arrayref (a, n), size_t(n)): arrszref(a)
// end of [arrszref_make_arrayref]
```

As an example, the following code creates an array-with-size containing all the decimal digits:

```
val DIGITS = (arrszref)$arrpsz{int}(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Note that `arrszref` is overloaded with `arrszref_make_arrpsz`.

For reading from and writing to an array-with-size, the function templates `arrszref_get_at` and `arrszref_set_at` can be used, respectively, which are assigned the following interfaces:

```
fun{a:t@ype}
arrszref_get_at (A: arrszref (a), i: size_t): (a)
fun{a:t@ype}
arrszref_set_at (A: arrszref (a), i: size_t, x: a): void
```

Given an array-with-size A, an index i and a value v, `arrszref_get_at(A, i)` and `arrszref_set_at(A, i, v)` can be written as `A[i]` and `A[i] := v`, respectively. Notice that array-bounds checking is performed at run-time whenever `arrszref_get_at` or `arrszref_set_at` is called, and the exception `ArraySubscriptExn` is raised in case of out-of-bounds array access being detected.

As a simple example, the following code implements a function that reverses the content of the array inside a given array-with-size:

```
fun{a:t@ype}
arrszref_reverse
(
  A: arrszref (a)
) : void = let
//
val n = A.size()
val n2 = half (n)
//
fun loop
  (i: size_t): void = let
in
  if i < n2 then let
    val tmp = A[i]
    val ni = pred(n)-i
  in
    A[i] := A[ni]; A[ni] := tmp; loop (succ(i))
  end else () // end of [if]
end // end of [loop]
//
in
  loop (i2sz(0))
end // end of [arrszref_reverse]
```

Arrays-with-size can be a good choice over arrays in a prototype implementation as it is often more demanding to program with arrays. Also, for programmers who are yet to become familiar with dependent types, it is definitely easier to work with arrays-with-size than arrays. When programming in ATS, I often start out with arrays-with-size and then replace them with arrays when I can see clear benefits from doing so.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 17. Persistent Matrices

A persistent matrix of dimension m by n is just a persistent array of size m*n. Like in C, the representation of a matrix in ATS is row-major. In other words, element (i, j) in a matrix of dimension m by n is element i*n+j in the underlying array that represents the matrix.

Given a viewtype VT and two integers M and N, the type `matrixref(VT, M, N)` is for persistent matrices of dimension M by N that contain elements of the viewtype VT. There is no dimension information attached to matrixref-values explicitly. The interfaces for various functions on persistent matrices can be found in the SATS file ***prelude/SATS/matrixref.sats***, which is automatically loaded by **atsopt**.

The following function is commonly used to create a matrixref-value:

```
fun{a:t0p}
matrixref_make_elt{m,n:int}
  (m: size_t m, n: size_t n, x0: a):<!wrt> matrixref (a, m, n)
// end of [matrixref_make_elt]
```

Given two sizes m and n plus an element x0, `matrixref_make_elt` returns a matrix of dimension m by n in which each cell is initialized with the element x0.

Also, the following cast function can be called to turn an array into a matrix:

```
castfn
arrayref2matrixref
  {a:vt0p}{m,n:nat} (A: arrayref (a, m*n)):<> matrixref (a, m, n)
// end of [arrayref2matrixref]
```

For accessing and updating the content of a matrix-cell, the following two functions `matrixref_get_at` and `matrixref_set_at` can be called:

```
//
fun{a:t0p}
matrixref_get_at
  {m,n:int}
(
  A: matrixref (a, m, n), i: sizeLt(m), n: size_t(n), j: sizeLt(n)
) :<!ref> (a) // end of [matrixref_get_at]
//
fun{a:t0p}
matrixref_set_at
```

```
  {m,n:int}
(
  A: matrixref (INV(a), m, n), i: sizeLt (m), n: size_t n, j: sizeLt (n), x: a
) :<!refwrt> void // end of [matrixref_set_at]
//
```

Note that it is not enough to just supply the coordinates of a matrix-cell in order to access it; the column dimension of the matrix needs to be supplied as well.

In the following presentation, I give an implementation of a function that turns a given square matrix into its transpose:

```
//
extern
fun{a:t0p}
matrixref_transpose
  {n:nat}
(
  M: matrixref (a, n, n), n: size_t (n)
) : void // end of [matrixref_transpose]
//
implement{a}
matrixref_transpose
  {n} (M, n) = let
//
macdef
mget (i, j) =
  matrixref_get_at (M, ,(i), n, ,(j))
macdef
mset (i, j, x) =
  matrixref_set_at (M, ,(i), n, ,(j), ,(x))
//
fun loop
  {i,j:nat |
   i < j; j <= n
  } .<n-i,n-j>.
(
  i: size_t (i), j: size_t (j)
) : void =
  if j < n then let
    val x = mget(i, j)
    val () = mset(i, j, mget(j, i))
    val () = mset(j, i, x)
  in
    loop (i, j+1)
  end else let
```

```
    val i1 = succ (i)
  in
    if i1 < n then loop (i1, succ(i1)) else ()
  end // end of [if]
//
in
  if n > 0 then loop (i2sz(0), i2sz(1)) else ()
end // end of [matrixref_transpose]
```

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 18. Persistent Matrices-with-size

I use the name *matrix-with-size* to refer to a persistent matrix with attached dimension information (that is, number of rows and number of columns). Given a viewtype VT, the type for a matrix-with-size that contains M rows and N columns of elements of viewtype VT is `mtrxszref(VT, M, N)`. Essentially, such a value is a boxed record of three components of types `arrayref(VT, N)`, `size_t(M)` and `size_t(N)`. The interfaces for various functions on persistent matrices-with-size can be found in ***prelude/SATS/matrixref.sats***.

The following function is commonly used to create a matrix-with-size:

```
fun{a:t0p}
mtrxszref_make_elt (m: size_t, n: size_t, x0: a): mtrxref (a)
// end of [mtrxszref_make_elt]
```

Given two sizes m and n plus an element x0, `mtrxszref_make_elt` returns a matrix-with-size of the dimension m by n in which each matrix-cell is initialized with the given element x0.

For accessing and updating the content of a matrix-cell, the following two functions `mtrxszref_get_at` and `mtrxszref_set_at` can be called:

```
fun{a:t0p}
mtrxszref_get_at (M: mtrxszref(a), i: size_t, j: size_t): (a)
fun{a:t0p}
mtrxszref_set_at (M: mtrxszref(a), i: size_t, j: size_t, x: a): void
```

Given a matrix-with-size M, two indices i and j, and a value v, `mtrxszref_get_at(M, i, j)` and `mtrxszref_set_at(M, i, j, v)` can be written as `M[i,j]` and `M[i,j] := v`, respectively. Notice that matrix-bounds checking is performed at run-time whenever `mtrxszref_get_at` or `mtrxszref_set_at` is called, and the exception `MatrixSubscriptExn` is raised in case of out-of-bounds matrix access being detected.

