# Introduction to Programming in ATS

**Hongwei Xi**

**hwxi AT cs DOT bu DOT edu**

**Introduction to Programming in ATS:**
by Hongwei Xi

# Dedication

To Jinning and Zoe.

# Table of Contents


**Preface** ........ ix

**I. Basic Functional Programming** ........ 1

1. Preparation for Starting ........ 1
   - A Running Program ........ 1
   - A Template for Single-File Programs ........ 1
   - A Makefile Template ........ 2
2. Elements of Programming ........ 5
   - Expressions and Values ........ 5
   - Names and Bindings ........ 5
   - Scopes for Bindings ........ 6
   - Environments for Evaluation ........ 7
   - Static Semantics ........ 7
   - Primitive Types ........ 7
   - Tuples and Tuple Types ........ 8
   - Records and Record Types ........ 9
   - Conditional Expressions ........ 10
   - Sequence Expressions ........ 11
   - Comments in Code ........ 11
3. Functions ........ 13
   - Functions as a Simple Form of Abstraction ........ 13
   - Function Arity ........ 14
   - Function Interface ........ 14
   - Evaluation of Function Calls ........ 15
   - Recursive Functions ........ 15
   - Evaluation of Recursive Function Calls ........ 16
   - Example: Coin Changes ........ 17
   - Tail-Call and Tail-Recursion ........ 18
   - Example: Solving the Eight Queens Puzzle ........ 18
   - Mutually Recursive Functions ........ 21
   - Mutual Tail-Recursion ........ 22
   - Envless Functions and Closure Functions ........ 23
   - Higher-Order Functions ........ 24
   - Example: Binary Search ........ 25
   - Currying and Uncurrying ........ 25
4. Datatypes ........ 27
   - Patterns ........ 27
   - Pattern-Matching ........ 27
   - Matching Clauses and Case-Expressions ........ 28
   - Enumerative Datatypes ........ 28
   - Recursive Datatypes ........ 29
   - Exhaustiveness of Pattern-Matching ........ 30
   - Example: Evaluating Integer Expressions ........ 31
5. Parametric Polymorphism ........ 35
   - Function Templates ........ 35
   - Polymorphic Functions ........ 36
   - Polymorphic Datatypes ........ 37
   - Example: Function Templates on Lists ........ 38
   - Example: Mergesort on Lists ........ 40
6. Summary ........ 43

**II. Support for Practical Programming** ........ 45

7. Effectful Programming Features ........ 45
   - Exceptions ........ 45
   - Example: Testing for Braun Trees ........ 47
   - References ........ 48
   - Example: Implementing Counters ........ 49
   - Arrays ........ 50
   - Example: Ordering Permutations ........ 51

Matrices ....53
Example: Estimating the Constant Pi ....54
Simple Input and Output ....55
8. Convenience in Programming ....59
Macro Definitions ....59
Compile-Time Directives ....61
Overloading ....61
9. Modularity ....63
Types as a Form of Specification ....63
Static and Dynamic ATS Files ....64
Generic Template Implementation ....66
Abstract Types ....66
Example: A Package for Rationals ....68
Example: A Functorial Package for Rationals ....69
Specific Template Implementation ....71
Example: A Temptorial Package for Rationals ....72
10. Library Support ....75
The prelude Library ....75
The libc Library ....75
The libats Library ....75
Contributed Packages ....75
11. Interaction with the C Programming Language ....77
12. Summary ....79
**III. Dependent Types for Programming ....81**
13. Introduction to Dependent Types ....81
Enhanced Expressiveness for Specification ....81
Constraint-Solving during Typechecking ....84
Example: String Processing ....85
Example: Binary Search on Arrays ....86
Termination-Checking for Recursive Functions ....87
Example: Dependent Types for Debugging ....89
14. Datatype Refinement ....91
Dependent Datatypes ....91
Example: Function Templates on Lists (Redux) ....93
Example: Mergesort on Lists (Redux) ....95
Sequentiality of Pattern Matching ....96
Example: Functional Random-Access Lists ....97
Example: Functional Red-Black Trees ....101
15. Theorem-Proving in ATS/LF ....105
Encoding Relations as Dataprops ....105
Constructing Proofs as Total Functions ....106
Example: Proving Distributivity of Multiplication ....107
Datasorts ....108
Example: Proving Properties on Braun Trees ....110
16. Programming with Theorem-Proving ....115
Circumventing Nonlinear Constraints ....115
Example: Safe Matrix Subscripting ....116
Specifying with Enhanced Precision ....117
Example: Another Verified Factorial Implementation ....118
Example: Verified Fast Exponentiation ....119
17. Summary ....123
**IV. Linear Types for Programming ....125**
18. Introduction to Views and Viewtypes ....125
Views for Memory Access through Pointers ....125
Viewtypes as a Combination of Views and Types ....127
Left-Values and Call-by-Reference ....128
Stack-Allocated Variables ....129
19. Dataviews as Linear Dataprops ....131
Optional Views ....131

Linear Arrays ....................132
Singly-Linked Lists ....................134
Proof Functions for View Changes ....................137
Example: Quicksort ....................140
20. Dataviewtypes as Linear Datatypes ....................145
Linear Optional Values ....................145
Linear Lists ....................146
Example: Mergesort on Linear Lists ....................149
Linear Binary Search Trees ....................152
Transition from Datatypes to Dataviewtypes ....................155
21. Abstract Views and Viewtypes ....................161
Memory Allocation and Deallocation ....................161
Simple Linear Objects ....................162
Example: Implementing an Array-Based Circular Buffer ....................165
From linearity to non-linearity ....................166
22. Summary ....................169

# Preface

ATS[1] is a statically typed programming language that unifies implementation with formal specification. Within ATS, there are two sublanguages: one for specification and the other for implementation, and there is also a theorem-proving subsystem for verifying whether an implementation indeed implements according to its specification. If I could associate only one single word with ATS, I would choose the word *precision*. Programming in ATS is about being precise and being able to effectively enforce precision. This point will be demonstrated concretely and repeatedly in this book.

In order to be precise in building software systems, we need to specify what such a system is expected to accomplish. In the current day and age, software specification, which we use in a rather loose sense, is often done in forms of varying degree of formalism, ranging from verbal discussions to pencil/paper drawings to various diagrams in modeling languages such as UML to formal specifications in specification languages such as Z[2], etc. Often the main purpose of software specification is to establish some mutual understanding among a team of developers. After the specification for a software system is done, either formally or informally, we need to implement the specification in a programming language. In general, it is exceedingly difficult to be reasonably certain whether an implementation actually meets its specification. Even if the implementation coheres well with its specification initially, it nearly inevitably diverges from the specification as the software system evolves. The dreadful consequences of such a divergence are all too familiar; the specification becomes less and less reliable for understanding the behavior of the software system while the implementation gradually turns into its own specification; for the developers, it becomes increasingly difficult and risky to maintain and extend the software system; for the users, it requires increased amount of time and effort to learn and use the software system.

Largely inspired by Martin-Loef's constructive type theory, which was originally developed for the purpose of establishing a foundation for mathematics, I designed ATS in an attempt to combine specification and implementation into a single programming language. There are a static component (statics) and a dynamic component (dynamics) in ATS. Intuitively, the statics and dynamics are each for handling types and programs, respectively. In particular, specification is done in the statics. Given a specification, how can we then effectively ensure that an implementation of the specification (type) indeed implements according to the specification? We request that the programmer who does the implementation also construct a proof in the theorem-proving subsystem of ATS to demonstrate it. This is a style of program verification that puts the programmer at the center, and thus we refer to it as a programmer-centric approach to program verification.

ATS is also a feature-rich programming language. It can support a variety of programming paradigms, including functional programming, imperative programming, object-oriented programming, concurrent programming, modular programming, etc. However, the core of ATS, which is based on a call-by-value functional language, is surprisingly simple, and this is where the journey of programming in ATS starts. In this book, I will demonstrate primarily through examples how various programming features in ATS can be employed effectively to facilitate the construction of high-quality programs. I will focus on programming practice instead of programming theory. If you are primarily interested in the type-theoretical foundation of ATS, then you have to find it elsewhere.

If you can implement, then you are a good programmer. In order to be a better programmer, you should also be able to explain what you implement. If you can guarantee what is implemented matches what is specified, then you are surely the best programmer. Hopefully, learning ATS will put you on a wonderful exploring journey to become the best programmer. Let that journey start now!

## Notes

1. http://www.ats-lang.org
2. http://www.afm.sbu.ac.uk/zbook

# Chapter 1. Preparation for Starting

It is most likely that you want to write programs in the programming language you are learning. You may also want to try some of the examples included in this book and see what really happens. So I will first show you how to write in ATS a single-file program, that is, a program contained in a single file, and compile it and then execute it.

## A Running Program

The following example is a program in ATS that prints out (onto the console) the string "Hello, world!" and a newline before it terminates:

```
val _void_ = print ("Hello, world!
")
implement main () = () // a dummy implementation for [main]
```

The keyword val initiates a binding between the variable _void_ and the function call print ("Hello, world! "). However, this binding is never used after it is introduced; its sole purpose is for the call to the print function to get evaluated.

The function main is of certain special meaning in ATS, which I will explain elsewhere. For a programmer who knows the C or Java programming language, I simply point out that the role of main is essentially the same as its counterpart of the same name in C or Java. The keyword implement initiates the implementation of a function whose interface has already been declared elsewhere. The declared interface for main in ATS is given as follows:

```
fun main (): void
```

which indicates that main is a nullary function, that is, a function that takes no arguments, and it returns no value (or it returns the void value). The double slash symbol (//) initiates a comment that terminates at the end of the current line.

Suppose that you have already installed the ATS programming language system. You can issue the following command-line to generate an executable named `hello` in the current working directory:

```
atscc -o hello hello.dats
```

where `hello.dats` is a file containing the above program. Note that the filename extension *.dats* should not be altered as it has already been assigned a special meaning that the compilation command **atscc** recognizes. Another special filename extension is *.sats*, which we will encounter elsewhere.

## A Template for Single-File Programs

The following code template, which is available on-line[1], is designed for constructing a single-file program in ATS:

```
(*
**
** This is a template for a single-file ATS program
**
*)

(* ****** ****** *)
```

```
(*
** please do not change unless you know what you do
*)
//
staload _(*anon*) = "libc/SATS/stdio.sats"
//
staload _(*anon*) = "prelude/DATS/array.dats"
staload _(*anon*) = "prelude/DATS/array0.dats"
//
staload _(*anon*) = "prelude/DATS/list.dats"
staload _(*anon*) = "prelude/DATS/list0.dats"
staload _(*anon*) = "prelude/DATS/list_vt.dats"
//
staload _(*anon*) = "prelude/DATS/matrix.dats"
staload _(*anon*) = "prelude/DATS/matrix0.dats"
//
staload _(*anon*) = "prelude/DATS/option.dats"
staload _(*anon*) = "prelude/DATS/option0.dats"
//
staload _(*anon*) = "prelude/DATS/pointer.dats"
//
staload _(*anon*) = "prelude/DATS/reference.dats"
//
(* ****** ****** *)

//
// please write you program in this section
//

(* ****** ****** *)

implement main () = () // a dummy implementation for [main]
```

Each line starting with the keyword staload essentially allows the ATS compiler **atsopt** to gain access to the definition of certain library functions. I will cover elsewhere in the book the topic on programming with library functions in ATS.

## A Makefile Template

The following Makefile template, which is available on-line[2], is provided to help you construct your own Makefile for compiling ATS programs. If you are not familiar with the **make** utility, you could readily find plenty resources on-line to help yourself learn it.

```
##
##
## A Makefile template for compiling ATS programs
##
##

######

ATSUSRQ="$(ATSHOME)"
ifeq ($(ATSUSRQ),"")
ATSUSRQ="/usr"
endif # end of [ifeq]

######

ATSCC=$(ATSUSRQ)/bin/atscc
ATSOPT=$(ATSUSRQ)/bin/atsopt

######
```

```
#
# Please uncomment the one you want, or skip it entirely
#
ATSCCFLAGS=
#ATSCCFLAGS=-O2
#
# [-flto] enables link-time optimization such as inlining lib functions
#
#ATSCCFLAGS=-O2 -flto

######

#
# HX: Please uncomment it if you need to run GC at run-time
#
ATSGCFLAG=
#ATSGCFLAG=-D_ATS_GCATS

######

distclean::

######

#
# Please uncomment the following three lines and replace the name [foo]
# with the name of the file you want to compile
#

# foo: foo.dats
#  $(ATSCC) $(ATSGCFLAG) $(ATSCCFLAGS) -o $@ $< || touch $@
# distclean:: ; $(RMF) foo

######

#
# You may find these rules useful
#

# %_sats.o: %.sats
#  $(ATSCC) $(ATSCCFLAGS) -c $< || touch $@

# %_dats.o: %.dats
#  $(ATSCC) $(ATSCCFLAGS) -c $< || touch $@

######

RMF=rm -f

######

clean:
 $(RMF) *~
 $(RMF) *_?ats.o
 $(RMF) *_?ats.c

distclean:: clean

###### end of [Makefile] ######
```

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_START/mytest.dats
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_START/Makefile_tem

# Chapter 2. Elements of Programming

The core of ATS is a call-by-value functional programming language. I will explain the meaning of *call-by-value* in a moment. As for functional programming, there is really no precise definition. The most important aspect of functional programming that I want to explore is the notion of binding, which relates names to expressions.

## Expressions and Values

ATS is feature-rich, and its grammar is probably more complex than most existing programming languages. In ATS, there are a large variety of forms of expressions, which I will introduce gradually.

Let us first start with some integer arithmetic expressions (IAEs): 1, ~2, 1+2, 1+2*3-4, (1+2)/(3-4), etc. Note that the negative sign is represented by the tilde symbol (~) in ATS. There is also support for floating point numbers, and some floating point constants are given here: 1.0, ~2.0, 3., 0.12345, 2.71828, 31416E-4, etc. Note that 3. and 31416E-4 are the same as 3.0 and 3.1416, respectively. What I really want to emphasize at this point is that 1 and 1.0 are two distinct numbers in ATS: the former is an integer while the latter is a floating point number.

There are also boolean constants: true and false. We can form boolean expressions such as 1 >= 0, not(2-1 >= 2), (1 < 2) andalso (2 < 3) and (~1 > 1) orelse (~1 <= 1), where not, andalso and orelse stand for negation, conjunction and disjunction, respectively.

Other commonly used constant values include characters and strings. For instance, here are some character constants: ’a’, ’B’, ’ ’ (newline), ’ ’ (tab), ’(’ (left parenthesis), ’)’ (right parenthesis), ’{’ (left curly brace), ’}’ (right curly brace), etc; here are some string constants: "My name is Zoe", "Don’t call me "Cloe"", "this is a newline: ", etc.

Given a (function) name, say, foo, and an expression exp, the expression foo(exp) is a function application or function call. The parentheses in foo(exp) may be dropped if no ambiguity is created by doing so. For instance, print("Hello") is a function application, which can also be written as print "Hello". If foo is a nullary function, then a function application foo() can be formed. If foo is a binary function, then a function application foo(exp1, exp2) can be formed for expressions exp1 and exp2. Functions of more arguments can be treated accordingly.

Note that we cannot write +(1,2) as the name + has already been given the infix status requiring that it be treated as an infix operator. However, we can write op+(1,2), where op is a keyword in ATS that can be used to temporarily suspend the infix status of any name immediately following it. I will explain in detail the issue of fixity (prefix, infix and postfix) elsewhere.

Values are essentially expressions of certain special forms, which can not be reduced or simplified further. For instance, integer constants such as 1 and ~2 are values, but the integer expression 1+2 is not a value, which can be reduced to the value 3. Evaluation refers to the computational process that reduces a given expression into a value. However, certain expressions such as 1/0 cannot be reduced to a value, and evaluating such an expression must abort at some point. I will gradually present more information on evaluation.

## Names and Bindings

A crucial aspect of a programming language is the mechanism it provides for binding names, which are themselves expressions, to expressions. For instance, a declaration is introduced by the following syntax that declares a binding between the name x, which is also referred to as a variable, and the expression 1+2:

```
val x = 1 + 2
```

Note that val is a keyword in ATS, and the declaration is classified as a val-declaration. Conceptually, what happens at run-time in a call-by-value language such as ATS is that the expression 1+2 is first evaluated to the value 3, and then the binding between x and 1+2 is *finalized* into a binding between x and 3. Essentially, call-by-value means that a binding between a name and an expression needs to be finalized into one between the name and the value of the expression before it can be used in evaluation subsequently. As another example, the following syntax declares three bindings, two of which are formed simultaneously in the first line:

```
val PI = 3.14 and radius = 10.0
val area = PI * radius * radius
```

Note that it is unspecified in ATS as to which of the first two bindings is finalized ahead of the other at run-time. However, it is guaranteed that the third binding is finalized after the first two are done.

## Scopes for Bindings

Each binding is given a fixed scope in which the binding is considered legal or effective. The scope of a toplevel binding in a file starts from the point where the binding is introduced until the very end of the file. The bindings introduced in the following example between the keywords let and in are effective until the keyword end is reached:

```
val area = let
  val PI = 3.14 and radius = 10.0 in PI * radius * radius
end // end of [let]
```

Such bindings are referred to as local bindings, and the names such as PI and radius are referred to as local names. This example can also be written in the following style:

```
val area =
  PI * radius * radius where {
  val PI = 3.14 and radius = 10.0 // simultaneous bindings
} // end of [where]
```

The keyword where appearing immediately after an expression introduces bindings that are solely effective for evaluating names contained in the expression. Note that expressions formed using the keywords let and where are often referred to as let-expressions and where-expressions, respectively. The former can always be translated into the latter directly and vice versa. Which style is better? I have not formed my opinion yet. The answer seems to entirely depend on the taste of the programmer.

The following example demonstrates an alternative approach to introducing local bindings:

```
local

val PI = 3.14 and radius = 10.0

in

val area = PI * radius * radius

end // end of [local]
```

where the bindings introduced between the keywords local and in are effective until the keyword end is reached. Note that the bindings introduced between the keywords in and end are themselves toplevel bindings. The difference between let and local should be clear: The former is used to form an expression while the latter is used to introduce a sequence of declarations.

## Environments for Evaluation

Evaluation is the computational process that reduces expressions to values. When performing evaluation, we need not only the expression to be evaluated but also a collection of bindings that map names in the expression to values. This collection of bindings, which is just a finite mapping, is often referred to as an environment (for evaluation). For instance, suppose that we want to evaluate the following expression:

```
let
  val PI = 3.14 and radius2 = 10.0 * 10.0 in PI * radius2
end
```

We start with the empty environment ENV0; we evaluate 3.14 to itself and 10.0 * 10.0 to 100.0 under the environment ENV0; we then extend ENV0 to ENV1 with two bindings mapping PI to 3.14 and radius2 to 100.0; we then evaluate PI * radius2 under ENV1 to 3.14 * radius2, then to 3.14 * 100.0, and finally to 314.0, which is the value of the let-expression.

## Static Semantics

ATS is a programming language equipped with a highly expressive type system rooted in the *Applied Type System* framework, which also gives ATS its name. I will gradually introduce the type system of ATS, which is probably the most outstanding and interesting part of this book.

It is common to treat a type as the set of values it classifies. However, I find it more approriate to treat a type as a form of meaning. There are formal rules for assigning types to expressions, which are referred to as typing rules. If a type T can be assigned to an expression, then I say that the expression possesses the static meaning (semantics) represented by the type T. Note that an expression may be assigned many distinct static meanings. An expression is well-typed if there exists a type T such that the expression can be assigned the type T.

If there is a binding between a name and an expression and the expression is of some type T, then the name is assumed to be of the type T in the effective scope of the binding. In other words, the name assumes the static meaning of the expression it refers to.

Let exp0 be an expression of some type T, that is, the type T can be assigned to exp0 according to the typing rules. If we can evaluate exp0 to exp1, then exp1 can also be assigned the type T. In other words, static meaning is an invariant under evaluation. This property is often referred to as *type preservation*, which is part of the soundness of the type system of ATS. Based on this property, we can readily infer that any value is of the type T if exp0 can be evaluated to it (in many steps).

Let exp0 be an expression of some type T. Assume that exp0 is not a value. Then exp0 can always be evaluated one step further to another expression exp1. This property is often referred to as *progress*, which is another part of the soundness of the type system of ATS.

## Primitive Types

The simplest types in ATS are primitive types, which are used to classify primitive values. For instance, we have the primitive types int and double, which classify integers (in a fixed range) and floating point numbers (of double precision), respectively.

In the current implementation of ATS (Anairiats), a program in ATS is first compiled into one in C, which can then be compiled to object code by a compiler for C such as gcc. In the compilation from ATS to C, the type int in ATS is translated to the type of the same name in C. Similarly, the type double in ATS is translated to the type of the same name in C.

There are many other primitive types in ATS, and I will introduce them gradually. Some commonly used primitive types are listed as follows:

- bool: This type is for boolean values true and false.
- char: This type is translated into the type in C for characters.
- schar: This type is translated into the type in C for signed characters.
- uchar: This type is translated into the type in C for unsigned characters.
- float: This type is translated into the type in C for floating point numbers of single precision.
- uint: This type is translated into the type in C for unsigned integers.
- lint: This type is translated into the type in C for long integers.
- ulint: This type is translated into the type in C for unsigned long integers.
- llint: This type is translated into the type in C for long long integers.
- ullint: This type is translated into the type in C for unsigned long long integers.
- size_t: This type is translated into the type in C of the same name, which is for unsigned integers of certain precision.
- string: This type is for strings, and its translation in C is the type void* (for pointers). I will explain this translation elsewhere.
- void: This type is for the void value, and its translation in C is the type void (for pointers). It should be noted that the void value is unspecified in ATS. I often say that a function returns no value if it returns the void value, and vice versa.

I will gradually present programming examples involving various primitive types and values.

## Tuples and Tuple Types

Given two types T1 and T2, we can form a tuple type (T1, T2), which can also be written as @(T1, T2). Assume that exp1 and exp2 are two expressions of the types T1 and T2, respectively. Then the expression (exp1, exp2), which can also be written as @(exp1, exp2), refers to a tuple of the tuple type (T1, T2). Accordingly, we can form tuples and tuple types of more components. In order for a tuple type to be assigned to a tuple, the tuple and tuple type must have the equal number of components.

When evaluating a tuple expression, we evaluate all of its components *sequentially*. Suppose that the expression contains n components, then the value of the expression is the tuple consisting of the n values of the n components listed in the order as the components themselves.

A tuple of length n for n >= 2 is just a record of field names ranging from 0 until n-1, inclusively. Given an expression exp of some tuple type (T1, T2), we can form expressions (exp).0 and (exp).1, which are of types T1 and T2, respectively. Note that the expression exp does not have to be a tuple expression. For instance, exp may be a name or a function application. If exp evaluates to a tuple of two values, then exp.0

evaluates to the first value and exp.1 the second value. Clearly, if the tuple type of exp contains more components, what is stated can be generalized accordingly.

In the following example, we first construct a tuple of length 3 and then introduce bindings between 3 names and all of the 3 components of the tuple:

```
val xyz = ('A', 1, 2.0)
val x = xyz.0 and y = xyz.1 and z = xyz.2
```

Note that the constructed tuple can be assigned the tuple type (char, int, double). Another method for selecting components in a given tuple is based on pattern matching, which is employed in the following example:

```
val xyz = ('A', 1, 2.0)
val (x, y, z) = xyz // x = 'A'; y = 1; z = 2.0
```

Note that (x, y, z) is a pattern that can match any tuples of exact 3 components. I will say more about pattern matching elsewhere.

The tuples introduced above are often referred to as flat tuples, native tuples or unboxed tuples. There is another kind of tuples supported in ATS, which are called boxed tuples. A boxed tuple is essentially a pointer pointing to some heap location where a flat tuple is stored.

Assume that exp1 and exp2 are two expressions of the types T1 and T2, respectively. Then the expression '(exp1, exp2), refers to a tuple of the tuple type '(T1, T2). Accordingly, we can form boxed tuples and boxed tuple types of fewer or more components. What should be noted immediately is that every boxed tuple is of the size of a pointer, and can thus be stored in any place where a pointer can. Using boxed tuples is rather similar to using unboxed ones. For instance, the meaning of the following code should be evident:

```
val xyz = '( 'A', 1, 2.0 )
val x = xyz.0 and y = xyz.1 and z = xyz.2
```

Note that a space is needed between '( and 'A' for otherwise the current parser (for Anairiats) would be confused.

Given the availability of flat and boxed tuples, one naturally wants to know whether there is a simple way to determine which kind is preferred over the other. Unfortunately, there is no simple way to do this as far as I can tell. In order to be certain, some kind of profiling is often needed. However, if we want to run code with no support of garbage collection (GC), then we should definitely avoid using boxed tuples.

## Records and Record Types

A record is just like a tuple except that each field name of the record is chosen by the programmer (instead of being fixed). Similarly, a record type is just like a tuple type. For instance, a record type point2D is defined as follows:

```
typedef point2D = @{ x= double, y= double }
```

where x and y are the names of the two fields in a record value of this type. We also refer to a field in a record as a component. The special symbol @{ indicates that the formed type is for flat/native/unboxed records. A value of the type point2D is constructed as follows and given the name theOrigin:

```
val theOrigin = @{ x= 0.0, y= 0.0 } : point2D
```

We can use the standard dot notation to extract out a selected component in a record, and this is shown in the next line of code:

```
val theOrigin_x = theOrigin.x and theOrigin_y = theOrigin.y
```

Alternatively, we can use pattern matching for doing component extraction as is done in the next line of code:

```
val @{ x= theOrigin_x, y= theOrigin_y } = theOrigin
```

In this case, the names theOrigin_x and theOrigin_y are bound to the components in theOrgin that are named x and y, respectively. If we only need to extract out a selected few of components (instead of all the available ones), we can make use of the following kind of patterns:

```
val @{ x= theOrigin_x, ... } = theOrigin // the x-component only
val @{ y= theOrigin_y, ... } = theOrigin // the y-component only
```

If you find all this syntax for component extraction to be confusing, then I suggest that you stick to the dot notation. I myself rarely use pattern matching on record values.

Compared with handling native/flat/unboxed records, the only change needed for handling boxed records is to replace the special symbol @{ with another one: '{, which is a quote followed immediately by a left curly brace.

## Conditional Expressions

A conditional expression consists of a test and two branches. For instance, the following expression is conditional:

```
if (x >= 0) then x else ~x
```

where if, then and else are keywords in ATS. In a conditional expression, the expression following if is the test and the expressions following then and else are referred to as the then-branch and the else-branch (of the conditional expression), respectively.

In order to assign a type T to a conditional expression, we need to assign the type bool to the test and the type T to both of the then-branch and the else-branch. For instance, the type int can be assigned to the above conditional expression if the name x is given the type int. One may think that the following conditional expression is ill-typed, that is, it cannot be given type:

```
if (x >= 0) then '0' else 1 // this expression can be given a type!
```

Actually, it is possible to find a type T in ATS that can be assigned to the conditional expression. I will explain the reason for this elsewhere.

Suppose that we have a conditional expression that is well-typed. When evaluating it, we first evaluate the test to a value, which is guaranteed to be either true or false; if the value is true, then we continue to evaluate the then-branch; otherwise, we continue to evaluate the else-branch.

It is also allowed to form a conditional expression where the else-branch is truncated. For instance, we can form something as follows:

```
if (x >= 0) then print(x)
```

which is equivalent to the following conditional expression:

```
if (x >= 0) then print(x) else ()
```

Note that () stands for the void value (of the type void). If a type can be assigned to a conditional expression in the truncated form, then the type must be void.

## Sequence Expressions

Assume that exp1 and exp2 are expressions of types T1 and T2 respectively, where T1 is void. Then a sequence expression (exp1; exp2) can be formed that is of the type T2. When evaluating the sequence expression (exp1; exp2), we first evaluate exp1 to the void value and then evaluate exp2 to some value, which is also the value of the sequence expression. When more expressions are sequenced, all of them but the last one need to be of the type void and the type of the last expression is also the type of the sequence expression being formed. Evaluating a sequence of more expressions is analogous to evaluating a sequence of two. The following example is a sequence expression:

```
(print 'H'; print 'e'; print 'l'; print 'l'; print 'o')
```

Evaluating this sequence expression prints out (onto the console) the 5-letter string "Hello". Instead of parentheses, we can also use the keywords begin and end to form a sequence expression:

```
begin
  print 'H'; print 'e'; print 'l'; print 'l'; print 'o'
end // end of [begin]
```

If we like, we may also add a semicolon immediately after the last expression in a sequence as long as the last expression is of the type void. For instance, the above example can also be written as follows:

```
begin
  print 'H'; print 'e'; print 'l'; print 'l'; print 'o';
end // end of [begin]
```

I also want to point out the following style of sequencing:

```
let
  val () = print 'H'
  val () = print 'e'
  val () = print 'l'
  val () = print 'l'
  val () = print 'o'
in
  // nothing
end // end of [begin]
```

which is rather common in functional programming.

## Comments in Code

ATS currently supports four forms of comments: line comment, block comment of ML-style, block comment of C-style, and rest-of-file comment.

- A line comment starts with the double slash symbol (//) and extends until the end of the current line.
- A block comment of ML-style starts and closes with the tokens (* and *), respectively. Note that nested block comments of ML-style are allowed, that is, one block comment of ML-style can occur within another one of the same style.
- A block comment of C-style starts and closes with the tokens /* and */, respectively. Note that block comments of C-style cannot be nested. The use of block comments of C-style is primarily in code that is supposed to be shared by ATS and C. In other cases, block comments of ML-style should be the preferred choice.
- A rest-of-file comment starts with the quadruple slash symbol (////) and extends until the end of the file. Comments of this style of are particularly useful for the purpose of debugging.

# Chapter 3. Functions

Functions play a foundational role in programming. While it may be theoretically possible to program without functions (but with loops), such a programming style is of little practical value. ATS does provide some language constructs for implementing for-loops and while-loops directly. I, however, strongly recommend that the programmer implement loops as recursive functions or more precisely, as tail-recursive functions. This is a programming style that matches well with more advanced programming features in ATS that will be presented in this book later.

The code employed for illustration in this chapter plus some additional code for testing is available on-line[1].

## Functions as a Simple Form of Abstraction

Given an expression exp of the type double, we can multiply exp by itself to compute its square. If exp is a complex expression, we may introduce a binding between a name and exp so that exp is only evaluated once. This idea is shown in the following example:

```
let val x = 3.14 * (10.0 - 1.0 / 1.4142) in x * x end
```

Now suppose that we have found a more efficient way to do squaring. In order to take full advantage of it, we need to modify each occurrence of squaring in the current program accordingly. This style of programming is clearly not modular, and it is of little chance to scale. To address this problem, we can implement a function as follows to compute the square of a given floating point number:

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

The keyword fn initiates the definition of a non-recursive function, and the name following it is for the function to be defined. In the above example, the function square takes one argument of the name x, which is assumed to have the type double, and returns a value of the type double. The expression on the right-hand side (RHS) of the symbol = is the body of the function, which is x * x in this case. If we have a more efficient way to do squaring, we can just re-implement the body of the function square accordingly to take advantage of it, and there is no other changes needed (assuming that squaring is solely done by calling square).

If square is a name, what is the expression it refers to? It turns out that the above function definition can also be written as follows:

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

where the RHS of the symbol = is a lambda-expression representing an anonymous function that takes one argument of the type double and returns a value of the type double, and the expression following the symbol => is the body of the function. If we wish, we can change the name of the function argument as follows:

```
val square = lam (y: double): double => y * y
```

This is called alpha-renaming (of function arguments), and the new lambda-expression is said to be alpha-equivalent to the original one.

A lambda-expression is a (function) value. Suppose we have a lambda-expression representing a binary function, that is, a function taking two arguments. In order to assign a type of the form (T1, T2) -> T to the lambda-expression, we need to verify that the body of the function can be given the type T if the two arguments of the function are assumed to have the types T1 and T2. What is stated also applies, *mutatis* *mutandis*, to lambda-expressions representing functions of fewer or more arguments. For instance, the lambda-expression lam (x: double): double => x * x can be assigned the function type (double) -> double, which may also be written as double -> double.

Assume that exp is an expression of some function type (T1, T2) -> T. Note that exp is not necessarily a name or a lambda-expression. If expressions $exp_1$ and $exp_2$ can be assigned the types T1 and T2, then the function application exp($exp_1$, $exp_2$), which may also be referred to as a function call, can be assigned the type T. Typing a function application of fewer or more arguments is handled similarly.

Let us now see an example that builds on the previously defined function square. The boundary of a ring consists of two circles centered at the same point. If the radii of the outer and inner circles are R and r, respectively, then the area of the ring can be computed by the following function area_of_ring:

```
fn area_of_ring
  (R: double, r: double): double = 3.1416 * (square(R) - square(r))
// end of [area_of_ring]
```

Given that the subtraction and multiplication functions (on floating point numbers) are of the type (double, double) -> double and square is of the type (double) -> double, it is a simple routine to verify that the body of area_of_ring can be assigned the type double.

## Function Arity

The arity of a function is the number of arguments the function takes. Functions of arity 0, 1, 2 and 3 are often called nullary, unary, binary and ternary functions, respectively. For example, the following function sqrsum1 is a binary function such that its two arguments are of the type int:

```
fn sqrsum1 (x: int, y: int): int = x * x + y * y
```

We can define a unary function sqrsum2 as follows:

```
typedef int2 = (int, int)
fn sqrsum2 (xy: int2): int =
  let val x = xy.0 and y = xy.1 in x * x + y * y end
// end of [sqrsum2]
```

The keyword typedef introduces a binding between the name int2 and the tuple type (int, int). In other words, int2 is treated as an abbreviation or alias for (int, int). The function sqrsum2 is unary as it takes only one argument, which is a tuple of the type int2. When applying sqrsum2 to a tuple consisting of 1 and ~1, we need to write sqrsum2 @(1, ~1). If we simply write sqrsum2 (1, ~1), then the typechecker is to report an error of arity mismatch as it assumes that sqrsum2 is applied to two arguments (instead of one that is a tuple).

Many functional languages (e.g., Haskell and ML) only allow unary functions. A function of multiple arguments is encoded in these languages as a unary function taking a tuple as its only argument or it is curried into a function that takes these arguments sequentially. ATS, however, provides direct support for functions of multiple arguments. There is even some limited support in ATS for variadic functions, that is, functions of indefinite number of arguments (e.g., the famous printf function in C). This is a topic I will cover elsewhere.

