A Functional Crash into ATS

Hongwei Xi

gmhwxiATgmailDOTcom

A Functional Crash into ATS

by Hongwei Xi

Table of Contents

Preface

This book consists of a set of lecture notes for a crash course on functional programming in ATS. The primary focus of these notes is set upon advocating a style of problem-solving that makes extensive use of list-processing and stream-processing functions. The average reader is expected to have already had some exposure to programming (e.g., having taken one semester of programming course based on Java or Python) but may not know much or anything about functional programming. After finishing this course, one will have become familiar with recursion in problem-solving and ready to formulate high-level combinator-based programming solutions.

Chapter 1. Introducing ATS

1.1. What is ATS?

ATS is a statically typed programming language with a functional programming core inspired by ML.

Aiming at unifying implementation with formal specification, ATS is equipped with a highly expressive type system rooted in the framework Applied Type System, which in turn gives the language its name. In particular, ATS supports practical use of both dependent types and linear types in capturing program invariants. Moreover, ATS supports a form of embeddable templates in the sense that such a template may be implemented inside the body of a function (so as to allow the implementation to have direct access to the arguments of the function), greatly facilitating code reuse.

There is also a subsystem ATS/LF in ATS for accommodating a form of (interactive) theorem-proving where proofs are constructed as total functions. With this subsystem, ATS is able to advocate a programmer-centric approach to program verification that combines programming with theorem-proving in a syntactically intertwined manner.

1.2. What can ATS bring?

Learning ATS can help a programmer greatly enhance his or her programming productivity.

For most people, learning basic programming is easy and fun but there seems to be no clear path for one to ever reach the level of a programming expert who can confidently design and implement large and complex software systems. It is very common to see that the initial excitement one receives from some sort of exposure to programming is quickly replaced with the endless need for debugging. And one may even draw the conclusion that the only way to obtain functioning code of quality is by going through a lengthy debugging process, which of course is far from the truth.

In this book, I will be primarily using ATS as a vehicle to teach a style of programing that is often referred to as functional programming (FP). In particular, I will be advocating the use of recursion in problem-solving as well as the need to prevent potential stack overflow caused by deeply nested recursive function calls. I will also be demonstrating concretely and repeatedly that one can drastically reduce the need for debugging by making extensive use of types in programming.

1.3. Installing ATS

There are many approaches to installing ATS. Please visit this link

(http://www.ats-lang.org/Downloads.html) for details.

I have a free account with Cloud-9 (http://c9.io/), which allows me to gain on-line access to Ubuntu boxes. With such a box, one can simply execute the following command-lines to install ATS in a couple minutes:

```
wget https://ats-lang.github.io/SCRIPT/C9-ATS2-install-cs320.sh
bash -v C9-ATS2-install-cs320.sh
source ~/.bashrc ## this one only needs to be executed once for all
```

After installation is done, one can execute:

```
which patscc
```

and expect to see the following line of output:

```
/home/ubuntu/workspace/ATS2/bin/patscc
```

Note that the home directory for ATS is `/home/ubuntu/workspace/ATS2`, which is stored in an environment variable of the name *PATSHOME*. Please use **which** to check that the commands **patsopt** and **myatscc** are also available.

By visiting ats-bucs320 (https://ide.c9.io/c9hwxi/ats-bucs320), one can browse the two installed directories of the names *ATS2* and *ATS2-contrib* and the files contained in them. If re-installing ATS is ever needed, please remember that the installed directories *ATS2* and *ATS2-contrib* should be removed before the steps outlined above for installation are repeated.

Please note that more scripts for installing ATS can be found on-line at Scripts_for_installing_ATS (http://www.ats-lang.org/Downloads.html#Scripts_for_installing_ATS_Postiats).

1.4. Compiling ATS code

ATS code is first compiled into code written in a very restricted subset of C. The generated C code from ATS source can then be compiled into object code or transpiled into code written in a variety of programming languages including Javascript, PHP, Perl, Python, Erlang, Scheme, Clojure, etc.

The command **patsopt** is for compiling ATS code into C code, which can then be compiled into object code by a standard C compiler such as *gcc* and *clang*. The command **patscc** is a convenience wrapper around **patsopt**, and it can be called, for instance, to compile ATS code into object code directly. For compiling a single-file ATS program into a standalone executable, the command **myatscc** can be used instead.

As an example, let us assume that the following code (for printing out the string "Hello, world!" plus a newline) is stored in a file of the name `hello.dats`:

```
//
val _ = print ("Hello, world!\\n")
//
implement main0 () = () // a dummy for [main]
//
```

One can produce an executable of the name `hello_dats` by simply executing the command-line below:

```
myatscc hello.dats
```

When the command **./hello_dats** is executed, the expected message is printed onto the console. Please give it a try!

When a project consisting of multiple files written in ATS needs to be compiled, I often construct a Makefile and then rely on the **make** utility to streamline the compilation process.

1.5. Accessing ATS libraries

There are many libraries in the ATS programming language system. For instance, the name `ATSLIB/prelude` refers to the prelude library in ATS that targets C code generation; the name `ATSLIB/libats` refers to the libats library in ATS that targets C code generation as well; the name `LIBATSCC2JS` refers to a library for targeting Javascript (JS) code generation; the name `LIBATSCC2PHP` refers to a library for targeting PHP code generation; etc.

If a function is implemented as a template, the ATS compiler needs to access the source of the function implementation in order to generate code for each instance call of the function. For instance, the following line can be added to the beginning of a file so as to indicate to the compiler that `ATSLIB/prelude` needs to be first loaded for compiling the rest of the ATS code in the file:

```
//
#include "share/atspre_staload.hats"
//
```

For accessing the library `ATSLIB/libats/ML` (which is somewhat an ML-like portion of `ATSLIB/libats`), the following line can be used:

```
//
#include "share/atspre_staload_libats_ML.hats"
//
```

Various library functions are to be introduced in the sample code presented in this book.

Chapter 2. Recursive Functions (1)

Recursion, which literally means *running back*, plays a pivotal role in problem-solving. For instance, a problem-solving strategy based on divide-and-conquer divides a given problem (that cannot be solved immediately) into subproblems of a similar nature so that the same strategy can be applied to these subproblems *recursively*; the obtained solutions to these subproblems are then combined in some manner to form a solution to the original problem.

As the first example, let us implement the factorial function that takes a natural number n and returns the product of the first n positive integers. In the following code, a function of the name fact is defined recursively:

```
//
fun
fact(n: int): int =
if n > 0 then n * fact(n-1) else 1
//
```

The keyword fun initiates the definition of a function, and the function header following fun specifies that fact is a function taking an integer as its argument and returning another integer as its result. For testing fact, let us include the following line:

```
val () = println! ("fact(10) = ", fact(10))
```

where println! is a function-like in ATS, which resembles a function but is not actually a function. In this case, println! can be thought of as a macro that calls a print function on each of its arguments and then prints out a newline at the end. Please do not forget the symbol ! in the name println!.

Suppose we need to call fact on all of the natural numbers less than a given one (e.g., 100). We can first define a function testfact as follows and then call testfact on the given natural number:

```
//
fun
testfact
(n: int): void =
if n > 0 then
(
  testfact(n-1);
  println! ("fact(", n-1, ") = ", n-1)
) (* end of [if] *)
//
```

The function header for testfact indicates that testfact takes an integer as its arguments and returns a void-value (of the type void). Often a function is said to return no value if its return is a void-value. Note that the semicolon symbol is for sequencing; a sequence of expressions separated by semicolons are evaluated from left to right and the value returned from evaluating the last expression is taken as the value from evaluating the sequence.

In imperative programming, a function like testfact is normally implemented in terms of a for-loop (instead of being defined recursively). While there is indeed direct support for for-loops and while-loops in ATS, I will not attempt to make use of the support in this book. I hope to make a convincing argument that making extensive use of recursion is a key to increasing one's programming producivity. In fact, I think that a functional programmer should develop a reflex for solving problems recursively.

As another example, the following function fibonacci is defined to compute Fibonacci numbers:

```
//
fun
fibo(n: int): int =
  if n > 2 then fibo(n-1)+fibo(n-2) else 1
//
```

The first and the second Fibonacci numbers are 1. Given a positive integer *n* greater than 2, the *n*th Fibonacci number equals the sum of the previous two Fibonacci numbers. This implementation of fibo is of expontial time complexity, and it is probably impractical to call it on any integer that is 50 or greater.

For a slightly more interesting example, please study the code in multable.dats (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/02/CODE/multable.dats), which can be executed to generate the html file multable.html (https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/02/CODE/multable.html) for displaying a multiplication table.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/02/CODE) the entirety the code used in this chapter. The mentioned URL link(s) can be found as follows:

- https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/02/CODE/multable.html

Chapter 3. Recursive Functions (2)

While recursion can be really powerful for solving problems, there is a serious issue of practicality with recursion. If we follow the semantics of C, then we know that the evaluation of a function call allocates a call frame on the run-time stack (for thread execution); this frame only unwinds after the evalutation returns; it is entirely possible that other function calls are made during the evaluation. Let us revisit the following implementation of the factorial function:

```
//
fun
fact(n: int): int =
if n > 0 then n * fact(n-1) else 1
//
```

Assume that we want to evaluate fact(1000000). Clearly, the evaluation of fact(1000000) calls fact(999999), and the evaluation of fact(999999) calls fact(999998), and so on. When fact(0) is called, there are 1000001 call frames on the run-time stack for evaluating fact(n), where n goes from 1000000 down to 0. In the case where the run-time stack is not deep enough for holding all of these call frames, stack overflow happens at some point of execution (and the evaluation of fact(1000000) terminates abnormally). Please write a program to evaluate fact(1000000) and see what actually happens when it is executed.

While evaluating fact(1000000) may be seen as a contrived example, stack overflow caused by deeply nested recursive calls is a very serious issue in practice. A common approach to addressing this issue makes use of tail-recursion as is illustrated in the following implementation of the factorial function:

```
//
fun
fact2(n: int): int = let
//
fun
loop(i: int, res: int): int =
if i < n
  then loop(i+1, (i+1)*res) else res
// end of [if]
//
in
  loop(0, n)
end // end of [fact2]
//
```

Note that loop is a recursive function. The recursive call in the body of loop is a tail-call in the sense that the return value of the call can be directly used as the return value of the caller. In contrast, the return value of the recursive call in the body of fact is not a tail-call (as the value needs to be multiplied by n in order to obtain the return value of the caller). Let us take a look at another example:

```
//
fun
```

```
f91(n: int): int =
  if x > 100 then x-10 else f91(f91(x+11))
//
```

In the body of f91, there are two recursive calls; the outer call is a tail-call but the inner call is not (as the return value of the inner call needs to be passed to be outer call in order to obtain the return value of the caller).

If all of the recursive calls in the body of a function are tail-calls, the function is referred to as being tail-recursive. By default, the ATS compiler translates each tail-recursive call into a local jump. When a tail-recursive function is called, the evaluation of the call does not need to allocate any call frames for handling subseqent recursive calls and therefore it runs no risk of stack overflow due to deeply nested recursion. Please write a program to evaluate fact2(1000000) and see what actually happens when it is executed.