As a simple example, the following code implements a function that transpose the content of the matrix inside a given matrix-with-size:

```
//
extern
fun{a:t0p}
mtrxszref_transpose (M: mtrxszref (a)): void

```

```
//
implement{a}
mtrxszref_transpose
  (M) = let
//
val n = M.nrow()
//
val ((*void*)) = assertloc (M.nrow() = M.ncol())
//
fun loop
(
  i: size_t, j: size_t
) : void =
  if j < n then let
    val x = M[i,j]
    val () = M[i,j] := M[j,i]
    val () = M[j,i] := x
  in
    loop (i, succ(j))
  end else let
    val i1 = succ (i)
  in
    if i1 < n then loop (i1, succ(i1)) else ()
  end // end of [if]
//
in
  if n > 0 then loop (i2sz(0), i2sz(1)) else ()
end // end of [mtrxszref_transpose]
```

Like arrays-with-size, matrices-with-size are easier to program with than dependently typed matrices. However, the latter can not only lead to more effective error detection at compile-time but also more efficent code execution at run-time. For someone programming in ATS, it is quite reasonable to start out with matrices-with-size and then replace them with matrices when there are clear benefits from doing so.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 19. Persistent Hashtables

Hashtables are commonly used to implement finite maps. In ATSLIB/libats, there are hashtable implementations based on linear chaining and linear probing. There is also support for linear hashtables as well as persistent hashtables. The linear ones can be safely freed by the programmer, and the persistent ones (which are directly based on linear ones) can only be safely reclaimed through garbage collection (GC). In this chapter, I show how persistent hashtables can be created and operated on.

Suppose that a map is needed for mapping keys of type `key_t` to items of type `itm_t`. The following code essentially sets up an interface for creating and operating on such a map based on a hashtable implementation in ATSLIB/libats:

```
local

typedef
key = key_t and itm = itm_t

in (* in-of-local *)

#include "libats/ML/HATS/myhashtblref.hats"

end // end of [local]
```

Please find ***on-line*** the HATS file mentioned in the code, which is just a convenience wrapper made to simplify programming with hashtables.

Assume that `key_t` is `string` and `itm_t` is `int`. The following line of code creates a hashtable with its initial capacity set to be 1000:

```
val mymap = myhashtbl_make_nil(1000)
```

Note that the capacity in this case is the size of the array associated with the created hashtable. The underlying hashtable implementation is based on linear chaining, and this hashtable can store up to 5000 (5*1000) items without need for resizing. When resizing is indeed needed, it is done automatically. The following few lines insert some key/item pairs into `mymap`:

```
//
val-~None_vt() = mymap.insert("a", 0)
val-~Some_vt(0) = mymap.insert("a", 1)
```

```
//
val-~None_vt() = mymap.insert("b", 1)
val-~Some_vt(1) = mymap.insert("b", 2)
//
val-~None_vt() = mymap.insert("c", 2)
val-~Some_vt(2) = mymap.insert("c", 3)
//
```

The dot-symbol `.insert` is overloaded with a function of the name `myhashtbl_insert`. Given a key and an item, `mymap.insert` inserts the key/item pair into `mymap`. If the key is in the domain of the map represented by `mymap` before insertion, then the original item associated with the key is returned. Otherwise, no item is returned. As can be expected, the size of `mymap` is 3 at this point:

```
val () = assertloc (mymap.size() = 3)
```

The dot-symbol `.size` is overloaded with a function of the name `myhashtbl_get_size`, which returns the number of key/item pairs stored in a given hashtable. During the course of debugging, one may want to print out the key/item pairs in a given hashtable:

```
//
val () =
  fprintln! (stdout_ref, "mymap = ", mymap)
//
```

where the symbol `fprint` is overloaded with `fprint_myhashtbl`. The next two lines of code show how search with a given key operates on a hashtable:

```
val-~None_vt() = mymap.search("")
val-~Some_vt(1) = mymap.search("a")
```

The dot-symbol `.search` is overloaded with a function of the name `myhashtbl_search`, which returns the item associated with a given key if it is found. The next few lines of code remove some key/item pairs from `mymap`:

```
//
val-true = mymap.remove("a")
val-false = mymap.remove("a")
//
val-~Some_vt(2) = mymap.takeout("b")
val-~Some_vt(3) = mymap.takeout("c")
//
```

The dot-symbol `.remove` is overloaded with a function of the name `myhashtbl_remove` for removing a key/item pair of a given key. If a key/item pair is removed, then the function returns true. Otherwise, it returns false to indicates that no key/item pair of the given key is stored in the hashtable being operated on. The dot-symbol `.takeout` is overloaded with a function of the name `myhashtbl_takeout`, which is similar to `myhashtbl_remove` excepting for returning the removed item. The next few lines of code make use of several less commonly used functions on hashtables:

```
//
val () = mymap.insert_any("a", 0)
val () = mymap.insert_any("b", 1)
val () = mymap.insert_any("c", 2)
val kxs = mymap.listize1((*void*))
val ((*void*)) = fprintln! (stdout_ref, "kxs = ", kxs)
val kxs = mymap.takeout_all((*void*))
val ((*void*)) = fprintln! (stdout_ref, "kxs = ", kxs)
//
val () = assertloc (mymap.size() = 0)
//
```

The dot-symbol `.insert_any` is overloaded with a function of the name `myhashtbl_insert_any`, which *always* inserts a given key/item pair regardless whether the key is already in use. One should really avoid using this function or only call it when it is absolutely sure that the given key is not already in use for otherwise the involved hashtable would be corrupted. The dot-symbols `.listize1` and `.takeout_all` are overloaded with two functions of the names `myhashtbl_listize1` and `myhashtbl_takeout_all`, respectively. Both of them return a list consisting of all the key/item pairs in a given hashtable; the former keeps the hashtable unchanged while the latter empties it. Last, I present as follows the interface for an iterator going over all the key/item pairs in a given hashtable:

```
//
extern
fun
myhashtbl_foreach_cloref
(
  tbl: myhashtbl
, fwork: (key, &(itm) >> _) -<cloref1> void
) : void // end-of-function
//
```

As an example, the following code prints out all the key/item pairs in a given hashtable:

```
//
```

```
val () =
myhashtbl_foreach_cloref
(
  mymap
, lam (k, x) => fprintln! (stdout_ref, "k=", k, " and ", "x=", x)
) (* myhashtbl_foreach_cloref *)
//
```

Please find the entirety of the code used in this chapter ***on-line***. Also, there is a hashtable-based implementation of symbol table available ***on-line***.