## Function Interface

The interface for a function specifies the type assigned to the function. It offers a means to describe a function that is both efficient and informative. Given a binary function foo of the type (T1, T2) -> T3, its interface can be written as follows:

```
fun foo (arg1: T1, arg2: T2): T3
```

where arg1 and arg2 may be replaced with any other legal identifiers for function arguments. For functions of more or fewer arguments, interfaces can be written in a similar fashion. For instance, we have the following interfaces for various functions on integers:

```
fun succ_int (x: int): int // successor
fun pred_int (x: int): int // predecessor

fun add_int_int (x: int, y: int): int // +
fun sub_int_int (x: int, y: int): int // -
fun mul_int_int (x: int, y: int): int // *
fun div_int_int (x: int, y: int): int // /

fun mod_int_int (x: int, y: int): int // modulo
fun gcd_int_int (x: int, y: int): int // greatest common divisor

fun lt_int_int (x: int, y: int): bool // <
fun lte_int_int (x: int, y: int): bool // <=
fun gt_int_int (x: int, y: int): bool // >
fun gte_int_int (x: int, y: int): bool // >=
fun eq_int_int (x: int, y: int): bool // =
fun neq_int_int (x: int, y: int): bool // <>

fun max_int_int (x: int, y: int): int // maximum
fun min_int_int (x: int, y: int): int // minimum

fun print_int (x: int): void
fun tostring_int (x: int): string
```

For now, I mostly use function interfaces for the purpose of presenting functions. I will show later how a function definition can be separated into two parts: a function interface and an implementation that implements the function interface. Note that separation as such is pivotal for constructing (large) programs in a modular style.

## Evaluation of Function Calls

Evaluating a function call is straightforward. Assume that we are to evaluate the function call abs(0.0 - 1.0) under some environment ENV0, where the function abs is defined as follows:

```
fn abs (x: double): double = if x >= 0 then x else ~x
```

We first evaluate the argument of the call to ~1.0 under ENV0; we then extend ENV0 to ENV1 with a binding between x and ~1.0 and start to evaluate the body of abs under ENV1; we evaluate the test x >= 0 to ~1.0 >= 0 and then to false, which indicates that we take the else-branch ~x to continue; we evaluate ~x to ~(~1.0) and then to 1.0; so the evaluation of the function call abs(0.0 - 1.0) returns 1.0.

## Recursive Functions

A recursive function is one that may make calls to itself in its body. Therefore, a non-recursive function is just a special kind of of recursive function: the kind that does not make any calls to itself in its body. I consider recursion the most enabling feature a programming language can provide. With recursion, we are enabled to do problem-solving based on a strategy of reduction: In order to solve a problem to which a solution is difficult to find immediately, we reduce the problem to problems that are similar but simpler, and we repeat this reduction process if needed until solutions become apparent. Let us now see some concrete examples of problem-solving that make use of this reduction strategy.

Suppose that we want to sum up all the integers ranging from 1 to n, where n is a given integer. This can be readily done by implementing the following recursive function sum1:

```
fun sum1 (n: int): int = if n >= 1 then sum1 (n-1) + n else 0
```

Note that the keyword fun initiates the definition of a recursive function. To find out the sum of all the integers ranging from 1 to n, we call sum1 (n). The reduction strategy for sum1 (n) is straightforward: If n is greater than 1, then we can readily find the value of sum1 (n) by solving a simpler problem, that is, finding the value of sum1 (n-1).

We can also solve the problem by implementing the following recursive function sum2 that sums up all the integers in a given range:

```
fun sum2 (m: int, n: int): int =
  if m <= n then m + sum2 (m+1, n) else 0
// end of [sum2]
```

This time, we call sum2 (1, n) in order to find out the sum of all the integers ranging from 1 to n. The reduction strategy for sum2 (m, n) is also straightforward: If m is less than n, then we can readily find the value of sum2 (m, n) by solving a simpler problem, that is, finding the value of sum2 (m+1, n). The reason for sum2 (m+1, n) being simpler than sum2 (m, n) is that m+1 is closer to n than m is.

Given integers m and n, there is another strategy for summing up all the integers from m to n: If m does not exceed n, we can find the sum of all the integers from m to (m+n)/2-1 and then the sum of all the integers from (m+n)/2+1 to n and then sum up these two sums and (m+n)/2. The following recursive function sum3 is implemented precisely according to this strategy:

```
fun sum3 (m: int, n: int): int =
  if m <= n then let
    val mn2 = (m+n)/2 in sum3 (m, mn2-1) + mn2 + sum3 (mn2+1, n)
  end else 0 // end of [if]
// end of [sum3]
```

It should be noted that the division involved in the expression (m+n)/2 is integer division for which rounding is done by truncation.

## Evaluation of Recursive Function Calls

Evaluating a call to a recursive function is not much different from evaluating one to a non-recursive function. Let fib be the following defined function for computing the Fibonacci numbers:

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

```
// end of [fib]
```

Suppose that we are to evaluate fib(2) under some environment ENV0. Given that 2 is already a value, we extend ENV0 to ENV1 with a binding between n and 2 and start to evaluate the body of fib under ENV1; clearly, this evaluation leads to the evaluation of fib(n-1) + fib(n-2); it is easy to see that evaluating fib(n-1) and fib(n-2) under ENV1 leads to 1 and 0, respectively, and the evaluation of fib(n-1) + fib(n-2) eventually returns 1 (as the result of 1+0); thus the evaluation of fib(2) under ENV0 yields the integer value 1.

Let us now evaluate fib(3) under ENV0; we extend ENV0 to ENV2 with a binding between n and 3, and start to evaluate the body of fib under ENV2; we then reach the evaluation of fib(n-1) + fib(n-2) under ENV2; evaluating fib(n-1) under ENV2 leads to the evaluation of fib(2) under ENV2, which eventually returns 1; evaluating fib(n-2) under ENV2 leads to the evaluation of fib(1) under ENV2, which eventually returns 1; therefore, evaluating fib(3) under ENV0 returns 2 (as the result of 1+1).

## Example: Coin Changes

Let S be a finite set of positive numbers. The problem we want to solve is to find out the number of distinct ways for a given integer x to be expressed as the sum of multiples of the positive numbers chosen from S. If we interpret each number in S as the denomination of a coin, then the problem asks how many distinct ways there exist for a given value x to be expressed as the sum of a set of coins. If we use cc(S, x) for this number, then we have the following properties on the function cc:

- cc(S, 0) = 1 for any S.
- If x < 0, then cc(S, x) = 0 for any S.
- If S is empty and x > 0, then cc(S, x) = 0.
- If S contains a number c, then cc(S, x) = cc($S_1$, x) + cc(S, x-c), where $S_1$ is the set formed by removing c from S.

In the following implementation, we fix S to be the set consisting of 1, 5, 10 and 25.

```
typedef int4 = (int, int, int, int)

val theCoins = (1, 5, 10, 25): int4

fun coin_get
  (n: int): int =
  if n = 0 then theCoins.0
  else if n = 1 then theCoins.1
  else if n = 2 then theCoins.2
  else if n = 3 then theCoins.3
  else ~1 (* erroneous value *)
// end of [coin_get]

fun coin_change (sum: int) = let
  fun aux (sum: int, n: int): int =
    if sum > 0 then
     (if n >= 0 then aux (sum, n-1) + aux (sum-coin_get(n), n) else 0)
    else (if sum < 0 then 0 else 1)
  // end of [aux]
in
  aux (sum, 3)
end // end of [coin_change]
```

The auxiliary function aux defined in the body of the function coin_change corresponds to the cc function mentioned above. When applied to 1000, the function coin_change returns 142511.

Note that the entire code in this section plus some additional code for testing is available on-line[2].

## Tail-Call and Tail-Recursion

Suppose that a function foo makes a call in its body to a function bar, where foo and bar may be the same function. If the return value of the call to bar is also the return value of foo, then this call to bar is a tail-call. If foo and bar are the same, then this is a (recursive) self tail-call. For instance, there are two recursive calls in the body of the function f91 defined as follows:

```
fun f91 (n: int): int =
  if n >= 101 then n - 10 else f91 (f91 (n+11))
// end of [f91]
```

where the outer recursive call is a self tail-call while the inner one is not.

If each recursive call in the body of a function is a tail-call, then this function is a tail-recursive function. For instance, the following function sum_iter is tail-recursive:

```
fun sum_iter (n: int, res: int): int =
  if n > 0 then sum_iter (n-1, n+res) else res
// end of [sum_iter]
```

A tail-recursive function is often referred to as an iterative function.

In ATS, the single most important optimization is probably the one that turns a self tail-call into a local jump. This optimization effectively turns every tail-recursive function into the equivalent of a loop. Although ATS provides direct syntactic support for constructing for-loops and while-loops, the preferred approach to loop construction in ATS is in general through the use of tail-recursive functions.

## Example: Solving the Eight Queens Puzzle

The eight queens puzzle is the problem of positioning on a 8x8 chessboard 8 queen pieces so that none of them can capture any other pieces using the standard chess moves defined for a queen piece. I will present as follows a solution to this puzzle in ATS, reviewing some of the programming features that have been covered so far. In particular, please note that every recursive function implemented in this solution is tail-recursive.

First, let us introduce a name for the integer constant 8 as follows:

```
#define N 8
```

After this declaration, each occurrence of the name N is to be replaced with 8. For representing board configurations, we define a type int8 as follows:

```
typedef int8 = (
  int, int, int, int, int, int, int, int
) // end of [int8]
```

A value of the type int8 is a tuple of 8 integers where the first integer states the column position of the queen piece on the first row (row 0), and the second integer states the column position of the queen piece on the second row (row 1), and so on.

In order to print out a board configuration, we define the following functions:

```
fun print_dots (i: int): void =
  if i > 0 then (print ". "; print_dots (i-1)) else ()
// end of [print_dots]

fun print_row (i: int): void = begin
  print_dots (i); print "Q "; print_dots (N-i-1); print "\n";
end // end of [print_row]

fun print_board (bd: int8): void = begin
  print_row (bd.0); print_row (bd.1); print_row (bd.2); print_row (bd.3);
  print_row (bd.4); print_row (bd.5); print_row (bd.6); print_row (bd.7);
  print_newline ()
end // end of [print_board]
```

The function print_newline prints out a newline symbol and then flushes the buffer associated with the standard output. If the reader is unclear about what buffer flushing means, please feel free to ignore this aspect of print_newline.

As an example, if print_board is called on the board configuration represented by @(0, 1, 2, 3, 4, 5, 6, 7), then the following 8 lines are printed out:

```
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q
```

Given a board and the row number of a queen piece on the board, the following function board_get returns the column number of the piece:

```
fun board_get
  (bd: int8, i: int): int =
  if i = 0 then bd.0
  else if i = 1 then bd.1
  else if i = 2 then bd.2
  else if i = 3 then bd.3
  else if i = 4 then bd.4
  else if i = 5 then bd.5
  else if i = 6 then bd.6
  else if i = 7 then bd.7
  else ~1 // end of [if]
// end of [board_get]
```

Given a board, a row number i and a column number j, the following function board_set returns a new board that are the same as the original board except for j being the column number of the queen piece on row i:

```
fun board_set
  (bd: int8, i: int, j:int): int8 = let
  val (x0, x1, x2, x3, x4, x5, x6, x7) = bd
in
```

```
  if i = 0 then let
    val x0 = j in (x0, x1, x2, x3, x4, x5, x6, x7)
  end else if i = 1 then let
    val x1 = j in (x0, x1, x2, x3, x4, x5, x6, x7)
  end else if i = 2 then let
    val x2 = j in (x0, x1, x2, x3, x4, x5, x6, x7)
  end else if i = 3 then let
    val x3 = j in (x0, x1, x2, x3, x4, x5, x6, x7)
  end else if i = 4 then let
    val x4 = j in (x0, x1, x2, x3, x4, x5, x6, x7)
  end else if i = 5 then let
    val x5 = j in (x0, x1, x2, x3, x4, x5, x6, x7)
  end else if i = 6 then let
    val x6 = j in (x0, x1, x2, x3, x4, x5, x6, x7)
  end else if i = 7 then let
    val x7 = j in (x0, x1, x2, x3, x4, x5, x6, x7)
  end else bd // end of [if]
end // end of [board_set]
```

Clearly, the functions board_get and board_set are defined in a rather unwieldy fashion. This is entirely due to the use of tuples for representing board configurations. If we could use an array to represent a board configuration, then the implementation would be much simpler and cleaner. However, we have not yet covered arrays at this point.

We now implement two testing functions safety_test1 and safety_test2 as follows:

```
fun safety_test1 (
  i0: int, j0: int, i1: int, j1: int
) : bool =
(*
** [abs]: the absolute value function
*)
  j0 <> j1 andalso abs (i0 - i1) <> abs (j0 - j1)
// end of [safety_test1]

fun safety_test2 (
  i0: int, j0: int, bd: int8, i: int
) : bool =
  if i >= 0 then
    if safety_test1 (i0, j0, i, board_get (bd, i))
      then safety_test2 (i0, j0, bd, i-1) else false
    // end of [if]
  else true // end of [if]
// end of [safety_test2]
```

The functionalities of these two functions can be described as such:

- The function safety_test1 tests whether a queen piece on row i0 and column j0 can capture another one on row i and column j.
- The function safety_test2 tests whether a queen piece on row i0 and column j0 can capture any pieces on a given board with a row number less than or equal to i.

We are now ready to implement the following function search based on a standard depth-first search (DFS) algorithm:

```
fun search (
  bd: int8, i: int, j: int, nsol: int
) : int =
  if j < N then
    if safety_test2 (i, j, bd, i-1) then let
```

```
      val bd1 = board_set (bd, i, j)
    in
      if i+1 = N then let
        val () = print!
          ("This is solution no. ", nsol+1, ":\n\n")
        val () = print_board (bd1) in search (bd, i, j+1, nsol+1)
      end else search (bd1, i+1, 0, nsol)
    end else search (bd, i, j+1, nsol)
  else if i > 0 then
    search (bd, i-1, board_get (bd, i-1) + 1, nsol)
  else nsol // end of [if]
// end of [search]
```

The return value of search is the number of distinct solutions to the eight queens puzzle. The symbol print! in the body of search is a special identifier in ATS: It takes an indefinite number of arguments and then applies print to each of them. Here is the first solution printed out by a call to the function search:

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

There are 92 distinct solutions in total.

Note that the entire code in this section plus some additional code for testing is available on-line[3].

## Mutually Recursive Functions

A collection of functions are defined mutually recursively if each function can make calls in its body to any functions in this collection. Mutually recursive functions are commonly encountered in practice.

As an example, let P be a function on natural numbers defined as follows:

- P(0) = 1
- P(n+1) = 1 + the sum of the products of i and P(i) for i ranging from 1 to n

Let us introduce a function Q such that Q(n) is the sum of the products of i and P(i) for i ranging from 1 to n. Then the functions P and Q can be defined mutually recursively as follows:

- P(0) = 1
- P(n+1) = 1 + Q(n)
- Q(0) = 0
- Q(n+1) = Q(n) + (n+1) * P(n+1)

The following implementation of P and Q is a direct translation of their definitions into ATS:

```
fun P (n:int): int = if n > 0 then 1 + Q(n-1) else 1
and Q (n:int): int = if n > 0 then Q(n-1) + n * P(n) else 0
```

Note that the keyword and is used to combine function definitions.

## Mutual Tail-Recursion

Suppose that foo and bar are two mutually defined recursive functions. In the body of foo or bar, a tail-call to foo or bar is a mutually recursive tail-call. For instance, the following two functions isevn and isodd are mutually recursive:

```
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
```

The mutually recursive call to isodd in the body of isevn is a tail-call, and the mutually recursive call to isevn in the body of isodd is also a tail-call. If we want that these two tail-calls be compiled into local jumps, we should replace the keyword fun with the keyword fn* as follows:

```
fn* 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
```

What the ATS compiler does in this case is to combine these two functions into a single one so that each mutually recursive tail-call in their bodies can be turned into a self tail-call, which is then ready to be compiled into a local jump.

When writing code corresponding to embedded loops in an imperative programming language such as C or Java, we often need to make sure that mutually recursive tail-calls are compiled into local jumps. The following function print_multable is implemented to print out a standard multiplication table for nonzero digits:

```
fun print_multable () = let
//
  #define N 9
//
  fn* loop1 (i: int): void =
    if i <= N then loop2 (i, 1) else ()
  and loop2 (i: int, j: int): void =
    if j <= i then let
      val () = if j >= 2 then print " "
      val () = printf ("%dx%d=%2.2d", @(j, i, j*i))
    in
      loop2 (i, j+1)
    end else let
      val () = print_newline () in loop1 (i+1)
    end // end of [if]
//
in
  loop1 (1)
end // end of [print_multable]
```

The functions loop1 and loop2 are defined mutually recursively, and the mutually recursive calls in their bodies are all tail-calls. The keyword fn* indicates to the ATS compiler that the functions loop1 and loop2 should be combined so that these tail-calls can be compiled into local jumps. In a case where N is a large number (e.g., 1,000,000), calling loop1 may run the risk of stack overflow if these tail-calls are not compiled into local jumps.

When called, the function print_multable prints out the following multiplication table:

```
1x1=01
1x2=02 2x2=04
1x3=03 2x3=06 3x3=09
1x4=04 2x4=08 3x4=12 4x4=16
1x5=05 2x5=10 3x5=15 4x5=20 5x5=25
1x6=06 2x6=12 3x6=18 4x6=24 5x6=30 6x6=36
```

```
1x7=07 2x7=14 3x7=21 4x7=28 5x7=35 6x7=42 7x7=49
1x8=08 2x8=16 3x8=24 4x8=32 5x8=40 6x8=48 7x8=56 8x8=64
1x9=09 2x9=18 3x9=27 4x9=36 5x9=45 6x9=54 7x9=63 8x9=72 9x9=81
```

In summary, the very ability to turn mutually recursive tail-calls into local jumps makes it possible to implement embedded loops as mutually tail-recursive functions. This ability is indispensable for advocating the practice of replacing loops with recursive functions in ATS.

## Envless Functions and Closure Functions

I use *envless* as a shorthand for environmentless, which is not a legal word but I guess you have no problem figuring out what it means.

An envless function is represented by a pointer pointing to some place in a code segment where the object code for executing a call to this function is located. Every function in the programming language C is envless. A closure function is also represented by a pointer, but the pointer points to some place in a heap where a tuple is allocated (at run-time). Usually, the first component of this tuple is a pointer representing an envless function and the rest of the components represent some bindings. A tuple as such is often referred to as a closure, which can be thought of as an envless function paired with an environment. It is possible that the environment of a closure function is empty, but this does not equate a closure function with an envless function. Every function in functional languages such as ML and Haskell is a closure function.

In the following example, the function sum, which is assigned the type (int) -> int, sums up all the integers between 1 and a given natural number:

```
fun sum (n: int): int = let
  fun loop (
    i: int, res: int
  ) :<cloref1> int =
    if i <= n then loop (i+1, res+i) else res
  // end of [loop]
in
  loop (1(*i*), 0(*res*))
end // end of [sum]
```

The inner function loop is a closure function as is indicated by the special syntax :<cloref1>, and the type assigned to loop is denoted by (int, int) -<cloref1> int. Hence, envless functions and closure functions can be distinguished at the level of types.

If the syntax :<cloref1> is replaced with the colon symbol : alone, the code can still pass typechecking but its compilation eventually leads to an error indicating that loop cannot be compiled into a toplevel function C. The reason for this error is due to the body of loop containing a variable n that is neither at toplevel nor a part of the arguments of loop itself. It is straightforward to make loop an envless function by including n as an argument in addition to the original ones:

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

As a matter of fact, what happens during compilation is that the first implementation of sum and loop gets translated, more or less, into the second implementation, and there is simply no creation of closures (for representing closure functions) at run-time.

The need for creating closures often appears when the return value of a function call is a function itself. For instance, the following defined function addx returns another function when applied to a given integer x, and the returned function is a closure function, which always adds the integer x to its own argument:

```
fun addx (x: int): int -<cloref1> int = lam y => x + y

val plus1 = addx (1) // [plus1] is of the type int -<cloref1> int
val plus2 = addx (2) // [plus2] is of the type int -<cloref1> int
```

It should be clear that plus1(0) and plus2(0) return 1 and 2, respectively. The closure that is assigned the name plus1 consists of an envless function and an environment binding x to 1. The envless function can essentially be described by the pseudo syntax lam (env, y) => env.x + y, where env and env.x refer to an environment and the value to which x is bound in that environment. When evaluating plus1(0), we can first bind env and y to the environment in plus1 and the argument 0, respectively, and then start to evaluate the body of the envless function in plus1, which is env.x + y. Clearly, this evaluation yields the value 1 as is expected.

Closures are often passed as arguments to functions that are referred to as higher-order functions. It is also not uncommon for closures to be embedded in data structures.

## Higher-Order Functions

A higher-order function is a function that take another function as its argument. For instance, the following defined function rtfind is a higher-order one:

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

Given a function from integers to integers, rtfind searches for the first natural number that is a root of the function. For instance, calling rtfind on the polynomial function lam x => x * x - x + 110 returns 11. Note that rtfind loops forever if it is applied to a function that does not have a root.

Higher-order functions can greatly facilitate code reuse, and I now present a simple example to illustrate this point. The following defined functions sum and prod compute the sum and product of the integers ranging from 1 to a given natural number, respectively:

```
fun sum (n: int): int = if n > 0 then sum (n-1) + n else 0
fun prod (n: int): int = if n > 0 then prod (n-1) * n else 1
```

The similarity between the functions sum and prod is evident. We can define a higher-function ifold and then implement sum and prod based on ifold:

```
fun ifold
```

```
  (n: int, f: (int, int) -> int, ini: int): int =
  if n > 0 then f (ifold (n-1, f, ini), n) else ini
// end of [ifold]

fun sum (n: int): int = ifold (n, lam (res, x) => res + x, 0)
fun prod (n: int): int = ifold (n, lam (res, x) => res * x, 1)
```

If we ever want to compute the sum of the squares of the integers ranging from 1 to a given natural number, we can readily define a function based on ifold to do it:

```
fun sqrsum (n: int): int = ifold (n, lam (res, x) => res + x * x, 0)
```

As more features of ATS are introduced, higher-order functions will become even more effective in facilitating code reuse.

## Example: Binary Search

While binary search is often performed on an ordered array to check whether a given element is stored in that array, it can also be employed to compute the inverse of an increasing or decreasing function on integers. In the following code, the defined function bsearch_fun returns an integer i0 such that f(i) <= x0 holds for i ranging from lb to i, inclusively, and x0 < f(i) holds for i ranging from i+1 to ub, inclusively:

```
//
// The type [uint] is for unsigned integers
//
fun bsearch_fun (
  f: int -<cloref1> uint
, x0: uint, lb: int, ub: int
) : int =
  if lb <= ub then let
    val mid = lb + (ub - lb) / 2
  in
    if x0 < f (mid) then
      bsearch_fun (f, x0, lb, mid-1)
    else
      bsearch_fun (f, x0, mid+1, ub)
    // end of [if]
  end else ub // end of [if]
// end of [bsearch_fun]
```

As an example, the following function isqrt is defined based on bsearch_fun to compute the integer square root of a given natural number, that is, the largest integer whose square is less than or equal to the given natural number:

```
//
// Assuming that [uint] is of 32 bits
//
val ISQRT_MAX = (1 << 16) - 1 // = 65535
fun isqrt (x: uint): int =
  bsearch_fun (lam i => square ((uint_of_int)i), x, 0, ISQRT_MAX)
// end of [isqrt]
```

Note that the function uint_of_int is for casting a signed integer into an unsigned integer and the function square returns the square of its argument.

Please find the entire code in this section plus some additional code for testing online[4].

## Currying and Uncurrying

Currying, which is named after the logician Haskell Curry, means to turn a function taking multiple arguments simultaneously into a function of the same body (modulo corresponding recursive function calls being changed accordingly) that takes these arguments sequentially. Uncurrying means precisely the opposite of currying. In the following code, both of the defined functions acker1 and acker2 implement the Ackermann's function (which is famous for being recursive but not primitive recursive):

```
fun acker1 (m: int, n: int): int =
  if m > 0 then
    if n > 0 then acker1 (m-1, acker1 (m, n-1)) else acker1 (m-1, 1)
  else n+1

fun acker2 (m: int) (n: int): int =
  if m > 0 then
    if n > 0 then acker2 (m-1) (acker2 m (n-1)) else acker2 (m-1) 1
  else n+1
```

The function acker2 is a curried version of acker1 while the function acker1 in an uncurried version of acker2. Applying acker2 to an integer value generates a linear function closure, which I will explain elsewhere.

In functional languages such as ML and Haskell, a function of multiple arguments needs to be either curried or translated into a corresponding unary function of a single argument that itself is a tuple. In such languages, currying often leads to better performance at run-time and thus is preferred. In ATS, functions of multiple arguments are supported directly. Also, given a function of multiple arguments, a curried version of the function is likely to perform less efficiently at run-time than the function itself (due to the treatment of curried functions by the ATS compiler **atsopt**). Therefore, the need for currying in ATS is greatly diminished. Unless convincing reasons can be given, currying is in general not a recommended programming style in ATS.

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_FUNCTIONS/
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_FUNCTIONS/coin.da
3. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_FUNCTIONS/queens
4. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_FUNCTIONS/bsearch

# Chapter 4. Datatypes

A datatype is like a tagged union type. For each datatype, there are some constructors associated with it, and these constructors are needed for constructing values of the datatype. As an example, the following syntax declares a datatype named intopt:

```
datatype intopt =
  | intopt_none of () | intopt_some of (int)
// end of [intopt]
```

There are two constructors associated with intopt: intopt_none, which is nullary, and intopt_some, which is unary. For instance, intopt_none() and intopt_some(1) are two values of the type intopt. In order for accessing components in such values, a mechanism often referred to as pattern-matching is provided in ATS. I will demonstrate through examples that datatypes plus pattern matching can offer not only great convenience in programming but also clarity in code.

The code employed for illustration in this chapter plus some additional code for testing is available on-line[1].

## Patterns

Patterns in ATS can be defined inductively as follows:

- Certain constant values such booleans, chars, floating point numbers, integers and strings are patterns.
- The void value () is a pattern.
- The underscore symbol _ represents a special wildcard pattern.
- Variables are patterns.
- A tuple of patterns, either boxed or unboxed, is a pattern.
- A record of patterns, either boxed or unboxed, is a pattern.
- Given a constructor C, a pattern can be formed by applying C to a given list of patterns.
- Given a variable x and a pattern pat, (x as pat) is a referenced pattern, where as is a keyword.
- Some other forms of patterns will be introduced elsewhere.

Each variable can occur at most once in a given pattern, and this is referred as the linearity restriction on variables in patterns. For instance, (x, x) is not a legal pattern as the variable x appears twice in it. However, this restriction does not apply to the variable _, which represents the wildcard pattern.

## Pattern-Matching

Pattern matching means matching values against patterns. In the case where a value matches a pattern, a collection of bindings are generated between the variables in the pattern and certain components in the value. Pattern-matching is performed according to the following set of rules:

- A value that matches a constant pattern must be the same constant, and this matching generates no bindings.
- The void value () only matches the pattern (), and this matching generate no bindings.

- Any value can match the wildcard pattern, and this matching generates no bindings.
- Any value can match a variable pattern, and this matching generates a binding between the variable and the value.
- A tuple value matches a tuple pattern if they are of the same length and each value component in the former matches the corresponding pattern component in the latter, and this matching generates a collection of bindings that is the union of the bindings generated from matching the value components in the tuple value against the pattern components in the tuple pattern.
- A record value matches a record pattern if they have the same field names and each value component in the former matches the corresponding pattern component in the latter, and this matching generates a collection of bindings that is the union of the bindings generated from matching the value components in the record value against the pattern components in the record pattern.
- Given a pattern formed by applying a constructor C to some pattern arguments, a value matches this pattern if the value is formed by applying C to some value arguments matching the pattern arguments, and this matching generates a collection of bindings that is the union of the binding generated from matching the value arguments against the pattern arguments.
- Given a referenced pattern (x as pat), a value matches the pattern if it matches pat, and this matching generates a collection of bindings that extends the bindings generated from matching the value against pat with a binding from x to the value.

Suppose we have a tuple value (0, 1, 2, 3) and a tuple pattern (1, _, x, y). Then the value matches the pattern and this matching yields bindings from x and y to 2 and 3, respectively.

## Matching Clauses and Case-Expressions

Given a pattern pat and an expression exp, (pat => exp) is a matching clause. The pattern pat and the expression exp are referred to as the guard and the body of the matching clause.

Given an expression exp0 and a sequence of matching clauses clseq, a case-expression can be formed as such: (case exp0 of clseq). To evaluate the case-expression under a given environment ENV0, we first evaluate exp0 under ENV0 to a value. If this value does not match the guard of any clause in clseq, then the evaluation of the case-expression aborts. Otherwise, we choose the first clause in clseq such that the value matches its guard. Let ENV1 be the environment that extends ENV0 with the bindings generated from this matching, and we evaluate the body of the chosen clause under ENV1. The value of obtained from this evaluation is the value of the case-expression we started to evaluate.

## Enumerative Datatypes

The simplest form of datatypes is for enumerating a finite number of constants. For instance, the following concrete syntax introduces a datatype of the name wday:

```
datatype wday =
  | Monday of ()
  | Tuesday of ()
  | Wednesday of ()
  | Thursday of ()
  | Friday of ()
  | Saturday of ()
  | Sunday of ()
```

```
// end of [wday]
```

where the first bar symbol (|) is optional. There are 7 nullary constructors introduced in the datatype declaration: Monday through Sunday, which are for constructing values of the type wday. For instance, Monday() is a value of the type wday. Given a nullary constructor C, we can write C for C() as a value. For instance, we can write Monday for Monday(). However, one should *not* assume that Tuesday is something like Monday+1.

The following code implements a function that tests whether a given value of the type wday is a weekday or not:

```
fun isWeekday
  (x: wday): bool = case x of
  | Monday () => true // the bar (|) is optional for the first clause
  | Tuesday () => true
  | Wednesday () => true
  | Thursday () => true
  | Friday () => true
  | Saturday () => false
  | Sunday () => false
// end of [isWeekday]
```

Given a unary constructor C, C() is a pattern that can only match the value C(). Note that C() *cannot* be written as C when it is used as a pattern. If Monday () is written as Monday in the body of the function isWeekday, then an error message is to be reported during typechecking, indicating that all the clauses after the first one are redundant. This is simply due to Monday being treated as a variable pattern, which is matched by any value. A likely more sensible implementation of isWeekday is given as follows:

```
fun isWeekday
  (x: wday): bool = case x of
  | Saturday () => false | Sunday () => false | _ => true
// end of [isWeekday]
```

This implementation works because pattern-matching is done sequentially at runtime: If a value of the type wday does not match either of Saturday () and Sunday (), then it must match one of Monday () , Tuesday () , Wednesday () , Thursday () and Friday ().

## Recursive Datatypes

A recursive datatype is one such that its associated constructors may form values by applying to values of the datatype itself. For instance, the following declared datatype charlst is recursive:

```
datatype charlst =
  | charlst_nil of () | charlst_cons of (char, charlst)
```

When applied to a character and a value of the type charlst, the constructor charlst_cons forms a value of the type charlst. As an example, the following value represents a character list consisting of 'a', 'b' and 'c':

```
char_cons ('a', char_cons ('b', char_cons ('c', char_nil ())))
```

We can define a function charlst_length as follows to compute the length of a given character list:

```
fun charlst_length (cs: charlst): int =
  case cs of
  | charlst_cons (_, cs) => 1 + charlst_length (cs)
  | charlst_nil () => 0
// end of [charlst_length]
```

Note that this implementation is recursive but not tail-recursive. By relying on the commutativity and associativity of integer addition, we can give the following implementation of charlst_length that is tail-recursive:

```
fun charlst_length
  (cs: charlst): int = let
  fun loop (cs: charlst, n: int): int = case cs of
    | charlst_cons (_, cs) => loop (cs, n+1) | charlst_nil () => n
  // end of [loop]
in
  loop (cs, 0)
end // end of [charlst_length]
```

Only a tail-recursive function should be given a name that suggests it is a loop. This is a naming convention I follow closely in this book and elsewhere.

## Exhaustiveness of Pattern-Matching

Given a type T and a set of patterns, if for any given value of the type T there is always at least one pattern in the set such that the value matches the pattern, then pattern-matching values of the type T against the set of patterns is exhaustive. Given a case-expression of the form (case exp0 of clseq), where exp0 is assumed to be of some type T, if pattern-matching values of the type T against the guards of the matching clauses in clseq is exhaustive, then the case-expression is said to be pattern-matching-exhaustive.

The following code implements a function that finds the last character in a non-empty character list:

```
fun charlst_last
  (cs: charlst): char = case cs of
  | charlst_cons (c, charlst_nil ()) => c
  | charlst_cons (_, cs1) => charlst_last (cs1)
// end of [charlst_last]
```

The body of charlst_last is a case-expression, which is not pattern-matching-exhaustive: If cs is bound to the value charlst_nil(), that is, the empty character list, than none of the matching clauses in the case-expression can be chosen. When the code is typechecked by atsopt, a warning message is issued to indicate the case-expression being non-pattern-matching-exhaustive. If the programmer wants an error message instead, the keyword case should be replaced with case+. If the programmer wants to suppress the warning message, the keyword case should be replaced with case-. I myself mostly use case+ when coding in ATS.

The function charlst_last can also be implemented as follows:

```
fun charlst_last
  (cs: charlst): char = case cs of
  | charlst_cons (c, cs1) => (case+ cs1 of
      charlst_nil () => c | charlst_cons _ => charlst_last (cs1)
    ) // end of [char_cons]
// end of [charlst_last]
```

In this implementation, the outer case-expression is not pattern-matching-exhaustive while the inner one is. Note that the pattern charlst_cons _ is just a shorthand for charlst_cons (_, _). In general, a pattern of the from (C _), where C is a constructor, can be matched by any value that is constructed by applying C to some values. For instance, the pattern charlst_nil () can also be written as charlst_nil _.

Suppose we have a case-expression containing only one matching clause, that is, the case-expression is of the form (case exp0 of pat => exp). Then we can also write this case-expression as a let-expression: (let val pat = exp0 in exp end). For instance, we give another implementation of the function charlst_last as follows:

```
fun charlst_last
  (cs: charlst): char = let
  val charlst_cons (c, cs1) = cs in case+ cs1 of
  | charlst_nil () => c | charlst_cons _ => charlst_last (cs1)
end // end of [charlst_last]
```

When this implementation is typechecked by atsopt, a warning message is issued to indicate the val-declaration being non-pattern-matching-exhaustive. If the programmer wants an error message instead, the keyword val should be replaced with val+. If the programmer wants to suppress the warning message, the keyword val should be replaced with val-.

As values formed by the constructors charlst_nil and charlst_cons are assigned the same type charlst, it is impossible to rely on typechecking to prevent the function charlst_last from being applied to an empty character list. This is a serious limitation. With dependent types, which allow data to be described much more precisely, we can ensure at the level of types that a function finding the last element of a list can only be applied to a non-empty list.