In ATS, mutually recursive functions can be defined by simply joining a sequence of function definitions together. For instance, the functions isevn and isodd are defined mutually recursively in the following code:

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

Note that the keyword and is used for joining function definitions. The call to isodd in the body of isevn is a tail-call, and so is the call to isevn in the body of isodd. In order for the ATS compiler to recognize that these two calls are tail-recursive calls (and thus can be translated into local jumps), the keyword fun needs to be replaced with fnx.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/03/CODE) the entirety of the code used in this chapter. Also, it can found in this article (http://ats-lang.sourceforge.net/EXAMPLE/EFFECTIVATS/loop-as-tailrec/main.html) a detailed explanation on tail-recursion and the support in ATS for turning tail-recursive calls into local jumps.

Chapter 4. Higher-Order Functions

In ATS, an anonymous function can be defined as a lambda-abstraction. For instance, the square function on integers can be defined as follows:

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

where the keyword lam is for constructing a lambda-abstraction. For defining a recursive anonymous function, the keyword fix needed. For instance, the factorial function can also be implemented as follows:

```
//
val fact =
fix f(x: int): int => if x > 0 then x * f(x) else 1
//
```

A function value can be passed as a function argument just like any other values, and a higher-order function refers to one that takes a function value as its argument. As far as terminology is concerned, a first-order function takes no function arguments; a second-order function takes a first-order function as its argument; a third-order function takes a second-order function as its argument; etc. In practice, higher-order functions are overwhelmingly second-order ones.

At this point, I want to digress a bit by advocating a so-called *build-your-own-library* approach to learning programming. Often a limitation faced by someone learning programming is that one does not have many opportunities to actually use the code written by oneself. For instance, we rarely see a case where someone makes extensive use of a data structure (such as hash table and associative map) implemented by his or her own. Most likely, one implements some data structure for the purpose of learning about it and then throws the code away afterwards. My own experience strongly indicates that one can learn a great deal more about programming if one insists on calling library functions implemented by oneself. From this point on, I will gradually build a library for this book and I encourage everyone reading the book to study the source code for the library on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/MYLIB).

As an example of higher-order function, let us implement a commonly used library function of the name int_foreach:

```
//
extern
fun
int_foreach
(n0: int, fwork: cfun(int, void)): void
//
```

Note that the type cfun(int, void) is just a shorthand for (int) -<cloref1> void, which is assigned to a closure-function that takes an integer argument and returns void. There are two kinds of functions in ATS: envless function and closure-function; the former is just a function in the sense of C: a function pointer to some memory location where a sequence of instructions is stored; the latter is essentially a function pointer paired with an environment (represented as a tuple of values) for certain paramemters. For an envless function from int to void, its type is written as (int) -> void or (int) -<fun1> void. While turning an envless function into a closure-function is straightforward, there is no generic method for turning a closure-function into an envless function. Note that each function argument of a higher-order function is usually chosen to be a closure-function so as to make the higher-order function more applicable.

A standard implementation of int_foreach is given as follows:

```
//
implement
int_foreach
(n0, fwork) =
loop(0) where {
  fun
  loop(i: int): void =
    if i < n0 then (fwork(i); loop(i+1)) else ()
  // end of [fun loop]
} (* end of [int_foreach] *)
//
```

For testing fact (that implements the factorial function), we can simply make a call to int_foreach as follows:

```
//
fun
testfact(n: int): void =
int_foreach(n, lam(i) => println!("fact(", i, ") = ", fact(i)))
//
```

As another example, let us implement a higher-order function of the name int_foldleft as follows:

```
//
extern
fun
{res:t@ype}
int_foldleft
( n0: int
, res: res, fopr: cfun(res, int, res)): res
//
implement
{res}(*tmp*)
```

```
int_foldleft
  (n0, res, fopr) =
  loop(res, 0) where
{
//
fun loop(res: res, i: int): res =
  if i < n0 then loop(fopr(res, i), i+1) else res
//
} (* end of [int_foldleft] *)
//
```

Note that int_foldleft is declared as a function template. For someone who knows the standard left-folding function on a list (for folding the list from left to right), int_foldleft does essentially the same as left-folding a list consisting of integers from 0 to n-1, where n is the first argument passed to int_foldleft. For instance, we can implement the factorial function as follows:

```
//
fun
fact(n: int): int =
int_foldleft<int>(n, 1, lam(res, i) => res * (i+1))
//
```

If we want a function sqsum to sum up the squares of the first n positive integers for any given natural number n, we can implement it with a call to int_foldleft as well:

```
//
fun
sq(i: int): int = i*i
fun
sqsum(n: int): int =
int_foldleft<int>(n, 0, lam(res, i) => res * sq(i+1))
//
```

Higher-order functions like int_foreach and int_foldleft are often referred to as combinators. By making use of combinators, one can often shorten the code needed for solving a particular problem. Much more importantly, combinators can help one formulate high-level solutions that tend to significantly reduce the need for directly handling minute programming details. As we all know too well, handling such details is a very rich source for programming errors.

As the last example in this chapter, let us see a combinator-based implementation of matrix multiplication. In imperative programming, the following style of code is common:

```
for (i = 0; i < m; i = i+1) for (j = 0; i < n; j = j+1) fwork(i, j);
```

where one for-loop is embedded in the body of another for-loop. A combinator int_cross_foreach is given as follows for doing the same:

```
extern
fun
int_cross_foreach
(m: int, n: int, fwork: cfun(int, int, void)): void
//
implement
int_cross_foreach
  (m, n, fwork) =
  int_foreach(m, lam(i) => int_foreach(n, lam(j) => fwork(i, j)))
//
```

The following function matrix_mulby takes six arguments p, q, r, A, B, and C, where it is assumed that A, B, and C are matrices of dimensions p by q, q by r, and p by r, respectively:

```
fun
{a:t@ype}
matrix_mulby
( p:int, q:int, r:int
, A:matrix0(a), B:matrix0(a), C:matrix0(a)
) : void = let
//
val add = gadd_val_val<a>
val mul = gmul_val_val<a>
//
fun
fwork(i: int, j: int): void =
(
  C[i,j] :=
  int_foldleft<a>
  ( q, C[i,j]
  , lam(res, k) => add(res, mul(A[i,k], B[k,j]))
  )
)
//
in
  int_cross_foreach(p, r, lam(i, j) => fwork(i, j))
end // end of [matrix_mulby]
```

Note that the function templates gadd_val_val and gmul_val_val are for addition and multiplication operations on generic numbers of type a. What matrix_mulby does is simply updating C with C plus the product of A and B.

In ATS code, dot-notation is often seen for calling combinators. For instance, we can introduce the following curried version of int_foreach and int_foldleft plus some overload declarations:

```
//
extern
fun
int_foreach_method
(n0: int)(fwork: cfun(int, void)): void
//
```

```
extern
fun
{res:t@ype}
int_foldleft_method
(n0: int, ty: TYPE(res))
(res: res, fopr: cfun(res, int, res)): res
//
overload .foreach with int_foreach_method
overload .foldleft with int_foldleft_method
//
```

We can then implement the functions fact and testfact as follows:

```
//
fun fact(n: int): int =
(n).foldleft(TYPE{int})(lam(res, i) => res*(i+1))
//
fun testfact(n: int): int =
(n).foreach()(lam(i) => println! ("fact(", i+1, ") = ", fact(i+1)))
//
```

Note that TYPE{int} is a form of type-annotation; it indicates to the typechecker of ATS that the template parameter for the involved instance of int_foldleft_method is int.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/04/CODE) the entirety of the code used in this chapter. The multable example mentioned previously is modified a bit so as to make use of the int_foreach_method combinator.

Chapter 5. Functional List-Processing (1)

Lists are by far the most commonly used data structure in functional programming: The functional programming language LISP is even named after *LISt Processor*. For someone with a background in imperative programming, it is important to note that a functional list is essentially an immutable linked-list: Such a list can never be changed after its creation. In particular, functional list-processing cannot modify any lists being processed.

In ATS, there is a datatype list0 declared as follows:

```
//
datatype
list0(a:t@ype) =
list0_nil of () | list0_cons of (a, list0(a))
//
```

Note that t@ype is a sort for static terms representing types of dynamic values of unspecified size. There is also a sort type in ATS, which is for types of boxed dynamic values (of the size of exactly one machine word). Given any type T, list0(T) is for lists consisting of elements of the type T.

Every datatype is associated with a set of data-constructors, which are called for constructing (boxed) values of the datatype. The declaration of list0 indicates that list0_nil and list0_cons are the two constructors assocated with list0; list0_nil is nullary; list0_cons is binary, which takes a given element and a given list to form a new list such that the given element and list are the head and tail of the newly formed list, respectively. For instance, the following code binds xs3 to a list of type list0(int) that contains three elements 3, 2, and 1:

```
val xs0 = list0_nil{int}() // xs = ()
val xs1 = list0_cons(1, xs0) // xs = (1)
val xs2 = list0_cons(2, xs1) // xs = (2, 1)
val xs3 = list0_cons(3, xs2) // xs = (3, 2, 1)
```

If list0_nil() is used instead, then the typechecker of ATS can only infer that the type of xs3 is list0(T) for some type T such that 3, 2, and 1 are values of T. Note that the type system of ATS is highly expressive and it is beyond the focus of this book to cover advanced use of types in ATS.

In ATS, nil0 and cons0 are declared aliases for list0_nil and list0_cons, respectively. Following the convention of ML, we can use :: for list0_cons as well:

```
#define :: list0_cons
val xs123 = 1 :: 2 :: 3 :: nil0{int}() // xs123 = (1, 2, 3)
```

In addition, another notation for constructing list0-values is used in the following sample code:

```
val xs123 = g0ofg1($list{int}(1, 2, 3)) // xs123 = (1, 2, 3)
val ys123 = g0ofg1($list{string}("1", "2", "3")) // ys123 = ("1", "2", "3")
```

Note that $list{T}(...) is for constructing a list-value in which each element is of type T and g0ofg1 is a cast function (of zero run-time cost) that casts a list-value into the corresponding list0-value.

The function for computing the length of a given list0-value can be implemented as follows:

```
fun
{a:t@ype}
list0_length
  (xs0: list0(a)): int =
(
case xs0 of
| list0_nil() => 0
| list0_cons(x0, xs1) => 1 + list0_length<a>(xs1)
) (* end of [list0_length] *)
```

Strictly speacking, list0_length is a function template (rather than a function). In the case-expression, there are two pattern matching clauses; each clause consists of a guard, which is pattern, and a body, which is an expression. When the value xs0 matches the guard of a clause, the body of the clause is chosen for subsequent evaluation. If xs0 is empty, then it is constructed with list0_nil and thus matches the pattern list0_nil(). If xs0 is non-empty, then then it is constructed with list0_cons and thus matches the pattern list0_cons(x0, xs1), resulting in the names x0 and xs1 bound to the head and tail of xs0, respectively.

Though the given implementation of list0_length is not tail-recursive, it should be clear that a tail-recursive implementation can be readily done. Note that the function list0_length is of O(n)-time for n being the length of its argument. Often I see someone writing the code list0_length(xs) > 0 for testing whether a given list xs is empty. This practice is terribly inefficient as checking whether a list is empty can be easily done in O(1)-time.