Chapter 20. Tail-Recursion

Please see ***this article*** for a detailed explanation on tail-recursion and the support in ATS for turning tail-recursive calls into local jumps.

Chapter 21. Higher-Order Functions

A higher-order function is one that takes another function as its argument. Let us use BT to range over base types such as `int`, `bool`, `char`, `double` and `string`. A simple type T is formed according to the following inductive definition:

- BT is a simple type.
- (T_1, ..., T_n) -> T_0 is a simple type if T_0, T_1, ... T_n are simple types.

Let *order* be a function from simple types to natural numbers defined as follows:

- *order*(BT) = 0
- *order*((T_1, ..., T_n) -> T_0) = *max*(*order*(T_0), 1 + *order*(T_1), ..., 1 + *order*(T_n))

Given a function f of some simple type T, let us say that f is a *n*th-order function if *order*(T) = *n*. For instance, a function of the type (int, int) -> int is 1st-order, and a function of the type int -> (int -> int) is also 1st-order, and a function of the type ((int -> int), int) -> int is 2nd-order. In practice, most functions are 1st-order and most higher-order functions are 2nd-order.

As an example, let us implement as follows a 2nd-order function `find_root` that takes as its only argument a function f from integers to integers and searches for a root of f by enumeration:

```
fn find_root
(
  f: int -<cloref1> int
) : int = let
//
fun loop
(
  f: int -<cloref1> int, n: int
) : int =
  if f (n) = 0 then n else (
    if n <= 0 then loop (f, ~n + 1) else loop (f, ~n)
  ) // end of [else] // end of [if]
in
  loop (f, 0)
end // end of [find_root]
```

The function `find_root` computes the values of f at 0, 1, -1, 2, -2, etc. until it finds the first integer n in this sequence that satisfies f(n) = 0.

As another example, let us implement as follows the famous Newton-Raphson's method for finding roots of functions on reals:

```
typedef
fdouble = double -<cloref1> double
//
macdef epsilon = 1E-6 (* precision *)
//
// [f1] is the derivative of [f]
//
fun
newton_raphson
(
  f: fdouble, f1: fdouble, x0: double
) : double = let
  fun loop (
    f: fdouble, f1: fdouble, x0: double
  ) : double = let
    val y0 = f x0
  in
    if abs (y0 / x0) < epsilon then x0 else
      let val y1 = f1 x0 in loop (f, f1, x0 - y0 / y1) end
    // end of [if]
  end // end of [loop]
in
  loop (f, f1, x0)
end // end of [newton_raphson]
```

With `newton_raphson`, both the square root function and the cubic root function can be readily implemented as follows:

```
// square root function
fn sqrt (c: double): double =
  newton_raphson (lam x => x * x - c, lam x => 2.0 * x, 1.0)
// cubic root function
fn cbrt (c: double): double =
  newton_raphson (lam x => x * x * x - c, lam x => 3.0 * x * x, 1.0)
```

Higher-order functions can be of great use in supporting a form of code sharing that is both common and flexible. As function arguments are often represented as heap-allocated closures that can only be reclaimed through garbage collection (GC), higher-order functions are used infrequently, if at all, in a setting where GC is not present. In ATS, linear closures, which can be freed explictly in a safe manner, are available to support higher-order functions in the absence of GC, making it possible to employ higher-order functions extensively in systems programming (where GC is unavailable or simply disallowed). The details on linear closures are to be given elsewhere.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 22. Stream-Based Lazy Evaluation

While the core of ATS is based on call-by-value evaluation, there is also direct support in ATS for lazy (that is, call-by-need) evaluation.

There is a special language construct `$delay` for delaying or suspending the evaluation of an expression (by forming a thunk), and there is also a special function `lazy_force` for resuming a suspended evaluation (represented by a thunk). The abstract type constructor `lazy` of the sort `(t@ype) => type` forms a (boxed) type when applied to a type. Given an expression exp of type T, the value `$delay(exp)` of the type `lazy(T)` represents the suspended evaluation of exp. Given a value V of the type `lazy(T)` for some type T, calling `lazy_force` on V resumes the suspended evaluation represented by V. If the call returns, then the returned value is of type T. The interface for the function template `lazy_force` is given as follows:

```
fun{a:t@ype} lazy_force (lazyval: lazy(a)):<!laz> a
```

where the symbol `!laz` indicates a form of effect associated with lazy-evaluation. Note that the special prefix operator `!` in ATS is overloaded with `lazy_force`.

In ***prelude/SATS/stream.sats***, the following types `stream_con` and `stream` are declared mutually recursively for representing lazy streams:

```
datatype
stream_con (a:t@ype+) =
  | stream_nil of ((*void*)) | stream_cons of (a, stream(a))
where stream (a:t@ype) = lazy (stream_con(a))
```

Also, a number of common functions on streams are declared in ***prelude/SATS/stream.sats*** and implemented in ***prelude/DATS/stream.dats***.

The following code gives a standard implementation of the sieve of Eratosthenes:

```
//
fun
from (n: int): stream (int) =
  $delay (stream_cons (n, from (n+1)))
//
fun sieve
(
  ns: stream(int)
) :<!laz> stream(int) = $delay let
//
// [val-] means no warning message from the compiler
//
  val-stream_cons(n, ns) = !ns
in
  stream_cons (n, sieve (stream_filter_cloref<int> (ns, lam x => x mod n > 0)))
end // end of [$delay let] // end of [sieve]
//
val thePrimes = sieve(from(2))
//
```

A stream is constructed consisting of all the integers starting from 2; the first element of the stream is kept and all the multiples of this element are removed from the tail of the stream; this process is then repeated on the tail of the stream recursively. Clearly, the final stream thus generated consists of all the prime numbers ordered ascendingly.

The function template `stream_filter_cloref` is of the following interface:

```
fun{a:t@ype}
stream_filter_cloref
  (xs: stream(a), pred: a -<cloref> bool):<!laz> stream(a)
// end of [stream_filter_cloref]
```

Given a stream and a predicate, `stream_filter_cloref` generates another stream consisting of all the elements in the given stream that satisfy the given predicate.