## Example: Evaluating Integer Expressions

For representing integer expressions, we declare a datatype IEXP as follows:

```
datatype IEXP =
  | IEXPnum of int // numeral
  | IEXPneg of (IEXP) // negative
  | IEXPadd of (IEXP, IEXP) // addition
  | IEXPsub of (IEXP, IEXP) // subtraction
  | IEXPmul of (IEXP, IEXP) // multiplication
  | IEXPdiv of (IEXP, IEXP) // division
// end of [IEXP]
```

The meaning of the constructors associated with IEXP should be obvious. A value of the type IEXP is often referred to as an abstract syntax tree. For instance, the abstract syntax tree for the expression (~1+(2-3)*4) is the following one:

```
IEXPadd(IEXPneg(IEXPnum(1)), IEXPmul(IEXPsub(IEXPnum(2), IEXP(3)), IEXP(4)))
```

Translating an integer expression written in some string form into an abstract syntax tree is called parsing, which we will not do here. The following defined function eval_iexp takes the abstract syntax tree of an integer expression and return an integer that is the value of the expression:

```
fun eval_iexp (e0: IEXP): int = case+ e0 of
  | IEXPnum n => n
  | IEXPneg (e) => ~eval_iexp (e)
  | IEXPadd (e1, e2) => eval_iexp (e1) + eval_iexp (e2)
  | IEXPsub (e1, e2) => eval_iexp (e1) - eval_iexp (e2)
  | IEXPmul (e1, e2) => eval_iexp (e1) * eval_iexp (e2)
  | IEXPdiv (e1, e2) => eval_iexp (e1) / eval_iexp (e1)
```

```
// end of [eval_iexp]
```

Suppose we also allow the construct if-then-else to be use in forming integer expressions. For instance, we may write an integer expression like (if 1+2 <= 3*4 then 5+6 else 7-8). Note that the test (1+2 <= 3*4) is a boolean expression rather than an integer expression. This indicates that we also need to declare a datatype BEXP for representing boolean expressions. Furthermore, IEXP and BEXP should be defined mutually recursively, which is shown in the following code:

```
datatype IEXP =
  | IEXPcst of int // integer constants
  | IEXPneg of (IEXP) // negative
  | IEXPadd of (IEXP, IEXP) // addition
  | IEXPsub of (IEXP, IEXP) // subtraction
  | IEXPmul of (IEXP, IEXP) // multiplication
  | IEXPdiv of (IEXP, IEXP) // division
  | IEXPif of (BEXP(*test*), IEXP(*then*), IEXP(*else*))
// end of [IEXP]

and BEXP = // [and] for combining datatype declarations
  | BEXPcst of bool // boolean constants
  | BEXPneg of BEXP // negation
  | BEXPconj of (BEXP, BEXP) // conjunction
  | BEXPdisj of (BEXP, BEXP) // disjunction
  | BEXPeq of (IEXP, IEXP) // equal-to
  | BEXPneq of (IEXP, IEXP) // not-equal-to
  | BEXPlt of (IEXP, IEXP) // less-than
  | BEXPlte of (IEXP, IEXP) // less-than-equal-to
  | BEXPgt of (IEXP, IEXP) // greater-than
  | BEXPgte of (IEXP, IEXP) // greater-than-equal-to
// end of [BEXP]
```

Evidently, we also need to evaluate boolean expressions when evaluating integer expressions. The following two functions eval_iexp and eval_bexp for evaluating integer and boolean expressions, respectively, are defined mutually recursively as is expected:

```
fun eval_iexp (e0: IEXP): int = case+ e0 of
  | IEXPcst n => n
  | IEXPneg (e) => ~eval_iexp (e)
  | IEXPadd (e1, e2) => eval_iexp (e1) + eval_iexp (e2)
  | IEXPsub (e1, e2) => eval_iexp (e1) - eval_iexp (e2)
  | IEXPmul (e1, e2) => eval_iexp (e1) * eval_iexp (e2)
  | IEXPdiv (e1, e2) => eval_iexp (e1) / eval_iexp (e1)
  | IEXPif (e_test, e_then, e_else) => let
      val b = eval_bexp (e_test) in eval_iexp (if b then e_then else e_else)
    end // end of [IEXPif]
// end of [eval_iexp]

and eval_bexp (e0: BEXP): bool = case+ e0 of
  | BEXPcst b => b
  | BEXPneg (e) => ~eval_bexp (e)
  | BEXPconj (e1, e2) => if eval_bexp (e1) then eval_bexp (e2) else false
  | BEXPdisj (e1, e2) => if eval_bexp (e1) then true else eval_bexp (e2)
  | BEXPeq (e1, e2) => eval_iexp (e1) = eval_iexp (e2)
  | BEXPneq (e1, e2) => eval_iexp (e1) <> eval_iexp (e2)
  | BEXPlt (e1, e2) => eval_iexp (e1) < eval_iexp (e2)
  | BEXPlte (e1, e2) => eval_iexp (e1) <= eval_iexp (e2)
  | BEXPgt (e1, e2) => eval_iexp (e1) > eval_iexp (e2)
  | BEXPgte (e1, e2) => eval_iexp (e1) >= eval_iexp (e2)
// end of [eval_bexp]
```

The integer and boolean expressions used in this example are all constant expressions containing no variables. Therefore, there is no need for an environment to evaluate them. I will present a more advanced example elsewhere to demonstrate how an evaluator for a simple call-by-value functional programming language like the core of ATS can be implemented.

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATATYPES/

# Chapter 5. Parametric Polymorphism

Code sharing is of paramount importance in programming language design. In a typed programming language, we often encounter a situation where the same functionality is needed for values of different types. For instance, we need a function to compute the length of a list while the elements in the list may be characters, integers, strings, etc. Evidently, we want to avoid implementing such a function for each element type as it would probably be the worst form of code duplication otherwise. We want to implement a single function that can be applied to any list to compute the length of the list. The length-computing function parameterizes over the element type of a list it is applied to, and it behaves uniformly regardless what the element type is. This is a form of code sharing that is given the name: parametric polymorphism, which should be distinguished from other forms of polymorphism such as inheritance polymorphism in object-oriented programming.

The code employed for illustration in this chapter plus some additional code for testing is available on-line[1].

## Function Templates

A function template is a code template that implements a function. In the following code, two functions are defined to swap values:

```
typedef charint = (char, int)
typedef intchar = (int, char)
fun swap_char_int (xy: charint): intchar = (xy.1, xy.0)
fun swap_int_char (xy: intchar): charint = (xy.1, xy.0)
```

If types are ignored, the bodies of swap_char_int and swap_int_char are identical. In order to avoid this kind of code duplication, we can first implement a function template swap as follows and then implement swap_char_int and swap_int_char based on swap:

```
fun{a,b:t@ype} swap (xy: (a, b)): (b, a) = (xy.1, xy.0)
fun swap_char_int (xy: charint): intchar = swap<char,int> (xy)
fun swap_int_char (xy: intchar): charint = swap<int,char> (xy)
```

It should be noted that a function template is not a first-class value in ATS: There is no expression for representing a function template. The syntax {a,b:t@ype} following the keyword fun represents template parameters or arguments. The unusual symbol t@ype is a sort for static terms representing types of unspecified size, where the size of a type is the number of bytes needed for representing a value of the type. There is another sort type in ATS, which is for static terms representing types of size equal to one word exactly, that is, 4 bytes on a 32-bit machine or 8 bytes on a 64-bit machine. The syntax swap<char,int>, where no space is allowed between swap and < , stands for an instance of the function template swap in which the parameters a and b are replaced with char and int, respectively. The syntax swap<int,char> is interpreted similarly.

A different style of implementation of swap is given as follows:

```
fun{a:t@ype}{b:t@ype} swap2 (xy: (a, b)): (b, a) = (xy.1, xy.0)
```

where the template parameters are given sequentially (instead of simultaneously). The following code shows how swap2 can be instantiated to for instances:

```
fun swap_char_int (xy: charint): intchar = swap2<char><int> (xy)
fun swap_int_char (xy: intchar): charint = swap2<int><char> (xy)
```

Note that >< is a special symbol (of the name GTLT) and no space is allowed between > and <.

As another example, a higher-order function template for composing (closure) functions is given as follows:

```
typedef cfun (t1:t@ype, t2:t@ype) = t1 -<cloref1> t2

fun{a,b,c:t@ype} compose
  (f: cfun (a, b), g: cfun (b, c)):<cloref1> cfun (a, c) = lam x => g(f(x))
// end of [compose]

val plus1 = lam (x:int): int =<cloref1> x+1
val times2 = lam (x:int): int =<cloref1> x*2

val f_2x_1: cfun (int, int) = compose (times2, plus1)
val f_2x_2: cfun (int, int) = compose (plus1, times2)
```

It should be clear that the value f_2x_1 represents the function that multiplies its integer argument by 2 and then adds 1 to it. Similarly, the value f_2x_2 represents the function that adds 1 to its integer argument and then multiplies it by 2.

In ATS, function templates are typechecked but not compiled. Only instances of a function template can be compiled. Suppose we have a function template foo taking one type parameter and two instances foo<T1> and foo<T2> are used in a program for some types T1 and T2. In general, one function in C is generated for each instance of foo when the program is compiled. However, if T1 and T2 have the same name, then the two instances share one function in C. In particular, if both T1 and T2 are boxed types, which are always given the same name, only one function in C is generated for them.

Please note that I may simply use the name function to refer to a function template from now on if no confusion is expected.

## Polymorphic Functions

A polymorphic function is rather similar to a function template. However, the former is a first-class value in ATS while the latter is not. As an example, the following defined function swap_boxed is polymorphic:

```
fun swap_boxed {a,b:type} (xy: (a, b)): (b, a) = (xy.1, xy.0)
```

The type variables a and b are often referred as static arguments while xy is a dynamic argument. Here is some code that makes use of the polymorphic function swap_boxed:

```
val AB = ("A", "B")
val BA1 = swap_boxed {string,string} (AB) // [string] is boxed
val BA2 = swap_boxed (AB) // this is fine, too
```

If swap_boxed is called on a pair of the type (T1, T2) for some types T1 and T2, both T1 and T2 are required to be boxed. Otherwise, a type-error is reported. For instance, calling swap_boxed on (0, 1) yields a type-error as the type int is not boxed.

When calling a polymorphic function, we often omit passing static arguments explicitly and expect them to be synthesized by the compiler. However, there are also occasions, which are not uncommon, where static arguments need to be supplied explicitly as either they cannot be successfully synthesized or what is synthesized is not exactly what is expected.

It is also possible to pass static arguments sequentially as is shown in the following style of implementation of a polymorphic function:

```
fun swap2_boxed {a:type} {b:type} (xy: (a, b)): (b, a) = (xy.1, xy.0)

val AB = ("A", "B")
val BA1 = swap2_boxed {string} {string} (AB)
val BA2 = swap2_boxed (AB) // this is fine, too
val BA3 = swap2_boxed {..} {string} (AB) // 1st static argument to be synthesized
val BA4 = swap2_boxed {string} {..} (AB) // 2nd static argument to be synthesized
val BA5 = swap2_boxed {..} {..} (AB) // both arguments to be synthesized
val BA6 = swap2_boxed {...} (AB) // every static argument to be synthesized
```

The special syntax {..} indicates to the typechecker that the static argument (or arguments) involved in the current application should be synthesized while the special syntax {...} means that the rest of static arguments should all be synthesized.

I have seen two kinds of errors involving polymorphic functions that are extremely common in practice.

- The first kind is depicted in the following example:

  ```
  fun swap_boxed {a,b:t@ype} (xy: (a, b)): (b, a) = (xy.1, xy.0)
  ```

  Notice that the sort for type variables a and b is t@ype (instead of type). While this example can pass typechecking, its compilation results in an error that may seem mysterious to many programmers. The simple reason for this error is that the compiler cannot figure out the size of a and b when trying to generate code in C as the sort t@ype is for types of unspecified size.

- The second kind is depicted in the following example:

  ```
  fun{a,b:type} swap_boxed (xy: (a, b)): (b, a) = (xy.1, xy.0)
  ```

  Strictly speaking, there is really no error in this case. If defined as such, swap_boxed is a function template instead of a polymorphic function. However, such a function template is severely restricted as it cannot be instantiated with types that are not boxed. While this could be intended, it is likely not.

Given the potential confusion, why do we need both function templates and polymorphic functions? At this stage, it is certainly plausible that we program only with function templates and make no use of polymorphic functions. However, polymorphic functions can hardly be missed in the presence dependent types. There will actually be numerous occasions where we encounter polymorphic function templates, that is, templates for polymorphic functions.

## Polymorphic Datatypes

Code sharing also applies to datatype declarations. For instance, a commonly used polymorphic datatype list0 is declared as follows:

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

More precisely, list0 is a type constructor. Given a type T, we can form a type list0(T) for lists consisting of elements of the type T. For instance, list0(char) is for character lists, list0(int) for integer lists, list0(list0(int)) for lists whose elements themselves are integer lists, etc. To a great extent, the need for function templates or polymorphic functions largely stems from the availability of polymorphic datatypes. As an example, a function template list0_length is implemented as follows for computing the length of any given list:

```
fun{a:t@ype}
list0_length (xs: list0 a): int = case+ xs of
  | list0_cons (_, xs) => 1 + list0_length (xs) | list0_nil () => 0
// end of [list0_length]
```

When applying list0_length to a list xs, we can in general write list0_length(xs), expecting the typechecker to synthesize a proper type parameter for list0_length. We may also write list0_length<T>(xs) if the elements of xs are of the type T. The latter style, though a bit more verbose, is likely to yield more informative messages in case type-errors occur.

Another commonly used polymorphic datatype option0 is declared as follows:

```
datatype option0 (a:t@ype) =
  | option0_none (a) of () | option0_some (a) of a
// end of [option0]
```

A typical use of option0 is to perform some kind of error-handling. Suppose that we are to implement a function doing integer division and we want to make sure that the function returns even if it is called in a case where the divisor equals 0. This can be done as follows:

```
fun divopt (x: int, y: int): option0 (int) =
  if y = 0 then option0_none () else option0_some (x/y)
// end of [divopt]
```

By inspecting what divopt returns, we can tell whether integer division has been done normally or an error of divsion-by-zero has occurred. A realistic use of option0 is shown in the following implementation of list0_last:

```
fun{a:t@ype}
list0_last (xs: list0 a): option0 (a) = let
  fun loop (x: a, xs: list0 a): a = case+ xs of
    | list0_nil () => x | list0_cons (x, xs) => loop (x, xs)
  // end of [loop]
in
  case+ xs of
  | list0_nil () => option0_none ()
  | list0_cons (x, xs) => option0_some (loop (x, xs))
end // end of [list0_last]
```

When applied to a list, list0_last returns an optional value. If the value matches the pattern option0_none(), then the list is empty. Otherwise, the value is formed by applying option0_some to the last element in the list.

## Example: Function Templates on Lists

In functional programming, lists are ubiquitous. We implement as follows some commonly used function templates on lists. It should be noted that these templates are all available in a library of ATS, where they may be implemented in a significantly more efficient manner due to the use of certain programming features that have not been covered so far.

Please find the entire code in this section plus some additional code for testing online[2].

## Appending: list0_append

Given two lists xs and ys of the type list0(T) for some type T, list0_append(xs, ys) returns a list that is the concatenation of xs and ys:

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

Clearly, this implementation of list0_append is not tail-recursive.

## Reverse Appending: list0_reverse_append

Given two lists xs and ys of the type list0(T) for some type T, list0_reverse_append(xs, ys) returns a list that is the concatenation of the reverse of xs and ys:

```
fun{a:t@ype}
list0_reverse_append (
  xs: list0 a, ys: list0 a
) : list0 a = case+ xs of
  | list0_cons (x, xs) =>
      list0_reverse_append (xs, list0_cons (x, ys))
  | list0_nil () => ys
// end of [list0_reverse_append]
```

Clearly, this implementation of list0_reverse_append is tail-recursive.

## Reversing: list0_reverse

Given a list xs, list0_reverse(xs) returns the reverse of xs:

```
fun{a:t@ype}
list0_reverse
  (xs: list0 a): list0 a = list0_reverse_append (xs, list0_nil)
// end of [list0_reverse]
```

## Mapping: list0_map

Given a list xs of the type list0(T1) for some type T1 and a closure function f of the type T1 -<cloref1> T2 for some type T2, list0_map(xs) returns a list ys of the type list0(T2):

```
fun{a:t@ype}{b:t@ype}
list0_map (
  xs: list0 a, f: a -<cloref1> b
) : list0 b = case+ xs of
  | list0_cons (x, xs) => list0_cons (f x, list0_map (xs, f))
  | list0_nil () => list0_nil ()
// end of [list0_map]
```

The length of ys equals that of xs and each element y in ys equals f(x), where x is the corresponding element in xs. Clearly, this implementation of list0_map is not tail-recursive.

### Zipping: list0_zip

Given two lists xs and ys of the types list0(T1) and list0(T2) for some types T1 and T2, respectively, list0_zip(xs, ys) returns a list zs of the type list0 @(T1, T2):

```
fun{a,b:t@ype}
list0_zip (
  xs: list0 a, ys: list0 b
) : list0 @(a, b) = case+ (xs, ys) of
  | (list0_cons (x, xs),
     list0_cons (y, ys)) => list0_cons ((x, y), list0_zip (xs, ys))
  | (_, _) => list0_nil ()
// end of [list0_zip]
```

The length of zs is the minimum of the lengths of xs and ys and each element z in zs @(x, y), where x and y are the corresponding elements in xs and ys, respectively. Clearly, this implementation of list0_zip is not tail-recursive.

### Zipping with: list0_zipwith

Given two lists xs and ys of the types list0(T1) and list0(T2) for some types T1 and T2, respectively, and a closure function f of the type (T1, T2) -<cloref1> T3 for some type T3, list0_zipwith(xs, ys, f) returns a list zs of the type list0(T3):

```
fun{a,b:t@ype}{c:t@ype}
list0_zipwith (
  xs: list0 a
, ys: list0 b
, f: (a, b) -<cloref1> c
) : list0 c = case+ (xs, ys) of
  | (list0_cons (x, xs), list0_cons (y, ys)) =>
      list0_cons (f (x, y), list0_zipwith (xs, ys, f))
  | (_, _) => list0_nil ()
// end of [list0_zipwith]
```

The length of zs is the minimum of the lengths of xs and ys and each element z in zs is f(x, y), where x and y are the corresponding elements in xs and ys, respectively. Clearly, this implementation of list0_zipwith is not tail-recursive. Note that list0_zipwith behaves exactly like list0_zip if its third argument f is replaced with lam (x, y) => @(x, y). This function template is also named list0_map2 for the obvious reason.

## Example: Mergesort on Lists

Mergesort is simple sorting algorithm that is guaranteed to be log-linear. It is stable in the sense that the order of two equal elements always stay the same after sorting. I give as follows a typical functional style of implementation of mergesort on lists.

First, let us introduce abbreviations for the list constructors list0_nil and list0_cons:

```
#define nil list0_nil // writing [nil] for list0_nil
#define :: list0_cons // writing [::] for list0_cons
#define cons list0_cons // writing [cons] for list0_cons
```

Note that the operator :: is already given the infix status. For instance, the list consisting of the first 5 natural numbers can be constructed as follows:

```
cons (0, cons (1, 2 :: 3 :: 4 :: nil ()))
```

In practice, there is of course no point in mixing cons with ::.

We next implement a function template merge to merge two given ordered lists into a single ordered one:

```
typedef lte (a:t@ype) = (a, a) -> bool

fun{a:t@ype}
merge (
  xs: list0 a, ys: list0 a, lte: lte a
) : list0 a =
  case+ xs of
  | x :: xs1 => (
    case+ ys of
    | y :: ys1 =>
        if x \lte y then
          x :: merge (xs1, ys, lte))
        else
          y :: merge (xs, ys1, lte))
        // end of [if]
    | nil () => xs
    ) // end of [::]
  | nil () => ys
// end of [merge]
```

For instance, suppose that the two given lists are (1, 3, 4, 8) and (2, 5, 6, 7, 9), and the comparison function (the third argument of merge) is the standard less-than-or-equal-to function on integers. Then the list returned by merge is (1, 2, 3, 4, 5, 6, 7, 8, 9). The syntax \lte means that the particular occurrence of lte following the backslash symbol (\) is given the infix status, and thus the expression x \lte y means the same as lte(x, y).

The following function template mergesort implements the standard mergesort algorithm:

```
fun{a:t@ype}
mergesort
  (xs: list0 a, lte: lte a): list0 a = let
//
  fun msort (
    xs: list0 a, n: int, lte: lte a
  ) : list0 a =
    if n >= 2 then split (xs, n, lte, n/2, nil) else xs
  and split (
    xs: list0 a, n: int, lte: lte a, i: int, xsf: list0 a
  ) : list0 a =
    if i > 0 then let
      val- cons (x, xs) = xs
    in
      split (xs, n, lte, i-1, cons (x, xsf))
    end else let
      val xsf = list0_reverse<a> (xsf) // make sorting stable!
      val xsf = msort (xsf, n/2, lte) and xs = msort (xs, n-n/2, lte)
    in
      merge (xsf, xs, lte)
    end // end of [if]
//
  val n = list0_length<a> (xs)
//
```

```
in
  msort (xs, n, lte)
end // end of [mergesort]
```

Suppose we want to sort the list (8, 3, 4, 1, 2, 7, 6, 5, 9); we first divide it into two lists: (8, 3, 4, 1) and (2, 7, 6, 5, 9); by performing mergesort on each of them, we obtain two ordered lists: (1, 3, 4, 8) and (2, 5, 6, 7, 9); by merging these two ordered list, we obtain the ordered list (1, 2, 3, 4, 5, 6, 7, 8, 9), which is a permutation of the original one.

Note that the function template merge is not tail-recursive as the call to merge in its body is not a tail-call. This is a serious problem in practice: It is almost certain that a stack overflow is to occur if the above implementation of mergesort is employed to sort a list that is very long (e.g., containing 1,000,000 elements or more). I will later give a tail-recursive implementation of the merge function in ATS that makes use of linear types.

Please find the entire code in this section plus some additional code for testing online[3].

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_POLYMORPHISM
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_POLYMORPHISM/lis
3. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_POLYMORPHISM/m

# Chapter 6. Summary

I have given a presentation of the core of ATS in this part of book. While this core is largely similar to the core of the functional language ML (and various other call-by-value functional languages), there are also crucial differences. I list some of these differences as follows:

- ATS shares the same native data representation with the C programming language. This differs profoundly from ML and most other functional languages, which all rely on some forms of boxed data representation. In ATS, there are native/flat/unboxed tuples and records as well as boxed ones.
- ATS supports both envless functions and closure functions while ML and most other functional languages only support the latter. In ATS, envless functions and closure functions can be differentiated at the level of types.
- In ATS, there are both function templates and polymorphic functions. The need for the former primarily stems from the need for handling native data representation directly.
- ATS provides explicit support for mutually recursive tail-call optimization.

Except for making a few uses of printing functions, I have intentionally stayed away from programming features that can generate effects when presenting the core of ATS. I will cover such programming features when addressing in the next part of the book the issue of supporting practical programming in ATS.

# Chapter 7. Effectful Programming Features

Effectful programming features are those that can generate effects at run-time. But what is really an effect? The answer to this question is rather complex as it depends on the model of evaluation. I will gradually introduce various kinds of effects in this book. In sequential programming, that is, constructing programs to be evaluated sequentially (in contrast to concurrently), an expression is effectless if there exists a value such that the expression and the value cannot be distinguished as far as evaluation is concerned. For instance, the expression 1+2 is effectless as it cannot be distinguished from the value 3. An effectless expression is also said to be pure. On the other hand, an effectful expression is one that can be distinguished from any given values. For instance, the expression print("Hello") is effectful as its evaluation results in an observable behavior that distinguishes the expression from any values. In this case, print("Hello") is said to contain certain I/O effect. If the evaluation of an expression never terminates, then the expression is also effectul. For instance, let us define a function loop as follows:

```
fun loop (): void = loop ()
```

Then the expression loop() can be distinguished from any values in the following context:

```
let val _ = [] in print ("Terminated") end
```

If the hole [] in the context is replaced with loop(), then the evaluation of the resulting expression continues forever. If the hole [] is replaced with any value, then the evaluation leads to the string "Terminated" being printed out. The expression loop is said to contain the non-termination effect.

I will cover programming features related to exceptional control-flow, persistent memory storage and simple I/O in this chapter, which are all of common use in practical programming.

The code employed for illustration in this chapter plus some additional code for testing is available on-line[1].

## Exceptions

The exception mechanism provides an efficient means for reporting a special condition encountered during program evaluation. Often such a special condition indicates an error, but it is not uncommon to use the exception mechanism for addressing issues that are not related to errors.

The type exn is predefined in ATS. One may think of exn as an extensible datatype for which new constructors can always be declared. For instance, two exception constructors are declared as follows:

```
exception FatalError0 of ()
exception FatalError1 of (string)
```

The constructor FatalError0 is nullary while the constructor FatalError1 is unary. Exception values, that is, values of the type exn can be formed by applying exception constructors to proper arguments. For instance, FatalError0() and FatalError1("division-by-zero") are two exception values (or simply exceptions). In the following program, a function for integer division is implemented:

```
exception DivisionByZero of ()
fun divexn (x: int, y: int): int =
  if y <> 0 then then x / y else $raise DivisionByZero()
```

```
// end of [divexn]
```

When the function call divexn(1, 0) is evaluated, the exception DivisionByZero() is raised. The keyword $raise in ATS is solely for raising exceptions.

A raise-expression is of the form ($raise exp) for some expression exp. Clearly, if the evaluation of exp returns a value, then the evaluation of ($raise exp) leads to a raised exception. Therefore, the evaluation of a raise-expression can never return a value, and this justifies that a raise-expression can be given any type.

A raised exception can be captured. If it is not captured, the raised exception aborts the program evaluation that issued it in the first place. In ATS, a try-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 matching clauses. 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 a try-expression, the with-part is often referred to as an exception-handler.

Let us now see an example that involves raising and capturing an exception. In the following program, three functions are defined to compute the product of the integers in a given list:

```
fun listprod1
  (xs: list0 (int)): int = case+ xs of
  | list0_cons (x, xs) => x * listprod1 (xs) | list0_nil () => 1
// end of [listprod1]

fun listprod2
  (xs: list0 (int)): int = case+ xs of
  | list0_cons (x, xs) => if x = 0 then 0 else x * listprod2 (xs)
  | list0_nil () => 1
// end of [listprod2]

fun listprod3
  (xs: list0 (int)): int = let
  exception ZERO of ()
  fun aux (xs: list0 (int)): int =
    case+ xs of
    | list0_cons (x, xs) =>
        if x = 0 then $raise ZERO() else x * aux (xs)
    | list0_nil () => 1
  // end of [aux]
in
  try aux (xs) with ~ZERO () => 0
end // end of [listprod3]
```

While these functions can all be defined tail-recursively, they are not so as to make a point that should be clear shortly. Undoubtedly, we all know the following simple fact:

- If the integer 0 occurs in a given list, then the product of the integers in the list is 0 regardless what other integers are.

The function listprod1 is defined in a standard manner, and it does not make any use of the fact. The function listprod2 is defined in a manner that makes only partial use of the fact. To see the reason, let us evaluate a call to listprod2 on [1, 2, 3, 0, 4, 5, 6], which denotes a list consisting of the 7 mentioned integers. The evaluation of this call eventually leads to the evaluation of 1*(2*(3*(listprod([0,4,5,6])))), which then leads to 1*(2*(3*0)), and then to 1*(2*0), and then to 1*0, and finally to 0. However, what we really want is for the evaluation to return 0 immediately once the integer 0 is encountered in the list, and this is accomplished by the function listprod3. When evaluating a call to listprod3 on [1, 2, 3, 0, 4, 5, 6], we eventually reach the evaluation of the following expression:

```
try 1*(2*(3*(aux([0,4,5,6])))) with ~ZERO() => 0
```

Evaluating aux([0,4,5,6]) leads to the exception ZERO() being raised, and the raised exception is caught and 0 is returned as the value of the call to listprod3. Note that the pattern guard of the matching clause following the keyword with is ~ZERO(). I will explain the need for the tilde symbol ~ elsewhere. For now, it suffices to say that exn is a linear type and each exception value is a linear value, which must be consumed or re-raised. The tilde symbol ~ indicates that the value matching the pattern following ~ is consumed (and the memory for holding the value is freed).

Exceptions are not a programming feature that is easy to master, and misusing exceptions is abundant in practice. So please be patient when learning the feature and be cautious when using it!

## Example: Testing for Braun Trees

Braun trees are special binary trees that can be defined inductively as follows:

- If a binary tree is empty, then it is a Braun tree.
- If both children of a binary tree are Braun trees and the size of the left child minus the size of the right child equals 0 or 1, then the binary tree is a Braun tree.

Given a natural number n, there is exactly one Braun tree whose size is n. It is straightforward to prove that Braun trees are balanced.

A polymorphic datatype is declared as follows for representing binary trees:

```
datatype tree (a:t@ype) =
  | tree_nil (a) of ()
  | tree_cons (a) of (a, tree(a)(*left*), tree(a)(*right*))
// end of [tree]
```

The following defined function brauntest0 tests whether a given binary tree is a Braun tree:

```
fun{a:t@ype}
size (t: tree a): int = case+ t of
  | tree_nil () => 0
  | tree_cons (_, tl, tr) => 1 + size(tl) + size(tr)
// end of [size]

fun{a:t@ype}
brauntest0 (t: tree a): bool = case+ t of
  | tree_nil () => true
  | tree_cons (_, tl, tr) => let
      val cond1 = brauntest0(tl) andalso brauntest0(tr)
    in
      if cond1 then let
        val df = size(tl) - size(tr) in (df = 0) orelse (df = 1)
      end else false
    end // end of [tree_cons]
// end of [brauntest0]
```

The implementation of brauntest0 follows the definition of Braun trees closely. If applied to binary trees of size n, the time-complexity of the function size is O(n) and the time-complexity of the function brauntest0 is O(n log(n)).

In the following program, the defined function brauntest1 also tests whether a given binary tree is a Braun tree:

```
fun{a:t@ype}
brauntest1 (t: tree a): bool = let
  exception Negative of ()
  fun aux (t: tree a): int = case+ t of
    | tree_nil () => 0
    | tree_cons (_, tl, tr) => let
        val szl = aux (tl) and szr = aux (tr)
        val df = szl - szr
      in
        if df = 0 orelse df = 1 then 1+szl+szr else $raise Negative()
      end // end of [tree_cons]
   // end of [aux]
in
  try let
    val _ = aux (t)
  in
    true // [t] is a Braun tree
  end with
    ~Negative() => false // [t] is not a Braun tree
  // end of [try]
end // end of [brauntest1]
```

Clearly, a binary tree cannot be a Braun tree if one of its subtrees, proper or improper, is not a Braun tree. The auxiliary function aux is defined to return the size of a binary tree if the tree is a Braun tree or raise an exception otherwise. When the evaluation of the try-expression in the body of brauntest1 starts, the call to aux on a binary tree t is first evaluated. If the evaluation of this call returns, then t is a Braun tree and the boolean value true is returned as the value of the try-expression. Otherwise, the exception Negative() is raised and then caught, and the boolean value false is returned as the value of the try-expression. The time complexity of brauntest1 is the same as that of aux, which is O(n).

The use of the exception mechanism in the implementation brauntest1 is a convincing one because the range between the point where an exception is raised and the point where the raised exception is captured can span many function calls. If this range is short (e.g., spanning only one function call) in a case, then the programmer should probably investigate whether it is a sensible use of the exception mechanism.

Please find the entire code in this section plus some additional code for testing on-line[2].

## References

A reference is just an array containing one element. Given a type T, a reference for storing a value of the type T is given the type ref(T). The following program makes use of all the essential functionalities on references:

```
val intr = ref<int> (0) // create a ref and init. it with 0
val () = !intr := !intr + 1 // increase the integer at [intr] by 1
```

The first line creates a reference for storing an integer and initializes it with the value 0 and then names it intr. Note that the creation of a reference cannot be separated from its initialization. The second line updates the reference intr with its current value plus 1. In general, given a reference r of type ref(T) for some T, the expression !r means to fetch the value stored at r, which is of the type T. However, !r can also be used as a left-value. For instance, the assignment (!r := exp) means to evaluate exp into a value and then store the value into r. Therefore, the value stored in intr is 1 after the second line in the above program is evaluated.

Various functions and function templates on references are declared in the file `prelude/SATS/reference.sats`[3], which is automatically loaded by **atsopt**. In particular, it is also possible to read from and write to a reference by using the function templates ref_get_elt and ref_set_elt of the following interfaces, respectively:

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

If you implement a program that makes use of references, please do not forget to include the following line somewhere in the program:

```
staload _(*anon*) = "prelude/DATS/reference.dats"
```

This line allows the ATS compiler **atsopt** to gain access to the defintion of various functions and function templates on references.

References are often misused in practice, especially, by beginners in functional programming who had some previous exposure to imperative programming languages such C and Java. Such programmers often think that they can just "translate" their programs in C or Java into functional programs. For example, the following defined function sum is such an example, which sums up all the integers between 1 and a given integer, inclusively:

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

This is a correct but poor implementation, and its style, though not the worst of its kind, is deplorable. As references are allocated in heap, reading from or writing to a reference can be much more time-consuming than reading from or writing to a register. So, this implementation of sum is unlikely to be time-efficient. Every call to sum creates two references in heap and leaves them there when it returns, and the memory allocated for such references can only be reclaimed by a garbage collector (GC). So, this implementation of sum is not memory-efficient. More importantly, a program making heavy use of references is often difficult to reason about.

I consider references a dangerous feature in functional programming. If you want to run your program without GC, please do not create references in the body of a function (besides many other restrictions). If you find that you are in need of references to "translate" imperative programs into functional ones, then it is most likely that you are lost and you have not learned well to program in a functional style yet.

## Example: Implementing Counters

We implement a counter like an object in object-oriented programming (OOP). The type counter for counters is defined as follows:

```
typedef counter = '{
```

```
  get= () -<cloref1> int
, inc= () -<cloref1> void
, reset= () -<cloref1> void
} // end of [counter]
```

The three fields of counter are closure functions that correspond to methods associated with an object: getting the count of the counter, increasing the count of the counter by 1 and resetting the count of the counter to 0. The following defined function newCounter is for creating a counter object (represented as a boxed record of closure functions):

```
fun newCounter
  (): counter = let
  val count = ref<int> (0)
in '{
  get= lam () => !count
, inc= lam () => !count := !count + 1
, reset= lam () => !count := 0
} end // end of [newCounter]
```

The state of each created counter object is stored in a reference, which can only be accessed by the three closure functions in the record that represents the object. This is often referred to as state encapsulation in OOP.

## Arrays

I mentioned earlier that a reference is just an array of size 1. I would now like to state that an array of size n is just n references allocated consecutively. These references can also be called cells, and they are numbered from 0 until n-1, inclusively.

Given an array of size n, an integer is a valid index for this array if it is a natural number strictly less than n. Otherwise, the integer is out of the bounds of the array. For an array named A, the expression A[i] means to fetch the content of the cell in A that is numbered i if i is a valid index for A. The expression A[i] can also be used as a left value. For instance, the assignment (A[i] := exp) means to evaluate exp to a value and then store the value into the cell in A that is numbered i if i is a valid index.