The function for concatenating two given list0-values can be implemented as follows:

```
fun
{a:t@ype}
list0_append
(xs: list0(a),
 ys: list0(a)): list0(a) =
(
case+ xs of
| list0_nil() => ys
| list0_cons(x, xs) => list0_cons(x, list0_append<a>(xs, ys))
) (* end of [list0_append] *)
```

Given xs and ys, list0_append returns a list that represents the concatenation of xs and ys. Clearly, list0_append is O(n)-time for n being the length of xs. The implementation of list0_append is functional in the sense that it does not alter either xs or ys for the construction of the concatenation. For clarification, I point out that there is a function of the name list_vt_append in ATS that consumes two given linear lists to construct their concatenation. When a call to list_vt_append returns, the two linear lists passed to the call are no longer available for subsequent use.

The function for constructing the reverse of a given list0-value can be implemented as follows:

```
fun
{a:t@ype}
list0_reverse
(xs: list0(a)): list0(a) =
list0_revappend<a>(xs, list0_nil())

and
list0_revappend
(xs: list0(a),
 ys: list0(a)): list0(a) =
(
case+ xs of
| list0_nil() => ys
| list0_cons(x, xs) => list0_revappend<a>(xs, list0_cons(x, ys))
) (* end of [list0_revappend] *)
```

Given a list0-value xs, list0_reverse returns a newly contructed list0-value that represents the reverse of xs. Given two list0-values xs and ys, list0_revappend returns a newly contructed list0-value that represents the concatenation of the reverse of xs and ys. Clearly, both list0_reverse and list0_revappend are O(n)-time functions, where n is the length of xs.

A commonly used list-combinator is the list0_foldleft function implemented as follows:

```
//
extern
fun
{r:t@ype}
{a:t@ype}
list0_foldleft
( xs: list0(a)
, r0: r, fopr: cfun(r, a, r)): r
//
implement
{r}{a}
list0_foldleft
( xs
, r0, fopr) =
loop(xs, r0) where
{
fun
loop
(
xs: list0(a), r0: r
) : r =
(
case+ xs of
| list0_nil() => r0
| list0_cons(x, xs) => loop(xs, fopr(r0, x))
) (* end of [loop] *)
}
```

```
//
```

where the elements in the given list xs are processed from left to right. Clearly, list0_foldleft is tail-recursive.

As an example, the function for computing the length of a given list0-value can be implemented with a direct call to list0_foldleft:

```
//
fun
{a:t@ype}
list0_length
  (xs: list0(a)): int =
  list0_foldleft<int><a>(xs, 0, lam(r, _) => r + 1)
//
```

As another example, the reverse-append function that returns the link of the reverse of a given list0-value with another given list0-value can also be implemented with a direct call to list0_foldleft:

```
fun
{a:t@ype}
list0_revappend
(
xs: list0(a),
ys: list0(a)
) : list0(a) =
  list0_foldleft<list0(a)><a>(xs, ys, lam(ys, x) => list0_cons(x, ys))
```

In contrast to list0_foldleft, the following function list0_foldright processes the elements in a given list from right to left:

```
//
extern
fun
{a:t@ype}
{r:t@ype}
list0_foldright
( xs: list0(a)
, fopr: cfun(a, r, r), r0: r): r
//
implement
{a}{r}
list0_foldright
( xs
, fopr, r0) =
auxlst(xs) where
{
fun
```

```
auxlst
(xs: list0(a)): r =
(
case+ xs of
| list0_nil() => r0
| list0_cons(x, xs) => fopr(x, auxlst(xs))
) (* end of [auxlst] *)
}
//
```

Please note that list0_foldright is not tail-recursive. As an example, the append function for concatenating two given list0-values can be implmented with a direct call to list0_foldright:

```
//
fun
{a:t@ype}
list0_append
(xs: list0(a),
 ys: list0(a)): list0(a) =
list0_foldright<a><list0(a)>
  (xs, lam(x, ys) => list0_cons(x, ys), ys)
//
```

As another example, the list-concatenate function for concatenating a list of list0-values can also be implemented with a direct call to list0_foldright:

```
//
fun
{a:t@ype}
list0_concat
(xss: list0(list0(a))): list0(a) =
list0_foldright<list0(a)><list0(a)>
  (xss, lam(xs, res) => list0_append<a>(xs, res), list0_nil())
//
```

In a case where a function (for example, list0_length) can be implemented with a call to either list0_foldleft or list0_foldright, it is clear (unless there is a very special reason) that the former should be chosen as it is tail-recursive but the latter is not.

Often I see a beginner of functional programming giving the following implementation of list0_reverse:

```
fun
{a:t@ype}
list0_reverse
(xs: list0(a)): list0(a) =
(
case+ xs of
| list0_nil() => list0_nil()
```

```
| list0_cons(x, xs) =>
  list0_append<a>(list0_reverse<a>(xs), list0_cons(x, list0_nil()))
)
```

While this implementation is functionally correctly, it is of $O(n^2)$-time complexity and thus can take a prohibitively long time for an input that is not so short (e.g., one containing 10K elements).

As the last example of this chapter, list0_insertion_sort is implemented as follows that applies insertion sort to a given list to return a sorted version of the list:

```
//
extern
fun
{a:t@ype}
list0_insertion_sort
(
xs: list0(a), cmp: cfun(a, a, int)
) : list0(a)
//
implement
{a}(*tmp*)
list0_insertion_sort
  (xs, cmp) = let
//
fun
insord
( x0: a
, xs: list0(a)): list0(a) =
(
case+ xs of
| list0_nil() =>
  list0_cons(x0, list0_nil())
| list0_cons(x1, xs1) =>
  (
    if cmp(x0, x1) >= 0
      then list0_cons(x1, insord(x0, xs1))
      else list0_cons(x0, xs)
    // end of [if]
  ) // end of [list0_cons]
) (* end of [insord] *)
//
in
//
list0_foldleft<list0(a)><a>
  (xs, list0_nil(), lam(res, x) => insord(x, res))
//
end // end of [list0_insertion_sort]
//
```

Note that the argument cmp is a comparion function that is expected to return -1, 1, or 0 when its first argument is less than, greater than, or equal to its second argument, respectively.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/05/CODE) the entirety of the code used in this chapter.

Chapter 6. Functional List-Processing (2)

Sometimes, a list-processing function is partial in the sense that it is not well-defined for all of the lists. For instance, the function list0_head for returning the head element of a given list is defined only if the given list is non-empty:

```
fun
{a:t@ype}
list0_head(xs: list0(a)): a =
(
case+ xs of
| list0_cons(x, _) => x
| _(* list0_nil() *) => $raise ListSubscriptExn()
)
```

Note that the code $raise ListSubscriptExn() is for raising an exception (in the case where xs is empty), which is to be explained in details later. Another approach to handling a partial function is to turn it into a total one that returns options. For instance, list0_head_opt is such a total function corresponding to list0_head:

```
//
datatype
option0(a:t@ype) = Some0 of (a) | None0 of ()
//
fun
{a:t@ype}
list0_head_opt
  (xs: list0(a)): option0(a) =
(
case+ xs of list0_cons(x, _) => Some0(x) | _ => None0()
)
```

The function returning the last element of a given list is also partial as it is only defined for a non-empty list. The following function list0_last_opt returns an option to indicate whether the last element of a given list is found:

```
//
fun
{a:t@ype}
list0_last_opt
(
xs: list0(a)
) : option0(a) = let
//
fun
loop
(x0: a, xs: list0(a)): a =
(
case+ xs of
```

```
list0_nil() => x0
| list0_cons(x1, xs) => loop(x1, xs)
)
//
in
//
case+ xs of
| list0_nil() => None0()
| list0_cons(x, xs) => Some0(loop(x, xs))
//
end // end of [list0_last_opt]
//
```

Note that the inner function loop in the body of list0_last_opt is tail-recursive.

Before moving on, I would like to point out a very common mistake in (functional) list-processing. First and foremost, a list is not meant to be used like an array. The following (partial) function list0_get_at does the so-called list-subscripting:

```
//
extern
fun
{a:t@ype}
list0_get_at
(xs: list0(a), n: int): a
//
implement
{a}(*tmp*)
list0_get_at
  (xs, n) =
(
case+ xs of
| list0_nil() =>
  $raise ListSubscriptExn()
| list0_cons(x, xs) =>
  if n <= 0 then x else list0_get_at<a>(xs, n-1)
)
//
```

For instance, given the list (1,10,100) and the index 1, list0_get_at return the element 10. Another common name for list0_get_at is list0_nth. Clearly, the time-complexity of list0_get_at is O(n) (while array-subscripting is O(1)-time). It is almost always a poor programming style to process the elements in a list by calling list0_get_at (as the resulting code is likely to be prohibitively inefficient time-wise). For the purpose of illustration, let us take a look at the following two functions for tallying the integers contained in a given list:

```
//
fun
list0_tally1
  (xs: list0(int)): int =
  list0_foldleft<int><int>(xs, 0, lam(res, x) => res + x)
```

```
//
fun
list0_tally2
  (xs: list0(int)): int =
  int_foldleft<int>
  (list0_length(xs), 0, lam(res, i) => res + list0_get_at<int>(xs, i))
//
```

Given a list xs of length n, the function list0_tally1 is O(n)-time while the function list0_tally2 is $O(n^2)$-time. List-processing should be done in the efficient style of list0_tally1 rather than in the terribly inefficient style of list0_tally2. In general, please try to avoid doing list-subscripting repeatedly if the involved index cannot be bounded by a small constant!

Let us see more functions for performing functional list-processing in the following presentation and an example of functional programming at the end that illustrates some typical use of list-processing functions.

A commonly used (higher-order) function is often referred to as list-map, which takes a list and a function and returns a newly constructed list consisting of all of the elements obtained from applying the function to each element in the given list:

```
//
extern
fun
{a:t@ype}
{b:t@ype}
list0_map
(xs: list0(a), fopr: cfun(a, b)): list0(b)
//
implement
{a}{b}
list0_map
(
  xs, fopr
) = auxlst(xs) where
{
//
fun
auxlst
(xs: list0(a)): list0(b) =
(
case+ xs of
| list0_nil() => list0_nil()
| list0_cons(x, xs) => list0_cons(fopr(x), auxlst(xs))
)
//
} (* end of [list0_map] *)
//
```

For instance, given the list (1, 2, 3, 4, 5) and the integer square function, list0_map returns the list consisting of 1, 4, 9, 16, and 25.

With list0_map, we can readily build list0_cross as follows for computing the cross product of two given lists:

```
//
extern
fun
{a,b:t@ype}
list0_cross
(xs: list0(a), ys: list0(b)): list0($tup(a, b))
//
implement
{a,b}(*tmp*)
list0_cross
  (xs, ys) = let
//
typedef ab = $tup(a, b)
//
in
//
list0_concat // for concatenating a list of lists
(
list0_map<a><list0(ab)>
  (xs, lam(x) => list0_map<b><ab>(ys, lam(y) => $tup(x, y)))
) (* end of [list0_concat] *)
//
end // end of [list0_cross]
//
```

For instance, given two lists (0, 1) and (2, 3, 4), list0_cross returns a newly constructed list consisting of the following six pairs: (0, 2), (0, 3), (0, 4), (1, 2), (2, 3), and (3, 4).