Let us see another example of lazy evaluation. The follow code demonstrates an interesting approach to computing the Fibonacci numbers:

```
//
val _0_ = $UNSAFE.cast{int64}(0)
val _1_ = $UNSAFE.cast{int64}(1)
//
val // the following values are defined mutually recursively
rec theFibs_0
  : stream(int64) = $delay (stream_cons(_0_, theFibs_1)) // fib0, fib1, ...
and theFibs_1
  : stream(int64) = $delay (stream_cons(_1_, theFibs_2)) // fib1, fib2, ...
and theFibs_2
  : stream(int64) = // fib2, fib3, fib4, ...
(
  stream_map2_fun<int64,int64><int64> (theFibs_0, theFibs_1, lam (x, y) => x + y)
) (* end of [val/and/and] *)
```

```
//
```

The function template `stream_map2_fun` is assigned the following interface:

```
fun{
a1,a2:t0p}{b:t0p
} stream_map2_cloref
(
  xs1: stream (a1), xs2: stream (a2), f: (a1, a2) -<fun> b
) :<!laz> stream (b) // end of [stream_map2_cloref]
```

Given two streams xs1 and xs2 and a binary function f, `stream_map2_fun` forms a stream xs such that xs[n] (that is, element n in xs), if exists, equals f(xs1[n], xs2[n]), where n ranges over natural numbers.

Let us see yet another example of lazy evaluation. A Hamming number is a positive natural number whose prime factors can contain only 2, 3 and 5. The following code shows a straightforward way to generate a stream consisting of all the Hamming numbers:

```
//
val
compare_int_int =
  lam (x1: int, x2: int): int =<fun> compare(x1, x2)
//
macdef
merge2 (xs1, xs2) =
  stream_mergeq_fun<int> (,(xs1), ,(xs2), compare_int_int)
//
val
rec theHamming
  : stream(int) = $delay
(
  stream_cons(1, merge2(merge2(theHamming2, theHamming3), theHamming5))
) (* end of [theHamming] *)

and theHamming2
  : stream(int) = stream_map_fun<int><int> (theHamming, lam x => 2 * x)
and theHamming3
  : stream(int) = stream_map_fun<int><int> (theHamming, lam x => 3 * x)
and theHamming5
  : stream(int) = stream_map_fun<int><int> (theHamming, lam x => 5 * x)
//
```

The function template `stream_mergeq_fun` is given the following interface:

```
fun{a:t0p}
stream_mergeq_fun
(
  xs1: stream (a), xs2: stream (a), (a, a) -<fun> int
) :<!laz> stream (a) // end of [stream_mergeq_fun]
```

Given two streams and an ordering (represented by a function) such that the two streams are listed ascendingly according to the ordering, `stream_mergeq_fun` returns a stream listed ascendingly that represents the union of the two given streams such that any elements in the second stream that also occur in the first stream are dropped.

With stream-based lazy evaluation, an illusion of infinite data can be readily created. This illusion is given a simple programming interface plus automatic support for memoization, enabling a programming style that can often be both elegant and intriguing.

In general, it is difficult to estimate the time-complexity and space-complexity of a program based on lazy evaluation. This is regarded as a serious weakness. With linear stream-based lazy evalution, this weakness can essentially be removed.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 23. Linearly Typed Lists

A linearly typed list in ATS is also referred to as a linear list, which essentially corresponds to a singly-linked list in C. The following linear datatype declaration introduces a linear type `list_vt` for linear lists:

```
//
datavtype
list_vt(a:vt@ype, int) =
| list_vt_nil(a, 0) of ()
| {n:nat}
  list_vt_cons(a, n+1) of (a, list_vt(a, n))
//
```

Note that the keyword `datavtype` can also be written as `dataviewtype`. Given a (possibly linear) type T and an integer N, the type `list_vt(T,N)` is for a list of length N that contains elements of type T. The interfaces for various functions on linear lists can be found in the SATS file ***prelude/SATS/list_vt.sats***, which is automatically loaded by **atsopt**.

The following function `list_vt_length` shows a typical way of handling a linear list in a read-only situation:

```
//
fun
{a:vt@ype}
list_vt_length
  (xs: !list_vt(a, n)): int(n) =
(
case+ xs of
| list_vt_nil() => 0
| list_vt_cons(x, xs2) => 1 + list_vt_length<a> (xs2)
)
//
```

When `xs` is matched with the pattern `list_vt_nil()`, the type of `xs` is `list_vt(a, 0)`. When `xs` is matched with the pattern `list_vt_cons(x, xs2)`, the type of `xs` is `list_vt(a, N+1)` for some natural number N and the types of `x` and `xs2` are `a` and `list_vt(a, N)`, respectively. Note that both `x` and `xs2` are names for values, and their types are required to stay unchanged.

The following function `list_vt_foreach` shows a typical way of modifying elements stored in a linear list:

```
//
fun
{a:vt@ype}
list_vt_foreach
(
  xs: !list_vt(a, n)
, fwork: (&(a) >> _) -<cloref1> void
) : void =
(
case+ xs of
| list_vt_nil() => ()
| @list_vt_cons(x, xs2) => (fwork(x); list_vt_foreach<a> (xs2, fwork); fold@(xs))
)
//
```

When `xs` is matched with the pattern `@list_vt_cons(x,xs2)`, the type of `xs` is `list_vt(a, N+1)` for some natural number N and the types of `x` and `xs2` are `a` and `list_vt(a, N)`, respectively. Note that both `x` and `xs2` are variables (that are a form of left-values). At the beginning of the body following the pattern `@list_vt_cons(x,xs2)`, the type of `xs` is assumed to be `list_vt_cons_unfold(L0, L1, L2)`, which is a viewtype for a list-node created by a call to `list_vt_cons` such that the node is located at L0 and the two arguments of `list_vt_cons` are located at L1 and L2 while the proofs for the at-views associated with L1 and L2 are put in the store for currently available proofs. Therefore, as left-values, `x` and `xs2` have addresses L1 and L2, respectively, and the views of the proofs associated with L1 and L2 are `a@L1` and `list_vt_cons(a, N)@L2`, respectively. The application `fold@(xs)` turns `xs` into a value of the type `list_vt(a, N+1)` while consuming the proofs associated with L1 and L2. Notice that the type of `xs` can be different from the original one assigned to it after folding. The following example shows a case as such:

```
//
fun
{a:vt@ype}
list_vt_append
  {m,n:nat}
(
  xs: list_vt(a, m), ys: list_vt(a, n)
) : list_vt(a, m+n) = let
//
fun
loop{m:nat}
(
  xs: &list_vt(a, m) >> list_vt(a, m+n), ys: list_vt(a, n)
) : void =
(
case+ xs of
| ~list_vt_nil() => (xs := ys)
| @list_vt_cons(x, xs2) => (loop(xs2, ys); fold@(xs))
)
//
in
  case+ xs of
  | ~list_vt_nil () => ys
  | @list_vt_cons (x, xs2) => (loop(xs2, ys); fold@(xs); xs)
end // end of [list_vt_append]
//
```

The meaning of the symbol `~` in front of a pattern is to be explained below. The implementation of `list_vt_append` exactly corresponds to the standard implementaion of concatenating two singly-linked lists in C: Let xs and ys be two given lists; if xs is empty, then ys is returned; otherwise, the last node in xs is located and ys is stored in the field of the node reserved for the next node.