What happens if the index i in A[i] is invalid, that is, it is out of the bounds of the array A? In this case, A[i] is referred to as out-of-bounds array subscription and evaluating A[i] leads to a raised exception where the exception is ArraySubscriptException(). One simple and reliable way to tell whether an integer is a valid index for a given array is to compare it with the size of the array at run-time. Given a type T, the type array0(T) is for an array paired with its size in which elements of the type T are stored. I will loosely refer to values of the type array0\ (T) as arrays from now on. In case there is a clear need to avoid potential confusion, I may also refer to them as array0-values.

Various functions and function templates on array0-values are declared in the file `prelude/SATS/array0.sats`[4], which is automatically loaded by **atsopt**. For instance, three function templates and one polymorphic function on arrays are depicted by the following interfaces:

```
fun{a:t@ype} // a template
array0_make_elt (asz: size_t, x: a): array0 a // array creation

// a polymorphic function
fun array0_size {a:t@ype} (A: array0 a): size_t // size of an array

fun{a:t@ype} // a template
array0_get_elt_at (A: array0 a, i: size_t): a // A[i]
```

```
fun{a:t@ype} // a template
array0_set_elt_at (A: array0 a, i: size_t, x: a): void // A[i] := x
```

If you implement a program that makes use of array0-values, please do not forget to include the following two lines somewhere in the program:

```
staload _(*anon*) = "prelude/DATS/array.dats"
staload _(*anon*) = "prelude/DATS/array0.dats"
```

These lines allow the ATS compiler **atsopt** to gain access to the definition of various functions and function templates on arrays and array0-values. The topic on programming with arrays that carry no size information will be covered after dependent types are introduced.

Like in C, there are many types of integer values in ATS. The type size_t is essentially for unsigned long integers. The functions for converting between the type int and the type size_t are int_of_size and size_of_int. Given a type T and two values asz and init of the types size_t and T, respectively, array0_make_elt<T> (asz, init) returns an array of the type array0 (T) such that the size of the array is asz and each cell in the array is initialized with the value init. Given an array A of the type array0 (T) for some T, array0_size(A) returns the size of A, which is of the type size_t.

In the following program, the function template insertion_sort implements the standard insertion sort on arrays:

```
fun{a:t@ype}
insertion_sort (
  A: array0 (a), cmp: (a, a) -> int
) : void = let
  val asz = array0_size (A)
  val n = int_of_size (asz)
  fun ins (x: a, i: int):<cloref1> void =
    if i >= 0 then
      if cmp (x, A[i]) < 0
        then (A[i+1] := A[i]; ins (x, i-1)) else A[i+1] := x
      // end of [if]
    else A[0] := x // end of [if]
  // end of [ins]
  fun loop (i: int):<cloref1> void =
    if i < n then (ins (A[i], i-1); loop (i+1)) else ()
  // end of [loop]
in
  loop (1)
end // end of [insertion_sort]
```

The comparison function cmp should return 1 if its first argument is greater than the second one, and -1 if its first argument is less than the second one, and 0 if they are equal.

Note that the entire code in this section plus some additional code for testing is available on-line[5].

## Example: Ordering Permutations

Given a natural number n, we want to print out all the permutations consisting of integers ranging from 1 to n, inclusively. In addition, we want to print them out according to the lexicographic ordering on integer sequences. For instance, we want the following output to be generated when n is 3:

```
1, 2, 3
1, 3, 2
```

```
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1
```

Let us first define a function as follows for printing out an array of integers:

```
fun print_intarray
  (A: array0 (int)): void = let
  val asz = array0_size (A) // get the size of the array
  val asz = int_of_size (asz) // turn [asz] to be of the type [int]
//
// The integers are to be separated by the string [sep]
//
  fun loop (i: int, sep: string):<cloref1> void =
    if i < asz then
      (if i > 0 then print sep; print A[i]; loop (i+1, sep))
    // end of [if]
in
  loop (0, ", ")
end // end of [print_intarray]
```

We next implement two functions lrotate and rrotate for rearranging the elements in a given integer array:

```
fun lrotate (
  A: array0 int, i: int, j: int
) : void = let
  fun lshift (
    A: array0 int, i: int, j: int
  ) : void =
  if i < j then (A[i] := A[i+1]; lshift (A, i+1, j))
in
  if i < j then let
    val tmp = A[i] in lshift (A, i, j); A[j] := tmp
  end // end of [if]
end // end of [lrotate]

fun rrotate (
  A: array0 int, i: int, j: int
) : void = let
  fun rshift (
    A: array0 int, i: int, j: int
  ) : void =
  if i < j then (A[j] := A[j-1]; rshift (A, i, j-1))
in
  if i < j then let
    val tmp = A[j] in rshift (A, i, j); A[i] := tmp
  end // end of [if]
end // end of [rrotate]
```

When applied to an array and two valid indexes i and j for the array such that i is less than or equal to j, lrotate moves simultaneously the content of cell i into cell j and the content of cell k to cell k-1 for k ranging from i+1 to j, inclusively. The function rrotate is similar to lrotate but shuffles elements in the opposite direction.

Given a natural number n, the following defined function permute prints out all the permutations consisting of integers ranging from 1 to n, inclusively while arranging the output according to the lexicographic ordering on integer sequences.

```
fun permute
```

```
  (n: int): void = let
//
  val asz = size_of_int (n)
  val A = array0_make_elt<int> (asz, 0)
//
// Initializing A with integers from 1 to n, inclusively
//
  val () = init (0) where {
    fun init (i: int):<cloref1> void =
      if i < n then (A[i] := i+1; init (i+1))
  } // end of [val]
//
  fun aux (
      i: int
    ) :<cloref1> void =
    if i <= n then aux2 (i, i) else (
      print_intarray (A); print_newline ()
    ) // end of [if]
  and aux2 (
      i: int, j: int
    ) :<cloref1> void =
    if j <= n then let
      val () = (
        rrotate (A, i-1, j-1); aux (i+1); lrotate (A, i-1, j-1)
      ) // end of [val]
    in
      aux2 (i, j+1)
    end // end of [if]
//
in
  aux (1)
end // end of [permute]
```

Note that where is a keyword, and the expression (exp where { decseq }) for some expression exp and declaration sequence decseq is equivalent to the let-expression of the form (let decseq in exp end). To understand the behavior of the function aux, let us evaluate aux(1) while assuming n is 4 and the 4 elements of the array A are 1, 2, 3, and 4. It should be fairly straightforward to see that this evaluation leads to the evaluation of aux(2) for 4 times: the array A contains (1, 2, 3, 4) for the first time, and (2, 1, 3, 4) for the second time, and (3, 1, 2, 4) for the third time, and (4, 1, 2, 3) for the fourth time. With some inductive reasoning, it should not be difficult to see that evaluating aux(1) indeed leads to all the permutations being output according to the lexicographic ordering on integer sequences.

Please find the entire code in this section plus some additional code for testing online[6].

## Matrices

A matrix in ATS is just a two-dimensional array but it is represented by a one-dimensional array and the representation is of the row-major style (in contrast to the column-major style). Given a type T, the type matrix0(T) is for a matrix combined with its number of rows and number of columns such that each element stored in the matrix is of the type T. I will loosely refer to values of the type matrix0(T) as matrices from now on. If there is a clear need to avoid potential confusion, I may also refer to them as matrix0-values.

Given a matrix M of dimension m by n, the expression M[i,j] means to fetch the content of the cell in M that is indexed by (i, j), where i and j are natural numbers strictly less than m and n, respectively. The expression M[i,j] can also be used as a left value. For instance, the assignment (M[i,j] := exp) means to evaluate exp to a value and then store the value into the cell in M that is indexed by (i, j).

Various functions and function templates on matrix0-values are declared in the file `prelude/SATS/matrix0.sats`[7], which is automatically loaded by **atsopt**. For instance, three function templates and two polymorphic functions on matrices are depicted by the following interfaces:

```
fun{a:t@ype} // template
matrix0_make_elt
  (row: size_t, col: size_t, x: a): matrix0 (a)

fun matrix0_row {a:t@ype} (M: matrix0 a): size_t // polyfun
fun matrix0_col {a:t@ype} (M: matrix0 a): size_t // polyfun

fun{a:t@ype}
matrix0_get_elt_at // template
  (M: matrix0 a, i: size_t, j: size_t): a // M[i,j]

fun{a:t@ype}
matrix0_set_elt_at // template
  (M: matrix0 a, i: size_t, j: size_t, x: a): void // M[i,j] := x
```

Given a type T and three values row, col and init of the types size_t, size_t and T, respectively, matrix0_make_elt<T> (row, col, init) returns a matrix of the type matrix0(T) such that the dimension of the matrix is row by col and each cell in the matrix is initialized with the value init. Given a matrix M of the type matrix0(T) for some T, matrix0_row(M) and matrix0_col(M) return the number of rows and the number of columns of M, respectively, which are both of the type size_t. Also, matrix access and update can be done by calling the function templates matrix0_get_elt_at and matrix0_set_elt_at, respectively.

As an example, the following defined function matrix0_transpose turns a given matrix into its transpose:

```
fun{a:t@ype}
matrix0_transpose
  (M: matrix0 a): void = let
//
  val nrow = matrix0_row (M)
//
  fn* loop1
    (i: size_t):<cloref1> void =
    if i < nrow then loop2 (i, 0) else ()
  and loop2
    (i: size_t, j: size_t):<cloref1> void =
    if j < i then let
      val tmp = M[i,j]
    in
      M[i,j] := M[j,i]; M[j,i] := tmp; loop2 (i, j+1)
    end else
      loop1 (i+1)
    // end of [if]
//
in
  loop1 (0)
end // end of [matrix0_transpose]
```

The matrix M is assumed to be a square, that is, its number of rows equals its number of columns. Note that the two functions loop1 and loop2 are defined mutually tail-recursively, and the keyword fn* indicates the need to combine the bodies of loop1 and loop2 so that mutual recursive tail-calls in these bodies can be compiled into local jumps.

## Example: Estimating the Constant Pi

I present as follows a Monte Carlo approach to estimating the constant Pi, the ratio of the circumference of a circle over its diameter.

Assume that we have a square of the dimension N by N, where N is a relatively large natural number (e.g., 1000), and a disk of radius 1 that is contained in the square. Let N2 stand for N*N, that is, the square of N. If we randomly choose a point inside the square, then the probability for the point to hit the disk is Pi/N2.

The experiment we use to estimate the constant Pi can be described as follows. Given a natural number K, let us randomly choose K points inside the square in K rounds. In each round, we choose exactly one point. If the point chosen in round k hits on the disk centered at a previously chosen point, then we record one hit. Clearly, the expected number of hits recorded in round k is (k-1)*Pi/N2 as k-1 points have already being chosen in the previous rounds. Therefore, in K rounds, the expected total number of hits is (K*(K-1)/2)*Pi/N2. If K is fixed to be N2, then the expected total number of hits is (N2-1)*Pi/2. It can be proven that the total number of hits divided by N2 converges to Pi/2 (with probability 1) as N approaches infinity.

If we implement the above experiment directly based on the given description, the time-complexity of the implementation is evidently proportional to N2*N2 as the time spent in round k is proportional to k, where k ranges from 1 to N2. An implementation as such is simply impractical for handling N around the order 1000 (and thus N2 around the order of 1,000,000). To address the issue, we can impose a grid on the square, dividing it into N2 unit squares (of the dimension 1 by 1). We then associate with each unit square a list of chosen points that are inside it. In each round, we first choose a point randomly inside the original square; we next locate the unit square that contains this point; we then only search the lists associated with the unit square or any of its neighbors to count the number of hits generated by the point chosen in this round as this point cannot hit any disks centered at points that are not on these lists. As each unit square can have at most 8 neighbors and the average length of the list associated with each square is less than 1 during the experiment, the time spent during each round is O(1), that is, bounded by a constant. Hence, the time taken by the entire experiment is O(N2).

An implementation that precisely matches the above description plus some testing code is available on-line[8].

## Simple Input and Output

Handling I/O in ATS properly requires the availability of both dependent types and linear types, which I will cover elsewhere. In this section, I only present a means for allowing the programmer to access certain very basic I/O functionalities.

A file handle essentially associates a stream (of bytes) with a file identifier (represented as an integer). In ATS, the type for file handles is FILEref. There are three standard file handles, which are listed as follows:

- stdin_ref: standard input
- stdout_ref: standard output
- stderr_ref: standard error output

Various functions on file handles are declared in the file `prelude/SATS/filebas.sats`[9], which is automatically loaded by **atsopt**. For instance, the functions for opening and closing file handles have the following interfaces:

```
fun open_file_exn
  (path: string, mode: file_mode): FILEref
// end of [open_file_exn]
```

```
fun close_file_exn (fil: FILEref): void
```

Note that these two functions abort immediately whenever an error occurs. The type file_mode is for values representing file modes, which are listed as follows:

- file_mode_r: opening a file for reading and positioning the associated stream at the beginning of the file.
- file_mode_rr: opening a file for both reading and and writing and positioning the associated stream at the beginning of the file.
- file_mode_w: truncating a given file to zero length or creating a new one for writing and positioning the associated stream at the beginning of the file.
- file_mode_ww: truncating a given file to zero length or creating a new one for both reading and writing and positioning the associated stream at the beginning of the file.
- file_mode_a: opening a file for writing and positioning the associated stream at the end of the file.
- file_mode_aa: opening a file for both reading and writing and positioning the associated stream at the beginning of the file for reading and at the end for writing.

As an example, the following short program opens a file handle, outputs the string "Hello, world!" plus a newline into the stream associated with the file handle and then closes the file handle:

```
//
// The following line is needed for the compiler
// to gain access to some library I/O functions:
//
staload _(*anon*) = "libc/SATS/stdio.sats"

implement main () = () where {
  val out = open_file_exn ("hello.txt", file_mode_w)
  val () = fprint_string (out, "Hello, world!\n")
  val () = close_file_exn (out)
} // end of [main]
```

After executing the program, we obtain a file of the name "hello.txt" in the current working directory containing the expected content. There are various fprint-functions in ATS for printing out data into the stream in a given file handle. Often the programmer can simply use the name fprint to refer to these functions due to the support for overloading in ATS.

Another two common I/O functions are given the following interfaces:

```
fun input_line (fil: FILEref): Stropt
fun output_line (fil: FILEref, line: string): void
```

The function input_line reads a line from the stream in a given file handle, and it returns a value of the type Stropt. For the moment, let us equate Stropt with the type option0(string). If the return value is constructed by option0_none, then the stream has reached the end when input_line is called. Otherwise, the return value is of the form option0_some(str), where str represents the line read from the stream minus the ending newline symbol. The function output_line writes its second argument, which is a string, and a newline symbol into the stream associated with its first argument, which is a file handle. As an example, the following short program echos each line received from the standard input onto the standard output:

```
staload _(*anon*) = "libc/SATS/stdio.sats"

implement
main () = loop () where {
  fun loop (): void = let
    val line = input_line (stdin_ref)
  in
    if stropt_is_some (line) then let
      val () = output_line (stdout_ref, stropt_unsome (line))
    in
      loop ()
    end else
      () // loop exits as the end-of-file is reached
    // end of [if]
  end (* end of [loop] *)
} // end of [main]
```

The function stropt_is_some essentially checks whether a given value is constructed by option0_some (if we equate Stropt with option0(string)) and the function stropt_unsome extracts out the argument of option0_some in a value constructed by option0_some. Often, typing the CTRL-D character can terminate the above program for echoing each line of input.

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_EFFECTFUL/
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_EFFECTFUL/braunte
3. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/reference.sats
4. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/array0.sats
5. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_EFFECTFUL/insort.da
6. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_EFFECTFUL/permord
7. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/matrix0.sats
8. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_EFFECTFUL/monteca
9. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/filebas.sats

# Chapter 8. Convenience in Programming

There are a variety of programming features in ATS that are primarily designed to provide convienience in programming. In this chapter, I will cover macros, compile-time directives and several forms of overloading.

## Macro Definitions

There are two kinds of macros in ATS: a C-like kind and a LISP-like kind respectively.

### C-like Macros

The following two declarations bind the identifiers N1 and N2 to the abstract syntax trees (not strings) representing 1024 and N1 + N1, respectively:

```
#define N1 1024
#define N2 N1 + N1
```

Suppose we have the following value declaration appearing in the scope of the above macro delarations:

```
val x = N1 * N2
```

Then N1 * N2 first expands into 1024 * (N1 + N1), which further expands into 1024 * (1024 + 1024). Note that if this example is done in C, then N1 * N2 expands into 1024 * 1024 + 1024, which is different from what we have here. Also note that it makes no difference if we reverse the order of the previous macro definitions:

```
#define N2 N1 + N1
#define N1 1024
```

If we now introduce the following declaration:

```
#define N3 %(N1 + N1) + N2
```

then the name N3 is bound to the abstract syntax tree of 2048 + N2. In general, an expression of the form %(exp) refers to the abstract syntax tree representing the value of exp.

If we declare a marco as follows:

```
#define LOOP (LOOP + 1)
```

then an infinite loop is entered (or more precisely, some macro expansion depth is to be reached) when the identifier LOOP is expanded. There is currently no guard against infinite macro expansion in ATS, and the propgrammer is fully responsible for avoiding it.

### LISP-like Macros

There are two forms of LISP-like macros in ATS: short form and long form. These (untyped) macros are highly flexible and expressive, and they can certainly be used in convoluted manners that should probably be avoided in the first place. Some commonly used macro definitions can be found on-line[1]. In order to use LISP-like macros in ATS effectively, the programmer may want to study some examples in LISP involving backquote-comma-notation.

### Macros in Long Form

As a macro in short form can simply be considered a special kind of macro in long form, we first give some explanantion on the latter. A macro definition in long form is introduced through the use of the keyword macrodef. For instance, the following syntax introduces a macro name one that refers to some code, that is, abstract syntax tree (AST) representing the integer number 1.

```
macrodef one = `(1)
```

The special syntax ‘(exp), where no space is allowed between the backquote (‘) and the left parenthsis ((), means to form an abstract syntax tree representing the expression written inside the parentheses. This is often referred to as backquote-notation. Intuitively, one may think that a backquote-notation exerts an effect that *freezes* everything inside it. Let us now define another macro as follows:

```
macrodef one_plus_one = `(1 + 1)
```

The defined macro name one_plus_one refers to some code (i.e., AST) representing 1 + 1. At this point, it is important to stress that the code representing 1 + 1 is different from the code representing 2. The macro name one_plus_one can also be defined as follows:

```
macrodef one_plus_one = `(,(one) + ,(one))
```

The syntax ,(exp), where no space is allowed between the comma (,) and the left parenthesis ((), indicates the need to expand (or evaluate) the expression written inside the parentheses. This is often referred to as comma-notation, which is only allowed inside a backquote-notation. Intuitively, a comma-notation cancels out the *freezing* effect of the enclosing backquote-notation.

In addition to macro names, we can also define macro functions. For instance, the following syntax introduces a macro function cube_mac:

```
macrodef cube_mac (x) =
  `(,(x) * ,(x) * ,(x)) // [x] should refer to some code
```

Here are some examples that make use of cube_mac:

```
fun cubesum (i:int, j: int): int =
  ,(cube_mac `(i)) +  ,(cube_mac `(j))
fun volOfSphere (r: double): double =
  4.0 * 3.1416 * ,(cube_mac `(r)) / 3
```

After macro expansion, the definitions of the functions cubesum and volOfSphere can be written as

```
fun cubesum (i: int, j: int): int = (i * i * i) + (j * j * j)
fun volOfSphere (r: double): double = 4.0 * 3.1416 * (r * r * r) / 3
```

### Macros in Short Form

The previous macro function cube_mac can also be defined as follows:

```
macdef cube_mac (x) = ,(x) * ,(x) * ,(x) // [x] should refer to some code
```

The keyword macdef introduces a macro definition in short form. The previous examples that make use of cube_mac can now be written as follows:

```
fun cubesum (i:int, j: int): int = cube_mac (i) +  cube_mac (j)
fun volOfSphere (r: double): double = 4.0 * 3.1416 * cube_mac (r) / 3
```

In terms of syntax, a macro function in short form is just like an ordinary function. In general, if a unary macro function fmac in short form is defined as as follows:

```
macdef fmac (x) = exp
```

where exp stands for some expression, then one may essentially think that a macro definition in long form is defined as follows:

```
macrodef lfmac (x) = `(exp) // please note the backquote
```

and each occurrence of fmac(arg) for some expresson arg is automatically rewritten into ,(lfmac(‘(arg))). Note that macro functions in short form with multiple arguments are handled analogously.

The primary purpose for introducing macros in short form is to provide a form of syntax that seems more accessible. While macros in long form can be defined recursively (as is to be explained later), macros in short form cannot.

## Compile-Time Directives

## Overloading

I will cover in this section several forms of overloading supported in ATS: static constant overloading, data constructor overloading and dynamic function overloading.

### Static Constant Overloading

### Data Constructor Overloading

### Dynamic Function Overloading

A symbol in ATS can be overloaded with multiple (dynamic) functions. The following syntax introduces a symbol of the name foo for overloading:

```
symintr foo // symbol introduction for overloading
```

Suppose that foo1, foo2 and foo3 are names of three functions in ATS. Then we can overload foo with these three functions as follows:

```
overload foo with foo1
overload foo with foo2
overload foo with foo3
```

An overloaded function symbol is resolved according to the number of arguments it takes and, if needed, the types of these arguments.

## Notes

1. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/macrodef.sats

# Chapter 9. Modularity

Generally speaking, modularity in programming means to organize programs in a modular fashion so that they each can be constructed in a relatively isolated manner and then be combined to function coherently. I will introduce in this section some features in ATS that are largely designed to facilitate program organization.

The code employed for illustration in this chapter plus some additional code for testing is available on-line[1].

## Types as a Form of Specification

The interface for a function or value specifies a type that any implementation of the function or value should possess. For instance, the following code defines a function fact for computing the factorial numbers:

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

It is also possible to first declare an interface for fact as follows:

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

where extern is a keyword in ATS that initiates the declaration of an interface. I will mention later an occasion where the keyword extern should be dropped. An alternative way to declare an interface for fact is given as follows:

```
extern val fact : (int) -> int
```

If fact is declared to be a function, then it is required to be applied when occurring in code. If it is declared to be a value, there is no such a restriction.

A function interface can be considered as a form of specification. For instance, the above interface for fact specifies that fact is a function that takes one integer argument and returns an integer value. What is so special about this form of specification is that it is formally enforced in ATS through typechecking: Any implementation of fact in ATS must possess the interface declared for it. Of course, this is not a precise specification for fact as there are (infinite) many functions that can be given the same interface. This kind of imprecision can, however, be reduced or even eliminated, sometimes. After dependent types are introduced, I will present an interface for fact such that any implementation of the interface is guaranteed to implement precisely the factorial function as is defined by the following two equations:

- fact(0) = 1
- fact(n) = n * fact (n-1) for each natural number n > 0

An implementation for fact as the following one can be given at any point where the declared interface for fact is accessible:

```
implement fact (x) = if x > 0 then x * fact (x-1) else 1
```

The keyword implement is for initiating an implementation of a function or value whose interface is already declared.

As an example of an interface for a value, fact10 is declared as follows to be a value of the type int:

```
extern val fact10 : int
```

The following implementation for fact10 can be given at any point where the declared interface for fact10 is accessible:

```
implement fact10 = fact (10)
```

As another example, the following code declares an interface for a polymorphic function named swap_boxed:

```
extern fun swap_boxed {a,b:type} (xy: (a, b)): (b, a)
```

Note that both type variables a and b are boxed. An implementation for swap_boxed is given as follows:

```
implement swap_boxed {a,b} (xy) = (xy.1, xy.0)
```

The syntax {a,b} is for passing static arguments a and b to swap_boxed simultaneously. As neither a nor b is actually used in the body of swap_boxed, it is allowed to drop {a,b} in this case.

It is a standard practice for a programmer to first design interfaces for the functions to be supported in a package before actually implementing any of these functions. When such interfaces are available, application programs can be constructed to check whether the interface design makes sense or is convenient for practical use. Please remember that a superb implementation of a poor design cannot make the design any better. Therefore, testing a design before actually implementing it is often of vital importance. This is especially true if the involved design is complex.

## Static and Dynamic ATS Files

The first letters in the ATS filename extensions *sats* and *dats* refer to the words *static* and *dynamic*, respectively. For instance, `foo.sats` is a name for a static file while `bar.dats` for a dynamic one. A static file usually contains interface declarations for functions and values, datatype declarations, type definitions, etc. The primary purpose of such a file is for allowing its content to be shared among various other ATS files, either static or dynamic.

Let us now go through a simple example to see a typical use of static files. Suppose that we want to implement the Ackermann's function, which is famous for being recursive but not primitive recursive. In a static file named `acker.sats` (or any other legal filename), we can declare the following function interface:

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

Please note that we cannot use the keyword extern when declaring an interface for either a function or a value in a static file. Then in a dynamic file named `acker.dats` (or any other legal filename), we can give the following implementation:

```
staload "acker.sats"

implement
acker (m, n) =
  if m > 0 then
    if n > 0 then acker (m-1, acker (m, n-1))
    else acker (m-1, 1)
  else n+1
// end of [acker]
```

The keyword staload indicates to the ATS typechecker that the file following it is to be statically loaded. Essentially, statically loading a file means to put the content of the file in a namespace that can be accessed by the code that follows. It is important to note that static loading is different from plain file inclusion. The latter is also supported in ATS, and it is a feature I will cover elsewhere.

It is also possible to give the following implementation for the declared function acker:

```
staload ACKER = "acker.sats"

implement $ACKER.acker
  (m, n) = acker (m, n) where {
  fun acker (m: int, n:int): int =
    if m > 0 then
      if n > 0 then acker (m-1, acker (m, n-1))
      else acker (m-1, 1)
    else n+1
} // end of [$ACKER.acker]
```

In this case, the namespace for storing the content of the file `acker.sats` is given the name ACKER, and the prefix $ACKER. (the dollar sign followed by ACKER followed by the dot symbol) must be attached to any name that refers an entity (a function, a value, a datatype, the constructors associated with a datatype, a type definition, etc.) declared in `acker.sats`. When there are many static files to be loaded, it is often a good practice to assign names to the namespaces holding these files so that the original source of each declared entity can be readily tracked down.

In another file named `test_acker.dats`, let use write the following code:

```
staload "acker.sats"
dynload "acker.dats"

implement
main () = () where {
//
// acker (3, 3) should return 61
//
  val () = assertloc (acker (3, 3) = 61)
} // end of [main]
```

The keyword dynload indicates to the ATS compiler to generate a call to the initializing function associated with the file `acker.dats`. This is mandatory as an error would otherwise be reported at link-time. Usually, calling the initializing function associated with a dynamic file is necessary only if there is a value implemented in the file. In this case, there is only a function implemented in `acker.dats`. If we include the following line somewhere inside `acker.dats`:

```
#define ATS_DYNLOADFLAG 0 // no need for dynloading at run-time
```

then the line starting with the keyword dynload in `test_acker.dats` is no longer needed. The function assertloc verifies at run-time that its argument evaluates to the boolean value true. In the case where the argument evaluates to false, the function call aborts and a message is reported that contains the name of the file, which is `test_acker.dats` in this example, and the location at which the source code of the call is found in the file. If this sounds a bit confusing, please try to execute a program that contains a call to assertloc on false and you will see clearly what happens.

The simplest way to compile the three files acker.sats, acker.dats and test_acker.dats is to issue the following command-line:

```
atscc -o test_acker acker.sats acker.dats test_acker.dats
```

The generate excutable `test_acker` is in the current working directory. The compilation can also be performed separately as is demonstrated below:

```
atscc -c acker.sats
atscc -c acker.dats
atscc -c test_acker.dats
atscc -o test_acker acker_sats.o acker_dats.o test_acker_dats.o
```

This style of separate compilation works particularly well when it is employed by the **make** utility.

## Generic Template Implementation

Interfaces for function templates are mostly similar to those for functions. For example, the following syntax declares an interface in a dynamic file for a function template of the name list0_fold_left:

```
extern fun{a:t@ype}{b:t@ype}
list0_fold_left (f: (a, b) -<cloref1> a, init: a, xs: list0 b): a
```

If the same interface is declared in a static file, the keyword extern should be dropped. Implementing an interface for a function template is also mostly similar to implementing one for a function. The above interface for list0_fold_left is given an implementation in the following code:

```
implement{a}{b}
list0_fold_left (f, init, xs) = let
  fun loop (init: a, xs: list0 b):<cloref1> a =
    case+ xs of
    | list0_cons (x, xs) => loop (f (init, x), xs)
    | list0_nil () => init
  // end of [loop]
in
  loop (init, xs)
end // end of [list0_fold_left]
```

Note that template parameters are required to appear immediately after the keyword implement, and they cannot be omitted. Template parameters can also be passed sequentially as is shown in the following short example:

```
extern fun{a,b:t@ype}{c:t@ype}
app2 (f: (a, b) -<cloref1> c, x: a, y: b): c

implement{a,b}{c} app2 (f, x, y) = f (x, y)
```

The style of template implementation presented in this section is referred to as generic template implementation. I will later present a different style of template implementation, which is often referred to as specific template implementation.

## Abstract Types

The name *abstract type* refers to a type such that values of the type are represented in a way that is completely hidden from users of the type. This form of information-hiding attempts to ensure that changes to the implementation of an abstract type cannot introduce type-errors into well-typed code that makes use of the abstract type. In

ATS as well as in many other programming languages, abstract types play a pivotal role in support of modular programming. I will present as follows a concrete example to illustrate a typical use of abstract types in practice.

Suppose that we are to implement a package to provide various funtionalities on finite sets of integers. We first declare an abstract type intset as follows for values representing finite sets of integers:

```
abstype intset // a boxed abstract type
```

The keyword abstype indicates that the declared abstract type intset is boxed, that is, the size of intset is the same as a pointer. There is a related keyword abst@ype for introducing unboxed abstract types, which will be explained elsewhere. We next present an interface for each function or value that we want to implement in the package:

```
// empty set
val intset_empty : intset

// singleton set of [x]
fun intset_make_sing (x: int): intset

// turning a list into a set
fun intset_make_list (xs: list0 int): intset

// turning a set into a list
fun intset_listize (xs: intset): list0 (int)

// membership test
fun intset_ismem (xs: intset, x: int): bool

// computing the size of [xs]
fun intset_size (xs: intset): size_t

// adding [x] into [xs]
fun intset_add (xs: intset, x: int): intset

// deleting [x] from [xs]
fun intset_del (xs: intset, x: int): intset

// union of [xs1] and [xs2]
fun intset_union (xs1: intset, xs2: intset): intset

// intersection of [xs1] and [xs2]
fun intset_inter (xs1: intset, xs2: intset): intset

// difference between [xs1] and [xs2]
fun intset_differ (xs1: intset, xs2: intset): intset
```

Let us now suppose that the declaration for intset and the above interfaces are all stored in a file named `intset.sats` (or any other legal name for a static file).

Usually, a realistic implementation for finite sets is based on some kind of balanced trees (e.g., AVL trees, red-black trees). For the purpose of illustration, we give an implementation in which finite sets of integers are represented as ordered lists of integers. This implementation is contained in a file named `intset.dats`, which is available on-line[2]. In order to construct values of an abstract type, we need to concretize it temporarily by using the following form of declaration:

```
assume intset = list0 (int)
```

where assume is a keyword. This assume-declaration equates intset with the type list0 (int) and this equation is valid until the end of the scope in which it is introduced.

As the assume-declaration is at the toplevel in `intset.dats`, the assumption that intset equals list0 (int) is valid until the end of the file. There is a global restriction in ATS that allows each abstract type to be concretized by an assume-declaration at most once. More specifically, if an abstract type is concretized in two files `foo1.dats` and `foo2.dats`, then these two files cannot be used together to generate an executable. The rest of implementation in intset is all standard. For instance, the union operation on two given sets of integers is implemented as follows:

```
implement
intset_union (xs1, xs2) =
  case+ (xs1, xs2) of
  | (list0_cons (x1, xs11), list0_cons (x2, xs21)) =>
      if x1 < x2 then
        list0_cons (x1, intset_union (xs11, xs2))
      else if x1 > x2 then
        list0_cons (x2, intset_union (xs1, xs21))
      else list0_cons (x1, intset_union (xs11, xs21))
  | (list0_nil (), _) => xs2
  | (_, list0_nil ()) => xs1
// end of [intset_union]
```

There is also some testing code available on-line[3] that makes use of some functions declared in `intset.sats`. Often testing code as such is constructed immediately after the interfaces for various functions and values in a package are declared. This allows these interfaces to be tried before they are actually implemented so that potential flaws can be exposed in a timely fashion.

## Example: A Package for Rationals

We are to represent a rational number as a pair of integers. If we declare a boxed abstract type rat for values representing rational numbers, then each value of the type rat is stored in heap-allocated memory, which can only be reclaimed through garbage collection (GC). Instead, we follow an alternative approach by declaring rat as an unboxed abstract type. Therefore, a declaration like the following one is expected:

```
abst@ype rat
```

The problem with this declaration is that it is too abstract. As there is not information given about the size of the type rat, the ATS compiler does not even know how much memory is needed for storing a value of the type rat. However, the programmer should not assume that such a form of declaration is useless. There are actually circumstances where a declaration of this form can be of great importance, and this is a topic I will cover elsewhere. For now, let us declare an unboxed abstract type as follows:

```
abst@ype rat = (int, int)
```

This declaration simply informs the ATS compiler that the representation for values of the type rat is the same as the one for values of the type (int, int). However, this information is not made available to the typechecker of ATS. In particular, if a value of the type rat is treated as a pair of integers in a program, then a type-error will surely occur.

The following code is contained in a file named `ratmod.sats`, which is available on-line[4].

```
exception Denominator
exception DivisionByZero
```

```
fun rat_make_int_int (p: int, q: int): rat

fun ratneg: (rat) -> rat // negation
fun ratadd: (rat, rat) -> rat // addition
fun ratsub: (rat, rat) -> rat // subtraction
fun ratmul: (rat, rat) -> rat // multiplication
fun ratdiv: (rat, rat) -> rat // division
```

The exception Denominator is for reporting an erroneous occasion where a rational number is to be formed with a denominator equal to zero. Given two integers representing the numerator and denominator of a rational number, the function rat_make_int_int returns a value representing the rational number. The following implementation of rat_make_int_int can be found in a file named `ratmod.dats`, which is also available on-line[5].

```
implement
rat_make_int_int (p, q) = let
  fun make (
    p: int, q: int
  ) : rat = let
    val r = gcd (p, q) in (p / r, q / r)
  end // end of [make]
//
  val () = if q = 0 then $raise Denominator
//
in
  if q > 0 then make (p, q) else make (~p, ~q)
end // end of [rat_make_int_int]
```

Given a pair of integers p and q such that q is not zero, the function rat_make_int_int returns another pair of integers $p_1$ and $q_1$ such that $q_1$ is positive, $p_1$ and $q_1$ are coprimes, that is, their greatest common divisor is 1, and $p_1/q_1$ equals p/q. With rat_make_int_int, it is straightforward to implement as follows the arithmetic operations on rational numbers:

```
implement ratneg (x) = (~x.0, x.1)

implement
ratadd (x, y) =
  rat_make_int_int (x.0 * y.1 + x.1 * y.0, x.1 * y.1)
// end of [ratadd]

implement
ratsub (x, y) =
  rat_make_int_int (x.0 * y.1 - x.1 * y.0, x.1 * y.1)
// end of [ratsub]

implement
ratmul (x, y) = rat_make_int_int (x.0 * y.0, x.1 * y.1)

implement
ratdiv (x, y) =
  if y.0 > 0 then rat_make_int_int (x.0 * y.1, x.1 * y.0)
  else $raise DivisionByZero
// end of [ratdiv]
```

There is also some testing code available on-line[6] that makes use of some functions declared in `ratmod.sats`.