A function rather similar to list0_map is list0_foreach, which takes a list and a procedure (i.e., a function returning void) and applies the procedure to each element in the list:

```
//
extern
fun
{a:t@ype}
list0_foreach
(xs: list0(a), fwork: cfun(a, void)): void
//
implement
{a}(*tmp*)
list0_foreach
(
  xs, fwork
) = loop(xs) where
{
//
fun
loop
(xs: list0(a)): void =
```

```
(
case+ xs of
| list0_nil() => ()
| list0_cons(x, xs) => (fwork(x); loop(xs))
)
//
} (* end of [list0_foreach] *)
//
```

Another commonly used (higher-order) function is often referred to as list-filter, which takes a list and a predicate (i.e., a function returning a boolean value) and returns a newly constructed list consisting of all of the elements in the given list that satisfy the given predicate:

```
//
extern
fun
{a:t@ype}
list0_filter
(xs: list0(a), test: cfun(a, bool)): list0(a)
//
implement
{a}(*tmp*)
list0_filter
(
  xs, test
) = auxlst(xs) where
{
//
fun
auxlst
(xs: list0(a)): list0(a) =
(
case+ xs of
| list0_nil() =>
  list0_nil()
| list0_cons(x, xs) =>
  if test(x)
    then list0_cons(x, auxlst(xs)) else auxlst(xs)
  // end of [if]
)
//
} (* end of [list0_filter] *)
//
```

For instance, given the list (1, 2, 3, 4, 5) and the predicate for testing whether an integer is even, list0_filter returns the list consisting of 2 and 4.

Given a list xs, the following function list0_remdup removes all of the elements in xs that have already appeared previously:

```
//
extern
fun
{a:t@ype}
list0_remdup
(xs: list0(a), eqfn: cfun(a, a, bool)): list0(a)
//
implement
{a}(*tmp*)
list0_remdup(xs, eqfn) =
(
case+ xs of
| list0_nil() =>
  list0_nil()
| list0_cons(x0, xs) =>
  list0_cons(x0, list0_remdup<a>(list0_filter<a>(xs, lam(x) => eqfn(x0, x)), eqfn))
)
//
```

The implementation of list0_remdup should clearly remind one of the sieve of Eratosthenes, which is to be given a stream-based implementation later.

The list-processing function list0_map processes every element in its list argument and so does list0_filter. Sometimes, we need a list-processing function that stops immediately after certain condition is met. For instance, we may want to locate the index of the first element in a given list that satisfies some test, which can done by calling the following function list0_find_index:

```
//
extern
fun
{a:t@ype}
list0_find_index
(xs: list0(a), test: cfun(a, bool)): int
//
implement
{a}(*tmp*)
list0_find_index
  (xs, test) = let
//
fun
loop
(xs: list0(a), i: int): int =
(
case+ xs of
| list0_nil() => ~1
| list0_cons(x, xs) =>
  if test(x) then i else loop(xs, i+1)
)
//
in
  loop(xs, 0)
end // end of [list0_find_index]
```

```
//
```

For instance, given the list (1, 2, 3) and the predicate for testing whether an integer is even, list0_find_index returns 1 (which is the index for the element 2 in the given list). Note that ~1 (negative 1) is returned if no element satisfying the given predicate is found.

Given a list0-value xs and a predicate test, list0_exist returns true if and only if there exists one element in xs satisfing test, and list0_forall returns true if and only if every element in xs satisfies test. Both of these two functions can be readily implemented based on a direct call to list0_find_index:

```
//
extern
fun
{a:t@ype}
list0_exists
(xs: list0(a), test: cfun(a, bool)): bool
extern
fun
{a:t@ype}
list0_forall
(xs: list0(a), test: cfun(a, bool)): bool
//
implement
{a}(*tmp*)
list0_exists(xs, test) =
list0_find_index<a>(xs, test) >= 0
implement
{a}(*tmp*)
list0_forall(xs, test) =
list0_find_index<a>(xs, lam(x) => not(test(x))) < 0
//
```

Given a list $(x_0, ..., x_{n-1})$ of length n, its indexed version refers to the list of pairs in which the elements are of the form (i, x_i) for i ranging from 0 to n-1. For a function that processes a given list, there is often a meaningful variant of the function that processes the indexed version of the list. For instance, the following function list0_imap is such a variant of list0_map:

```
//
extern
fun
{a:t@ype}
{b:t@ype}
list0_imap
(xs: list0(a), fopr: cfun(int, a, b)): list0(b)
//
implement
{a}{b}
list0_imap
(
```

```
  xs, fopr
) = auxlst(0, xs) where
{
//
fun
auxlst
(i: int, xs: list0(a)): list0(b) =
(
case+ xs of
| list0_nil() => list0_nil()
| list0_cons(x, xs) => list0_cons(fopr(i, x), auxlst(i+1, xs))
)
//
} (* end of [list0_imap] *)
//
```

Let us use list0_iexists and list0_iforall to refer to the variants of list0_exists and list0_forall, respectively, for processing indexed lists. The code implementing list0_iexists and list0_iforall is omitted for brevity.

I have so far presented a variety of functions for processing lists in a functional style. I would like to conclude the chapter with an example of functional programming that can concretely demonstrate some typical use of list-processing functions in practice.

The famous 8-queen puzzle asks the player to find ways to put eight queen pieces on a chess board such that no queen piece can attack any other ones. In other words, no two queen pieces can be put on the same row, the same column, or the same diagnal. This puzzle can be readily solved with a tree-based search. Let us introduce an abstract type as follows:

```
abstype node = ptr
```

where ptr indicates that node is boxed. Intuitively, a node represents a partial solution where there are n queen pieces (for some n less than or equal to 8) on the first n rows of the chess board such that no one piece can attack any other pieces. Given a node, another node extending it with one more queen piece is considered its child. The following declared function node_get_children is supposed to be called to obtain all of the child nodes of a given node:

```
extern
fun
node_get_children(node): list0(node)
overload .children with node_get_children
```

With node_get_children, we can readily implement node_dfsenum as follows for enumerating in the depth-first manner all of the nodes contained in the tree rooted at a given node:

```
//
extern
fun
node_dfsenum(node): list0(node)
```

```
//
implement
node_dfsenum
(nx0) =
list0_cons
(
nx0
,
list0_concat<node>
(
list0_map<node><list0(node)>(nx0.children(), lam(nx) => node_dfsenum(nx))
) (* list0_concat *)
) (* node_dfsenum *)
//
```

In order to find all of the solutions to the 8-queen puzzle, we just need to keep all of the nodes of length 8 that are contained in the tree rooted at the initial node (representing the empty partial soloution):

```
//
extern
fun
QueenSolve(): list0(node)
//
#define N 8
//
implement
QueenSolve() =
list0_filter<node>(node_dfsenum(node_init()), lam(nx) => node_length(nx) = N)
//
```

where node_init returns the initial node and node_length returns the length of a node (that is, the number queen pieces in the partial solution represented by the node). By treating node as list0(int), we can implement node_init and node_length as follows:

```
//
assume node = list0(int)
//
implement
node_init() = list0_nil()
//
implement
node_length(nx) = list0_length(nx)
//
```

A node is represented as a list of integers; the length of the list refers to the number of queen pieces on the chess board; the integers (between 0 and N-1) in the list refer to the reversely listed column positions of the queen pieces. The function node_get_children is implemented as follows:

```
//
fun
```

```
test_safety
(
xs: list0(int)
) : bool = let
//
val-
list0_cons(x0, xs) = xs
//
in
//
list0_iforall<int> // abs: absolute value
  (xs, lam(i, x) => (x0 != x && abs(x0-x) != (i+1)))
//
end // end of [test_safety]
//
implement
node_get_children
  (nx) =
  list0_filter<node>
  (int_list0_map<node>
    (N, lam(x) => list0_cons(x, nx)), lam(nx) => test_safety(nx)
  )
//
```

The function test_safety checks whether a column position is safe for the next queen piece. Note that applying int_list0_map to an integer n is like applying list0_map to the list consisting of all of the integers between 0 and n-1, inclusive.

The presented code for solving 8-queen puzzle is of the kind of high-level functional programming style I intend to advocate throughout the book. It should soon be clear that we can reap even more benefits from programming in this way when linear lazy streams are used in place of functional lists.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/06/CODE) the entirety of the code used in this chapter.

Chapter 7. Through ATS to Javascript

Through-one-to-all is a catchy phrase often used to describe ATS. When given a programming language X, a programmer often assumes automatically that X is just meant for constructing programs *manually*. Sometimes, a (much) more productive approach to writing code in X is to write some code in another programming language for generating the needed code in X. In this chapter, I plan to demonstrate a style of co-programming with ATS and Javascript (JS). In practice (of this style of co-programming), ATS is primarily intended for high-level programming (that, for instance, makes extensive use of combinators) and JS for relatively low-level programming (that, for instance, handles direct interactions with the browser running JS code).

Let us take a look at a simple webpage for computing factorials on-line (https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/06-10/CODE/Factorial.html). After inputing a natural number, one can click the *Evaluate* button to see some output mentioning the factorial of the number. Please find the HTML source for the webpage here (https://github.com/ats-lang/ats-lang.github.io/tree/master/DOCUMENT/ATS2FUNCRASH/LECTURE/06-10/CODE/Factorial.html). Note that the following JS scripts are needed for evaluating JS code generated from ATS source:

```
<script type="text/javascript"
 src="https://ats-lang.github.io/LIBRARY/libatscc2js/ATS2-0.3.2/libatscc2js_all.js">
</script>
<script type="text/javascript"
  src="https://ats-lang.github.io/LIBRARY/libatscc2js/ATS2-0.3.2/libatscc2js_print_store_ca
</script>
```

The JS code in the file `Factorial_dats.js` is generated from compiling the ATS source stored in `Factorial.dats`. The command **patsopt** is called on the ATS source to compile it onto C code, and the command **atscc2js** is subsequently called on the generated C code to transpile it into JS code:

```
patsopt -o Factorial_dats.c -d Factorial.dats
atscc2js -o Factorial_dats.js -i Factorial_dats.c
```

At the beginning of `Factorial.dats`, the following code is present:

```
//
#define ATS_MAINATSFLAG 1
#define ATS_DYNLOADNAME "Factorial__dynload"
//
```

which indicates to patsopt that a dynload-function of the name Factorial__dynload is to be generated when the ATS source contained in `Factorial.dats` is compiled into C. Then this dynload-function is transpiled into JS by **atscc2js**, and it is supposed to be called first (to perform initialization) before any function in the generated JS code is put into use.

The following lines in `Factorial.dats` are added for accessing the LIBATSCC2JS library, which is needed for compiling ATS to JS:

```
//
#define
LIBATSCC2JS_targetloc
"$PATSHOME/contrib\\
/libatscc2js/ATS2-0.3.2"
//
#include "{$LIBATSCC2JS}/staloadall.hats" // for prelude stuff
#staload "{$LIBATSCC2JS}/SATS/print.sats" // for printing into a store
//
```

The version we use here is ATS2-0.3.2, which is the latest stable release of LIBATSCC2JS.