The following function `list_vt_free` frees a given linear list containing non-linear elements:

```
//
fun
{a:vt@ype}
list_vt_free
  {n:nat}
(
  xs: list_vt(a?, n)
) : void =
(
case+ xs of
| ~list_vt_nil() => ()
| ~list_vt_cons(x, xs2) => list_vt_free<a> (xs2)
)
//
```

When `xs` is matched with the pattern `~list_vt_nil()`, the type of `xs` changes to a special one indicating that `xs` is no longer available for subsequent use. When `xs` is matched with the pattern `~list_vt_cons(x,xs2)`, the type of `xs` changes again to a special one indicating that `xs` is no longer available for subsequent use. In the latter case, the two values representing the head and tail of the list referred to as `xs` can be subsequently referred to as `x` and `xs2`, respectively. So what is really freed here is the memory for the first list-node in the list referred to as `xs`.

Please find ***on-line*** the entirety of the code used in this chapter plus some testing code.

II. Advanced Tutorial Topics

Table of Contents

Chapter 24. Extvar-Declaration

ATS puts great emphasis on interacting with other programming languages.

Suppose that I have in some C code a (global) integer variable of the name `foo` and I want to increase in some ATS code the value stored in `foo` by 1. This can be done as follows:

```
val x0 = $extval(int, "foo") // get the value of foo
val p_foo = $extval(ptr, "&foo") // get the address of foo
val () = $UNSAFE.ptr_set<int> (p_foo, x0 + 1) // update foo
```

where the address-of operator (&) in C is needed for taking the address of `foo`. If I want to interact in ATS with a language that does not support the address-of operator (e.g., JavaScript and Python), then I can do it as follows:

```
extvar "foo" = x0 + 1
```

where the keyword `extvar` indicates that the string following it refers to an external variable (or left-value) that should be updated with the value of the expression on the right-hand side of the equality symbol following the string. Of course, this works for languages like C that do support the address-of operator as well. This so-called extvar-declaration can also be written as follows:

```
extern var "foo" = x0 + 1
```

where `extvar` expands into `extern var`.

As for another example, let us suppose that `foo2` is a record variable that contains two integer fields named `first` and `second`. Then the following code assigns integers 1 and 2 to these two fields of `foo2`:

```
extvar "foo2.first" = 1
extvar "foo2.second" = 2
```

By its very nature, the feature of extvar-declaration is inherently unsafe, and it should only be used with caution.

Please find ***on-line*** the entirety of the code presented in this chapter.

Chapter 25. Linear Closure-Functions

A closure-function is a boxed record that contains a pointer to an envless function plus bindings that map certain names in the body of the envless function to values. In practice, a function argument of a higher-order function is often a closure-function (instead of an envless function). For instance, the following higher-order function `list_map_cloref` takes a closure-function as its second argument:

```
fun{
a:t@ype}{b:t@ype
} list_map_cloref{n:int}
  (xs: list (a, n), f: (a) -<cloref> b): list_vt (b, n)
```

Closure-functions can be either linear or non-linear, and linear ones can be explicitly freed in a safe manner. The keyword `-<cloref>` is used to form a type for non-linear closure-functions. As a variant of `list_map_cloref`, the following higher-order function `list_map_cloptr` takes a linear closure-function as its second argument:

```
fun{
a:t@ype}{b:t@ype
} list_map_cloptr{n:int}
  (xs: list (a, n), f: !(a) -<cloptr> b): list_vt (b, n)
```

As can be easily guessed, the keyword `-<cloptr>` is used to form a type for linear closure-functions. Note that the symbol `!` indicates that the second argument is still available after a call to `list_map_cloptr` returns.

A typical example making use of `list_map_cloptr` is given as follows:

```
fun foo{n:int}
(
  x0: int, xs: list (int, n)
) : list_vt (int, n) = res where
{
//
val f = lam (x) =<cloptr> x0 + x
val res = list_map_cloptr (xs, f)
val () = cloptr_free ($UNSAFE.cast{cloptr(void)}(f))
//
} (* end of [foo] *)
```

Note that a linear closure is first created in the body of the function `foo`, and it is explicitly freed after its use. The function `cloptr_free` is given the following interface:

```
fun cloptr_free {a:t0p} (pclo: cloptr (a)): void
```

where `cloptr` is abstract. The cast `$UNSAFE.cast{cloptr(void)}(f)` can certainly be replaced with something safer but it would make programming more curbersome.

There is also some interesting interaction between currying and linear closure-functions. In functional programming, currying means turning a function taking multiple arguments simutaneously into a corresponding one that takes these arguments sequentially. For instance, the function `acker2` in the following code is a curried version of the function `acker`, which implements the famous Ackermann function (that is recursive but not primitive recursive):

```
fun
acker(m:int, n:int): int =
(
  case+ (m, n) of
  | (0, _) => n + 1
  | (m, 0) => acker (m-1, 1)
  | (_, _) => acker (m-1, acker (m, n-1))
) (* end of [acker] *)

fun acker2 (m:int) (n:int): int = acker (m, n)
```

Suppose that we apply `acker2` to two integers 3 and 4: `acker2(3)(4)`; the application `acker2(3)` evaluates to a (non-linear) closure-function; the application of this closure-function to 4 evaluates to `acker(3,4)`, which further evaluates to the integer 125. Note that the closure-function generated from evaluating `acker2(3)` becomes a heap-allocated value that is no longer accessible after the evaluation of `acker2(3)(4)` finishes, and the memory for such a value can only to be safely reclaimed through garbage collection (GC).

It is also possible to define a curried version of `acker` as follows:

```
fun acker3 (m:int) = lam (n:int): int =<cloptr1> acker (m, n)
```

While the evaluation of `acker3(3)(4)` yields the same result as `acker2(3)(4)`, the compiler of ATS (ATS/Postiats) inserts code that automatically frees the linear closure-function generated from evaluating `acker3(3)` after the evaluation of `acker3(3)(4)` finishes.