## Example: A Functorial Package for Rationals

The previous package for rational numbers contains a serious limitation: The type for the integers employed in the representation of rational numbers is fixed to be int. If we ever want to represent rational numbers based on integers of a different type (for instance, lint for long integers or llint for long long integers), then we need to implement another package for rationals based on such integers. It is clearly advantageous to avoid this style of programming as it involves code duplication to a great extent.

The approach we take in this section to implement a package for rational numbers that can address the aforementioned limitation follows the idea of functors in the programming language Standard ML (SML). We first introduce a type definition as follows:

```
typedef
intmod (a:t@ype) = '{
  ofint= int -> a
, fprint= (FILEref, a) -> void
, neg= (a) -> a // negation
, add= (a, a) -> a // addition
, sub= (a, a) -> a // subtraction
, mul= (a, a) -> a // multiplication
, div= (a, a) -> a // division
, mod= (a, a) -> a // modulo
, cmp= (a, a) -> int // comparison
} // end of [intmod]
```

Given a type T, intmod(T) is a boxed record type in which each field is a function type. A value of the type intmod(T) is supposed to represent a module of integer operations on integers represented by values of the type T. Similarly, we introduce another type definition as follows:

```
abst@ype rat (a:t@ype) = (a, a)

typedef
ratmod (a:t@ype) = '{
  make= (a, a) -<cloref1> rat a
, fprint= (FILEref, rat a) -<cloref1> void
, numer= rat a -> a // numerator
, denom= rat a -> a // denominator
, neg= (rat a) -<cloref1> rat a // negation
, add= (rat a, rat a) -<cloref1> rat a // addition
, sub= (rat a, rat a) -<cloref1> rat a // subtraction
, mul= (rat a, rat a) -<cloref1> rat a // multiplication
, div= (rat a, rat a) -<cloref1> rat a // division
, cmp= (rat a, rat a) -<cloref1> int // comparison
} // end of [ratmod]
```

Given a type T, a value of the type ratmod(T) is supposed to represent a module of rational operations on rationals represented by values of the type rat(T). The function we need to implement can now be given the following interface:

```
fun{a:t@ype} ratmod_make_intmod (int: intmod a): ratmod a
```

If applied to a given module of integer operations, the function ratmod_make_intmod returns a module of rational operations such that the integers in the former and the latter modules have the same representation. Therefore, ratmod_make_intmod behaves like a functor in SML. In the following code, we implement two modules ratmod_int and ratmod_dbl of rational operations in which integers are represented as values of the types int and double, respectively:

```
staload M = "libc/SATS/math.sats" // for [fmod]

val intmod_int = '{
  ofint= lam (i) => i
, fprint= lam (out, x) => fprintf (out, "%i", @(x))
, neg= lam (x) => ~x
, add= lam (x, y) => x + y
, sub= lam (x, y) => x - y
, mul= lam (x, y) => x * y
, div= lam (x, y) => x / y
, mod= lam (x, y) => op mod (x, y)
, cmp= lam (x, y) => compare (x, y)
} : intmod (int) // end of [val]

val ratmod_int = ratmod_make_intmod<int> (intmod_int)

val intmod_dbl = '{
  ofint= lam (i) => double_of (i)
, fprint= lam (out, x) => fprintf (out, "%0.f", @(x))
, neg= lam (x) => ~x
, add= lam (x, y) => x + y
, sub= lam (x, y) => x - y
, mul= lam (x, y) => x * y
, div= lam (x, y) => $M.trunc (x/y) // trunc: truncation
, mod= lam (x, y) => $M.fmod (x, y) // the modulo function
, cmp= lam (x, y) => compare (x, y)
} : intmod (double) // end of [val]

val ratmod_dbl = ratmod_make_intmod<double> (intmod_dbl)
```

An implementation of the function ratmod_make_intmod is available on-line[7] and there is some related testing code available on-line[8] as well.

## Specific Template Implementation

Implementing an interface for a function template specifically means to give an implementation for a fixed instance of the template. For instance, the following interface is for a function template of the name eq_elt_elt:

```
fun{a:t@ype} eq_elt_elt (x: a, y: a): bool // a generic equality
```

There is no meaningful generic implementation for eq_elt_elt as equality test for values of a type T depends on T. Two specific template implementations are given as follows for the instances eq_elt_elt<int> and eq_elt_elt<double>:

```
implement eq_elt_elt<int> (x, y) = eq_int_int (x, y)
implement eq_elt_elt<double> (x, y) = eq_double_double (x, y)
```

where eq_int_int and eq_double_double are equality functions for values of the type int and double, respectively. It is also possible to give the implementations as follows:

```
implement eq_elt_elt<int> (x, y) = x + y
implement eq_elt_elt<double> (x, y) = x + y
```

This is allowed as the symbol + is already overloaded with eq_int_int and eq_double_double (in addition to many other functions).

Let us now see a typical use of specific template implementation. The following defined function template listeq implements an equality function on lists:

```
fun{a:t@ype}
listeq (xs: list0 a, ys: list0 a): bool =
  case+ (xs, ys) of
  | (list0_cons (x, xs), list0_cons (y, ys)) =>
      if eq_elt_elt<a> (x, y) then listeq (xs, ys) else false
  | (list0_nil (), list0_nil ()) => true
  | (_, _) => false
// end of [listeq]
```

Given two lists xs and ys, listeq returns true if and only if xs and ys are of the same length and each element in xs equals the corresponding one in ys based on a call to eq_elt_elt. Given a type T, it is clear that the instance eq_elt_elt<T> is needed if listeq is called on two lists of the type list0(T). In other words, a specific implementation for eq_elt_elt<T> should be given if a call to listeq is to be made on two lists of the type list0(T). Note that the implementation for an instance of a function template is required to be present in the same file where the instance is called.

As a comparison, the following defined function template listeqf also implements equality test on two given lists:

```
fun{a:t@ype}
listeqf (
  xs: list0 a, ys: list0 a, eq: (a, a) -> bool
) : bool =
  case+ (xs, ys) of
  | (list0_cons (x, xs), list0_cons (y, ys)) =>
      if eq (x, y) then listeqf (xs, ys, eq) else false
  | (list0_nil (), list0_nil ()) => true
  | (_, _) => false
// end of [listeqf]
```

In this case, listeqf takes an additional argument eq that tests whether two list elements are equal. Both listeq and listeqf have advantages over the other. The former is a first-order function while the latter is a higher-order one, and thus it is likely for the former to be compiled into more efficient object code. However, the latter often seems more convenient for use in practice.

Please find the code presented in this section plus some additional testing code available on-line[9].

## Example: A Temptorial Package for Rationals

As I have great difficulty in naming the style of programming I am about to present in this section, I coin a word *temptorial* (as a shorthand for template-functorial) to refer to this style, which makes essential use of function templates that are implemented generically as well as specifically.

Suppose that we have interfaces for two function templates foo and bar. We give a generic template implementation for bar that makes use of foo but we cannot or do not want to implement foo generically. When an instance of bar is called, certain instances of foo may need to be implemented specifically so as to support the call. Let us now imagine a design where foo and bar are replaced with collections of function templates corresponding to various operations on integers and rationals. This is precisely the idea employed in the design of a temptorial package for rationals.

The following interfaces for function templates are declared in a file named `ratfun_tmp.sats`, which is available on-line[10]:

```
fun{a:t@ype} ofint: int -> a
fun{a:t@ype} fprint_int (out: FILEref, x: a): void

fun{a:t@ype} intneg: a -> a
```

```
fun{a:t@ype} intadd: (a, a) -> a
fun{a:t@ype} intsub: (a, a) -> a
fun{a:t@ype} intmul: (a, a) -> a
fun{a:t@ype} intdiv: (a, a) -> a
fun{a:t@ype} intmod: (a, a) -> a

fun{a:t@ype} intcmp: (a, a) -> int

(* ^^^^^^ ^^^^^^ *)
//
// the following templates are implemented based on the above ones
//
(* vvvvvv vvvvvv *)

fun{a:t@ype} intgcd: (a, a) -> a

fun{a:t@ype} intlt: (a, a) -> bool
fun{a:t@ype} intlte: (a, a) -> bool
fun{a:t@ype} intgt: (a, a) -> bool
fun{a:t@ype} intgte: (a, a) -> bool
fun{a:t@ype} inteq: (a, a) -> bool
fun{a:t@ype} intneq: (a, a) -> bool

abst@ype rat (a: t@ype) = (a, a)

fun{a:t@ype} rat_make_int_int (p: int, q: int): rat a
fun{a:t@ype} rat_make_numer_denom (p: a, q: a): rat a

fun{a:t@ype} fprint_rat (out: FILEref, x: rat a): void

fun{a:t@ype} rat_numer (x: rat a): a
fun{a:t@ype} rat_denom (x: rat a): a

fun{a:t@ype} ratneg: (rat a) -> rat a
fun{a:t@ype} ratadd: (rat a, rat a) -> rat a
fun{a:t@ype} ratsub: (rat a, rat a) -> rat a
fun{a:t@ype} ratmul: (rat a, rat a) -> rat a
fun{a:t@ype} ratdiv: (rat a, rat a) -> rat a

fun{a:t@ype} ratcmp: (rat a, rat a) -> int
```

In another file named `ratfun_tmp.dats`, which is also available on-line[11], the second half of the above interfaces for function templates are implemented based on the function templates for which the interfaces are declared in the first half. As an example, the function template rat_make_numer_denom is implemented as follows:

```
implement{a}
rat_make_numer_denom (p, q) = let
  fun make (p: a, q: a): rat a = let
    val r = intgcd (p, q) in (p \intdiv r, q \intdiv r)
  end // end of [make]
in
  if intgtz (q)
    then make (p, q) else make (intneg p, intneg q)
  // end of [if]
end // end of [rat_make_numer_denom]
```

Note that the backslash symbol (\) temporarily assigns the infix status to the identifier immediately following it. For instance, the syntax p \intdiv r simply stands for the function application intdiv(p, r).

There is some testing code available on-line[12] that demonstrates a typical way to use a temptorial package. For instance, if we want to use a package for rationals where integers are represented as values of the type double, then we can first give the following specific implementations for instances of function templates corresponding to certain integer operations:

```
staload M = "libc/SATS/math.sats" // for [fmod]

typedef T = double
implement ofint<T> (x) = double_of (x)
implement fprint_int<T> (out, x) = fprintf (out, "%.0f", @(x))

implement intneg<T> (x) = ~x
implement intadd<T> (x, y) = x + y
implement intsub<T> (x, y) = x - y
implement intmul<T> (x, y) = x * y
implement intdiv<T> (x, y) = $M.trunc (x/y) // trunc: truncation
implement intmod<T> (x, y) = $M.fmod (x, y) // modulo operation

implement intcmp<T> (x, y) = compare (x, y)
```

With these implementations, we can call the corresponding instances of those function templates in `ratfun_tmp.sats` (e.g., ratadd<double> and ratsub<double>) that have already been implemented generically.

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/intse
3. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/test_
4. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/ratm
5. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/ratm
6. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/test_
7. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/ratfu
8. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/test_
9. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/listeq
10. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/ratfu
11. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/ratfu
12. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_MODULARITY/test_

# Chapter 10. Library Support

There are three built-in libraries in the ATS programming language system: `prelude`[1], `libc`[2], and `libats`[3].

## The prelude Library

The prelude library provides support for funcions that are commonly needed in practice. The files containing the interfaces for these functions are mostly stored in the directory `prelude/SATS`[4]. I list some of these files as follows and also briefly explain what these files are for.

- `prelude/SATS/bool.sats`[5]: This package is for boolean values of the type bool, which is mapped to the type int in the C programming language.
- `prelude/SATS/char.sats`[6]: This package is for character values of various types: char for chars, which may or may not be signed, and uchar for unsigned chars, and schar for signed chars. These types are mapped to the types char, uchar and schar in the C programming language.
- `prelude/SATS/filebas.sats`[7]: This package constains some functions for basic file operations.
- `prelude/SATS/float.sats`[8]: This package is for floating point numbers of various types: float for single precision represention, double for double precision represention, and ldouble for long double precision represention. These types are mapped to the following types in the C programming language: float, double, and long double.
- `prelude/SATS/integer.sats`[9]: This package is for finite precision integers of all sorts of types: int for signed integers, uint for unsigned integers, lint for signed long integers, ulint for unsigned long integers, llint for signed long long integers, and ullint for unsigned long long integers. These types are mapped to the following types in the C programming lanuguage: int, unsigned int, long int, unsigned long int, long long int, unsigned long long int.
- `prelude/SATS/string.sats`[10]: This package constains various functions for processing strings.
- `prelude/SATS/array0.sats`[11]: This package is for arrays with size information attached. The functions in it are relatively easy to use but they may incur run-time array bounds checking.
- `prelude/SATS/list0.sats`[12]: This package is for values representing functional lists that are assgined the type list0(T) for some type T.
- `prelude/SATS/matrix0.sats`[13]: This package is for matrices, that is, two-dimensional arrays, with dimension information, that is, the number of rows and the number of columns, attached. The functions in it are relatively easy to use but they may incur run-time array bounds checking.

Note that programming with functions in the prelude library often requires the support of dependent types and linear types, which I will cover elsewhere.

## The libc Library

## The libats Library

## Contributed Packages

## Notes

1. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/
2. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/libc/
3. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/libats/
4. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/
5. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/bool.sats
6. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/char.sats
7. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/filebas.sats
8. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/float.sats
9. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/integer.sats
10. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/string.sats
11. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/array0.sats
12. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/list0.sats
13. http://www.ats-lang.org/DOCUMENTATION/ANAIRIATS/prelude/SATS/matrix0.sats

# Chapter 11. Interaction with the C Programming Language

ATS and C share precisely the same native/flat/unboxed data representation. As a consequence, there is no need for wrapping/unwrapping or boxing/unboxing when calling from C a function in ATS or vice versa, and there is also no run-time overhead for doing so.

# Chapter 12. Summary

# Chapter 13. Introduction to Dependent Types

The types we have encountered so far in this book cannot offer adequate precision for capturing programming invariants. For instance, if we assign the type int to both of 0 and 1, then we simply cannot distinguish 0 from 1 at the level of types. This means that 0 and 1 are interchangeable as far as typechecking is concerned. In other words, we cannot expect a program error to be caught during typechecking if the error is caused by 0 being mistyped as 1. This form of imprecision in types is a crippling limitation if we ever want to build a type-based specification language that is reasonably expressive for practical use.

Please find on-line[1] the code employed for illustration in this chapter plus some additional code for testing.

## Enhanced Expressiveness for Specification

The primary purpose of introducing dependent types into the type system of ATS is to greatly enhance the expressiveness of types so that they can be employed to capture program invariants with much more precision. Generally speaking, dependent types are types dependent on values of expressions. For instance, bool is a type constructor in ATS that forms a type bool(b) when applied to a given boolean value b. As this type can only be assigned to a boolean expression of the value b, it is often referred to as a singleton type, that is, a type for exactly one value. Clearly, the meaning of bool(b) depends on the boolean value b. Similarly, int is a type constructor in ATS that forms a type int(i) when applied to a given integer i. This type is also a singleton type as it can only be assigned to an integer expression of the value i. Note that both bool and int are overloaded as they also refer to (non-dependent) types. I will gradually introduce many other examples of dependent types. In particular, I will present a means for the programmer to declare dependent datatypes.

The statics of ATS is a simply-typed language, and the types in this language are called *sorts* so as to avoid some potential confusion (with the types for dynamic terms). The following four listed sorts are commonly used:

- *bool*: for static terms of boolean values
- *int*: for static terms of integer values
- *type*: for static terms representing boxed types (for dynamic terms)
- *t@ype*: for static terms representing unboxed types (for dynamic terms)

The sorts *bool* and *int* are classified as predicative sorts while the sorts *type* and *t@ype* are impredicative. A boxed type is a static term of the sort *type* while an unboxed type is a static term of the sort *t@ype*. As types, bool and int are static terms of the sort *t@ype*. As type constructors, bool and int are static terms of the sorts (*bool*) -> *t@ype* and (*int*) -> *t@ype*, respectively. Also note that the type constructor list0 is of the sort (*t@ype*) -> *type*, which indicates that list0 forms a boxed type when applied to an unboxed one. There are also various built-in static functions in ATS for constructing static terms of the sorts *bool* and *int*, and I list as follows some of these functions together with the sorts assigned to them:

- ~ (integer negation): (*int*) -> *int*
- + (addition): (*int*, *int*) -> *int*
- - (subtraction): (*int*, *int*) -> *int*
- * (multiplication): (*int*, *int*) -> *int*
- / (division): (*int*, *int*) -> *int*
- > (greater-than): (*int*, *int*) -> *bool*

- >= (greater-than-or-equal-to): (*int*, *int*) -> *bool*
- < (less-than): (*int*, *int*) -> *bool*
- <= (less-than-or-equal-to): (*int*, *int*) -> *bool*
- == (equal-to): (*int*, *int*) -> *bool*
- <> (not-equal-to): (*int*, *int*) -> *bool*
- ~ (boolean negation): (*bool*) -> *bool*
- || (disjunction): (*bool*, *bool*) -> *bool*
- && (conjunction) : (*bool*, *bool*) -> *bool*

By combining a sort with one or more predicates, we can define a subset sort. For instance, the following subset sorts are defined in the file `prelude/sortdef.sats`[2], which is automatically loaded by the ATS compiler:

```
sortdef nat = {a: int | a >= 0} // for natural numbers
sortdef pos = {a: int | a > 0}  // for positive numbers
sortdef neg = {a: int | a < 0}  // for negative numbers

sortdef two = {a: int | 0 <= a; a <= 1} // for 0 or 1
sortdef three = {a: int | 0 <= a; a <= 2} // for 0, 1 or 2
```

Note that predicates can be sequenced together with the semicolon symbol (;). It is also possible to define the subset sorts *two* and *three* as follows:

```
sortdef two = {a: int | a == 0 || a == 1} // for 0 or 1
sortdef three = {a: int | 0 <= a && a <= 2} // for 0, 1 or 2
```

Another possibility is to define a subset sort based on an existing one plus some predicates. For instance, the subset sorts *two* and *three* can also be given the following definitions:

```
sortdef two = {a: nat | a <= 1} // for 0 or 1
sortdef three = {a: nat | a <= 2} // for 0, 1 or 2
```

In order to unleash the expressiveness of dependent types, we need to employ both universal and existential quantification over static variables. For instance, the type Int in ATS is defined as follows:

```
typedef Int = [a:int] int (a) // for unspecified integers
```

where the syntax [a:int] means existential quantification over a static variable a of the sort *int*. Essentially, this means that for each value of the type Int, there exists an integer I such that the value is of the type int(I). Therefore, any value that can be given the type int can also be given the type Int. A type like Int is often referred to as an existentially quantified type. As another example, the type Nat in ATS is defined as follows:

```
typedef Nat = [a:int | a >= 0] int (a) // for natural numbers
```

where the syntax [a:int | a >= 0] means existential quantification over a static variable a of the sort *int* that satisfies the constraint a >= 0. Therefore, each value of the type Nat can be assigned the type int(I) for some integer I satisfying I >= 0. Given that int(I) is a singleton type, the value equals I and thus is a natural number. This means that the type Nat is for natural numbers. The definition of Nat can also be given as follows:

```
typedef Nat = [a:nat] int (a) // for natural numbers
```

where the syntax [a:nat] is just a form of syntactic sugar that automatically expands into [a:int | a >= 0].

At this point, types have already become much more expressive. For instance, we could only assign the type (int) -> int to a function that maps integers to natural numbers (e.g., the function that computes the absolute value of a given integer), but we can now do better by assigning it the type (Int) -> Nat, which is clearly more precise. In order to relate at the level of types the return value of a function to its arguments, we need universal quantification. For instance, the following universally quantified type is for a function that returns the successor of its integer argument:

```
{i:int} int (i) -> int (i+1)
```

where the syntax {i:int} means universal quantification over a static variable i of the sort int and the scope of this quantification is the function type following it. One may think that this function type is also a singleton type as the only function of this type is the successor function on integers. Actually, there are infinitely may partial functions that can be given this type as well. For the successor function on natural numbers, we can use the following type:

```
{i:int | i >= 0} int (i) -> int (i+1)
```

where the syntax {i:int | i >= 0} means universal quantification over a static variable i of the sort *int* that satisfies the constraint i >= 0. This type can also be written as follows:

```
{i:nat} int (i) -> int (i+1)
```

where the syntax {i:nat} automatically expands into {i:int | i >= 0}. I list as follows the interfaces for some commonly used functions on integers:

```
fun ineg {i:int} (i: int i): int (~i) // negation
fun iadd {i,j:int} (i: int i, j: int j): int (i+j) // addition
fun isub {i,j:int} (i: int i, j: int j): int (i-j) // subtraction
fun imul {i,j:int} (i: int i, j: int j): int (i*j) // multiplication
fun idiv {i,j:int} (i: int i, j: int j): int (i/j) // division

fun ilt {i,j:int} (i: int i, j: int j): bool (i < j) // less-than
fun ilte {i,j:int} (i: int i, j: int j): bool (i <= j) // less-than-or-equal-to
fun gt {i,j:int} (i: int i, j: int j): bool (i > j) // greater-than
fun gte {i,j:int} (i: int i, j: int j): bool (i >= j) // greater-than-or-equal-to
fun eq {i,j:int} (i: int i, j: int j): bool (i == j) // equal-to
fun neq {i,j:int} (i: int i, j: int j): bool (i <> j) // not-equal-to
```

These interfaces are all declared in the file `prelude/SATS/integer.sats`[3], which is automatically loaded by the ATS compiler.

It is now a proper moment for me to raise an interesting question: What does a dependently typed interface for the factorial function look like? After seeing the above examples, it is natural for one to expect something along the following line:

```
fun ifact {i:nat} (i: int i): int (fact (i))
```

where *fact* is a static version of the factorial function. The problem with this solution is that a static function like *fact* cannot be defined in ATS. The statics of ATS is a simply-typed language that does not allow any recursive means to be employed in the construction of static terms. This design is adopted primarily to ensure that the equality on static terms can be decided based on a practical algorithm. As type-checking involving dependent types essentially turns into verifying whether a set of equalities (and some built-in predicates) on static terms hold, such a design is of vital importance to the goal of supporting practical programming with dependent types. In order to assign an interface to the factorial function that precisely matches the definition of the function, we need to employ a mechanism in ATS for combining programming with theorem-proving. This is a topic I will cover later.

## Constraint-Solving during Typechecking

Typechecking in ATS involves generating and solving constraints. As an example, the code below gives an implementation of the factorial function:

```
fun fact {n:nat}
  (x: int n): [r:nat] int r = if x > 0 then x * fact (x-1) else 1
// end of [fact]
```

In this implementation, the function fact is assigned the following type:

```
{n:nat} int(n) -> [r:nat] int(r)
```

which means that fact returns a natural number r when applied to a natural number n. When the code is typechecked, the following constraints need to be solved:

- For each natural number n, n > 0 implies n - 1 >= 0
- For each natural number n and each natural number $r_1$, n > 0 implies n * $r_1$ >= 0
- For each natural number n, 1 >= 0 holds.

The first constraint is generated due to the call fact(x-1), which requires that x-1 be a natural number. The second constraint is generated in order to verify that x * fact(x-1) is a natural number under the assumption that fact(x-1) is a natural number. The third constraint is generated in order to verify that 1 is a natural number. The first and the third constraints can be readily solved by the constraint solver in ATS, which is based on the Fourier-Motzkin variable elimination method. However, the second constraint cannot be handled by the constraint solver as it is nonlinear: The constraint cannot be turned into a linear integer programming problem due to the occurrence of the nonlinear term ($n*r_1$). While nonlinear constraints cannot be handled automatically by the constraint solver in ATS, the programmer can verify them by constructing proofs in ATS explicitly. I will coven the issue of explicit proof construction in an elaborated manner elsewhere.

As a more interesting example, the following code implements MacCarthy's famous 91-function:

```
fun f91 {i:int} (x: int i)
  : [j:int | (i < 101 && j==91) || (i >= 101 && j==i-10)] int (j) =
  if x >= 101 then x-10 else f91 (f91 (x+11))
// end of [f91]
```

The type assigned to f91 clearly indicates that the function always returns 91 if its argument is less than 101 or it returns the difference between its argument and 10. The constraints generated during typechecking in this example are all linear and can be readily solved by the the constraint-solver employed by the typechecker of ATS.

Currently, the constraint-solver implemented for ATS/Anairiats makes use of machine-level arithmetic (in contrast to the standard arithmetic of infinite precision). This is done primarily for the sake of efficiency. In the presence of machine-level arithmetic overflow during constraint-solving, results returned by the constraint-solver are likely to be incorrect. While such cases can be readily constructed, their appearances in practice seem rare.

## Example: String Processing

A string in ATS is represented in the same manner as in C: It is sequence of adjacently stored non-null characters followed by the null character, and its length is the number of non-null characters in the sequence. Conventionally, such strings are often referred to as C-style strings, which are notoriously difficult to be processed safely (as indicated by so many bugs and breaches due to misusing such strings). As a matter of fact, ATS is the first practical programming language that I know can fully support safe processing of C-style strings. In ATS, string is a type constructor of the sort *(int) -> type* . Given a static integer n, string(n) is the type for strings of the length n. Note that string also refers to a non-dependent type for strings of unspecified length, which is basically equivalent to the type String defined as follows:

```
typedef String = [n:nat] string (n)
```

The following two functions are commonly used for traversing a given string:

```
fun string_is_at_end
  {n:int} {i:nat | i <= n}
  (str: string n, i: size_t i): bool (i == n)
// end of [string_is_at_end]

fun string_isnot_at_end
  {n:int} {i:nat | i <= n}
  (str: string n, i: size_t i): bool (i < n)
// end of [string_isnot_at_end]
```

Obviously, either one of them can be implemented based on the other. As an example, the following code implements a function that computes the length of a string:

```
fun string_length {n:nat}
  (str: string n): size_t n = let
  fun loop {i:nat | i <= n}
    (str: string n, i: size_t i): size_t (n) =
    if string_isnot_at_end (str, i) then loop (str, i+1) else i
  // end of [loop]
in
  loop (str, 0)
end // end of [string_length]
```

Note that the function loop in the body of string_length is defined tail-recursively. Although this implementation of string_length looks fairly plain now, it was actually an exciting achievement in the pursuit of supporting practical programming with dependent types.

The following two functions are for accessing and updating characters stored in strings:

```
typedef c1har = [c:char | c <> '
```

The type constructor char is of the sort *(char) -> t@ype*, which takes a static character c to form a singleton type char(c) for the only character equal to c. Thus, the type c1har is for all non-null characters. The following defined function string_find traverses a string from left to right to see if a given character occurs in the string:

```
typedef sizeLt (n:int) = [i:nat | i < n] size_t (i)

fun string_find {n:nat}
  (str: string n, c0: char): option0 (sizeLt n) = let
  fun loop {i:nat | i <= n}
    (str: string n, c0: char, i: size_t i): option0 (sizeLt n) =
    if string_isnot_at_end (str, i) then
      if (c0 = str[i]) then option0_some (i) else loop (str, c0, i+1)
    else option0_none () // end of [if]
  // end of [loop]
in
  loop (str, c0, 0)
end // end of [string_find]
```

If the character c0 occurs in the string str, then a value of the form option0_some(i) is returned, when i refers to the position of the first occurrence of c0 (counting from left to right). Otherwise, the value option0_none() is returned.

There is some inherent inefficiency in the implementation of string_find: A given position i is first checked to see if it is strictly less than the length of the string str by calling string_isnot_at_end, and, if it is, the character stored at the position in the string is fetched by calling string_get_char_at. These two function calls are merged into one function call in the following implementation:

```
//
// This implementation does the same as [string_find]
// but should be more efficient.
//
fun string_find2 {n:nat}
  (str: string n, c0: char): option0 (sizeLt n) = let
  fun loop {i:nat | i <= n} (
    str: string n, c0: char, i: size_t i
  ) : option0 (sizeLt n) = let
    val c = string_test_char_at (str, i)
  in
    if c != '
```

The interface for the function string_test_char_at is given as follows:

```
fun string_test_char_at {n:int}
  {i:nat | i <= n} (str: string n, i: size_t i)
  : [c:char | (c <> NUL && i < n) || (c == NUL && i >= n)] char c
// end of [string_test_char_at]
```

By checking the return value of a call to string_test_char_at, we can readily tell whether the position i is at the end of the string str.

Handling strings safely and efficiently is a complicated matter in programming language design, and a great deal of information about a programming language can often be revealed by simply studying its treatment of strings. In ATS, properly processing C-style strings also makes essential use of linear types, which I will cover in another part of this book.

## Example: Binary Search on Arrays

Given a type T (of the sort *t@ype*) and a static integer I (i.e., a static term of the sort *int*), array(T, I) is a boxed type for arrays of size I in which each stored element is of the type T. Note that such arrays are without size information attached to them.

The following interface is for a function template array_make_elt that can be called to create an array (with no size information attached to it):

```
fun{a:t@ype}
array_make_elt {n:nat} (asz: size_t n, elt: a): array (a, n)
```

Given a static integer I, the type size_t(I) is a singleton type for the value of the type size_t in C that represents the integer equal to I. The function templates for reading from and writing to an array cell have the following interfaces:

```
fun{a:t@ype}
array_get_elt_at {n:int}
  {i:nat | i < n} (A: array (a, n), i: size_t i): a
overload [] with array_get_elt_at

fun{a:t@ype}
array_set_elt_at {n:int}
  {i:nat | i < n} (A: array (a, n), i: size_t i, x: a): void
overload [] with array_set_elt_at
```

Note that these two function templates do not incur any run-time array-bounds checking: The types assigned to them guarantee that each index used for array subscripting is always legal, that is, within the bounds of the array being subscripted.

As a convincing example of practical programming with dependent types, the following code implements the standard binary search algorithms on an ordered array:

```
fun{a:t@ype}
bsearch_arr {n:nat} (
  A: array (a, n), n: int n, x0: a, cmp: (a, a) -> int
) : int = let
  fun loop
    {i,j:int |
     0 <= i; i <= j+1; j+1 <= n} (
    A: array (a, n), l: int i, u: int j
  ) :<cloref1> int =
    if l <= u then let
      val m = l + (u - l) / 2
      val x = A[m]
      val sgn = cmp (x0, x)
    in
      if sgn >= 0 then loop (A, m+1, u) else loop (A, l, m-1)
    end else u // end of [if]
  // end of [loop]
in
  loop (A, 0, n-1)
end // end of [bsearch_arr]
```

The function loop defined in the body of bsearch_arr searches the segment of the array A between the indices l and u, inclusively. Clearly, the type assigned to loop indicates that the integer values i and j of the arguments l and u must satisfy the precondition consisting of the constraints 0 <= i, i <= j+1, and j+1 <= n, where n is the size of the array being searched. The progress we have made by introducing dependent types into ATS should be evident in this example: We can not only specify much more precisely than before but also enforce effectively the enhanced precision in specification.

Please find on-line[4] the code employed for illustration in this section plus some additional code for testing.

## Termination-Checking for Recursive Functions

There is a mechanism in ATS that allows the programmer to supply termination metrics for checking whether recursively defined functions are terminating. It will soon become clear that this mechanism of termination-checking plays a fundamental role in the design of ATS/LF, a theorem-proving subsystem of ATS, where proofs are constructed as total functional programs.

A termination metric is just a tuple of natural numbers and the standard lexicographic ordering on natural numbers is used to order such tuples. In the following example, a singleton metric n is supplied to verify that the recursive function fact is terminating, where n is the value of the integer argument of fact:

```
fun fact {n:nat} .<n>.
  (x: int n): int = if x > 0 then x * fact (x-1) else 1
// end of [fact]
```

Note that the metric attached to the recursive call fact(x-1) is n-1, which is strictly less than the initial metric n. Essentially, termination-checking in ATS verifies that the metric attached to each recursive call in the body of a function is strictly less that the initial metric attached to the function.

A more difficult and also more interesting example is given as follows, where the MacCarthy's 91-function is implemented:

```
fun f91 {i:int} .<max(101-i,0)>. (x: int i)
  : [j:int | (i < 101 && j==91) || (i >= 101 && j==i-10)] int (j) =
  if x >= 101 then x-10 else f91 (f91 (x+11))
// end of [f91]
```

The metric supplied to verify the termination of f91 is max(101-i,0), where i. is the value of the integer argument of f91. Please try to verify manually that this metric suffices for verifying the termination of f91.

As another example, the following code implements the Ackermann's function, which is well-known for being recursive but not primitive recursive:

```
fun acker
  {m,n:nat} .<m,n>.
  (x: int m, y: int n): Nat =
  if x > 0 then
    if y > 0 then acker (x-1, acker (x, y-1)) else acker (x-1, 1)
  else y + 1
// end of [acker]
```

The metric supplied for verifying the termination of acker is a pair (m,n), where m and n are values of the two integer arguments of acker. The metrics attached to the three recursive calls to acker are, from left to right, (m-1,k) for some natural number k, (m,n-1), and (m-1,1). Clearly, these metrics are all strictly less than the initial metric (m,n) according to the lexicographic ordering on pairs of natural numbers.

Termination-checking for mutually recursive functions is similar. In the following example, isevn and isodd are defined mutually recursively:

```
fun isevn {n:nat} .<2*n>.
  (n: int n): bool = if n = 0 then true else isodd (n-1)
and isodd {n:nat} .<2*n+1>.
  (n: int n): bool = not (isevn n)
```

The metrics supplied for verifying the termination of isevn and isodd are 2*n and 2*n+1, respectively, where n is the value of the integer argument of isevn and also the value of the integer argument of isodd. Clearly, if the metrics (n, 0) and (n, 1) are supplied for isevn and isodd, respectively, the termination of these two functions can also be verified. Note that it is required that the metrics for mutually recursively defined functions be tuples of the same length.