In the following code, a function of the name funarg1_get is declared in ATS and implemented in JS:

```
//
extern
fun
funarg1_get(): int = "mac\#"
//
\%{^
function funarg1_get()
{
  return parseInt(document.getElementById("funarg1").value);
}
\%} (* end of external code *)
//
```

The function funarg1_get essentially locates a DOM-element identified by funarg1 and then converts into an integer the string stored in the value-field of the DOM-element. Note that the use of the string "mac#" simply indicates to **patsopt** that the function funarg1_get is to be given the same name when compiled into C. In contrast, the name automatically chosen (by **patsopt**) for funarg1_get contains a long prefix (for carrying some namespace information) if the string "mac#" is not present. In case a different name is needed for funarg1_get in C, the name should be explicitly mentioned after the symbol #. For instance, if the string "mac#\JS_funarg1_get" is used, then the name JS_funarg1_get is chosen (by **patsopt**) for funarg1_get. In ATS source, the code written between \%{^ and \%} is considered external and is directly pasted by **patsopt** into the generated C code. The symbol ^ means that the pasted code is to appear near the beginning of the generated code. If the symbol ^ is omitted or replaced with $, the pasted code is to appear near the bottom of the generated code.

The following function is called when the *Evaluate* button is clicked:

```
//
extern
fun
Factorial__evaluate
  ((*void*)): void = "mac#"
//
```

```
implement
Factorial__evaluate
  ((*void*)) = let
  val () =
  the_print_store_clear()
  val arg = funarg1_get()
  val () =
  println!
  ("The factorial of ", arg, " is ", fact(arg))
  // end of [val]
  val theOutput =
    document_getElementById("theOutput")
  // end of [val]
in
  xmldoc_set_innerHTML(theOutput, the_print_store_join())
end // end of [Factorial__evaluate]
//
```

There is a global store, an array of strings, for receiving the output generated from calling various print-functions. The function the_print_store_clear is for clearing this store and the function the_print_store_join for joining the strings contained in the store into a single one. Given a name, document_getElementById locates the DOM-element identified by the name. Given a DOM-element and a string, xmldoc_set_innerHTML updates the innerHTML-field of the DOM-element with the string.

For a more involved example, please visit on-line (https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/06-10/CODE/QueenPuzzle.html) a webpage for animating the process that searches (in the depth-first fashion) for solutions to the 8-queen puzzle. The HTML source for the webpage can be viewed here (https://github.com/ats-lang/ats-lang.github.io/tree/master/DOCUMENT/ATS2FUNCRASH/LECTURE/06-10/CODE/QueenPuzzle.html), and the code implementing depth-first search is simply adapted from some sample ATS code presented in a previous chapter.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/06-10/CODE) the entirety of the code used in this chapter. The mentioned URL link(s) can be found as follows:

- https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/06-10/CODE/Factorial.html
- https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/06-10/CODE/QueenPuzzle.html

Chapter 8. Functional List-Processing (3)

In this chapter, I intend to present a few more list-processing functions. More importantly, I would like to argue for the practice of actively identifying the need for generic list-processing functions during problem-solving.

There are many useful list-processing functions for handling two lists simultaneously. For instance, the following function list0_zip takes a pair of lists and returns a list of pairs:

```
//
extern
fun {
a,b:t@ype
} list0_zip
  (xs: list0(a), ys: list0(b)): list0($tup(a, b))
//
implement
{a,b}
list0_zip(xs, ys) =
(
case xs of
| list0_nil() =>
  list0_nil()
| list0_cons(x, xs) =>
  (
    case+ ys of
    | list0_nil() =>
      list0_nil()
    | list0_cons(y, ys) =>
      list0_cons($tup(x, y), list0_zip<a,b>(xs, ys))
  )
)
//
```

For instance, given two lists (1, 3, 5) and (2, 4), list0_zip returns the list consisting of $tup(1, 2) and $tup(3,4). Note that $tup is a keyword in ATS (for constructing tuples). For a list-processing function handling one list, there is often a meaningful variant that essentially acts like passing to the list-processing function a list of pairs obtained from applying list0_zip to two given lists. For instance, the following function list0_map2 is such a variant of list0_map:

```
//
extern
fun
{a
,b:t@ype}
{c:t@ype}
list0_map2
(xs: list0(a), ys: list0(b), fopr: cfun(a, b, c)): list0(c)
//
```

```
implement
{a,b}{c}
list0_map2
(xs, ys, fopr) =
(
case xs of
| list0_nil() =>
  list0_nil()
| list0_cons(x, xs) =>
  (
    case+ ys of
    | list0_nil() =>
      list0_nil()
    | list0_cons(y, ys) =>
      list0_cons(fopr(x, y), list0_map2<a,b><c>(xs, ys, fopr))
  ) (* end of [list0_cons] *)
)
//
```

For instance, given two lists (1, 3, 5) and (2, 4, 6) and the multiplication function, list0_map2 returns the list (2, 12, 30). Sometimes, list0_map2 is referred to as list0_zipwith.

There are various list-processing functions operating on ordered lists. For instance, the following function list0_merge is for merging two ordered lists into one:

```
//
extern
fun
{a:t@ype}
list0_merge
( xs: list0(a)
, ys: list0(a), cmp: cfun(a, a, int)): list0(a)
//
implement
{a}(*tmp*)
list0_merge
(xs0, ys0, cmp) = let
//
fun
auxlst
( xs0: list0(a)
, ys0: list0(a)
) : list0(a) = (
//
case+ xs0 of
| list0_nil() => ys0
| list0_cons
    (x1, xs1) => (
    case+ ys0 of
    | list0_nil() => xs0
    | list0_cons
        (y1, ys1) => let
```

```
        val sgn = cmp(x1, y1)
      in
        if (sgn <= 0)
          then list0_cons(x1, auxlst(xs1, ys0))
          else list0_cons(y1, auxlst(xs0, ys1))
        // end of [if]
      end // end of [list0_cons]
  ) (* end of [list0_cons] *)
//
) (* end of [auxlst] *)
//
in
  auxlst(xs0, ys0)
end // end of [list0_merge]
//
```

With list0_merge, we can readily implement as follows the well-known mergesort algorithm to sort a given list0-value:

```
//
extern
fun
{a:t@ype}
list0_mergesort
(xs: list0(a), cmp: cfun(a, a, int)): list0(a)
//
implement
{a}(*tmp*)
list0_mergesort
  (xs, cmp) = let
//
// [msort]:
// It is assumed
// that length(xs) = n
//
fun
msort
(xs: list0(a), n: int): list0(a) =
if
(n >= 2)
then let
  val n1 = n / 2
  val xs1 = list0_take_exn(xs, n1)
  val xs2 = list0_drop_exn(xs, n1)
in
  list0_merge<a>(msort(xs1, n1), msort(xs2, n-n1), cmp)
end // end of [then]
else (xs) // end of [else]
//
in
  msort(xs, list0_length<a>(xs))
end // end of [list0_mergesort]
```

```
//
```

Note that list0_take_exn(xs, n) returns the prefix of xs that is of length n and list0_drop_exn(xs, n) returns the suffix of xs that excludes the first n elements of xs. It is guaranteed that list0_mergesort is log-linear (that is, it is of O(n(log(n)))-time for n being the length of its argument). Also, list0_mergesort is stable in the sense that the order of the elements considered equal in the input is not changed in the output. Clearly, list0_merge is not tail-recursive, potentially running the risk of stack overflow when list0_mergesort is applied to a long list (e.g., one containing 1 million elements). This is a very serious issue with non-tail-recursion in practice, and some approaches to addressing it are to be presented later.

During problem-solving, it often pays if one actively looks for opptunities to generalize a specific function into one that can be given a meaningful description in a broader context. As an example, if one encounters a need to process all of the pairs formed with elements chosen from a given list, then one may want to implement the following function list0_choose2:

```
//
extern
fun
{a:t@ype}
list0_choose2
(xs: list0(a)): list0($tup(a, a))
//
extern
fun
{a:t@ype}
list0_choose2
(xs: list0(a)): list0($tup(a, a))
//
implement
{a}(*tmp*)
list0_choose2
  (xs) = let
//
typedef aa = $tup(a, a)
//
in
//
case+ xs of
| list0_nil() =>
  list0_nil()
| list0_cons(x0, xs) =>
  list0_append<aa>
  (list0_map<a><aa>(xs, lam(x) => $tup(x0, x)), list0_choose2(xs))
//
end // end of [list0_choose2]
//
```

For instance, given the list (1, 2, 3), list0_choose2 returns the list of 3 pairs: (1, 2), (1, 3), and (2, 3). And list0_choose2 can be further generalized into the following function list0_nchoose for listing all of the tuples of a given length that are formed with elements chosen from a given list:

```
//
extern
fun
{a:t@ype}
list0_nchoose
(xs: list0(a), n: int): list0(list0(a))
//
implement
{a}(*tmp*)
list0_nchoose
  (xs, n) =
  auxlst(xs, n) where
{
//
typedef xs = list0(a)
//
fun
auxlst
(
xs: xs, n: int
) : list0(xs) =
(
if
(n <= 0)
then
list0_sing(list0_nil())
else
(
case+ xs of
| list0_nil() =>
  list0_nil()
| list0_cons(x0, xs) =>
  list0_append<xs>(list0_mapcons(x0, auxlst(xs, n-1)), auxlst(xs, n))
) (* end of [else] *)
)
//
} (* end of [list0_nchoose] *)
//
```

Sometimes, we may need to process tuples consisting of elements chosen from a given list as well as the complements of these tuples. The following function list0_nchoose_rest generalizes list0_nchoose in this regard:

```
//
extern
fun
{a:t@ype}
list0_nchoose_rest
(xs: list0(a), n: int): list0($tup(list0(a), list0(a)))
//
implement
{a}(*tmp*)
```

```
list0_nchoose_rest
  (xs, n) =
  auxlst(xs, n) where
{
//
typedef xs = list0(a)
typedef xsxs = $tup(xs, xs)
//
fun
auxlst
(
xs: xs, n: int
) : list0(xsxs) =
(
if
(n <= 0)
then
list0_cons
($tup(list0_nil(), xs), list0_nil())
else
(
case+ xs of
| list0_nil() =>
  list0_nil()
| list0_cons(x0, xs) => let
    val res1 =
    list0_map<xsxs><xsxs>
    ( auxlst(xs, n-1)
    , lam($tup(xs1, xs2)) => $tup(list0_cons(x0, xs1), xs2)
    )
    val res2 =
    list0_map<xsxs><xsxs>
    ( auxlst(xs, n-0)
    , lam($tup(xs1, xs2)) => $tup(xs1, list0_cons(x0, xs2))
    )
  in
    list0_append<xsxs>(res1, res2)
  end // end of [list0_cons]
) (* end of [else] *)
)
//
} (* end of [list0_nchoose_rest] *)
```

For instance, given the list (1, 2, 3) and integer 2, list0_choose_rest returns the list of 3 pairs: $tup((1, 2), (3)), $tup((1, 3), (2)), and $tup((2, 3), (1)).