In ATS1, linear closure-functions play a pivotal role in supporting programming with higher-order functions in the absence of GC. Due to advanced support for templates in ATS2, the role played by linear closure-functions in ATS1 is greatly diminished. However, if closure-functions need to be stored in a data structure but GC is unavailable or undesirable, then using linear closure-functions can lead to a solution that avoids the risk of generatig memory leaks at run-time.

Please find ***on-line*** the entirety of the code used in this chapter.

Chapter 26. Automatic Code Generation

In practice, one often encounters a need to write boilerplate code or code that tends to follow certain clearly recognizable patterns. It is commonly seen that meta-programming (of various forms) is employed to automatically generate such code, thus not only increasing programming productivity but also potentially eliminating bugs that would otherwise be introduced due to manual code construction.

In the following presentation, I am to show that the ATS compiler can be directed to generate the code for certain functions on values of a declared datatype. Following is the datatype used for illustration:

```
//
datatype expr =
  | Int of int
  | Var of string
  | Add of (expr, expr)
  | Sub of (expr, expr)
  | Mul of (expr, expr)
  | Div of (expr, expr)
  | Ifgtz of (expr, expr, expr) // if expr > 0 then ... else ...
  | Ifgtez of (expr, expr, expr) // if expr >= 0 then ... else ...
//
```

which is for some kind of abstract syntax trees representing arithmetic expressions.

Generating a datcon-function

Given a datatype, its datcon-function is the one that takes a value of the datatype and then returns a string representing the name of the (outmost) constructor in the construction of the value. We can use the following directive to indicate (to the ATS compiler) that the datcon-function for the datatype `expr` needs to be generated:

```
#codegen2("datcon", expr)
```

By default, the name of the generated function is `datcon_expr`. If a different name is needed, it can be supplied as the third argument of the `#codegen2`-directive. For instance, the following directive indicates that the generated function is of the given name `my_datcon_expr`:

```
#codegen2("datcon", expr, my_datcon_expr)
```

Assume that a file of the name `expr.dats` contains the following directive (as a toplevel declaration):

```
#codegen2("datcon", expr)
```

and the definition for `expr` is accessible at the point where the `codegen2`-directive is declared. By executing the following command-line:

```
patscc --codegen-2 -d expr.dats
```

we can see some output of ATS code that implements `datcon_expr`:

```
(* ****** ****** *)
//
implement
{}(*tmp*)
datcon_expr
  (arg0) =
(
case+ arg0 of
| Int _ => "Int"
| Var _ => "Var"
| Add _ => "Add"
| Sub _ => "Sub"
| Mul _ => "Mul"
| Div _ => "Div"
| Ifgtz _ => "Ifgtz"
| Ifgtez _ => "Ifgtez"
)
//
(* ****** ****** *)
```

If the output needs to be stored in a file of the name `fprint_expr.hats`, we can issue the following command-line:

```
patscc -o fprint_expr.hats --codegen-2 -d expr.dats
```

Note that the funtion template `datcon_expr` is required to be declared somewhere in order for the generated code to be compiled properly:

```
fun{} datcon_expr : (expr) -> string // a function template
```

Please find ***on-line*** the entirety of this presented example plus a Makefile (for illustrating the code generation process).

Generating a datcontag-function

A datcontag-function is very similar to a datcon-function. Given a datatype, its datcontag-function is the one that takes a value of the datatype and then returns the tag (which is a small integer) assigned to the (outmost) constructor in the construction of the value. We can use the following directive to indicate (to the ATS compiler) that the datcontag-function for the datatype `expr` needs to be generated:

```
#codegen2("datcontag", expr)
```

By default, the name of the generated function is `datcontag_expr`. If a different name is needed, it can be supplied as the third argument of the `#codegen2`-directive. For instance, the following directive indicates that the generated function is of the given name `my_datcontag_expr`:

```
#codegen2("datcontag", expr, my_datcontag_expr)
```

The following ATS code is expected to be generated that implements `datcontag_expr`:

```
(* ****** ****** *)
//
implement
{}(*tmp*)
datcontag_expr
  (arg0) =
(
case+ arg0 of
| Int _ => 0
| Var _ => 1
| Add _ => 2
| Sub _ => 3
| Mul _ => 4
| Div _ => 5
| Ifgtz _ => 6
| Ifgtez _ => 7
)
//
(* ****** ****** *)
```

Note that the funtion template `datcontag_expr` is required to be declared somewhere in order for the generated code to be compiled properly:

```
fun{} datcontag_expr : (expr) -> intGte(0) // a function template
```

Please find ***on-line*** the entirety of this presented example plus a Makefile (for illustrating the code generation process).

Generating a fprint-function

A fprint-function takes a file-handle (of the type `FILEref`) and a value and then outputs a text representation of the value to the file-handle. Given a datatype, one is often in need of a function that can output certain kind of text representation for values of this datatype. For instance, such a function can be of great use in debugging.

Let us first declare a function template `fprint_expr` as follows:

```
fun{} fprint_expr : (FILEref, expr) -> void // a function template
```

We can then use the directive below to indicate (to the ATS compiler) that the fprint-function for the datatype `expr` needs to be generated:

```
#codegen2("fprint", expr, fprint_expr)
```