## Example: Dependent Types for Debugging

Given an integer x >= 0, the integer square root of x is the greatest integer i satisfying i * i <= x. An implementation of the integer square root function is given as follows based on the method of binary search:

```
fun isqrt (x: int): int = let
  fun search (x: int, l: int, r: int): int = let
    val diff = r - l
  in
    case+ 0 of
    | _ when diff > 0 => let
        val m = l + (diff / 2)
      in
        // x < m * m is more efficient but can overflow easily
        if x / m < m then search (x, l, m) else search (x, m, r)
      end // end of [if]
    | _ => l (* the result is found *)
  end // end of [search]
in
  search (x, 0, x)
end // end of [isqrt]
```

This implementation passes typechecking, but it seems to be looping forever when tested. Instead of going into the standard routine of debugging (e.g., by inserting calls to some printing functions), let us attempt to identify the cause for infinite looping by proving the termination of the function search through the use of dependent types. Clearly, the function search is assigned the function type (int, int, int) -> int, meaning that search takes three integers as its arguments and returns an integer as its result, and there is not much else that can be gathered from a non-dependent type as such. However, the programmer may have thought that the function search should possess the following invariants (if implemented correctly):

- l * l <= x and x <= r * r must hold when search(x, l, r) is called.
- Assume l * l <= x < r * r for some integers x, l, r. If a recursive call search(x, l1, r1) for some integers l1 and r1 is encountered in the body of search(x, l, r), then r1-l1 < r-l must hold. This invariant implies that search is terminating.

Though the first invariant can be captured in the type system of ATS, it is somewhat involved to do so due to the need for handling nonlinear constraints. Instead, we try to assign search the following dependent function type:

```
{x:nat} {l,r:nat | l < r} .<r-l>. (int(x), int(l), int(r)) -> int
```

which captures a weaker invariant stating that l < r must hold when search(x, l, r) is called. The termination metric .<r-l>. is provided for checking that the function search is terminating. When we assign search the dependent function type, we have to modify its body as certain errors are otherwise reported during typechecking. The following code we obtain after proper modification does pass typechecking:

```
fun isqrt {x:nat}
  (x: int x): int = let
  fun search
    {x,l,r:nat | l < r} .<r-l>.
```

```
    (x: int x, l: int l, r: int r): int = let
    val diff = r - l
  in
    case+ 0 of
    | _ when diff > 1 => let
        val m = l + (diff / 2)
      in
        if x / m < m then search (x, l, m) else search (x, m, r)
      end // end of [if]
    | _ => l (* the result is found *)
  end // end of [search]
in
  if x > 0 then search (x, 0, x) else 0
end // end of [isqrt]
```

It is now rather clear that infinite looping in the previous implementation of search may happen if search(x, l, r) is called in a situaltion where r-l equals 1 as this call can potentially lead to another call to search of the same arguments. However, such a call leads to a type-error after search is assigned the aforementioned dependent function type.

By being precise and being able to enforce precision effectively, the programmer will surely notice that his or her need for run-time debugging is diminishing rapidly.

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DEPTYPES/
2. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/sortdef.sats
3. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/integer.sats
4. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DEPTYPES/bsearch_a

# Chapter 14. Datatype Refinement

The datatype mechanism in ATS is adopted from ML directly, and it is really a signatory feature in functional programming. However, the datatypes we have seen so far are largely imprecise when employed to classify values. For instance, given a type T, the type list0(T) is for values representing both empty and non-empty lists consisting of elements of the type T. Therefore, empty and non-empty lists cannot be distinguished at the level of types. This limitation severely diminishes the effectiveness of datatypes of ML-style in capturing program invariants. In ATS, datatypes of ML-style can be refined into dependent datatypes of DML-style, where DML refers to the programming language Dependent ML, the immediate precursor of ATS. With such refinement, datatypes can classify values with greatly enhanced precision.

The code employed for illustration in this chapter plus some additional code for testing is available on-line[1].

## Dependent Datatypes

The syntax for declaring dependent datatypes is mostly similar to the syntax for declaring non-dependent datatypes: For instance, the dependent datatype list in ATS is declared as follows:

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

More precisely, list is declared as a type constructor of the sort *(t@ype, int) -> type*, which means that list takes an unboxed type and a static integer to form a boxed type. The keyword t@ype+ indicates that list is covariant at its first parameter (of the sort *t@ype*), that is, list(T1, I) is considered a subtype of list(T2, I) if T1 is a subtype of T2. There is also the keyword t@ype- for indicating the declared type constructor being contravariant at a parameter, but it is rarely used in practice. Keywords like type+ and type- are interpreted similarly.

There two data (or value) constructors list_nil and list_cons associated with list, which are declared to be of the following types:

```
list_nil : {a:t@ype} () -> list(a, 0)
list_cons : {a:t@ype} {n:nat} (a, list(a, n)) -> list(a, n+1)
```

Given a type T and a static integer I, the type list(T, I) is for values representing lists of length I in which each element is of the type T. Hence, the types of list_nil and list_cons mean that list_nil forms a list of length 0 and list_cons forms a list of length n+1 if applied to an element and a list of length n. Note that it is also possible to declare list as follows in a more concise style:

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

The use of a:t@ype+ (instead of t@ype+) introduces implicitly a universal quantifier over a for the type assigned to each data constructor associated with the declared type constructor list.

As an example of programming with dependent datatypes, the following code implements a function template for computing the length of a given list:

```
fun{a:t@ype}
list_length {n:nat} .<n>.
  // .<n>. is a termination metric
```

```
  (xs: list (a, n)): int (n) = case+ xs of
  | list_cons (_, xs1) => 1 + list_length (xs1)
  | list_nil () => 0
// end of [list_length]
```

The type assigned to the function list_length indicates that the function takes a list of length n for any natural number n and returns an integer of value n. In addition, the function is verified to be terminating. Therefore, list_length is guaranteed to implement the function that computes the length of a given list. I now briefly explain how typechecking can be performed on the definition of list_length. Let us first see that the the following clause typechecks:

```
  | list_cons (_, xs1) => 1 + list_length (xs1)
```

What we need to verify is that the expression on the righthand side of the symbol => can be assigned the type int(n) under the assumption that xs matches the pattern on the lefthand side of the symbol =>. Let us assume that xs1 is of the type list(a, n1) for some natural number n1, and this assumption implies that the pattern list_cons(_, xs1) is of the type list(a, n1+1). Clearly, matching xs against the pattern list_cons(_, xs1) generates a condition n=n1+1. It is also clear that list_length(xs1) is of the type int(n1) and thus 1 + list_length(xs1) is of the type int(1+n1). As the condition n=n1+1 implies n=1+n1, 1 + list_length(xs1) can be given the type int(n). So this clause typechecks. Note that matching xs against the pattern list_nil() generates the assumption n=0, which implies that 0 is of the type int(n). Therefore, the following clause typechecks:

```
  | list_nil () => 0
```

Given that the two clauses typecheck properly, the case-expression containing them can be assigned the type int(n). Therefore, the definition of list_length typechecks.

As the recursive call in the body of the above defined function list_length is not a tail-call, the function may not be able to handle a long list (e.g., one that contains 1 million elements). The following code gives another implementation for computing the length of a given list:

```
fun{a:t@ype}
list_length {n:nat} .<>.
  (xs: list (a, n)): int (n) = let
  // loop: {i,j:nat} (list (a, i), int (j)) -> int (i+j)
  fun loop {i,j:nat} .<i>.
    // .<i>. is a termination metric
    (xs: list (a, i), j: int j): int (i+j) = case+ xs of
    | list_cons (_, xs1) => loop (xs1, j+1) | list_nil () => j
  // end of [loop]
in
  loop (xs, 0)
end // end of [list_length]
```

This time, list_length is based on a tail-recursive function loop and thus can handle lists of any length in constant stack space. Note that the type assigned to loop indicates that loop takes a list of length i and an integer of value j for some natural numbers i and j and returns an integer of value i+j. Also, loop is verified to be terminating.

There is also a dependent datatype option in ATS that corresponds to option0:

```
datatype
option (a:t@ype+, bool) =
  | Some (a, true) of a | None (a, false) of ()
// end of [option]
```

As an example, the following function template list_last tries to find the last element in a given list:

```
fn{a:t@ype}
list_last
  {n:nat} (
  xs: list (a, n)
) : option (a, n > 0) = let
  fun loop {n:pos} .<n>.
    (xs: list (a, n)): a = let
    val+ list_cons (_, xs1) = xs
  in
    case+ xs1 of
    | list_cons _ => loop (xs1)
    | list_nil () => let
        val+ list_cons (x, _) = xs in x
      end // end of [list_nil]
  end // end of [loop]
in
  case+ xs of
  | list_cons _ => Some (loop (xs)) | list_nil () => None ()
end // end of [list_last]
```

The inner function loop is evidently tail-recursive and it is verified to be terminating.

After a programmer becomes familar with list and option, there is little incentive for him or her to use list0 and option0 anymore. Internally, values of list and list0 have exactly the same representation and there are cast functions of zero run-time cost in ATS for translating between values of list and list0. The same applies to values of list0 and option0 as well.

## Example: Function Templates on Lists (Redux)

I have presented previously implementation of some commonly used function templates on lists formed with the constructors list0_nil and list0_cons. This time, I present as follows implementation of the corresponding function templates on lists formed with the constructors list_nil and list_cons, which make it possible to assign more accurate types to these templates.

Please find the entire code in this section plus some additional code for testing on-line[2].

### Appending: list_append

Given two lists xs and ys of the types list(T, I1) and list(T, I2) for some type T and integers I1 and I2, list_append(xs, ys) returns a list of the type list(T, I1+I2) that is the concatenation of xs and ys:

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

Clearly, this implementation of list_append is not tail-recursive.

## Reverse Appending: list_reverse_append

Given two lists xs and ys of the type list(T, I1) and list(T, I2) for some type T and integers I1 and I2, list_reverse_append(xs, ys) returns a list of the type list(T, I1+I2) that is the concatenation of the reverse of xs and ys:

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

Clearly, this implementation of list_reverse_append is tail-recursive.

## Reversing: list_reverse

Given a list xs, list_reverse(xs) returns the reverse of xs, which is of the same length as xs:

```
fun{a:t@ype}
list_reverse {n:nat} .<>. // defined non-recursively
  (xs: list (a, n)): list (a, n) = list_reverse_append (xs, list_nil)
// end of [list_reverse]
```

## Mapping: list_map

Given a list xs of the type list(T1, I) for some type T1 and integer I and a closure function f of the type T1 -<cloref1> T2 for some T2, list_map(xs) returns a list ys of the type list(T2, I):

```
fun{a:t@ype}{b:t@ype}
list_map {n:nat} .<n>. (
  xs: list (a, n), f: a -<cloref1> b
) : list (b, n) = case+ xs of
  | list_cons (x, xs) => list_cons (f x, list_map (xs, f))
  | list_nil () => list_nil ()
// end of [list_map]
```

Each element y in ys equals f(x), where x is the corresponding element in xs. Clearly, this implementation of list_map is not tail-recursive.

## Zipping: list_zip

Given two lists xs and ys of the types list(T1, I) and list(T2, I) for some types T1 and T2 and integer I, respectively, list_zip(xs, ys) returns a list zs of the type list((T1, T2), I):

```
fun{a,b:t@ype}
list_zip {n:nat} .<n>. (
  xs: list (a, n), ys: list (b, n)
) : list ((a, b), n) = case+ (xs, ys) of
  | (list_cons (x, xs),
```

```
     list_cons (y, ys)) => list_cons ((x, y), list_zip (xs, ys))
  | (list_nil (), list_nil ()) => list_nil ()
// end of [list_zip]
```

Each element z in zs equals the pair (x, y), where x and y are the corresponding elements in xs and ys, respectively. Clearly, this implementation of list_zip is not tail-recursive.

### Zipping with: list_zipwith

Given two lists xs and ys of the types list(T1, I) and list(T2, I) for some types T1 and T2 and integer I, respectively, and a closure function f of the type (T1, T2) -<cloref1> T3 for some type T3, list_zipwith(xs, ys, f) returns a list zs of the type list(T3, I):

```
fun{a,b:t@ype}{c:t@ype}
list_zipwith
  {n:nat} .<n>. (
  xs: list (a, n)
, ys: list (b, n)
, f: (a, b) -<cloref1> c
) : list (c, n) = case+ (xs, ys) of
  | (list_cons (x, xs), list_cons (y, ys)) =>
      list_cons (f (x, y), list_zipwith (xs, ys, f))
  | (list_nil (), list_nil ()) => list_nil ()
// end of [list_zipwith]
```

Each element z in zs equals f(x, y), where x and y are the corresponding elements in xs and ys, respectively. Clearly, this implementation of list_zipwith is not tail-recursive.

## Example: Mergesort on Lists (Redux)

I have previously presented an implementation of mergesort on lists that are formed with the constructors list0_nil and list0_cons. In this section, I give an implementation of mergesort on lists formed with the constructors list_nil and list_cons. This implementation is based on the same algorithm as the previous one but it assigns a type to the implemented sorting function that guarantees the function to be length-preserving, that is, the function always returns a list of the same length as the list it sorts.

The following defined function merge combines two ordered list (according to a given ordering) into a single ordered list:

```
typedef lte (a:t@ype) = (a, a) -> bool

fun{a:t@ype}
merge {m,n:nat} .<m+n>. (
  xs: list (a, m), ys: list (a, n), lte: lte a
) : list (a, m+n) =
  case+ xs of
  | list_cons (x, xs1) => (
    case+ ys of
    | list_cons (y, ys1) =>
        if x \lte y then
          list_cons (x, merge (xs1, ys, lte))
        else
          list_cons (y, merge (xs, ys1, lte))
        // end of [if]
    | list_nil () => xs
    ) // end of [list_cons]
```

```
  | list_nil () => ys
// end of [merge]
```

Clearly, the type assigned to merge indicates that the function returns a list whose length equals the sum of the lengths of the two input lists. Note that a termination metric is present for verifying that merge is a terminating function.

The following defined function mergesort mergesorts a list according to a given ordering:

```
fun{a:t@ype}
mergesort {n:nat} (
  xs: list (a, n), lte: lte a
) : list (a, n) = let
  fun msort {n:nat} .<n,n>. (
    xs: list (a, n), n: int n, lte: lte a
  ) : list (a, n) =
    if n >= 2 then split (xs, n, lte, n/2, list_nil) else xs
  // end of [msort]
  and split
    {n:int | n >= 2} {i:nat | i <= n/2} .<n,i>. (
    xs: list (a, n-n/2+i)
  , n: int n, lte: lte a, i: int i, xsf: list (a, n/2-i)
  ) : list (a, n) =
    if i > 0 then let
      val+ list_cons (x, xs) = xs
    in
      split (xs, n, lte, i-1, list_cons (x, xsf))
    end else let
      val xsf = list_reverse<a> (xsf) // make sorting stable!
      val xsf = list_of_list_vt (xsf) // linear list -> nonlinear list
      val xsf = msort (xsf, n/2, lte) and xs = msort (xs, n-n/2, lte)
    in
      merge (xsf, xs, lte)
    end // end of [if]
  // end of [split]
  val n = list_length<a> (xs)
in
  msort (xs, n, lte)
end // end of [mergesort]
```

The type assigned to mergesort indicates that mergesort returns a list of the same length as its input list. The two inner functions msort and split are defined mutually recursively, and the termination metrics for them are pairs of natural numbers that are compared according to the standard lexicographic ordering on integer sequences. The type assigned to msort clearly indicates that its integer argument is required to be the length of its list argument, and a mismatch between the two surely results in a type-error. The type assigned to split is particularly informative, depicting a relation between the arguments and the return value of the function that can be of great help for someone trying to understand the code. The function list_reverse returns a linear list that is the reverse of its input, and the cast function list_of_list_vt turns a linear list into a nonlinear one (while incuring no cost at run-time). I will explain what linear lists are elsewhere.

Please find the entire code in this section plus some additional code for testing online[3].

## Sequentiality of Pattern Matching

In ATS, pattern matching is performed sequentially at run-time. In other words, a clause is selected only if a given value matches the pattern guard associated with this clause but the value fails to match the pattern associated with any clause ahead of it. Naturally, one may expect that the following implementation of list_zipwith also typechecks:

```
fun{a1,a2:t@ype}{b:t@ype}
list_zipwith {n:nat} (
  xs1: list (a1, n)
, xs2: list (a2, n)
, f: (a1, a2) -<cloref1> b
) : list (b, n) =
  case+ (xs1, xs2) of
  | (list_cons (x1, xs1), list_cons (x2, xs2)) =>
      list_cons (f (x1, x2), list_zipwith (xs1, xs2, f))
  | (_, _) => list_nil ()
// end of [list_zipwith]
```

This, however, is not the case. In ATS, typechecking clauses is done nondeterministically (rather than sequentially). In this example, the second clause fails to typecheck as it is done without the assumption that the given value pair (xs1, xs2) fails to match the pattern guard associated with the first clause. The second clause can be modified slightly as follows to pass typechecking:

```
  | (_, _) =>> list_nil ()
```

The use of the symbol =>> (in place of =>) indicates to the typechecker that this clause needs to be typechecked under the sequentiality assumption that the given value that matches it does not match the pattern guards associated with any previous clauses. Therefore, when the modified second clause is typechecked, it can be assumed that the value pair (xs1, xs2) matching the pattern (_, _) must match one of the following three patterns:

- (list_cons (_, _), list_nil ())
- (list_nil (), list_cons (_, _))
- (list_nil (), list_nil ())

Given that xs1 and xs2 are of the same length, the typechecker can readily infer that (xs1, xs2) cannot match either of the first two patterns. After these two patterns are ruled out, typechecking is essentially done as if the second clause was written as follows:

```
  | (list_nil (), list_nil ()) => list_nil ()
```

One may be wondering why typechecking clauses is not required to be done sequentially by default. The simple reason is that this requirement, if fully enforced, can have a great negative impact on the efficiency of typechecking. Therefore, it is a reasonable design to provide the programmer with an explict means to occasionally make use of the sequentiality assumption needed for typechecking a particular clause.

## Example: Functional Random-Access Lists

The data structure I implement in this section is based on one presented in the book titled *Pure Funtional Data Structures* by Chris Okasaki, where more elaboration on data structures of this kind can be found.

Let us first declare a datatype as follows:

```
typedef pt (a:t@ype) = '(a, a)

datatype
ralist (a:t@ype+, n:int) =
  | RAnil (a, 0)
  | {n:pos} RAevn (a, 2*n) of ralist (pt a, n)
  | {n:nat} RAodd (a, 2*n+1) of (a, ralist (pt a, n))
// end of [ralist]
```

Given an unboxed type T and a static integer, the type ralist(T, I) is a boxed type for values representing lists of length I. The meaning of the three constructors RAnil, RAevn and RAodd can be briefly explained as follows:

- The constructor RAnil is for constructing a value representing the empty list.
- In order to construct a value representing a list of 2*I elements for some I > 0, we first construct a value representing a list of I pairs of adjacent elements in the (original) list and then apply the constructor RAevn to the value.
- In order to construct a value representing a list of 2*I+1 elements for some I >= 0, we take out the head of the list and construct a value representing a list of I pairs of adjacent elements in the tail of the (original) list and then apply the constructor RAodd to the head element and the value.

For example, the list of the first 5 positive integers is represented by the following value:

```
RAodd(1, RAevn(RAodd( '( '(2, 3), '(4, 5) ), RAnil())))
```

Values constructed by the constructors RAnil, RAevn and RAodd represent lists that support operations of logrithmic time for accessing and updating list elements, and such lists are often referred to as random-access lists.

Note that the datatype ralist is not supported in ML even if the index representing list length is erased. This kind of datatypes are often referred to as nested datatypes, which are also supported in Haskell.

The following defined function ralist_length computes the length of a random-access list:

```
fun{a:t@ype}
ralist_length // O(log n)
  {n:nat} .<n>.
  (xs: ralist (a, n)): int n =
  case+ xs of
  | RAnil () => 0
  | RAevn xxs => 2 * ralist_length<pt(a)> (xxs)
  | RAodd (_, xxs) => 2 * ralist_length<pt(a)> (xxs) + 1
// end of [ralist_length]
```

While the implementation of ralist_length is clearly not tail-recursive, this is hardly of any practical concern as the time-complexity of ralist_length is O(log(n)).

Consing means to form a new list with an element and a list such that the element and the list are the head and tail of the new list. The following defined function ralist_cons implements consing for random-access lists:

```
fun{a:t@ype}
ralist_cons // O(1), amortized
  {n:nat} .<n>.
  (x0: a, xs: ralist (a, n)): ralist (a, n+1) =
  case+ xs of
  | RAnil () => RAodd (x0, RAnil)
```

```
  | RAevn (xxs) => RAodd (x0, xxs)
  | RAodd (x1, xxs) =>
      RAevn (ralist_cons<pt(a)> ( '(x0, x1), xxs ))
    // end of [RAodd]
// end of [ralist_cons]
```

In the worst case (where the length of xs is a power of 2 minus 1), ralist_cons takes O(log(n))-time to finish. However, it can be proven that the amortized time-complexity of consing on random-access lists is O(1).

Unconsing does the opposite of consing: It returns a pair consisting of the head and tail of a given non-empty list. The following defined function ralist_uncons implements unconsing for random-access lists:

```
fun{a:t@ype}
ralist_uncons // O(1), amortized
  {n:pos} .<n>.
  (xs: ralist (a, n)): (a, ralist (a, n-1)) =
  case+ xs of
  | RAevn (xxs) => let
      val (xx, xxs) = ralist_uncons<pt(a)> (xxs)
    in
      (xx.0, RAodd (xx.1, xxs))
    end // end of [RAevn]
  | RAodd (x, xxs) => (case+ xxs of
(*
// Note: [=>>] is needed for enforcing sequentiality
// during typechecking:
*)
      RAnil () => (x, RAnil) | _ =>> (x, RAevn (xxs))
    ) // end of [RAodd]
// end of [ralist_uncons]
```

Like ralist_cons, ralist_uncons takes O(log(n))-time to finish when the length of xs is a power of 2. However, its amortized time-complexity is also O(1). It is highly probable for a programmer to implement the second matching clause in the body of ralist_uncons as follows:

```
  | RAodd (x, xxs) => (x, RAevn (xxs))
```

For instance, I myself did this. This is a program error as the invariant can potentially be broken that states RAevn being only applied to a value representing a non-empty list. The error is readily caught by the typechecker of ATS but it is most likely to go unnoticed in a setting where the invariant on the constructor RAevn can not be captured at compile-time.

Given a random-access list xs of length n, the elements in it are numbered from 0 to n-1, inclusively. We can find element i in the list xs, where i is assumed to be a natural number less than n, by implementing the following algorithm:

- Assume the length n is even. Then xs is of the form RAevn(xxs), where xxs is a list of pairs. Let i2 be i/2 and we find element i2 in xxs, which is a pair. Let xx refer to this pair. If i is even, then the left component of xx is element i in xs. Otherwise, the right component is.
- Assume the length n is odd. Then xs is of the form RAodd(x, xxs), where xxs is a list of pairs. If i equals 0, the x is element i in xs. Otherwise, let i1 be i-1 and i2 be i1/2 and we find element i2 in xxs, which is a pair. Let xx refer to this pair. If i1 is even, then the left component of xx is element i in xs. Otherwise, the right component is.

The following function ralist_lookup is implemented precisely according to the given algorithm:

```
fun{a:t@ype}
ralist_lookup // O(log(n))-time
  {n:int} {i:nat | i < n} .<n>.
  (xs: ralist (a, n), i: int i): a =
  case+ xs of
  | RAevn xxs => let
      val i2 = i / 2
      val lr = i - 2 * i2
      val xx = ralist_lookup<pt(a)> (xxs, i2)
    in
      if lr = 0 then xx.0 else xx.1
    end // end of [RAevn]
  | RAodd (x, xxs) =>
      if i > 0 then let
        val i1 = i - 1
        val i2 = i1 / 2
        val lr = i1 - 2 * i2
        val xx = ralist_lookup<pt(a)> (xxs, i2)
      in
        if lr = 0 then xx.0 else xx.1
      end else x
    // end of [RAodd]
// end of [ralist_lookup]
```

Clearly, the time complexity of ralist_lookup is O(log(n)).

Given a random-access list xs of length n, an index that is a natural number less than n and an element x0, the following defined function ralist_update returns another random-access that is the same as xs except element i in it being x0:

```
fun{a:t@ype}
ralist_update // O(log(n))-time
  {n:int} {i:nat | i < n} .<n>. (
    xs: ralist (a, n), i: int i, x0: a
  ) : ralist (a, n) = let
//
  fun{a:t@ype} fupdate
    {n:int} {i:nat | i < n} .<n,1>. (
      xs: ralist (a, n), i: int i, f: a -<cloref1> a
    ) : ralist (a, n) =
    case+ xs of
    | RAevn xxs =>
        RAevn (fupdate2 (xxs, i, f))
    | RAodd (x, xxs) =>
        if i > 0 then
          RAodd (x, fupdate2 (xxs, i-1, f))
        else RAodd (f(x), xxs)
   (* end of [fupdate] *)
//
   and fupdate2
     {n2:int} {i:nat | i < n2+n2} .<2*n2,0>. (
       xxs: ralist (pt(a), n2), i: int i, f: a -<cloref1> a
     ) : ralist (pt(a), n2) = let
     val i2 = i / 2
     val lr = i - 2 * i2
     val f2 = (
       if lr = 0 then
         lam xx => '(f(xx.0), xx.1)
       else
         lam xx => '(xx.0, f(xx.1))
     ) : pt(a) -<cloref1> pt(a)
   in
     fupdate<pt(a)> (xxs, i2, f2)
   end // end of [fupdate2]
//
in
```

```
  fupdate (xs, i, lam _ => x0)
end // end of [ralist_update]
```

Note that the functions fupdate and fupdate2 are higher-order. Given a random-access list xs of length n, an index i that is a natural number less than n and a (closure) function f, fupdate returns another random-access list that is the same as xs except element i in it being f(x), where x is element i in xs. It is straightforward to see that the time-complexity of ralist_update is O(log(n)). I leave the reader to figure out further details on this interesting implementation.

The code employed for illustration in this section plus some additional code for testing is available on-line[4].

## Example: Functional Red-Black Trees

A red-black tree is defined as a binary tree such that each node in it is colored red or black and every path from the root to a leaf has the same number of black nodes while containing no occurrences of two red nodes in a row. Clearly, the length of a longest path in each red-black tree is bounded by 2 times the length of a shortest path in it. Therefore, red-black trees are a family of balanced trees. The number of black nodes occurring on each path in a red-black tree is often referred to as the *black height* of the tree.

Formally, a datatype precisely for red-black trees can be declared in ATS as follows:

```
#define BLK 0
#define RED 1
sortdef clr = {c:nat | c <= 1}

datatype rbtree
  (a:t@ype+, int(*clr*), int(*bh*)) =
  | {c,cl,cr:clr | cl <= 1-c; cr <= 1-c} {bh:nat}
    rbtree_cons (a, c, bh+1-c) of (int c, rbtree (a, cl, bh), a, rbtree (a, cr, bh))
  | rbtree_nil (a, BLK, 0)
// end of [rbtree]
```

The color of a tree is the color of its root node or is black if the tree is empty. Given a type T, a color C (represented by a integer) and an integer BH, the type rbtree(T, C, BH) is for red-black trees carrying elements of the type T that is of the color C and the black height BH.

When implementing various operations (such as insertion and deletion) on a red-black tree, we often need to first construct intermediate trees that contain color violations caused by a red node being followed by another red node and then employ tree rotations to fix such violations. This need makes the above datatype rbtree too rigid as it cannot be assigned to any intermediate trees containing color violations. To address this issue, we can declare rbtree as follows:

```
datatype
rbtree (
  a:t@ype+, int(*clr*), int(*bh*), int(*v*)
) = // element type, color, black height, violations
  | rbtree_nil (a, BLK, 0, 0) of ()
  | {c,cl,cr:clr} {bh:nat} {v:int}
      {c==BLK && v==0 || c == RED && v==cl+cr}
    rbtree_cons (a, c, bh+1-c, v) of (
      int c, rbtree0 (a, cl, bh), a, rbtree0 (a, cr, bh)
    ) // end of [rbtree_cons]
// end of [rbtree]

where rbtree0 (a:t@ype, c:int, bh:int) = rbtree (a, c, bh, 0)
```

We count each occurrence of two red nodes in a row as one color violation. Given a type T, a color C (represented by a integer), an integer BH and an integer V, the type rbtree(T, C, BH, V) is for trees carrying elements of the type T that is of the color C and the black height BH and contains exactly V color violations. Therefore, the type rbtree(T, C, BH, 0) is for valid red-black trees (containing no color violations).

Given a tree containing at most one color violation, an element and another tree containing no violations, the following operation constructs a valid red-black tree:

```
fn{a:t@ype}
insfix_l // right rotation for fixing left insertion
  {cl,cr:clr} {bh:nat} {v:nat} (
  tl: rbtree (a, cl, bh, v), x0: a, tr: rbtree (a, cr, bh, 0)
) : [c:clr] rbtree0 (a, c, bh+1) = let
  #define B BLK; #define R RED; #define cons rbtree_cons
in
  case+ (tl, x0, tr) of
  | (cons (R, cons (R, a, x, b), y, c), z, d) =>
      cons (R, cons (B, a, x, b), y, cons (B, c, z, d)) // shallow rot
  | (cons (R, a, x, cons (R, b, y, c)), z, d) =>
      cons (R, cons (B, a, x, b), y, cons (B, c, z, d)) // deep rotation
  | (a, x, b) =>> cons (B, a, x, b)
end // end of [insfix_l]
```

By simply reading the interface of insfix_l, we can see that the two tree arguments are required to be of the same black height bh for some natural number bh and the returned tree is of the black height bh+1.

The following operation insfix_r is just the mirror image of insfix_l:

```
fn{a:t@ype}
insfix_r // left rotation for fixing right insertion
  {cl,cr:clr} {bh:nat} {v:nat} (
  tl: rbtree (a, cl, bh, 0), x0: a, tr: rbtree (a, cr, bh, v)
) : [c:clr] rbtree0 (a, c, bh+1) = let
  #define B BLK; #define R RED; #define cons rbtree_cons
in
  case+ (tl, x0, tr) of
  | (a, x, cons (R, b, y, cons (R, c, z, d))) =>
      cons (R, cons (B, a, x, b), y, cons (B, c, z, d)) // shallow rot
  | (a, x, cons (R, cons (R, b, y, c), z, d)) =>
      cons (R, cons (B, a, x, b), y, cons (B, c, z, d)) // deep rotation
  | (a, x, b) =>> cons (B, a, x, b)
end // end of [insfix_r]
```

The preparation for implementing insertion on a red-black tree is all done by now, and we are ready to see an implementation of insertion guaranteeing that the tree obtained from inserting an element into a given red-black tree is always a valid red-black tree itself. This guarantee is precisely captured in the following interface for insertion:

```
extern
fun{a:t@ype}
rbtree_insert
  {c:clr} {bh:nat} (
  t: rbtree0 (a, c, bh), x0: a, cmp: cmp a
) : [bh1:nat] rbtree0 (a, BLK, bh1)
```

Interestingly, this interface also implies that the tree returned by a call to rbtree_insert is always black. The code presented below gives an implementation of rbtree_insert:

```
implement{a}
rbtree_insert
```

```
  (t, x0, cmp) = let
  #define B BLK; #define R RED
  #define nil rbtree_nil; #define cons rbtree_cons
  fun ins
    {c:clr} {bh:nat} .<bh,c>. (
    t: rbtree0 (a, c, bh), x0: a
  ) :<cloref1> [cl:clr; v:nat | v <= c] rbtree (a, cl, bh, v) =
    case+ t of
    | cons (c, tl, x, tr) => let
        val sgn = compare (x0, x, cmp)
      in
        if sgn < 0 then let
          val [cl,v:int] tl = ins (tl, x0)
        in
          if c = B then insfix_l (tl, x, tr)
            else cons {..}{..}{..}{cl} (R, tl, x, tr)
          // end of [if]
        end else if sgn > 0 then let
          val [cr,v:int] tr = ins (tr, x0)
        in
          if c = B then insfix_r (tl, x, tr)
            else cons {..}{..}{..}{cr} (R, tl, x, tr)
          // end of [if]
        end else t // end of [if]
      end // end of [cons]
    | nil () => cons {..}{..}{..}{0} (R, nil, x0, nil)
  // end of [ins]
  val t = ins (t, x0)
in
  case+ t of cons (R, tl, x, tr) => cons (B, tl, x, tr) | _ =>> t
end // end of [rbtree_insert]
```

Note that the type assigned to the inner function ins is so informative that it literally gives an formal explanation about the way in which insertion works on a red-black tree. Many a programmer implements red-black trees by simply following an algorithm written in some format of pseudo code while having little understanding about the innerworkings of the algorithm. For instance, if the above inner function ins is implemented in C, few programmers are likely to see that the function always maintain the black height of a given red-black tree after insertion but may introduce one color violation if the root of the tree is red. On the other hand, knowing this invariant is essential to gaining a thorough understanding of the insertion algorithm on red-black trees.

The insertion operation implemented above does not insert an element if it is already in the given red-black tree. It may be desirable to require that the operation inform the caller if such a case occurs. For instance, an exception can be raised for this purpose. An alternative is to give rbtree_insert a call-by-reference argument so that a flag can be returned in it to indicate whether the element to be inserted is actually inserted. I will explain elsewhere what call-by-reference is and how it is supported in ATS.

Often deleting an element from a binary search tree is significantly more difficult to implement than inserting one. This is especially so in the case of a red-black tree. I refer the interested reader to the libats library of ATS for some code implementing a deletion operation on red-black trees that can guarantee based on types each tree returned by the operation being a valid red-black tree (containing no color violations).

Please find the entire code in this section plus some additional code for testing online[5].

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DEPREFDTS/
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DEPREFDTS/listfun.d

3. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DEPREFDTS/merges
4. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DEPREFDTS/ralist.da
5. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DEPREFDTS/rbtree.d

# Chapter 15. Theorem-Proving in ATS/LF

Within the ATS programming langauge system, there is a component named ATS/LF for supporting (interactive) therorem-proving. In ATS/LF, theorem-proving is done by constructing proofs as total functional programs. It will soon become clear that this style of theorem-proving can be combined cohesively with functional programming to yield a programming paradigm that is considered the signature of ATS: *programming with theorem-proving*. Moreover, ATS/LF can be employed to encode various deduction systems and their meta-properties.

Please find on-line[1] the code employed for illustration in this chapter plus some additional code for testing.

## Encoding Relations as Dataprops

In the statics of ATS, there is a built-in sort *prop* for static terms that represent types for proofs. A static term of the sort *prop* can also be referred to as a type or more accurately, a prop-type or just a prop. A dataprop can be declared in a manner that is mostly similar to the declaration of a datatype. For instance, a prop construct FIB is introduced in the following dataprop declaration:

```
dataprop FIB (int, int) =
  | FIB0 (0, 0) of () // [of ()] can be dropped
  | FIB1 (1, 1) of () // [of ()] can be dropped
  | {n:nat} {r0,r1:nat}
    FIB2 (n+2, r0+r1) of (FIB (n, r0), FIB (n+1, r1))
// end of [FIB]
```

The sort assigned to FIB is *(int, int) -> prop*, indicating that FIB takes two static integers to form a prop-type. There are three data (or proof) constructors associated with FIB: FIB0, FIB1 and FIB2, which are assigned the following function types (or more accurately, prop-types):

- FIB0: () -> FIB (0, 0)
- FIB1: () -> FIB (1, 1)
- FIB2: {n:nat} {r0,r1:int} (FIB(n, r0), FIB(n+1, r1)) -> FIB(n+2, r0+r1)

Given a natural number n and an integer r, it should be clear that FIB(n, r) encodes the relation fib(n) = r, where fib is defined by the following three equations:

- fib(0) = 0, and
- fib(1) = 1, and
- fib(n+2) = fib(n+1) + fib(n) for n >= 2.