The problem of enumerating all of the permutations of a given list is often used to test one's ability in constructing recursively defined functions. Let us see a solution to this problem given as follows:

```
//
fun
{a:t@ype}
```

```
list0_permute
(
xs: list0(a)
) : list0(list0(a)) =
(
case+ xs of
| list0_nil() =>
  list0_cons(nil0(), nil0())
| list0_cons _ => let
    typedef xs = list0(a)
    typedef out = list0(xs)
    typedef inp = $tup(xs, xs)
  in
    list0_concat<xs>
    (
     list0_map<inp><out>
     ( list0_nchoose_rest<a>(xs, 1)
     , lam($tup(ys, zs)) => list0_mapcons<a>(ys[0], list0_permute<a>(zs))
     )
    ) (* list0_concat *)
  end (* end of [list0_cons] *)
)
//
```

For instance, given the list (1, 2, 3), list0_permute returns a list of length 6 in which each element is a permutation of (1, 2, 3). Note that ys[0] refers to the first element in ys and list0_mapcons prepends its first argument to each element in its second argument (which is a list of lists). A demo of list0_permute can be seen by following this link (https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/07/CODE/Permute.html).

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/07/CODE) the entirety of the code used in this chapter. The mentioned URL link(s) can be found as follows:

- https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/07/CODE/Permute.html

Chapter 9. Example: Game of Twenty-four

Given four integers n1, n2, n3 and n4, one chooses two and uses them to produce a rational number r1 by applying either addition, subtraction, multiplication or division; one mixes r1 with the remaining two numbers and chooses two of them to produce a rational number r2 by applying either addition, subtraction, multiplication or division; one then takes r2 and the last remaining number to produce a rational number r3 by applying addition, subtraction, multiplication, or division; if there exists a way to make r3 equal 24, then (n1, n2, n3, n4) is said to be a good quad. For instance, (10,10,4,4) is a good quad since we have: (10*10-4)/4 = 24; (5,7,7,11) is a good quad since we have: (5-11/7)*7 = 24. Game-of-24 is a game that determines whether four given integers form a good quad or not. For a demo of Game-of-24, please visit this link (https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/07-10/CODE/Game-of-24.html).

In the following presentation of this chapter, I intend to present an implementation of Game-of-24 in pure functional style, solidifying various functional programming concepts covered so far. Please pay attention to the support for overloading in ATS, which helps make the presented code significantly easier to access.

Instead of making direct use of rational numbers in the implementation, I choose to use doubles (that is, floating point numbers of double precision) to approaximate them. The following declarations introduce int2dbl and dbl2int for casting integers to doubles and vice versa, respectively:

```
//
#define
int2dbl g0int2float_int_double
#define
dbl2int g0float2int_double_int
//
```

In the case where four given integers are claimed to form a good quad, we may want to see a proof to justify the claim. The following datatype expr is meant for providing such a proof:

```
//
datatype expr =
  | EXPRnum of double
  | EXPRbop of (string(*opr*), expr, expr)
//
typedef exprlst = list0(expr)
//
```

For instance, the expression (10*10-4)/4 can be represented as a value of the type expr:

```
//
val myproof =
```

```
EXPRbop("/", EXPRbop("-", EXPRbop("*", EXPRnum(10), EXPRnum(10)), EXPRnum(4)), EXPRnum(4))
//
```

Note that values of the type expr is often referred as abstract syntax trees.

Given an expression (that is, an expr-value), the following function eval_expr returns the value of the expression:

```
//
extern
fun
eval_expr(x0: expr): double
//
overload eval with eval_expr
//
implement
eval_expr(x0) =
(
case+ x0 of
| EXPRnum(v0) => v0
| EXPRbop(opr, x1, x2) => let
    val v1 = eval_expr(x1)
    val v2 = eval_expr(x2)
  in
    case+ opr of
    | "+" => v1 + v2
    | "-" => v1 - v2
    | "*" => v1 * v2
    | "/" => v1 / v2
    | _(*unrecognized*) =>
      let val () = assertloc(false) in 0.0 end
  end
) (* end of [eval_expr] *)
//
```

The following functions print_expr and fprint_expr are needed for displaying an expression (that is, an expr-value):

```
//
extern
fun
print_expr : (expr) -> void
extern
fun
fprint_expr : (FILEref, expr) -> void
//
overload print with print_expr
overload fprint with fprint_expr
//
```

```
(* ****** ****** *)
//
implement
print_expr(x0) =
fprint_expr(stdout_ref, x0)
//
implement
fprint_expr(out, x0) =
(
case x0 of
| EXPRnum(v0) =>
  fprint(out, dbl2int(v0))
| EXPRbop(opr, x1, x2) =>
  fprint!(out, "(", x1, opr, x2, ")")
)
//
```

The following functions are introduced for constructing abstract syntax trees to represent the 4 arithmetic operations: addition, subtraction, multiplication, and division:

```
//
extern
fun
add_expr_expr
  : (expr, expr) -> expr
and
sub_expr_expr
  : (expr, expr) -> expr
and
mul_expr_expr
  : (expr, expr) -> expr
and
div_expr_expr
  : (expr, expr) -> expr
//
overload + with add_expr_expr
overload - with sub_expr_expr
overload * with mul_expr_expr
overload / with div_expr_expr
//
(* ****** ****** *)
//
implement
add_expr_expr(x1, x2) = EXPRbop("+", x1, x2)
implement
sub_expr_expr(x1, x2) = EXPRbop("-", x1, x2)
implement
mul_expr_expr(x1, x2) = EXPRbop("*", x1, x2)
implement
div_expr_expr(x1, x2) = EXPRbop("/", x1, x2)
```

```
//
```

By overloading familiar symbols like +, -, *, and /, I expect that the presented code should be easier to access (even for someone who knows very little about the programming language in which the code is written).

When using floating point numbers, one often needs to pay special attention equality test. For testing whether the value of an expression equals a given integer, the following function eq_expr_int checks whether the absolute difference between the value and the integer is less than a tiny fraction EPSILON:

```
//
#define EPSILON 1E-8
//
extern
fun
eq_expr_int(expr, int): bool
//
overload = with eq_expr_int
//
implement
eq_expr_int(x0, i0) = abs(eval_expr(x0) - i0) < EPSILON
//
```

I must say that choosing EPSILON to be 10^{-8} is an empirical decision (aimed at handling inputs that are small integers (e.g., between 0 and 100)).

What is done so far can be thought of as providing basic setup for the task of implementing Game-of-24. It is time for us to switch to the algorithmic part of the task next.

Given two expressions, the function combine_expr_expr returns a list consisting of every expression that can be constructed by applying addition, subtraction, multiplication, or division to the two given ones:

```
//
extern
fun
combine_expr_expr(expr, expr): exprlst
//
overload combine with combine_expr_expr
//
(* ****** ****** *)
//
implement
combine_expr_expr
  (x1, x2) = res where
{
//
val res = list0_nil()
val res = list0_cons(x1+x2, res)
val res = list0_cons(x1-x2, res)
val res = list0_cons(x2-x1, res)
```

```
val res = list0_cons(x1*x2, res)
val res = if x2 = 0 then res else list0_cons(x1/x2, res)
val res = if x1 = 0 then res else list0_cons(x2/x1, res)
//
val res = list0_reverse(res)
//
} (* end of [combine_expr_expr] *)
//
```

Please note that the divisor must be non-zero if the division operation is applied.

In practice, the following higher-order function list0_mapjoin is of common use:

```
//
extern
fun
{a:t@ype}
{b:t@ype}
list0_mapjoin
( xs: list0(INV(a))
, fopr: cfun(a, list0(b))): list0(b)
//
implement
{a}{b}
list0_mapjoin
  (xs, fopr) =
(
  list0_concat<b>(list0_map<a><list0(b)>(xs, fopr))
)
//
```

Given a list xs and a closure-function fopr, list0_mapjoin calls list0_map to obtain a list of lists and then calls list0_concat to flatten this obtained list of lists.

Let task be an alias for list0(expr). Given a task-value xs of the type task, we can choose two expressions out of xs to form a new expression (by applying addition, subtraction, multiplication or division) and then combine it with the rest to construct a new task-value. By calling do_one on xs, we obtain all of the possible new task-values:

```
//
typedef task = list0(expr)
//
extern
fun
do_one(xs: task): list0(task)
//
implement
do_one(xs) = let
//
val x1x2xss =
list0_nchoose_rest<expr>(xs, 2)
```

```
//
in
//
list0_mapjoin<$tup(exprlst, exprlst)><task>
(
x1x2xss
,
lam
($tup(x1x2, xs)) =>
list0_map<expr><task>
 (combine(x1x2[0], x1x2[1]), lam(x) => list0_cons(x, xs))
)
//
end // end of [do_one]
//
```

Given four integers, the following function play_game returns a list of expressions that are proofs showing the four integers being a good quad:

```
//
extern
fun
do_ones(n: int, xss: list0(task)): list0(task)
//
implement
do_ones(n, xss) =
if n >= 2
  then
  do_ones
  ( n-1
  , list0_mapjoin<task><task>(xss, lam(xs) => do_one(xs))
  ) (* do_ones *)
  else xss
//
(* ****** ****** *)
//
extern
fun
play_game
( n1: int
, n2: int
, n3: int
, n4: int): exprlst
//
implement
play_game
  (n1, n2, n3, n4) = let
//
  val x1 = EXPRnum(int2dbl(n1))
  val x2 = EXPRnum(int2dbl(n2))
  val x3 = EXPRnum(int2dbl(n3))
  val x4 = EXPRnum(int2dbl(n4))
```

```
in
//
list0_mapopt<task><expr>
(
do_ones(4, list0_sing(g0ofg1($list{expr}(x1, x2, x3, x4))))
,
lam(xss) => if (xss[0] = 24) then Some0(xss[0]) else None0()
)
//
end // end of [play_game]
```

Whether four given integers are a good quad depends precisely on whether the list returned by a call to play_game on them is non-empty.

The code presented in this chapter targets C. After a bit of adaption (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/07-10/CODE/Game-of-24-js.dats), the code can also be compiled into JS for interpretation inside a browser. Please see a demo of Game-of-24 on-line (https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/07-10/CODE/Game-of-24.html), which is directly based on the presented code.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/07-10/CODE) the entirety of the code used in this chapter. The mentioned URL link(s) can be found as follows:

- https://ats-lang.github.io/DOCUMENT/ATS2FUNCRASH/LECTURE/07-10/CODE/Game-of-24.html
- https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/07-10/CODE/Game-of-24-js.dats

Chapter 10. Persistent Arrays and References

Let us start with references. A reference is precisely an (initialized) singleton array, that is, an array of size 1, and its typical use is for implementing a global variable that can be updated.