The third argument of the `codegen2`-directive can be omitted in this case as it coincides with the default. The generated code that implements `fprint_expr` is listed as follows:

```
(* ****** ****** *)
//
extern
fun{}
fprint_expr$Int: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Var: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Add: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Sub: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Mul: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Div: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtz: $d2ctype(fprint_expr<>)
extern
```

```
fun{}
fprint_expr$Ifgtez: $d2ctype(fprint_expr<>)
//
(* ****** ****** *)
//
implement{}
fprint_expr
  (out, arg0) =
(
case+ arg0 of
| Int _ => fprint_expr$Int<>(out, arg0)
| Var _ => fprint_expr$Var<>(out, arg0)
| Add _ => fprint_expr$Add<>(out, arg0)
| Sub _ => fprint_expr$Sub<>(out, arg0)
| Mul _ => fprint_expr$Mul<>(out, arg0)
| Div _ => fprint_expr$Div<>(out, arg0)
| Ifgtz _ => fprint_expr$Ifgtz<>(out, arg0)
| Ifgtez _ => fprint_expr$Ifgtez<>(out, arg0)
)
//
(* ****** ****** *)
//
extern
fun{}
fprint_expr$sep: (FILEref) -> void
implement{}
fprint_expr$sep(out) = fprint(out, ",")
//
extern
fun{}
fprint_expr$lpar: (FILEref) -> void
implement{}
fprint_expr$lpar(out) = fprint(out, "(")
//
extern
fun{}
fprint_expr$rpar: (FILEref) -> void
implement{}
fprint_expr$rpar(out) = fprint(out, ")")
//
extern
fun{a:t0p}
fprint_expr$carg: (FILEref, INV(a)) -> void
implement{a}
fprint_expr$carg(out, arg) = fprint_val<a>(out, arg)
//
(* ****** ****** *)
```

```
//
extern
fun{}
fprint_expr$Int$con: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Int$lpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Int$rpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Int$arg1: $d2ctype(fprint_expr<>)
//
implement{}
fprint_expr$Int(out, arg0) =
{
//
val () = fprint_expr$Int$con<>(out, arg0)
val () = fprint_expr$Int$lpar<>(out, arg0)
val () = fprint_expr$Int$arg1<>(out, arg0)
val () = fprint_expr$Int$rpar<>(out, arg0)
//
}
implement{}
fprint_expr$Int$con(out, _) = fprint(out, "Int")
implement{}
fprint_expr$Int$lpar(out, _) = fprint_expr$lpar(out)
implement{}
fprint_expr$Int$rpar(out, _) = fprint_expr$rpar(out)
implement{}
fprint_expr$Int$arg1(out, arg0) =
  let val-Int(arg1) = arg0 in fprint_expr$carg(out, arg1) end
//
extern
fun{}
fprint_expr$Var$con: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Var$lpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Var$rpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Var$arg1: $d2ctype(fprint_expr<>)
//
```

```
implement{}
fprint_expr$Var(out, arg0) =
{
//
val () = fprint_expr$Var$con<>(out, arg0)
val () = fprint_expr$Var$lpar<>(out, arg0)
val () = fprint_expr$Var$arg1<>(out, arg0)
val () = fprint_expr$Var$rpar<>(out, arg0)
//
}
implement{}
fprint_expr$Var$con(out, _) = fprint(out, "Var")
implement{}
fprint_expr$Var$lpar(out, _) = fprint_expr$lpar(out)
implement{}
fprint_expr$Var$rpar(out, _) = fprint_expr$rpar(out)
implement{}
fprint_expr$Var$arg1(out, arg0) =
  let val-Var(arg1) = arg0 in fprint_expr$carg(out, arg1) end
//
extern
fun{}
fprint_expr$Add$con: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Add$lpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Add$rpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Add$sep1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Add$arg1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Add$arg2: $d2ctype(fprint_expr<>)
//
implement{}
fprint_expr$Add(out, arg0) =
{
//
val () = fprint_expr$Add$con<>(out, arg0)
val () = fprint_expr$Add$lpar<>(out, arg0)
val () = fprint_expr$Add$arg1<>(out, arg0)
val () = fprint_expr$Add$sep1<>(out, arg0)
val () = fprint_expr$Add$arg2<>(out, arg0)
val () = fprint_expr$Add$rpar<>(out, arg0)
//
}
implement{}
fprint_expr$Add$con(out, _) = fprint(out, "Add")
implement{}
fprint_expr$Add$lpar(out, _) = fprint_expr$lpar(out)
implement{}
fprint_expr$Add$rpar(out, _) = fprint_expr$rpar(out)
implement{}
fprint_expr$Add$sep1(out, _) = fprint_expr$sep<>(out)
implement{}
fprint_expr$Add$arg1(out, arg0) =
  let val-Add(arg1, _) = arg0 in fprint_expr$carg(out, arg1) end
implement{}
fprint_expr$Add$arg2(out, arg0) =
  let val-Add(_, arg2) = arg0 in fprint_expr$carg(out, arg2) end
//
extern
fun{}
fprint_expr$Sub$con: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Sub$lpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Sub$rpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Sub$sep1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Sub$arg1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Sub$arg2: $d2ctype(fprint_expr<>)
//
implement{}
fprint_expr$Sub(out, arg0) =
{
//
val () = fprint_expr$Sub$con<>(out, arg0)
val () = fprint_expr$Sub$lpar<>(out, arg0)
val () = fprint_expr$Sub$arg1<>(out, arg0)
val () = fprint_expr$Sub$sep1<>(out, arg0)
val () = fprint_expr$Sub$arg2<>(out, arg0)
val () = fprint_expr$Sub$rpar<>(out, arg0)
//
}
implement{}
fprint_expr$Sub$con(out, _) = fprint(out, "Sub")
implement{}
fprint_expr$Sub$lpar(out, _) = fprint_expr$lpar(out)
implement{}
fprint_expr$Sub$rpar(out, _) = fprint_expr$rpar(out)
implement{}
fprint_expr$Sub$sep1(out, _) = fprint_expr$sep<>(out)
implement{}
fprint_expr$Sub$arg1(out, arg0) =
  let val-Sub(arg1, _) = arg0 in fprint_expr$carg(out, arg1) end
implement{}
fprint_expr$Sub$arg2(out, arg0) =
  let val-Sub(_, arg2) = arg0 in fprint_expr$carg(out, arg2) end
//
extern
fun{}
fprint_expr$Mul$con: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Mul$lpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Mul$rpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Mul$sep1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Mul$arg1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Mul$arg2: $d2ctype(fprint_expr<>)
//
implement{}
fprint_expr$Mul(out, arg0) =
{
//
val () = fprint_expr$Mul$con<>(out, arg0)
val () = fprint_expr$Mul$lpar<>(out, arg0)
val () = fprint_expr$Mul$arg1<>(out, arg0)
val () = fprint_expr$Mul$sep1<>(out, arg0)
val () = fprint_expr$Mul$arg2<>(out, arg0)
val () = fprint_expr$Mul$rpar<>(out, arg0)
```

```
//
}
implement{}