A proof value of the prop FIB(n, r) can be constructed if and only if fib(n) equals r. For instance, the proof value FIB2(FIB0(), FIB1()) is assigned the prop FIB(2, 1), attesting to fib(2) equaling 1.

As another example of dataprop, the following declaration introduces a prop constructor MUL together with three associated proof constructors:

```
dataprop MUL (int, int, int) =
  | {n:int} MULbas (0, n, 0) of ()
  | {m:nat} {n:int} {p:int}
    MULind (m+1, n, p+n) of MUL (m, n, p)
  | {m:pos} {n:int} {p:int}
    MULneg (~m, n, ~p) of MUL (m, n, p)
```

```
// end of [MUL]
```

Given three integers m, n and p, the prop MUL(m, n, p) encodes the relation m*n = p. As for MULbas, MULind and MULneg, they correspond to the following three equations, respectively:

- 0*n = 0 for every integer n, and
- (m+1)*n = m*n + n for each pair of integers m and n, and
- (~m)*n = ~(m*n) for each pair of integers m and n.

In other words, the dataprop declaration for MUL encodes the relation that represents the standard multiplication function on integers.

It can be readily noticed that the process of encoding a functional relation (i.e., a relation representing a function) as a dataprop is analogous to implementing a function in a logic programming language such as Prolog.

## Constructing Proofs as Total Functions

Theorems are represented as types (or more accurately, props) in ATS/LF. For instance, the following prop states that integer multiplication is commutative:

```
{m,n:int} {p:int} MUL (m, n, p) -<prf> MUL (n, m, p)
```

Constructing a proof for a theorem in ATS/LF means implementing a total value (which is most likely to be a total function) of the type that is the encoding of the theorem in ATS/LF, where being total means being pure and terminating. Please note that this style of theorem-proving may seem rather peculiar to those who have never tried it before.

As a simple introductory example, let us first construct a proof function in ATS/LF that is given the following interface:

```
prfun mut_istot {m,n:int} (): [p:int] MUL (m, n, p)
```

The keyword prfun indicates that the interface is for a proof function (in contrast to a non-proof function). Note that mul_istot is declared to be of the following type (or more accurately, prop):

```
{m,n:int} () -<prf> [p:int] MUL (m, n, p)
```

which essentially states that integer multiplication is a total function: Given any two integers m and n, there exists an integer p such that m, n and p are related according to the structurally inductively defined relation MUL. The following code gives an implementation of mul_istot:

```
implement
mul_istot {m,n} () = let
  prfun istot
    {m:nat;n:int} .<m>. (): [p:int] MUL (m, n, p) =
    sif m > 0 then MULind (istot {m-1,n} ()) else MULbas ()
  // end of [istot]
in
  sif m >= 0 then istot {m,n} () else MULneg (istot {~m,n} ())
end // end of [mul_istot]
```

Note that the inner proof function istot encodes a proof showing that there exists an integer p for any given natural number m and integer n such that m, n and p are related (according to MUL). The keyword sif is used for forming a conditional (proof) expression in which the test is a static expression. The proof encoded by istot proceeds by induction on m; if m > 0 holds, then there exists an integer p1 such that m-1, n and p1 are related by induction hypothesis (on m-1) and thus m, n and p are related for p = p1+n according to the rule encoded by MULind; if m = 0, then m, n and p are related for p = 0. The proof encoded by the implementation of mul_istot goes like this: if m is a natural number, then the lemma proven by istot shows that m, n and some p are related; if m is negative, then the same lemma shows that ~m, n and p1 are related for some integer p1 and thus m, n and p are related for p = ~p1 according to the rule encoded by MULneg.

As another example of theorem-proving in ATS/LF, a proof function of the name mul_isfun is given as follows:

```
prfn mul_isfun
  {m,n:int} {p1,p2:int} (
  pf1: MUL (m, n, p1), pf2: MUL (m, n, p2)
) : [p1==p2] void = let
  prfun isfun
    {m:nat;n:int} {p1,p2:int} .<m>. (
    pf1: MUL (m, n, p1), pf2: MUL (m, n, p2)
  ) : [p1==p2] void =
    case+ pf1 of
    | MULind (pf1prev) => let
        prval MULind (pf2prev) = pf2 in isfun (pf1prev, pf2prev)
      end // end of [MULind]
    | MULbas () => let
        prval MULbas () = pf2 in ()
      end // end of [MULbas]
  // end of [isfun]
in
  case+ pf1 of
  | MULneg (pf1nat) => let
      prval MULneg (pf2nat) = pf2 in isfun (pf1nat, pf2nat)
    end // end of [MULneg]
  | _ =>> isfun (pf1, pf2)
end // end of [mul_isfun]
```

The keyword prfn is used for defining a non-recursive proof function, and the keyword prval for introducing bindings that relate names to proof expressions, that is, expressions of prop-types. As far as pattern matching exhaustiveness is concerned, prval is equivalent to val+ (as proofs cannot contain any effects such as failures of pattern matching).

What mul_isfun proves is that the relation MUL is functional on its first two arguments: If m, n and p1 are related according to MUL and m, n and p2 are also related according to MUL, then p1 and p2 are equal. The statement is first proven by the inner proof function isfun under the assumption that m is a natural number, and then the assumption is dropped. Let us now take a look at the first matching clause in the body of isfun. If the clause is chosen, then pf1 matches the pattern MULind(pf1prev) and thus pf1prev is of the type MUL(m1, n1, q1) for some natural number m1 and integer n1 and integer p1 such that m=m1+1, n=n1, and p1=q1+n1. This means that pf2 must be of the form MULind(pf2prev) for some pf2prev of the type MUL(m2, n2, q2) such that m2+1=m, n2=n and p2=q2+n2. By calling isfun on pf1prev and pf2prev, which amounts to invoking the induction hypothesis on m-1, we establish q1=q2, which implies p1=p2. The second matching clause in the body of isfun can be readily understood, which corresponds to the base case in the inductive proof encoded by isfun.

## Example: Proving Distributivity of Multiplication

The distributivity of multiplication over addition means that the following equation holds

```
m * (n1 + n2) = m * n1 + m * n2
```

for m, n1 and n2 ranging over integers. Clearly, a direct encoding of the equation is given by the following (proof) function interface:

```
extern
prfun mul_distribute {m:int} {n1,n2:int} {p1,p2:int}
  (pf1: MUL (m, n1, p1), pf2: MUL (m, n2, p2)): MUL (m, n1+n2, p1+p2)
// end of [mul_distribute]
```

Plainly speaking, the encoding states that the product of m and (n1+n2) is p1+p2 if the product of m and n1 is p1 and the product of m and n2 is p2. An implementation of mul_distribute is given as follows:

```
implement
mul_distribute
  {m} {n1,n2} {p1,p2} (pf1, pf2) = let
  prfun aux
    {m:nat}
    {n1,n2:int}
    {p1,p2:int}
    .<m>. (
    pf1: MUL (m, n1, p1), pf2: MUL (m, n2, p2)
  ) : MUL (m, n1+n2, p1+p2) =
    case+ (pf1, pf2) of
    | (MULbas (), MULbas ()) => MULbas ()
    | (MULind pf1, MULind pf2) => MULind (aux (pf1, pf2))
  // end of [aux]
in
  sif m >= 0 then
    aux (pf1, pf2)
  else let
    prval MULneg (pf1) = pf1
    prval MULneg (pf2) = pf2
  in
    MULneg (aux (pf1, pf2))
  end
end // end of [mul_distribute]
```

In essence, the inner function aux encodes a straighforward proof based on mathematical induction that establishes the following equation:

```
m * (n1 + n2) = m * n1 + m * n2
```

for m ranging over natural numbers and n1 and n2 ranging over integers. The function mul_distribute can then be implemented immediately based on aux.

## Datasorts

A datasort is rather similar to a dataype. However, the former is declared in the statics of ATS while the latter in the dynamics of ATS. To see a typical need for datasorts, let us try to encode a theorem in ATS stating that s is strictly less than $2^h$ if s and h are the size and height, respectively, of a given binary tree. To represent binary trees in the statics, we first declare a datasort as follows:

```
datasort tree = E of () | B of (tree, tree)
```

The name of the declared datasort is tree and there are two constructor associated with it: E and B, where E forms the empty tree and B forms a tree by joining two given trees. For instance, B(E(), E()) is a static term of the sort tree that represents a singleton tree, that is, a tree consisting of exactly one node. Please note that the trees formed by E and B are really just tree skeletons carrying no data.

We now declare two dataprops as follows for capturing the notion of size and height of trees:

```
dataprop SZ (tree, int) =
  | SZE (E (), 0) of ()
  | {tl,tr:tree} {sl,sr:nat}
    SZB (B (tl, tr), 1+sl+sr) of (SZ (tl, sl), SZ (tr, sr))
// end of [SZ]

dataprop HT (tree, int) =
  | HTE (E (), 0) of ()
  | {tl,tr:tree} {hl,hr:nat}
    HTB (B (tl, tr), 1+max(hl,hr)) of (HT (tl, hl), HT (tr, hr))
// end of [HT]
```

Given a tree t and an integer s, SZ(t, s) encodes the relation that the size of t equals s. Similiarly, given a tree t and an integer h, HZ(t, h) encodes the relation that the height of t equals h.

As the power function (of base 2) is not available in the statics of ATS, we declare a dataprop as follows to capture it:

```
dataprop POW2 (int, int) =
  | POW2bas (0, 1)
  | {n:nat} {p:int} POW2ind (n+1, p+p) of POW2 (n, p)
// end of [POW2]
```

Given two integers h and p, POW2 (h, p) encodes the relation that $2^h$ equals p.

It should be clear by now that the following proof function interface encodes the theorem stating that s is strictly less than $2^h$ if s and h are the size and height of a given binary tree:

```
extern
prfun lemma_tree_size_height
  {t:tree} {s,h:nat} {p:int} (
  pf1: SZ (t, s), pf2: HT (t, h), pf3: POW2 (h, p)
) : [s < p] void // end of [prfun]
```

Let us now construct an implementation of this proof function as follows.

We first establish some elementary properties on the power function (of base 2):

```
prfun pow2_istot
  {h:nat} .<h>. (): [p:int] POW2 (h, p) =
  sif h > 0 then POW2ind (pow2_istot {h-1} ()) else POW2bas ()
// end of [pow2_istot]

prfun pow2_pos
  {h:nat} {p:int} .<h>.
  (pf: POW2 (h, p)): [p > 0] void =
  case+ pf of
```

```
  | POW2ind (pf1) => pow2_pos (pf1) | POW2bas () => ()
// end of [pow2_pos]

prfun pow2_inc
  {h1,h2:nat | h1 <= h2} {p1,p2:int} .<h2>.
  (pf1: POW2 (h1, p1), pf2: POW2 (h2, p2)): [p1 <= p2] void =
  case+ pf1 of
  | POW2ind (pf11) => let
      prval POW2ind (pf21) = pf2 in pow2_inc (pf11, pf21)
    end
  | POW2bas () => pow2_pos (pf2)
// end of [pow2_inc]
```

Clearly, pow2_istot shows that the relation encoded by the dataprop POW2 is a total relation; pow2_pos proves that the power of each natural number is positive; pow2_inc establishes that the power function is increasing.

The function lemma_tree_size_height can be implemented as follows:

```
implement
lemma_tree_size_height
  (pf1, pf2, pf3) = let
//
prfun lemma
  {t:tree} {s,h:nat} {p:int} .<t>. (
  pf1: SZ (t, s), pf2: HT (t, h), pf3: POW2 (h, p)
) : [p > s] void =
  scase t of
  | B (tl, tr) => let
      prval SZB (pf1l, pf1r) = pf1
      prval HTB {tl,tr} {hl,hr} (pf2l, pf2r) = pf2
      prval POW2ind (pf31) = pf3
      prval pf3l = pow2_istot {hl} ()
      prval pf3r = pow2_istot {hr} ()
      prval () = lemma (pf1l, pf2l, pf3l)
      prval () = lemma (pf1r, pf2r, pf3r)
      prval () = pow2_inc (pf3l, pf31)
      prval () = pow2_inc (pf3r, pf31)
    in
      // nothing
    end // end of [B]
  | E () => let
      prval SZE () = pf1
      prval HTE () = pf2
      prval POW2bas () = pf3
   in
     // nothing
   end // end of [E]
//
in
  lemma (pf1, pf2, pf3)
end // end of [lemma_tree_size_height]
```

The inner function lemma, which is given a termination metric consisting of a static term of the sort tree, corresponds to a proof based on structural induction (where the involved structure is the binary tree t). Given two terms t1 and t2 of the sort tree, t1 is (strictly) less than t2 if t1 is a (proper) substructure of t2. Evidently, this is a well-founded ordering. The keyword scase is used to form a dynamic expression that does case-analysis on a static term (built by constructors associated with some declared datasort). So the relation between sif and scase is essentially parallel to that between if and case. Please find the entirety of the above code on-line[2].

## Example: Proving Properties on Braun Trees

As stated previously in this book, a binary tree is a Braun tree if it is empty or if its left and right subtrees are Braun trees and the size of the left one minus the size of the right one is either 0 or 1. Formally, we can declare the following dataprop isBraun to capture the notion of Braun trees:

```
dataprop isBraun (tree) =
  | isBraunE (E) of ()
  | {tl,tr:tree}
    {sl,sr:nat | sr <= sl; sl <= sr + 1}
    isBraunB (
      B(tl, tr)) of (isBraun tl, isBraun tr, SZ (tl, sl), SZ (tr, sr)
    ) // end of [isBraunB]
// end of [isBraun]
```

We first prove that there exists a Braun tree of any given size. This property can be encoded as follows in ATS:

```
extern
prfun lemma_existence {n:nat} (): [t:tree] (isBraun (t), SZ (t, n))
```

Literally, the type assigned to lemma_existence means that there exists a tree t for any given natural number n such that t is a Braun tree and the size of t is n. The following code gives an implementation of lemma_existence:

```
implement
lemma_existence {n} () = let
  prfun lemma {n:nat} .<n>.
    (): [t:tree] (isBraun (t), SZ (t, n)) =
    sif n > 0 then let
      stadef nl = n / 2 // size for the left subtree
      stadef nr = n - 1 - nl // size for the right subtree
      val (pfl1, pfl2) = lemma {nl} ()
      and (pfr1, pfr2) = lemma {nr} ()
    in
      (isBraunB (pfl1, pfr1, pfl2, pfr2), SZB (pfl2, pfr2))
    end else
      (isBraunE (), SZE ())
    // end of [sif]
in
  lemma {n} ()
end // end of [lemma_existence]
```

Note that stadef is a keyword in ATS for introducting a static binding between a name and a static term (of any sort). If one prefers, this keyword can be chosen to replace the keyword typedef (for introducing a name and a static term of the sort t@ype).

Next we show that two Braun trees of the same size are identical. This property can be encoded as follows:

```
extern
prfun lemma_unicity {n:nat} {t1,t2:tree} (
  pf1: isBraun t1, pf2: isBraun t2, pf3: SZ (t1, n), pf4: SZ (t2, n)
) : EQ (t1, t2) // end of [lemma_unicity]
```

where EQ is a prop-constructor introduced by the following dataprop declaration:

```
dataprop EQ (tree, tree) =
  | EQE (E, E) of ()
```

```
  | {t1l,t1r:tree} {t2l,t2r:tree}
    EQB (B (t1l, t1r), B (t2l, t2r)) of (EQ (t1l, t2l), EQ (t1r, t2r))
// end of [EQ]
```

Clearly, EQ is the inductively defined equality on trees. An implementation of the proof function lemma_unicity is presented as follows:

```
implement
lemma_unicity (pf1, pf2, pf3, pf4) = let
  prfun lemma {n:nat} {t1,t2:tree} .<n>. (
    pf1: isBraun t1, pf2: isBraun t2, pf3: SZ (t1, n), pf4: SZ (t2, n)
  ) : EQ (t1, t2) =
    sif n > 0 then let
      prval SZB (pf3l, pf3r) = pf3
      prval SZB (pf4l, pf4r) = pf4
      prval isBraunB (pf1l, pf1r, pf1lsz, pf1rsz) = pf1
      prval isBraunB (pf2l, pf2r, pf2lsz, pf2rsz) = pf2
      prval () = SZ_istot (pf1lsz, pf3l) and () = SZ_istot (pf1rsz, pf3r)
      prval () = SZ_istot (pf2lsz, pf4l) and () = SZ_istot (pf2rsz, pf4r)
      prval pfeql = lemma (pf1l, pf2l, pf3l, pf4l)
      prval pfeqr = lemma (pf1r, pf2r, pf3r, pf4r)
    in
      EQB (pfeql, pfeqr)
    end else let
      prval SZE () = pf3 and SZE () = pf4
      prval isBraunE () = pf1 and isBraunE () = pf2
    in
      EQE ()
    end // end of [sif]
in
  lemma (pf1, pf2, pf3, pf4)
end // end of [lemma_unicity]
```

Note that the proof function SZ_istot in this implementation of lemma_unicity is given the following interface:

```
extern
prfun SZ_istot {t:tree} {n1,n2:int}
  (pf1: SZ (t, n1), pf2: SZ (t, n2)): [n1==n2] void
```

which simply states that SZ is a functional relation with respect to its first parameter, that is, there is at most one n for every given t such that t and n are related according to SZ. Clearly, the mathematical proof corresponding to this implementation is of induction on the size n of the two given trees t1 and t2. In the base case where n is 0, t1 and t2 are equal as they both are the empty tree. In the inductive case where n > 0, it is proven that n1l and n2l are of the same value where n1l and n2l are the sizes of the left subtrees of t1 and t2, respecitvely; similarly, it is also proven that n1r and n2r are of the same value where n1r and n2r are the sizes of the right subtrees of t1 and t2, respectively; by induction hypothesis on n1l, the left substrees of t1 and t2 are the same; by induction hypothesis on n1r, the right substrees of t1 and t2 are the same; by the definition of tree equality (encoded by EQ), t1 and t2 are the same.

As a comparison, the following code gives another implementation of lemma_unicity that is of a different (and rather unusual) style:

```
implement
lemma_unicity (pf1, pf2, pf3, pf4) = let
  prfun lemma {n:nat} {t1,t2:tree} .<t1>. (
    pf1: isBraun t1, pf2: isBraun t2, pf3: SZ (t1, n), pf4: SZ (t2, n)
  ) : EQ (t1, t2) =
    case+ (pf1, pf2) of
    | (isBraunE (), isBraunE ()) => EQE ()
```

```
    | (isBraunB (pf11, pf12, pf13, pf14),
       isBraunB (pf21, pf22, pf23, pf24)) => let
//
        prval SZB (pf31, pf32) = pf3
        prval SZB (pf41, pf42) = pf4
//
        prval () = SZ_istot (pf13, pf31)
        prval () = SZ_istot (pf23, pf41)
//
        prval () = SZ_istot (pf14, pf32)
        prval () = SZ_istot (pf24, pf42)
//
        prval pfeq1 = lemma (pf11, pf21, pf31, pf41)
        prval pfeq2 = lemma (pf12, pf22, pf32, pf42)
      in
        EQB (pfeq1, pfeq2)
      end
    | (isBraunE _, isBraunB _) =/=> let
        prval SZE _ = pf3 and SZB _ = pf4 in (*none*)
      end
    | (isBraunB _, isBraunE _) =/=> let
        prval SZB _ = pf3 and SZE _ = pf4 in (*none*)
      end
in
  lemma (pf1, pf2, pf3, pf4)
end // end of [lemma_unicity]
```

This implementation corresponds to a proof by induction on the structure of the given tree t1. Note that the use of the special symbol =/=>, which is a keyword in ATS, is to indicate to the typechecker of ATS that the involved clause of patter matching is *unreachable*: It is the responsibility of the programmer to establish the falsehood on the right-hand side of the clause. Please find the entirety of the above code on-line[3].

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_THMPRVING/
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_THMPRVING/tree.da
3. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_THMPRVING/braunt

# Chapter 16. Programming with Theorem-Proving

*Programming with Theorem-Proving* (PwTP) is a rich and broad programming paradigm that allows cohesive construction of programs and proofs in a syntactically intwined manner. The support for PwTP in ATS is a signatory feature of ATS, and the novelty of ATS largely stems from it. For people who are familiar with the so-called Curry-Howard isomorphism, I emphasize that PwTP as is supported in ATS makes little, if any, essential use of this isomorphism (between proofs and programs): The dynamics of ATS in which programs are written is certainly not pure and the proofs encoded in ATS/LF are not required to be constructive, either. However, that proof construction in ATS can be done in a style of (functional) programming is fundamentally important in terms of syntax design for ATS, for the need to combine programs with proofs would otherwise be greatly more challenging.

In this chapter, I will present some simple but convincing examples to illustrate the power and flexibility of PwTP as is supported in ATS. However, the real showcase for PwTP will not arrive until after the introduction of linear types in ATS, when linear proofs can be combined with programs to track and safely manipulate resources such as memory and objects (e.g, file handles). In particular, PwTP is to form the cornersone of the support for imperative programming in ATS.

Please find on-line[1] the code employed for illustration in this chapter plus some additional code for testing.

## Circumventing Nonlinear Constraints

The constraint-solver of ATS is of rather diminished power. In particular, constraints containing nonlinear integer terms (e.g., those involving the use of multiplication (of variables)) are immediately rejected. This weakness must be properly addressed for otherwise it would become a crippling limitation on practicality of the type system of ATS. I now use a simple example to demonstrate how theorem-proving can be employed to circumvent the need for handling nonlinear constraints directly.

A function template list_concat is implemented as follows:

```
//
// [list_concat] does not typecheck!!!
//
fun{a:t@ype}
list_concat {m,n:nat}
  (xss: list (list (a, n), m)): list (a, m * n) =
  case+ xss of
  | list_cons (xs, xss) => list_append<a> (xs, list_concat xss)
  | list_nil () => list_nil ()
// end of [list_concat]
```

where the interface for list_append is given below:

```
fun{a:t@ype} list_append {n1,n2:nat}
  (xs: list (a, n1), ys: list (a, n2)): list (a, n1+n2)
```

Given a list xss of length m in which each element is of the type list(T,n) for some type T, list_concat<T>(xss) constructs a list of the type list(T,m*n). When the first matching clause in the code for list_concat is typechecked, a constraint is generated that is essentially like the following one:

```
m = m1 + 1 implying n + (m1 * n) = m * n holds for all natural numbers m, m1 and n.
```

This contraint may look simple, but it is rejected by the ATS constraint solver as it contains nonlinear integer terms (e.g., m1*n and m*n). In order to overcome (or rather circumvent) the limitation, we make use of theorem-proving. Another implementation of list_concat is given as follows:

```
fun{a:t@ype}
list_concat {m,n:nat} (
  xss: list (list (a, n), m)
) : [p:nat] (MUL (m, n, p) | list (a, p)) =
  case+ xss of
  | list_cons (xs, xss) => let
      val (pf | res) = list_concat (xss)
    in
      (MULind pf | list_append<a> (xs, res))
    end
  | list_nil () => (MULbas () | list_nil ())
// end of [list_concat]
```

Given a list xss of the type list(list(T,n),m), list_concat(xss) now returns a pair (pf | res) such that pf is a proof of the prop-type MUL(m,n,p) for some natural number p and res is a list of the type list(T,p), where the symbol bar (|) is used to separate proofs from values. In other words, pf acts as a witness to the equality p=m*n. After proof erasure is performed, this implementation of list_concat is essentially translated into the previous one (as far as dynamic semantics is concerned). In particular, there is no proof construction at run-time and no need for it, either.

## Example: Safe Matrix Subscripting

Internally, a matrix of the dimension m by n is represented as an array of the size m*n. For matrix subscripting, we need to implement a function template of the following interface:

```
extern fun{a:t@ype}
matget {m,n:nat} {i,j:nat | i < m; j < n}
  (A: array (a, m*n), col: int n, i: int i, j: int j): a
// end of [matget]
```

Assume that the matrix is row-major. Then the element indexed by i and j in the matrix is the element indexed by i*n + j in the array that represents the matrix, where i and j are natural numbers less than m and n, respectively. However, the following implementation fails to pass typechecking:

```
implement{a} matget (A, n, i, j) = A[i*n+j] // it fails to typecheck!!!
```

as the ATS constraint solver cannot automatically verify that i*n+j is a natural number strictly less than m*n. An implementation of matget that typechecks can be given as follows:

```
implement{a}
matget {m,n} {i,j} (A, n, i, j) = let
//
  val (pf_in | _in) = i imul2 n // pf_in: MUL (i, n, _in)
  prval () = mul_nat_nat_nat (pf_in) // _in >= 0
//
  prval pf_mn = mul_istot {m,n} () // pf1_mn: MUL (m, n, _mn)
  prval () = mul_elim (pf_mn) // _mn = m*n
  prval MULind (pf_m1n) = pf_mn // _m1n = (m-1)*n = m*n-n
//
```

```
  stadef i1 = m-1-i
  prval pf_i1n = mul_istot {i1,n} () // pf_i1n: MUL (i1, n, _i1n)
  prval () = mul_nat_nat_nat (pf_i1n) // _i1n >= 0
//
  prval pf2_m1n = mul_distribute2 (pf_in, pf_i1n) // _m1n = _in + _i1n
  prval () = mul_isfun (pf_m1n, pf2_m1n) // _mn - n = _in + _i1n
//
in
  A[_in+j]
end // end of [matget]
```

where the functions called in the body of matget are assigned the following interfaces:

```
fun imul2 {i,j:int}
  (i: int i, j: int j): [ij:int] (MUL (i, j, ij) | int ij)

prfun mul_istot {i,j:int} (): [ij:int] MUL (i, j, ij)

prfun mul_isfun {i,j:int} {ij1, ij2: int}
  (pf1: MUL (i, j, ij1), pf2: MUL (i, j, ij2)): [ij1==ij2] void

prfun mul_elim
  {i,j:int} {ij:int} (pf: MUL (i, j, ij)): [i*j==ij] void

prfun mul_nat_nat_nat
  {i,j:nat} {ij:int} (pf: MUL (i, j, ij)): [ij >= 0] void

prfun mul_distribute2
  {i1,i2:int} {j:int} {i1j,i2j:int}
  (pf1: MUL (i1, j, i1j), pf2: MUL (i2, j, i2j)): MUL (i1+i2, j, i1j+i2j)
```

Note that the keyword stadef is for introducing a binding between a name and a static term (of any sort). Assume that m and n are natural numbers and i and j are natural numbers less than m and n, respectively. The method employed in the implementation of matget to show i*n+j < m*n essentially proves that m*n = (m-1)*n + n, (m-1)*n = i*n + (m-1-i)*n and (m-1-i)*n >= 0, which in turn imply that m*n >= i*n+n > i*n+j.

Note that there are a variety of proof functions declared in `prelude/SATS/arith.sats`[2] for helping prove theorems involving arithmetic operations. For examples of proof construction in ATS, please find the implementation of some of these proof functions in `prelude/DATS/arith.dats`[3].

The entirety of the above presented code is available on-line[4].

## Specifying with Enhanced Precision

The integer addition function can be assigned the following (dependent) type in ATS to indicate that it returns the sum of its two integer arguments:

```
{i,j:int} (int (i), int (j)) -> int (i+j)
```

This type gives a full specification of integer addition as the only (terminating) function that can be given the type is the integer addition function. However, the factorial function, which yields the product of the first n positive integers when applied to a natural number n, cannot be given the following type:

```
{n:nat} int (n) -> int (fact(n))
```

as fact, which refers to the factorial function, does not exist in the statics of ATS. Evidently, a highly interesting and relevant question is whether a type can be formed in ATS that fully captures the functional relation specified by fact? The answer is affirmative. We can not only construct such a type but also assign it to a (terminating) function implemented in ATS.

Let us recall that the factorial function can be defined by the following two equations:

```
fact(0) = 1
fact(n) = n * fact(n-1) (for all n > 0)
```

Naturally, these equations can be encoded by the constructors associated with the dataprop FACT declared as follows:

```
dataprop FACT (int, int) =
  | FACTbas (0, 1)
  | {n:nat} {r1,r:int} FACTind (n, r) of (FACT (n-1, r1), MUL (n, r1, r))
// end of [FACT]
```

Note that for any given natural number n and integer r, FACT(n, r) can be assigned to a proof if and only if fact(n) = r. Therefore, the following type

```
{n:nat} int (n) -> [r:int] (FACT (n, r) | int (r))
```

can only be assigned to a function that, if applied to a natural number n, returns a proof and an integer such that the proof attests to the integer being equal to fact(n). For instance, the following defined function ifact is assigned this type:

```
fun ifact {n:nat} .<n>.
  (n: int n): [r:int] (FACT (n, r) | int r) =
  if n > 0 then let
    val (pf1 | r1) = ifact (n-1) // pf1: FACT (n-1, r1)
    val (pfmul | r) = n imul2 r1 // pfmul: FACT (n, r1, r)
  in (
    FACTind (pf1, pfmul) | r
  ) end else (
    FACTbas () | 1 // the base case
  ) // end of [if]
// end of [ifact]
```

After proof erasure, ifact precisely implements the factorial function.

Please find the entirety of the above presented code plus some testing code on-line[5].

## Example: Another Verified Factorial Implementation

The function ifact presented in the section on specifying with enhanced precision is a verified implementation of the factorial function as its type guarantees that ifact implements the specification of factorial encoded by the dataprop FACT. Clearly, the implementation of ifact closely follows the declaration of FACT. If we think of the latter as a logic program, then the former is essentially a functional version extracted from the logic program. However, the implementation of a specification in practice can often digress far from the specification algorithmically. For instance, we may want to have a verified implementation of factorial that is also tail-recursive. This can be done as follows:

```
fun ifact2 {n:nat} .<>.
  (n: int n): [r:int] (FACT (n, r) | int r) = let
```

```
  fun loop
    {i:nat | i <= n} {r:int} .<n-i>. (
    pf: FACT (i, r) | n: int n, i: int i, r: int r
  ) : [r:int] (FACT (n, r) | int r) =
    if n - i > 0 then let
      val (pfmul | r1) = (i+1) imul2 r in loop (FACTind (pf, pfmul) | n, i+1, r1)
    end else (pf | r) // end of [if]
  // end of [loop]
in
  loop (FACTbas () | n, 0, 1)
end // end of [ifact2]
```

The function ifact2 is assigned a type indicating that ifact2 is a verified implementation of factorial, and it is defined as a call to the inner function loop that is clearly tail-recursive. If we erase types and proofs, the function ifact2 is essentially defined as follows:

```
fun ifact2 (n) = let
  fun loop (n, i, r) =
    if n - i > 0 then let
      val r1 = (i+1) * r in loop (n, i+1, r1)
    end else r
  // end of [loop]
in
  loop (n, 0, 1)
end // end of [ifact2]
```

When the inner function loop is called on three arguments n, i and r, the precondition for this call is that i is natural number less than or equal to n and r equals fact(i), that is, the value of the factorial function on i. This precondition is captured by the type assigned to loop and thus enforced at each call site of loop in the implementation of ifact2.

Please find on-line[6] the entirety of the above presented code plus some testing code.

## Example: Verified Fast Exponentiation

Given an integer x, pow(x, n), the nth power of x, can be defined inductively as follows:

```
pow (x, 0) = 1
pow (x, n) = x * pow (x, n-1) (for all n > 0)
```

A direct implementation of this definition is given as follows:

```
fun ipow {n:nat} .<n>.
  (x: int, n: int n): int = if n > 0 then x * ipow (x, n-1) else 1
// end of [ipow]
```

which is of time-complexity O(n) (assuming multiplication is O(1)). A more efficient implmentation can be given as follows:

```
fun ifastpow {n:nat} .<n>.
  (x: int, n: int n): int =
  if n > 0 then let
    val n2 = n/2; i = n-(2*n2)
  in
    if i > 0 then pow (x*x, n2) else x * pow (x*x, n2)
  end else 1
// end of [ifastpow]
```

which makes use of the property that pow(x, n) equals pow(x*x, n/2) if n is even or x * pow(x*x, n/2) if n is odd. This is referred to as fast exponentiation. Note that ifastpow is of time-complexity O(log(n)).

Clearly, what is done above is not restricted to exponentiation on integers. As long as the underlying multiplication is associative, fast exponentiation can be employed to compute powers of any given element. In particular, powers of square matrices can be computed in this way. I now present as follows a verified generic implementation of fast exponentiation.

Handling generic data properly in a verified implementation often requires some finesse with the type system of ATS. Let us first introduce an abstract type constructor E as follows:

```
sortdef elt = int // [elt] is just an alias for [int]
abst@ype E (a:t@ype, x:elt) = a // [x] is an imaginary stamp
```

This is often referred to as *stamping*. For each type T and stamp x, E(T, x) is just T as far as data representation is concerned. The stamps are imaginary and they are solely used for the purpose of specification. We next introduce an abstract prop-type MUL and a function template mul_elt_elt:

```
absprop MUL (elt, elt, elt) // abstract mul relation

fun{a:t@ype}
mul_elt_elt {x,y:elt}
  (x: E (a, x), y: E (a, y)): [xy:elt] (MUL (x, y, xy) | E (a, xy))
// end of [mul_elt_elt]
```

Please do not confuse MUL with the one of the same name that is declared in prelude/SATS/arith.sats[7]. To state that the encoded multiplication is associative, we can introduce the following proof function:

```
praxi mul_assoc
  {x,y,z:elt} {xy,yz:elt} {xy_z,x_yz:elt} (
  pf1: MUL (x, y, xy), pf2: MUL (xy, z, xy_z)
, pf3: MUL (y, z, yz), pf4: MUL (x, yz, x_yz)
) : [xy_z==x_yz] void
```

The keyword praxi indicates that mul_assoc is treated as a form of axiom, which is not expected to be implemented.