The following function is for creating a reference:

```
//
fun
{a:t@ype}
ref_make_elt(x0: a): ref(a)
//
```

where ref is a type constructor in ATS that takes a type T to form a reference type ref(T) for any reference holding a value of the type T. The functions ref_get_elt and ref_set_elt are for fetching and updating the value stored in a reference, respectively:

```
//
fun
{a:t@ype}
ref_get_elt(r: ref(a)): a
fun
{a:t@ype}
ref_set_elt(r: ref(a), x: a): void
//
```

A shorthand for ref_make_elt is ref. Also, both ! and [] are overloaded with ref_get_elt and ref_set_elt. For instance, the following code prints out 0, 1, and 3 twice:

```
//
val r0 = ref<int>(0)
val () = println! (!r0)
val () = (!r0 := !r0 + 1)
val () = println! (!r0)
val () = (!r0 := !r0 + 2)
val () = println! (!r0)
//
val r1 = ref<int>(0)
val () = println! (r1[])
val () = (r1[] := r1[] + 1)
val () = println! (r1[])
val () = (r1[] := r1[] + 2)
val () = println! (r1[])
//
```

Programmers often misuse references. This is especially true for those with a background in imperative programming. For instance, the following example shows a typical poor use of references that is often resulted from someone learning functional programming by "translating" code written in imperative style:

```
//
fun
fact_ref
(n: int): int = let
//
val i = ref<int>(0)
val r = ref<int>(1)
//
fun loop(): void =
  if !i < n then (!i := !i+1; !r := !r * !i; loop())
//
in
  let val () = loop() in !r end
end (* end of [fact_ref] *)
//
```

There are some obvious drawbacks in this implementation of fact_ref. For evaluating each call to fact_ref, two references are created, which become garbage after the call returns. As a reference is allocated on heap, it is not mapped to a register when compiled. Accessing a reference involves memory traffic, which is much more expensive than accessing a register. Compared to a tail-recursive implementation of the factorial function in functional style, fact_ref is of great inefficiency both time-wise and memory-wise.

There is a type constructor array0 in ATS that takes a type T to form the array type array0(T) for an array storing elements of the type T. We refer to such an array as an array0-value, which is essentially a pair containing a pointer (to the memory location where elements are stored) and an integer (indicating the capacity of the array). For instance, the following function can be called to create an array0-value:

```
//
fun
{a:t@ype}
array0_make_elt(asz: int, x0: a): array0(a)
//
```

Given a non-negative integer asz and an element x0, array0_make_elt returns an array0-value in which the array is of size asz and each of its cells is initialized with x0.

Another commonly used function for creating an array0-value is array0_tabulate of the following interface:

```
//
fun
{a:t@ype}
array0_tabulate(asz: int, fopr: cfun(int, a)): array0(a)
//
```

Given a non-negative integer asz and a closure-function fopr, array0_tabulate returns an array0-value in which the array is of size asz and the array cells are initialized with the values of fopr at the valid indices.

The functions array0_get_at and array0_set_at are for fetching and updating the value stored in an array0-value at a given index:

```
//
fun
{a:t@ype}
array0_get_at(A: array0(a), i: int): a
fun
{a:t@ype}
array0_set_at(A: array0(a), i: int, x: a): void
//
```

Note that array0_get_at raises an exception (ArraySubscriptExn()) if the given index is invalid, that is, not between 0 and the array size minus 1, inclusive. And the same happens with respect to array0_set_at. Also please note that the brackets [] is overloaded with both array0_get_at and array0_set_at. For instance, evaluating the following code prints out two lines: the first consisting of the text 000 and the second consisting of the text 123:

```
//
val A =
array0_make_elt<int>(3, 0)
//
val () =
println! (A[0], A[1], A[2])
//
val () = A[0] := 1
val () = A[1] := A[0] + 1
val () = A[2] := A[1] + 1
//
val () =
println! (A[0], A[1], A[2])
//
```

Like list0-values, there are many commonly used functions for processing array0-values. For instance, the following function array0_foreach corresponds to the previouly presented list0_foreach:

```
//
extern
fun
{a:t@ype}
array0_foreach
(A: array0(a), fwork: cfun(a, void)): void
//
implement
{a}(*tmp*)
```

```
array0_foreach(A, fwork) =
(
  int_foreach<>(sz2i(A.size()), lam(i) => fwork(A[i]))
) (* end of [array0_foreach] *)
//
```

Given an array0-value A, the expression A.size() is written in dot-notation, which returns the size of the array contained in A. The name sz2i refers to a cast function from the type size_t to the type int.

The following function array0_foldleft corresponds to the previouly presented list0_foldleft:

```
//
extern
fun
{r:t@ype}
{a:t@ype}
array0_foldleft
(A: array0(a), r0: r, fopr: cfun(r, a, r)): r
//
implement
{r}{a}
array0_foldleft
  (A, r0, fopr) =
(
//
int_foldleft<r>
  (sz2i(A.size()), r0, lam(r, i) => fopr(r, A[i]))
//
) (* end of [array0_foldleft] *)
//
```

The following function array0_foldright corresponds to the previouly presented list0_foldright:

```
//
extern
fun
{r:t@ype}
{a:t@ype}
array0_foldright
(A: array0(a), fopr: cfun(a, r, r), r0: r): r
//
implement
{r}{a}
array0_foldright
  (A, fopr, r0) = let
  val asz = sz2i(A.size())
in
  int_foldleft<r>(asz, r0, lam(r, i) => fopr(A[asz-i-1], r))
end (* end of [array0_foldright] *)
//
```

Unlike list0_foldright, array0_foldright is tail-recursive.

In practice, matrices (that is, two dimensional arrays) are also a very common data structure. In ATS, there is a type constructor matrix0, which takes a type T to form the type matrix0(T) for a matrix storing elements of the type T. We refer to such a matrix as a matrix0-value, which is essentially a tuple containing a pointer (to the memory location where elements are stored) and two integers (indicating the number of rows and the number of columns of the matrix). For instance, the following function can be called to create a matrix0-value:

```
//
fun
{a:t@ype}
matrix0_make_elt(nrow: int, ncol: int, x0: a): matrix0(a)
//
```

Given two non-negative integers m and n and an element x0, matrix0_make_elt returns a matrix0-value in which the matrix is of dimension m by n and each of its cells is initialized with x0.

Please note that the so-called row-major representation is chosen for the matrix contained in each created matrix0-value. As the elements in each row are stored adjacently in row-major representation, it can be (much) more efficient to process the elements in the row-by-row fashion (compared to the column-by-column fashion) due to the memory-cache effect.

Like array0_tabulate, the following function matrix0_tabulate is often called to create a matrix0-value:

```
//
fun
{a:t@ype}
matrix0_tabulate
  (nrow: int, ncol: int, fopr: cfun(int, int, a)): matrix0(a)
//
```

Given two non-negative integers m and n and a closure-function fopr, matrix0_tabulate returns a matrix0-value in which the matrix is of dimension m by n and the matrix cells are initialized with the values of fopr at the valid indices.

The functions matrix0_get_at and matrix0_set_at are for fetching and updating the value stored in a matrix0-value at a given position (represented by a row index and a column index):

```
//
fun{a:t0p}
matrix0_get_at
  (M: matrix0(a), i: int, j: int): a
fun{a:t0p}
matrix0_set_at
  (M: matrix0(a), i: int, j: int, x: a): void
//
```

Note that matrix0_get_at raises an exception (MatrixSubscriptExn()) if one of the given indices is invalid. And the same happens with respect to matrix0_set_at. Also please note that the brackets [] is overloaded with both matrix0_get_at and matrix0_set_at.

Like array0_foreach, the following function matrix0_foreach applies to each element in a given matrix M a given closure-function fwork:

```
//
extern
fun
{a:t@ype}
matrix0_foreach
(M: matrix0(a), fwork: cfun(a, void)): void
//
implement
{a}(*tmp*)
matrix0_foreach
  (M, fwork) = let
  val nrow = sz2i(M.nrow())
  val ncol = sz2i(M.ncol())
in
  int_cross_foreach<>(nrow, ncol, lam(i, j) => fwork(M[i,j]))
end (* end of [matrix0_foreach] *)
//
```

The expression M.nrow() is written in dot-notation, which returns the number of rows in the matrix contained in M. And M.ncol() is for the number of columns in the matrix.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/08/CODE) the entirety of the code used in this chapter.

Chapter 11. Raising and Catching Exceptions

Exceptions provide a mechanism for altering the (normal) control-flow during program execution. Raising an exception is somewhat like executing a goto-statement. A handler for handling a raised exception is essentially a pattern matching clause guarded by a pattern (for matching exceptions). Intuitively speaking, a raised exception passes through a stack of handlers; the raised exception is handled by a handler if the exception matches the guard of the handler, or it simply tries the next handler to see if it can be handled; the raised exception terminates program execution abnormally if it is not handled by any of the handlers.

In ATS, a try-expression (or try-with-expression) is of the form (try exp with clseq), where try is a keyword, exp is an expression, with is also a keyword, and clseq is a sequence of pattern matching clauses (used as exception-handlers). When evaluating such a try-expression, we first evaluate exp. If the evaluation of exp leads to a value, then the value is also the value of the try-expression. If the evaluation of exp leads to a raised exception, then we match the exception against the guards of the matching clauses in clseq. If there is a match, the raised exception is caught and we continue to evaluate the body of the first clause whose guard is matched. If there is no match, the raised exception is uncaught. In the following example, list0_exists is implemented based on list0_foreach:

```
//
implement
{a}(*tmp*)
list0_exists
  (xs, test) = let
//
exception True of ()
//
in
//
try let
//
val () =
list0_foreach<a>
( xs
, lam(x) => if test(x) then $raise True()
)
//
in
  false
end with ~True() => true
//
end // end of [list0_exists]
//
```

Given a list and a predicate, list0_exists returns a boolean value to indicate whether there exists an element in the list that satisfies the predicate. There is a built-in type exn in ATS, which is somewhat like an extensible datatype in the sense that a new constructor associated with exn can be introduced through an exception-declaration. For instance, True is introduced as a nullary exception-constructor in the body of list0_exists. Given a list and a function, list0_foreach normally traverses until the end of the list while

appying the function to each encountered element. When the call to list0_foreach in the body of list0_exists is evaluated, an exception (True()) is raised to stop further traversing if an element satisfying the predicate test is encountered. If the call to to list0_foreach returns, then the value false is the return value of list0_exists. If True() is raised during the evaluation of the call, then this raised exception is to be caught by the handler following the keyword with, resulting in the value true becoming the return value of list0_exists. Note that a raised exception is considered a resource in ATS and it needs to be properly freed or re-raised after being caught. The symbol ~ in the pattern ~True() indicates that the caught execption is freed.

By raising an exception, a function can efficiently pass some information gathered during its evaluation. Let us see a typical example of this kind as follows. A datatype tree is decared for representing binary trees:

```
//
datatype tree(a:t@ype) =
| tree_nil of ()
| tree_cons of (tree(a), a, tree(a))
//
```

For instance, the function for computing the height of a given binary tree can be implemented as follows:

```
//
fun
{a:t@ype}
tree_height(t0: tree(a)): int =
(
case+ t0 of
| tree_nil() => 0
| tree_cons(tl, _, tr) =>
  1 + max(tree_height<a>(tl), tree_height<a>(tr))
)
//
```

Note that tree_height is O(n)-time, where n is the size of its argument. A binary is perfect if it is empty or both of its children are perfect and of the same height. For instance, the following function tree_is_perfect checks whether a given binary tree is perfect:

```
//
fun
{a:t@ype}
tree_is_perfect
  (t0: tree(a)): bool =
(
case+ t0 of
| tree_nil() => true
| tree_cons(tl, _, tr) =>
  tree_is_perfect<a>(tl) &&
  tree_is_perfect<a>(tr) &&
  (tree_height<a>(tl) = tree_height<a>(tr))
```

```
)
//
```

Clearly, the time complexity of tree_is_perfect is O(n(log(n))) where n is the size of its argument. By making use of exception, we can readily implement the following function of O(n)-time that also checks whether a given tree is perfect:

```
//
fun
{a:t@ype}
tree_is_perfect2
  (t0: tree(a)): bool = let
//
exception NotPerfect of ()
//
fun
aux(t0: tree(a)): int =
case+ t0 of
| tree_nil() => 0
| tree_cons(tl, _, tr) => let
    val hl = aux(tl) and hr = aux(tr)
  in
    if hl = hr then 1+max(hl, hr) else $raise NotPerfect
  end
//
in
  try let val _ = aux(t0) in true end with ~NotPerfect() => false
end // end of [tree_is_perfect2]
//
```

The inner function aux inside tree_is_perfect2 returns the height of a given tree only if the tree is perfect. Otherwise, an exception (NotPerfect()) is raised. In other words, if a call to aux on a tree returns, we know that the tree is pefect while obtaining its height. This is often dubbed a *hitting-two-birds-with-one-stone* scenario.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/09/CODE) the entirety of the code used in this chapter.