fprint_expr$Mul$con(out, _) = fprint(out, "Mul")
implement{}
fprint_expr$Mul$lpar(out, _) = fprint_expr$lpar(out)
implement{}
fprint_expr$Mul$rpar(out, _) = fprint_expr$rpar(out)
implement{}
fprint_expr$Mul$sep1(out, _) = fprint_expr$sep<>(out)
implement{}
fprint_expr$Mul$arg1(out, arg0) =
  let val-Mul(arg1, _) = arg0 in fprint_expr$carg(out, arg1) end
implement{}
fprint_expr$Mul$arg2(out, arg0) =
  let val-Mul(_, arg2) = arg0 in fprint_expr$carg(out, arg2) end
//
extern
fun{}
fprint_expr$Div$con: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Div$lpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Div$rpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Div$sep1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Div$arg1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Div$arg2: $d2ctype(fprint_expr<>)
//
implement{}
fprint_expr$Div(out, arg0) =
{
//
val () = fprint_expr$Div$con<>(out, arg0)
val () = fprint_expr$Div$lpar<>(out, arg0)
val () = fprint_expr$Div$arg1<>(out, arg0)
val () = fprint_expr$Div$sep1<>(out, arg0)
val () = fprint_expr$Div$arg2<>(out, arg0)
val () = fprint_expr$Div$rpar<>(out, arg0)
//
```

```
}
implement{}
fprint_expr$Div$con(out, _) = fprint(out, "Div")
implement{}
fprint_expr$Div$lpar(out, _) = fprint_expr$lpar(out)
implement{}
fprint_expr$Div$rpar(out, _) = fprint_expr$rpar(out)
implement{}
fprint_expr$Div$sep1(out, _) = fprint_expr$sep<>(out)
implement{}
fprint_expr$Div$arg1(out, arg0) =
  let val-Div(arg1, _) = arg0 in fprint_expr$carg(out, arg1) end
implement{}
fprint_expr$Div$arg2(out, arg0) =
  let val-Div(_, arg2) = arg0 in fprint_expr$carg(out, arg2) end
//
extern
fun{}
fprint_expr$Ifgtz$con: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtz$lpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtz$rpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtz$sep1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtz$sep2: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtz$arg1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtz$arg2: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtz$arg3: $d2ctype(fprint_expr<>)
//
implement{}
fprint_expr$Ifgtz(out, arg0) =
{
//
val () = fprint_expr$Ifgtz$con<>(out, arg0)
val () = fprint_expr$Ifgtz$lpar<>(out, arg0)
val () = fprint_expr$Ifgtz$arg1<>(out, arg0)
val () = fprint_expr$Ifgtz$sep1<>(out, arg0)
val () = fprint_expr$Ifgtz$arg2<>(out, arg0)
val () = fprint_expr$Ifgtz$sep2<>(out, arg0)
val () = fprint_expr$Ifgtz$arg3<>(out, arg0)
val () = fprint_expr$Ifgtz$rpar<>(out, arg0)
//
}
implement{}
fprint_expr$Ifgtz$con(out, _) = fprint(out, "Ifgtz")
implement{}
fprint_expr$Ifgtz$lpar(out, _) = fprint_expr$lpar(out)
implement{}
fprint_expr$Ifgtz$rpar(out, _) = fprint_expr$rpar(out)
implement{}
fprint_expr$Ifgtz$sep1(out, _) = fprint_expr$sep<>(out)
implement{}
fprint_expr$Ifgtz$sep2(out, _) = fprint_expr$sep<>(out)
implement{}
fprint_expr$Ifgtz$arg1(out, arg0) =
  let val-Ifgtz(arg1, _, _) = arg0 in fprint_expr$carg(out, arg1) end
implement{}
fprint_expr$Ifgtz$arg2(out, arg0) =
  let val-Ifgtz(_, arg2, _) = arg0 in fprint_expr$carg(out, arg2) end
implement{}
fprint_expr$Ifgtz$arg3(out, arg0) =
  let val-Ifgtz(_, _, arg3) = arg0 in fprint_expr$carg(out, arg3) end
//
extern
fun{}
fprint_expr$Ifgtez$con: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtez$lpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtez$rpar: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtez$sep1: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtez$sep2: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtez$arg1: $d2ctype(fprint_expr<>)
extern
```

```
fun{}
fprint_expr$Ifgtez$arg2: $d2ctype(fprint_expr<>)
extern
fun{}
fprint_expr$Ifgtez$arg3: $d2ctype(fprint_expr<>)
//
implement{}
fprint_expr$Ifgtez(out, arg0) =
{
//
val () = fprint_expr$Ifgtez$con<>(out, arg0)
val () = fprint_expr$Ifgtez$lpar<>(out, arg0)
val () = fprint_expr$Ifgtez$arg1<>(out, arg0)
val () = fprint_expr$Ifgtez$sep1<>(out, arg0)
val () = fprint_expr$Ifgtez$arg2<>(out, arg0)
val () = fprint_expr$Ifgtez$sep2<>(out, arg0)
val () = fprint_expr$Ifgtez$arg3<>(out, arg0)
val () = fprint_expr$Ifgtez$rpar<>(out, arg0)
//
}
implement{}
fprint_expr$Ifgtez$con(out, _) = fprint(out, "Ifgtez")
implement{}
fprint_expr$Ifgtez$lpar(out, _) = fprint_expr$lpar(out)
implement{}
fprint_expr$Ifgtez$rpar(out, _) = fprint_expr$rpar(out)
implement{}
fprint_expr$Ifgtez$sep1(out, _) = fprint_expr$sep<>(out)
implement{}
fprint_expr$Ifgtez$sep2(out, _) = fprint_expr$sep<>(out)
implement{}
fprint_expr$Ifgtez$arg1(out, arg0) =
  let val-Ifgtez(arg1, _, _) = arg0 in fprint_expr$carg(out, arg1) end
implement{}
fprint_expr$Ifgtez$arg2(out, arg0) =
  let val-Ifgtez(_, arg2, _) = arg0 in fprint_expr$carg(out, arg2) end
implement{}
fprint_expr$Ifgtez$arg3(out, arg0) =
  let val-Ifgtez(_, _, arg3) = arg0 in fprint_expr$carg(out, arg3) end
//
(* ****** ****** *)
```

The code for `fprint_expr` is entirely template-based. This style makes the code extremely flexible for adaption through template re-mplementation. As the datatype `expr` is recursively defined, the following template implementation needs to be added in order to make `fprint_expr` work:

```
implement fprint_expr$card<expr> = fprint_expr
```

For instance, applying `fprint_expr` to the expression `Add(Int(10),Mul(Int(1),Int(2)))` outputs the same text representation. As an example of adaptation, let us add the following template implementations:

```
implement
fprint_expr$Add$con<> (_, _) = ()
implement
fprint_expr$Add$sep1<> (out, _) = fprint! (out, "+")
```

When `fprint_expr` is applied to the expression `Add(Int(10),Mul(Int(1),Int(2)))` this time, the output is expected to read `(Int(10)+Mul(Int(1),Int(2)))`.

After proper adaptation is done, one can introduce a (non-template) function as follows:

```
//
extern
fun
my_fprint_expr(FILEref, expr): void
implement
my_fprint_expr (out, x) = fprint_expr<> (out, x)
//
```

In this way, only one instance of `fprint_expr` is compiled even if repeated calls to `my_fprint_expr` are made.

Please find ***on-line*** the entirety of this presented example plus a Makefile (for illustrating the code generation process).

Done.