Chapter 12. Lazy Stream-Processing

A stream is like a list but it is lazy in the sense that the elements in a stream are not required to be made available at the moment when the stream is formed. Instead, each element only needs to be produced at the point where the element is actually needed for evaluation. Given a type T, the type stream(T) is for a value representing a stream of elements of the type T. Let us use the name stream-value to refer to such a value, which is completely opaque for it does not even reveal whether the stream it represents is empty or not. Internally, a stream-value is represented as a thunk, that is, a nullary closure-function. Evaluating a stream-value (that is, calling the thunk representing it) yields a stream-con value of type stream_con(T) for some T, where stream_con is the following declared datatype:

```
datatype
stream_con(a:t@ype) =
  | steam_nil of ()
  | stream_cons of (a, stream(a))
```

With pattern matching, we can inspect whether a stream-con value represents an empty stream or not. If the stream is non-empty, then its first element can be extracted. Formally speaking, we have the following type definition in ATS:

```
typedef stream(a:t@ype) = lazy(stream_con(a))
```

where lazy is a special type constructor. Given a type T, the type lazy(T) is essentially for a thunk that returns a value of the type T. Often such a thunk is referred to as a suspended computation of the type T, which can be resumed by simply calling the thunk.

As an example, the following function int_stream_from takes an integer n and returns a stream that enumerates ascendingly all of the integers greater than or equal to n:

```
//
fun
int_stream_from
  (n: int): stream(int) =
  $delay(stream_cons(n, int_stream_from(n+1)))
//
```

The keyword $delay indicates the construction of a thunk based on the expression appearing as its argument.

Accessing elements in a given stream is shown in the following example:

```
//
fun
{a:t@ype}
stream_get_at
  (xs: stream(a), n: int): a =
(
```

```
case+ !xs of
| stream_nil() =>
  (
    $raise StreamSubscriptExn()
  )
| stream_cons(x, xs) =>
  (
    if n <= 0 then x else stream_get_at<a>(xs, n-1)
  )
)
//
```

The function stream_get_at is the stream-version of list_get_at: Given a stream and a position (denoted by an integer), stream_get_at returns the element in the stream at the given position. Note that the symbol ! refers to the function lazy_force, which evaluates a given stream value into a stream-con value in this case. Like list_get_at, it is almost always a poor style of programming to make use of stream_get_at excessively (for instance, inside the body of a loop).

Let us see what is involved in the evaluation of the following two lines of code:

```
//
val xs = int_stream_from(0)
val x0 = stream_get_at<int>(xs, 1000000)
//
```

The call to int_stream_from returns immediately as all that is essentially done is creating a thunk (that is, a nullary closure-function). The call to stream_get_at returns 1000000 after creating 1000001 nodes to store the first 1000001 elements in the stream bound to the name xs. While it takes only a tiny amount of memory to store the initial stream bound to the name xs, it requires a large amount of memory to store the expanded stream created during the evaluation of the call to stream_get_at. In general, a call to lazy_force evaluates its argument (of type lazy(T)) to a value (of type T) and then caches the value internally so that the value can be returned immediately when another call to lazy_force on the same argument evaluates later. In other words, some form of memoization happens when the evaluation of a suspended computation is performed. Memoization can be expensive memory-wise and unpredicatable time-wise. There is another kind of stream (to be presented later) that is referred to as *linear stream*, which involves no memoization when evaluated. As a matter of fact, the internal representation of a linear stream only needs memory for storing one node (that contains the head element of the stream), resulting in great memory-efficiency.

Basically, for each a list-processing function, there is a corresponding version of stream-processing function. Let us see some concrete examples. For instance, the following function stream_append (corresponding to list0_append) concatenates two given streams:

```
//
extern
fun
{a:t@ype}
stream_append
(xs: stream(a), ys: stream(a)): stream(a)
```

```
//
implement
{a}(*tmp*)
stream_append
(xs, ys) = $delay
(
case+ !xs of
| stream_nil() => !ys
| stream_cons(x, xs) =>
  stream_cons(x, stream_append<a>(xs, ys))
)
//
```

Please note the symbol ! in front of !ys: The expression following $delay should evaluate to a stream-con value (instead of a stream value).

With streams, we can facilitate the use of (general) recursion in problem-solving by eliminating the risk of stack overflow caused by deeply nested recursive calls. In the case of stream_append, the evaluation of a call to stream_append returns immediately as all it does essentially is to form a thunk (for representing the resulting stream). On the other hand, the implementation of list0_append presented previously can potentially cause stack overflow if its argument is a long list (e.g., one consisting of 1000000 elements). By the way, the actual implementation of list0_append in `ATSLIB/prelude` (the prelude library of ATS) cannot cause stack overflow due to its being tail-recursive. However, there is no free lunch here as this library implementation is written in an advanced style that is somewhat difficult to adopt in practice.

The following function stream_map is the stream-version of list0_map:

```
//
extern
fun
{a:t@ype}
{b:t@ype}
stream_map
(xs: stream(a), fopr: cfun(a, b)): stream(b)
//
implement
{a}{b}
stream_map
  (xs, fopr) = $delay
(
case+ !xs of
| stream_nil() =>
  stream_nil()
| stream_cons(x, xs) =>
  stream_cons(fopr(x), stream_map<a><b>(xs, fopr))
)
//
```

The following function stream_filter is the stream-version of list0_filter:

```
//
extern
fun
{a:t@ype}
stream_filter
(xs: stream(a), test: cfun(a, bool)): stream(a)
//
implement
{a}(*tmp*)
stream_filter
  (xs, test) = $delay
(
case+ !xs of
| stream_nil() =>
  stream_nil()
| stream_cons(x, xs) =>
  if test(x)
    then
    stream_cons
      (x, stream_filter<a>(xs, test))
    // end of [then]
    else !(stream_filter<a>(xs, test))
  // end of [if]
)
//
```

A function like stream_map and stream_filter is often referred to as being fully lazy as its evaluation does nothing except for creating a thunk (to represent a suspended computation).

The following function sieve implements the sieve of Eratosthenes for enumerating prime numbers:

```
//
fun
sieve(): stream(int) = let
//
fun
auxmain
(
xs: stream(int)
) : stream(int) = $delay
(
case- !xs of
| stream_cons(x0, xs) =>
  stream_cons(x0, auxmain(stream_filter(xs, lam(x) => x % x0 > 0)))
)
//
in
```

```
  auxmain(int_stream_from(2))
end // end of [sieve]
//
```

A call to sieve returns a stream consisting of all the primes enumerated in the ascending order: 2, 3, 5, 7, 11, etc. Note that case- is used in place of case for the purpose of suppressing a warning message that would otherwise be issued due to pattern matching being non-exhaustive (as the case stream_nil() is not covered).

As a simple experiment, please try to evaluate the following code:

```
//
val
thePrimes = sieve()
val () = println! ("stream_get_at(thePrimes, 5000):")
val () = println! (stream_get_at<int>(thePrimes, 5000))
val () = println! ("stream_get_at(thePrimes, 5000):")
val () = println! (stream_get_at<int>(thePrimes, 5000))
//
```

Note that the number *48619* is to be printed out twice. There is a clear pause before it is done for the first time, but there is virtually no delay between the first time and the second time, showing clearly the effect of memoization performed during the first call to stream_get_at on thePrimes.

Lastly, let us see a simple but telling example that demonstrates a stream-based approach to addressing the potential risk of stack overflow due to deeply nested non-tail-recursive calls. The following implementation of list0_map is standard:

```
//
implement
{a}{b}
list0_map
(xs, fopr) =
auxmain(xs) where
{
//
fun
auxmain
(
xs: list0(a)
) : list0(b) =
(
case+ xs of
| list0_nil() => list0_nil()
| list0_cons(x, xs) => list0_cons(fopr(x), auxmain(xs))
)
//
} (* end of [list0_map] *)
//
```

Clearly, this implementation is not tail-recursive. When applied to a long list (e.g., one consisting of 1000000 elements), list0_map runs a high risk of causing stack overflow. This can become a very serious issue in practice if we ever want to apply functional programming to a domain like machine learning where processing large data is a norm rather than an exception. One possibility is to insist on using only tail-recursion in the implementation of a function like list0_map that may need to be applied to large data, but such a requirement or restriction can clearly exert negative impact on the use of recursion in problem-solving. After all, there are numerous algorithms that are natrually expressed in terms of (general) recursion. And non-trivial effort is often needed in order to implement such an algorithm based on tail-recursion only, likely diminishing programming productivity.

Another implementation of list0_map is given as follows:

```
//
implement
{a}{b}
list0_map(xs, fopr) = let
//
fun
auxmain
(
xs: list0(a)
) : stream(b) = $delay
(
case+ xs of
| list0_nil() => stream_nil()
| list0_cons(x, xs) => stream_cons(fopr(x), auxmain(xs))
)
//
in
  g0ofg1(stream2list(auxmain(xs)))
end // end of [list0_map]
//
```

What is special about this implementation of list0_map lies in the implementation of of the inner function auxmain, which turns a list into a stream. Calling auxmain runs no risk of stack overflow as it simply creates a thunk (without issuing any recursive calls). The library function stream2list turns a stream into a (linear) list, which can be cast into a list0-value in O(1)-time by a call to g0ofg1. As stream2list is implemented tail-recursively, it can be safely called to generate a long list. Consequently, this stream-based implementation of list0_map can be applied to a long list with no concern of causing stack overflow. After linear lazy streams are introduced, we are to see a significantly improved version of this stream-based approach to resolving a common type of risk of stack overflow caused by calling recursively defined functions for generating long lists.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/10/CODE) the entirety of the code used in this chapter.

Chapter 13. Linear Lazy Stream-Processing

As the name suggests, linear lazy streams are the linear version of lazy streams. In ATS, a linear value is one that cannot be shared; it must be consumed properly for otherwise a type error is reported during typechecking. Before moving on to introduce programming with linear lazy streams, I would like to first explain a serious drawback in programming with (non-linear) lazy streams.

Please find on-line (https://github.com/ats-lang/ats-lang.github.io/blob/master/DOCUMENT/ATS2FUNCRASH/LECTURE/11/CODE) the entirety of the code used in this chapter.