The abstract power function can be readily specified in terms of the abstract prop-type MUL:

```
dataprop POW (
  elt(*base*), int(*exp*), elt(*res*)
) = // res = base^exp
  | {x:elt} POWbas (x, 0, 1(*unit*))
  | {x:elt} {n:nat} {p,p1:elt}
    POWind (x, n+1, p1) of (POW (x, n, p), MUL (x, p, p1))
// end of [POW]
```

As can be expected, generic fast exponentiation is given the following interface:

```
fun{a:t@ype}
fastpow_elt_int {x:elt} {n:nat}
  (x: E (a, x), n: int n): [p:elt] (POW (x, n, p) | E (a, p))
// end of [fastpow_elt_int]
```

With the preparation done above, a straightforward implementation of fastpow_elt_int can now be presented as follows:

```
implement{a}
fastpow_elt_int (x, n) = let
//
// lemma: (x*x)^n = x^(2n)
//
extern prfun
lemma {x:elt} {xx:elt} {n:nat} {y:elt}
  (pfxx: MUL (x, x, xx), pfpow: POW (xx, n, y)): POW (x, 2*n, y)
//
in
  if n > 0 then let
    val n2 = n / 2; val i = n - (n2+n2) // i = 0 or 1
    val (pfxx | xx) = mul_elt_elt (x, x) // xx = x*x
    val (pfpow2 | res) = fastpow_elt_int<a> (xx, n2) // xx^n2 = res
    prval pfpow = lemma (pfxx, pfpow2) // pfpow: x^(2*n2) = res
  in
    if i > 0 then let
      val (pfmul | xres) = mul_elt_elt<a> (x, res) // xres = x*res
    in
      (POWind (pfpow, pfmul) | xres)
    end else (pfpow | res)
  end else let
    val res = mulunit<a> () in (POWbas () | res) // res = 1
  end (* end of [if] *)
end // end of [fastpow_elt_int]
```

Note that this implementation of fastpow_elt_int is not tail-recursive. The function template mulunit, which is called to produce a unit for the underlying multiplication, is assigned the following interface:

```
fun{a:t@ype} mulunit (): E (a, 1(*stamp*))
```

The proof function lemma simply establishes that pow(x, 2*n)=pow(x*x, n) for each natural number n. I have made an implementation of lemma available on-line but I suggest that the interested reader give it a try first before taking a look. Note that the following axioms are needed to implement lemma:

```
praxi mul_istot // [MUL] is total
  {x,y:elt} (): [xy:elt] MUL (x, y, xy)
praxi mul_isfun {x,y:elt} {z1,z2:elt} // MUL is functional
  (pf1: MUL (x, y, z1), pf2: MUL (x, y, z2)): [z1==z2] void
```

Another interesting (and possibly a bit challenging) exercise is to implement fastpow_elt_int in a tail-recursive fashion.

Please find on-line the two files fastexp.sats[8] and fastexp.dats[9] that contain the entirety of the above presented code.

Now we have implemented fastpow_elt_int. How can we use it? Please find on-line[10] an example in which fastpow_elt_int is called to implement fast exponentiation on a 2-by-2 matrix so that Fibonacci numbers can be computed in a highly efficient manner.

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_PwTP/
2. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/arith.sats
3. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/DATS/arith.dats

4. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_PwTP/matget.dats
5. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_PwTP/ifact.dats
6. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_PwTP/ifact23.dats
7. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/arith.sats
8. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_PwTP/fastexp.sats
9. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_PwTP/fastexp.dats
10. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_PwTP/test_fastexp.da

# Chapter 17. Summary

# Chapter 18. Introduction to Views and Viewtypes

Probably the single most forceful motivation shaping the development of ATS is to make ATS a programming language that can be employed effectively to construct safe and reliable programs running in the kernels of operating systems. Instead of following seemingly natural approaches that often carve out a "safe" subset of C and/or put wrappers around "unsafe" programming features in C, ATS relies on the paradigm of programming with theorem-proving to prevent resources such as memory from being misused or mismanaged, advocating an approach to safety that is both general and flexible. For example, a well-typed program constructed in ATS cannot cause buffer overrun at run-time even though pointer arithmetic is fully supported in ATS. More specifically, if a pointer is to be dereferenced, ATS requires that a proof be given that attests to the safety of the dereferencing operation. Proofs of this kind are constructed to demonstrate the validity of linear propositions, which are referred to as views in ATS, for classifying resources as well as capabilities.

Please find on-line[1] the code presented for illustration in this chapter.

## Views for Memory Access through Pointers

A view is a linear version of prop, where the word *linear* comes from *linear logic*, a resource-aware logic invented by Jean-Yves Girard. There is a built-in sort view for static terms representing views. Given a type T and a memory location L, a view of the form T@L can be formed to indicate a value of the type T being stored in the memory at the location L, where @ is a special infix operator. Views of this form are extremely common in practice, and they are often referred to as @-views (or at-views). As an example, the following function templates ptr_get0 and ptr_set0, which reads and writes through a given pointer, are assigned types containing @-views:

```
fun{a:t@ype}
ptr_get0 {l:addr} (pf: a @ l | p: ptr l): (a @ l | a)

fun{a:t@ype}
ptr_set0 {l:addr} (pf: a? @ l | p: ptr l, x: a): (a @ l | void)
```

Note that ptr is a type constructor that forms a type ptr(L) when applied to a static term L of the sort addr, and the only value of the type ptr(L) is the pointer that points to the location denoted by L.

Given a type T, the function ptr_get0<T> is assigned the following type:

```
{l:addr} (T @ l | ptr (l)) -> (T @ l | T)
```

This type indicates that the function ptr_get0<T> returns a proof of the view T@L and a value of the type T when applied to a proof of the view T@L and a pointer of the type ptr(L) for some L. Intuitively speaking, a proof of the view T@L, which is a form of resource as T@L is linear, is *consumed* when it is passed to ptr_get0<T>, and another proof of the view T@L is generated when ptr_get0<T> returns. Notice that a proof of the view T@L must be returned for otherwise subsequent accesses to the content stored at the memory location L would have been precluded.

Similarly, the function ptr_set0<T> is assigned the following type:

```
{l:addr} (T? @ l | ptr (l)) -> (T @ l | void)
```

Note that T? is a type for values of size sizeof(T) that are assumed to be uninitialized. The function ptr_set0<T> returns a proof of the view T@L when applied to a proof of the view T?@L, a pointer of the type ptr(L) and a value of the type T. The use of the view T?@L indicates that the memory location at L is assumed to be uninitialized when ptr_set0<T> is called.

As an example, a function template swap0 is implemented below for swapping memory contents at two given locations:

```
fn{a:t@ype}
swap0 {l1,l2:addr} (
  pf1: a @ l1, pf2: a @ l2 | p1: ptr l1, p2: ptr l2
) : (a @ l1, a @ l2 | void) = let
  val (pf1 | tmp1) = ptr_get0<a> (pf1 | p1)
  val (pf2 | tmp2) = ptr_get0<a> (pf2 | p2)
  val (pf1 | ()) = ptr_set0<a> (pf1 | p1, tmp2)
  val (pf2 | ()) = ptr_set0<a> (pf2 | p2, tmp1)
in
  (pf1, pf2 | ())
end // end of [swap0]
```

Compared to a corresponding implementation in C, the verbosity of this one in ATS is evident. In particular, the need for *threading* linear proofs through calls to functions that make use of resources can often result in a lot of *administrative* code to be written. I now present some special syntax to significantly alleviate the need for such administrative code.

The function templates ptr_get1 and ptr_set1 are given the following interfaces:

```
fun{a:t@ype}
ptr_get1 {l:addr} (pf: !a @ l >> a @ l | p: ptr l): a

fun{a:t@ype}
ptr_set1 {l:addr} (pf: !a? @ l >> a @ l | p: ptr l, x: a): void
```

Clearly, for each type T, the function ptr_get1<T> is assigned the following type:

```
{l:addr} (!T @ l >> T @ l | ptr(l)) -> T
```

Given a linear proof pf of the view T@L for some L and a pointer p of the type ptr(L), the function call ptr_get1<T>(pf, p) is expected to return a value of the type T. However, the proof pf is not consumed. Instead, it is still a proof of the view T@L after the function call returns. Similarly, the function ptr_set1<T> is assigned the following type:

```
{l:addr} (!T? @ l >> T @ l | ptr(l), T) -> void
```

Given a linear proof pf of the view T?@L for some L, a pointer p of the type ptr(L) and a value v of the type T, the function call ptr_set1<T>(pf, p, v) is expected to return the void value while changing the view of pf from T?@L to T@L. In general, assume that f is given a type of the following form for some views V1 and V2:

```
(...,!V1 >> V2, ...) -> ...
```

Then a function call f(..., pf, ...) on some proof variable pf of the view V1 is to change the view of pf into V2 upon its return. In the case where V1 and V2 are the same, !V1 >> V2 can simply be written as !V1. As an example, a function template swap1 for swapping the contents at two given memory locations is implemented as follows:

```
fn{a:t@ype}
swap1 {l1,l2:addr} (
  pf1: !a @ l1, pf2: !a @ l2 | p1: ptr l1, p2: ptr l2
) : void = let
  val tmp = ptr_get1<a> (pf1 | p1)
  val () = ptr_set1<a> (pf1 | p1, ptr_get1<a> (pf2 | p2))
  val () = ptr_set1<a> (pf2 | p2, tmp)
in
```

```
  // nothing
end // end of [swap1]
```

Clearly, this implementation is considerably cleaner when compared to the above implementation of swap0.

A further simplied implementation of swap1 is given as follows:

```
fn{a:t@ype}
swap1 {l1,l2:addr} (
  pf1: !a @ l1, pf2: !a @ l2 | p1: ptr l1, p2: ptr l2
) : void = let
  val tmp = !p1 in !p1 := !p2; !p2 := tmp
end // end of [swap1]
```

Given a pointer p of the type ptr(L) for some L, !p yields the value stored at the memory location L. The typechecker first searches for a proof of the view T@L for some T among all the currently available proofs when typechecking !p; if such a proof pf is found, then !p is essentially elaborated into ptr_get1(pf | p) and then typechecked. As !p is a left-value, it can also be used to form an assignment like !p := v for some value v. The typechecker elaborates !p := v into ptr_set1(pf | p, v) for the sake of typechecking if a proof of the at-view T@L can be found for some type T among all the currently available proofs. Note that this implementation of swap1 makes no use of administrative code for handling linear proofs explicitly.

## Viewtypes as a Combination of Views and Types

A linear type in ATS is given the name *viewtype*, which is chosen to indicate that a linear type consists of two parts: one part for views and the other for types. For instance, given a view V and a type T, then the tuple (V | T) is a viewtype, where the bar symbol (|) is a separator (just like a comma) to separate views from types. What seems a bit surprising is the opposite: For each viewtype VT, we may assume the existence of a view V and a type T such that VT is equivalent to (V | T). Formally, this T can be referred as VT?! in ATS. This somewhat unexpected interpretation of linear types is a striking novelty of ATS, which stresses that the linearity of a viewtype comes *entirely* from the view part residing within it.

The built-in sorts viewtype and viewt@ype are for static terms representing viewtypes whose type parts are of the sorts type and t@ype, respectively. In other words, the former is assigned to viewtypes for linear values of the size equal to that of a pointer and the latter to viewtypes for linear values of unspecified size. For example, tptr is defined as follows that takes a type and an address to form a viewtype (of the sort viewtype):

```
viewtypedef tptr (a:t@ype, l:addr) = (a @ l | ptr l)
```

Given a type T and an address L, the viewtype tptr(T, L) is for a pointer to L paired with a linear proof stating that a value of the type T is stored at L. If we think of a counter as a pointer paired with a proof stating that the pointer points to an integer (representing the count), then the following defined function getinc returns the current count of a given counter after increasing it by 1:

```
fn getinc
  {l:addr} {n:nat} (
  cnt: !tptr (int(n), l) >> tptr (int(n+1), l)
) : int(n) = n where {
  val n = ptr_get1<int(n)> (cnt.0 | cnt.1)
  val () = ptr_set1<int(n+1)> (cnt.0 | cnt.1, n+1)
} // end of [getinc]
```

A particularly interesting example of a viewtype is the following one:

```
viewtypedef cloptr
  (a:t@ype, b:t@ype, l:addr) =
  [env:t@ype] (((&env, a) -> b, env) @ l | ptr l)
// end of [cloptr]
```

Given two types A and B, a pointer to some address L where a closure function is stored that takes a value of the type A to a value of the type B can be given the viewtype cloptr(A, B, L). Note that a closure function is just an envless function paired with an environment containing bindings for variables in the body of the closure function that are introduced from outside. In the function type (&env, a) -> b, the symbol & indicates that the corresponding function argument is passed by reference, that is, the argument is required to be a left-value and what is actually passed is the address of the left-value. I will cover the issue of call-by-reference elsewhere in more details. The following piece of code demonstrates a pointer to a closure function being called on a given argument:

```
fun{a:t@ype}{b:t@ype}
cloptr_app {l:addr} (
  pclo: !cloptr (a, b, l), x: a
) : b = let
  val p = pclo.1
  prval pf = pclo.0 // takeout pf: ((&env, a) -> b, env) @ l
  val res = !p.0 (!p.1, x)
  prval () = pclo.0 := pf // return pf
in
  res
end // end of [cloptr]
```

Note that the linear proof in pclo is first taken out so that the code for dereferencing p (denoted by the syntax !p) can pass typechecking, and it is then returned so that the type of pclo is restored to its original one. The very ability to explain within ATS programming features such as closure function is a convincing indication of the expressiveness of the type system of ATS.

## Left-Values and Call-by-Reference

In its simplest form, a left-value is just a pointer paired with a linear proof attesting to a value (of some type) being stored at the location to which the pointer points. The name *left-value* stems from such a value being able to appear on the left-hand side of an assignment statement (in languages like C). Often, a left-value is intuitively explained as a value with an address attached to it. Note that whatever representation chosen for a left-value must make it possible to identify the pointer and the linear proof (of some at-view) that are associated with the left-value.

In ATS, the simplest expression representing a left-value is !p, where ! is a special symbol and p a value of the type ptr(L) for some address L. When this expression is typechecked, a proof of T@L for some type T is required to be found among the currently available proofs. I will introduce additional forms of left values gradually.

The default strategy for passing a function argument in ATS is call-by-value. However, it is also allowed in ATS to specify that call-by-reference is chosen for passing a particular function argument. By call-by-reference, it is meant that the argument to be passed must be a left-value and what is actually passed is the address of the left-value (instead of the value stored at the address). For example, the following defined function swap2 makes essential use of call-by-reference:

```
fn{a:t@ype}
swap2 (
```

```
  x1: &a, x2: &a
) : void = let
  val tmp = x1 in x1 := x2; x2 := tmp
end // end of [swap2]
```

Note that the special symbol & in front of the type of a function argument indicates that the argument needs to be passed according to the call-by-reference strategy. The following code implements swap1 based on swap2:

```
fn{a:t@ype}
swap1 {l1,l2:addr} (
  pf1: !a @ l1, pf2: !a @ l2 | p1: ptr l1, p2: ptr l2
) : void = swap2 (!p1, !p2)
```

When the call swap2 (!p1, !p2) is evaluated at run-time, the parameters actually being passed are the two pointers p1 and p2 (rather than the values stored at the locations to which these two pointers point).

## Stack-Allocated Variables

Given a type T and an address L, how can a proof of the view T@L be obtained in the first place? There are actually a variety of methods for obtaining such proofs in practice, and I present one as follows that is based on stack-allocation of local variables.

In the body of the following function foo, some stack-allocated local variables are declared:

```
fn foo (): void = let
  var x0: int // view@ (x0): int? @ x0
  val () = x0 := 0 // view@ (x0): int(0) @ x0
  var x1: int = 1 // view@ (x1): int(1) @ x1
//
// [with] is a keyword in ATS
//
  var y: int with pfy // pfy is an alias of view@ (y): int? @ y
  val () = y := 2 // pfy = view@ (y): int(2) @ y
  var z: int with pfz = 3 // pfz is an alias of view@ (z): int(3) @ z
in
  // nothing
end // end of [f]
```

The keyword var is for declaring a local variable. When a variable is declared, either its type or its initial value needs to be given. If a variable is declared without a type, then the type of its initial value is assumed to be its type. Assume that a variable x is declared of type T. Then the pointer to the location of the variable is denoted by &x, and its associated linear proof (of some at-view) can be referred to as view@(x), where view@ is a keyword. A variable is another form of left-value in ATS. In the body of foo, x0 is declared to be a variable of the type int and then it is initialized with the integer 0; x1 is declared to be a variable of the type int that is given the initial value 1; y is declared to be a variable of the type int while pfy is introduced as an alias of view@(y), and then y is initialized with the integer 2; z is declared to be a variable of the type int that is given the initial value 3 while pfz is introduced as an alias of view@(z).

The following code gives an implementation of the factorial function:

```
fn fact {n:nat}
  (n: int n): int = let
  fun loop {n:nat} {l:addr} .<n>.
    (pf: !int @ l | n: int n, res: ptr l): void =
```

```
    if n > 0 then let
      val () = !res := n * !res in loop (pf | n-1, res)
    end // end of [if]
  // end of [loop]
  var res: int with pf = 1
  val () = loop (pf | n, &res) // &res: the pointer to res
in
  res
end // end of [fact]
```

Note that the variable res holds the intermediate result during the execution of the loop. As res is stack-allocated, there is no generated garbage after a call to fact is evaluated. When this style of programming is done in C, there is often a concern about the pointer to res being derefenced after a call to fact returns, which is commonly referred to as derefencing a dangling pointer. This concern is completely eliminated in ATS as it is required by the type system of ATS that a linear proof of the at-view associated with the variable res be present at the end of legal scope for res. More specifically, if x is a declared variable of the type T, then a linear proof of the view T?@L, where L is the address of x, must be available when typechecking reaches the end of the scope for x. This requirement ensures that a variable can no longer be accessed after the portion of the stack in which it is allocated is reclaimed as no linear proof of the at-view associated with the variable is ever available from that point on.

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_VVTINTRO/

# Chapter 19. Dataviews as Linear Dataprops

The at-views of the form T@L for types T and addresses L are building blocks for constructing other forms of views. One mechanism for putting together such building blocks is by declaring dataviews, which is mostly identical to declaring dataprops. I now present in this chapter some commonly encountered dataviews and their uses.

Please find on-line[1] the code presented for illustration in this chapter.

## Optional Views

The dataview option_v is declared as follows:

```
dataview option_v (v:view+, bool) =
  | Some_v (v, true) of (v) | None_v (v, false) of ()
// end of [option_v]
```

By the declaration, the dataview option_v is covariant in its first argument and there are two proof constructors associated with it: Some_v and None_v. Given a view V, option_v(V, b) is often called an optional view, where b is a boolean. Clearly, a proof of the view option_v(V, true) contains a proof of the view V while a proof the view option_v(V, false) contains nothing. As an example, let us take a look at the following function template interface:

```
fun{a:t@ype}
ptr_alloc_opt (): [l:addr] (option_v (a? @ l, l > null) | ptr l)
```

Given a type T, the function ptr_alloc_opt<T> returns a pointer paired with a proof of an optional view; if the returned pointer is not null, then the proof can be turned into a proof of the view T?@L, where L is the location to which the pointer points; otherwise, there is no at-view associated with the returned pointer.

To see another example, let us assume that get_width is given the interface below:

```
fun get_width (x: &window): int
```

where window is some (unboxed) abstract type. One may think that get_width returns the width of a window object. The following code shows a typical use of an optional view:

```
viewdef optat
  (a:t@ype, l:addr) = option_v (a @ l, l > null)
// end of [optat]

fun get_width_alt {l:addr} (
  pf: !optat (int?, l) >> optat (int, l)
| x: &window, p: ptr l
) : void =
  if p > null then let
    prval Some_v (pf1) = pf
    val () = !p := get_width (x)
  in
    pf := Some_v (pf1)
  end else let
    prval None_v () = pf in pf := None_v ()
  end // end of [val]
// end of [get_width_alt]
```

The function get_width_alt is a variation of get_width. In addtion to the window object, it takes a pointer which, if not null, points to the location where the width information should be stored.

Please find the entirety of the above presented code on-line[2].

## Linear Arrays

Unlike in most programming languages, arrays are not a primitive data structure in ATS. More specifically, persistent arrays can be implemented based on linear arrays, which allow for being freed safely by the programmer, and linear arrays can be implemented based on at-views. Intuitively, the view for an array containing N elements of type T consists of N at-views: T@$L_0$, T@$L_1$, ..., and T@$L_{N-1}$, where $L_0$ is the starting address of the array and each subsequent L equals the previous one plus the size of T, that is, the number of bytes needed to store a value of the type T. The following declared dataview array_v precisely captures the intuituion:

```
dataview array_v
  (a:t@ype+, int(*size*), addr(*beg*)) =
  | {n:nat} {l:addr}
    array_v_cons (a, n+1, l) of (a @ l, array_v (a, n, l+sizeof(a)))
  | {l:addr} array_v_nil (a, 0, l)
// end of [array_v]
```

Given a type T, an integer N and an address L, array_v(T, N, L) is a view for the array starting at L that contains N elements of the type T. As can be readily expected, the function templates for array accessing and array updating are given the following interfaces:

```
fun{a:t@ype}
arrget {n,i:nat | i < n} {l:addr}
  (pf: !array_v (a, n, l) | p: ptr l, i: int i): a
// end of [arrget]

fun{a:t@ype}
arrset {n,i:nat | i < n} {l:addr}
  (pf: !array_v (a, n, l) | p: ptr l, i: int i, x: a): void
// end of [arrset]
```

Before implementing arrget and arrset, I present as follows some code that implements a function template to access the first element of a nonempty array:

```
fun{a:t@ype}
arrgetfst {n:pos} {l:addr} (
  pf: !array_v (a, n, l) | p: ptr l
) : a = x where {
  prval array_v_cons (pf1, pf2) = pf
  // pf1: a @ l; pf2: array_v (a, n-1, l+sizeof(a))
  val x = !p
  prval () = pf := array_v_cons (pf1, pf2)
} // end of [arrgetfst]
```

Obviously, the function template arrget can be implemented based on arrgetfst:

```
implement{a}
arrget (pf | p, i) =
  if i > 0 then let
    prval array_v_cons (pf1, pf2) = pf
    val x = arrget (pf2 | p+sizeof<a>, i-1)
    prval () = pf := array_v_cons (pf1, pf2)
  in
    x
  end else
    arrgetfst (pf | p)
  // end of [if]
```

This implementation is of time-complexity O(n), and it is tail-recursive (after the proofs are erased). However, the very point of having arrays is to support O(1)-time accessing and updating operations. My initial effort spent on implementing such operations immediately dawned on me the need to construct proof functions for supporting view changes (of no run-time cost).

Clearly, an array starting at L that contains N elements of type T can also be thought of as two arrays: one starting at L that contains I elements while the other starting at L+I*sizeof(T) that contains N-I elements, where I is an natural number less that or equal to N. Formally, this statement can be encoded in the type of the proof function array_v_split:

```
prfun array_v_split
  {a:t@ype} {n,i:nat | i <= n} {l:addr} {ofs:int} (
  pfmul: MUL (i, sizeof(a), ofs), pfarr: array_v (a, n, l)
) : (array_v (a, i, l), array_v (a, n-i, l+ofs))
```

The other direction of the above statement can be encoded in the type of the proof function array_v_unsplit:

```
prfun array_v_unsplit
  {a:t@ype} {n1,n2:nat} {l:addr} {ofs:int} (
  pfmul: MUL (n1, sizeof(a), ofs)
, pf1arr: array_v (a, n1, l), pf2arr: array_v (a, n2, l+ofs)
) : array_v (a, n1+n2, l)
```

With array_v_split and array_v_unsplit, we can readily give implementations of arrget and arrset that are O(1)-time:

```
implement{a}
arrget (pf | p, i) = x where {
  val tsz = int1_of_size1 (sizeof<a>)
  val (pfmul | ofs) = i imul2 tsz
  prval (pf1, pf2) = array_v_split {a} (pfmul, pf)
  prval array_v_cons (pf21, pf22) = pf2
  val x = ptr_get1<a> (pf21 | p+ofs)
  prval pf2 = array_v_cons (pf21, pf22)
  prval () = pf := array_v_unsplit {a} (pfmul, pf1, pf2)
} // end of [arrget]

implement{a}
arrset (pf | p, i, x) = () where {
  val tsz = int1_of_size1 (sizeof<a>)
  val (pfmul | ofs) = i imul2 tsz
  prval (pf1, pf2) = array_v_split {a} (pfmul, pf)
  prval array_v_cons (pf21, pf22) = pf2
  val () = ptr_set1<a> (pf21 | p+ofs, x)
  prval pf2 = array_v_cons (pf21, pf22)
  prval () = pf := array_v_unsplit {a} (pfmul, pf1, pf2)
} // end of [arrset]
```

Note that the function int1_of_size1 is called to turn a size (i.e., an integer of the type size_t) into an int (i.e., an integer of the type int). Of course, the proof functions array_v_split and array_v_split are still to be implemented, which I will do when covering the topic of view change.

Given a type T and a natural number N, a proof of the view array_v(T?, N, L) for some address L can be obtained and released by calling the functions malloc and free, respectively, which are to be explained in details elsewhere. I now give as follows an implemention of a function template for array intialization:

```
typedef natLt (n:int) = [i:nat | i < n] int (i)
```

```
fun{a:t@ype}
array_ptr_tabulate
  {n:nat} {l:addr} (
  pf: !array_v (a?,n,l) >> array_v (a,n,l)
| p: ptr (l), n: int (n), f: natLt(n) -<cloref1> a
) : void = let
  fun loop {i:nat | i <= n} {l:addr} .<n-i>. (
    pf: !array_v (a?,n-i,l) >> array_v (a,n-i,l)
  | p: ptr l, n: int n, f: natLt(n) -<cloref1> a, i: int i
  ) : void =
    if i < n then let
      prval array_v_cons (pf1, pf2) = pf
      val () = !p := f (i)
      val () = loop (pf2 | p+sizeof<a>, n, f, i+1)
    in
      pf := array_v_cons (pf1, pf2)
    end else let
      prval array_v_nil () = pf in pf := array_v_nil {a} ()
    end // end of [if]
  // end of [loop]
in
  loop (pf | p, n, f, 0)
end // end of [array_ptr_tabulate]
```

Given a natural number n, the type natLt(n) is for all natural numbers less than n. Given a type T, the function array_ptr_tabulate<T> takes a pointer to an uninitialized array, the size of the array and a function f that maps each natural number less than n to a value of the type T, and it initializes the array with the sequence of values of f(0), f(1), ..., and f(n-1). In other words, the array contains a tabulation of the given function f after the initialization is over.

Given a type T and an integer N, @[T][N] is a built-in type in ATS for N consecutive values of the type T. Therefore, the at-view @[T][N]@L for any given address L is equivalent to the array-view array_v(T, N, L). By making use of the feature of call-by-reference, we can also assign the following interfaces to the previously presented functions arrget and arrset:

```
fun{a:t@ype}
arrget {n,i:nat | i < n} (A: &(@[a][n]), i: int i): a

fun{a:t@ype}
arrset {n,i:nat | i < n} (A: &(@[a][n]), i: int i, x: a): void
```

These interfaces are more concise as they obviate the need to mention explicitly where the array argument is located.

Please find the entirety of the above presented code on-line[3].

## Singly-Linked Lists

The following dataview slseg_v captures the notion of a singly-linked list segment:

```
dataview
slseg_v (
  a:t@ype+ // covariant argument
, int(*len*)
, addr(*beg*)
, addr(*end*)
) =
  | {l:addr} slseg_v_nil (a, 0, l, l) of ()
  | {n:nat} {l_fst:agz} {l_nxt,l_end:addr}
    slseg_v_cons (a, n+1, l_fst, l_end) of
      ((a, ptr l_nxt) @ l_fst, slseg_v (a, n, l_nxt, l_end))
```

```
// end of [slseg]_v
```

There are two proof constructors slseg_v_nil and slseg_v_cons associated with slseg_v, which are assigned the following types:

```
slseg_v_nil :
  {a:t@ype} {l:addr} () -> slseg_v (a, 0, l, l)
slseg_v_cons :
  {a:t@ype} {n:nat} {l_fst:agz} {l_nxt,l_end:addr}
  ((a, ptr l_nxt) @ l_fst, slseg_v (a, l_nxt, l_end)) -> slseg_v (a, n+1, l_fst, l_end)
```

Note that agz is a subset sort for addresses that are not null. Given a type T, a natural number N and two addresses L1 and L2, the view slseg_v (T, N, L1, L2) is for a singly-linked list segment containing N elements of the type T that starts at L1 and finishes at L2. In the case where L2 is the null pointer, then the list segment is considered a list as is formally defined below:

```
viewdef sllst_v
  (a:t@ype, n:int, l:addr) = slseg_v (a, n, l, null)
// end of [sllst_v]
```

Given a type T, a pointer pointing to L plus a proof of the view sllst_v(T, N, L) for some natural number N is essentially the same as a pointer to a struct of the following declared type sllst_struct in C:

```
typedef struct sllst {
  T data ; /* [T] matches the corresponding type in ATS */
  struct sllst *next ; /* pointing to the tail of the list */
} sllst_struct ;
```

Let us now see a simple example involving singly-linked lists:

```
fn{a:t@ype}
sllst_ptr_length
  {n:nat} {l:addr} (
  pflst: !sllst_v (a, n, l) | p: ptr l
) : int (n) = let
  fun loop {i,j:nat} {l:addr} .<i>. (
    pflst: !sllst_v (a, i, l) | p: ptr l, j: int (j)
  ) : int (i+j) =
    if p > null then let
      prval slseg_v_cons (pfat, pf1lst) = pflst
      val res = loop (pf1lst | !p.1, j+1) // !p.1 points to the tail
      prval () = pflst := slseg_v_cons (pfat, pf1lst)
    in
      res
    end else let // the length of a null list is 0
      prval slseg_v_nil () = pflst in pflst := slseg_v_nil (); j
    end (* end of [if] *)
  // end of [loop]
in
  loop (pflst | p, 0)
end // end of [sllst_ptr_length]
```

The function template sllst_ptr_length computes the length of a given singly-linked list. Note that the inner function loop is tail-recursive. The above implementation of sllst_ptr_length essentially corresponds to the following implementation in C:

```
int sllst_ptr_length (sllst_struct *p) {
  int res = 0 ;
  while (p != NULL) { res = res + 1 ; p = p->next ; }
  return res ;
```

```
} // end of [sllst_ptr_length]
```

As another example, the following function template sllst_ptr_reverse turns a given linked list into its reverse:

```
fn{a:t@ype}
sllst_ptr_reverse
  {n:nat} {l:addr} (
  pflst: sllst_v (a, n, l) | p: ptr l
) : [l:addr] (sllst_v (a, n, l) | ptr l) = let
  fun loop
    {n1,n2:nat}
    {l1,l2:addr} .<n1>. (
    pf1lst: sllst_v (a, n1, l1)
  , pf2lst: sllst_v (a, n2, l2)
  | p1: ptr l1, p2: ptr l2
  ) : [l:addr] (sllst_v (a, n1+n2, l) | ptr l) =
    if p1 > null then let
      prval slseg_v_cons (pf1at, pf1lst) = pf1lst
      val p1_nxt = !p1.1
      val () = !p1.1 := p2
    in
      loop (pf1lst, slseg_v_cons (pf1at, pf2lst) | p1_nxt, p1)
    end else let
      prval slseg_v_nil () = pf1lst in (pf2lst | p2)
    end // end of [if]
in
  loop (pflst, slseg_v_nil | p, null)
end // end of [sllst_ptr_reverse]
```

By translating the tail-recursive function loop into a while-loop, we can readily turn the implementation of sllst_ptr_reverse in ATS into the following implementation in C:

```
sllst_struct *sllst_ptr_reverse (sllst_struct *p) {
  sllst_struct *tmp, *res = NULL ;
  while (p != NULL) {
    tmp = p->next ; p->next = res ; res = p ; p = tmp ;
  }
  return res ;
} // end of [sllst_ptr_reverse]
```

Let us see yet another example. List concatenation is a common operation on lists. This time, we first give an implementation of list concatenation in C:

```
sllst_struct *sllst_ptr_append
  (sllst_struct *p, sllst_struct *q) {
  sllst_struct *p1 = p ;
  if (p1 == NULL) return q ;
  while (p1->next != NULL) p1 = p1->next ; p1->next = q ;
  return p ;
} // end of [sllst_ptr_append]
```

The algorithm is straightforward. If p is null, then q is returned. Otherwise, the last node in the list pointed to by p is first found and its field of the name next then replaced with q. This implementation of sllst_ptr_append in C can be translated directly into to following one in ATS:

```
fn{a:t@ype}
sllst_ptr_append
  {n1,n2:nat} {l1,l2:addr} (
  pf1lst: sllst_v (a, n1, l1)
, pf2lst: sllst_v (a, n2, l2)
```

```
| p1: ptr l1, p2: ptr l2
) : [l:addr] (sllst_v (a, n1+n2, l) | ptr l) = let
  fun loop
    {n1,n2:nat}
    {l1,l2:addr | l1 > null} .<n1>. (
    pf1lst: sllst_v (a, n1, l1)
  , pf2lst: sllst_v (a, n2, l2)
  | p1: ptr l1, p2: ptr l2
  ) : (sllst_v (a, n1+n2, l1) | void) = let
    prval slseg_v_cons (pf1at, pf1lst) = pf1lst
    val p1_nxt = !p1.1
  in
    if p1_nxt > null then let
      val (pflst | ()) = loop (pf1lst, pf2lst | p1_nxt, p2)
    in
      (slseg_v_cons (pf1at, pflst) | ())
    end else let
      val () = !p1.1 := p2
      prval slseg_v_nil () = pf1lst
    in
      (slseg_v_cons (pf1at, pf2lst) | ())
    end (* end of [if] *)
  end // end of [loop]
in
  if p1 > null then let
    val (pflst | ()) = loop (pf1lst, pf2lst | p1, p2)
  in
    (pflst | p1)
  end else let
    prval slseg_v_nil () = pf1lst in (pf2lst | p2)
  end (* end of [if] *)
end // end of [sllst_ptr_append]
```

In the above examples, it is evident that the code in ATS is a lot more verbose than its counterpart in C. However, the former is also a lot more robust than the latter in the following sense: If a minor change is made to the code in ATS (e.g., renaming identifiers, reordering function arguments), it is most likely that a type-error is to be reported when the changed code is typechecked. On the other hand, the same thing cannot be said about the code written in C. For instance, the following erroneous implementation of sllst_ptr_reverse in C is certainly type-correct:

```
/*
** This implementation is *incorrect* but type-correct:
*/
sllst_struct *sllst_ptr_reverse (sllst_struct *p) {
  sllst_struct *tmp, *res = NULL ;
  while (p != NULL) {
    tmp = p->next ; res = p ; p->next = res ; p = tmp ;
  }
  return res ;
} // end of [sllst_ptr_reverse]
```

I now point out that the dataview slseg_v is declared here in a manner that does not address the issue of allocating and freeing list nodes, and it is done so for the sake of a less involved presentation. A dataview for singly-linked lists that does handle allocation and deallocation of list nodes can be found in the libats library of ATS.

There is another method for handling singly-linked lists in ATS that is based on the notion of dataviewtype (i.e., linear datatype). When compared to the one presented above, this alternative is significantly simpler. However, dataviews are more general and flexible than dataviewtypes, and there are many common data structures (e.g. doubly-linked lists) that can only be properly handled with the former in ATS.

## Proof Functions for View Changes

By the phrase *view change*, I mean applying a function to proofs of a set of views to turn them into proofs of another set of views. As this function itself is a proof function, there is no run-time cost associated with each view change. For instance, a proof of the at-view int32@L for any address L can be turned into a proof of a tuple of 4 at-views: int8@L, int8@L+1, int8@L+2 and int8@L+3, where int32 and int8 are types for 32-bit and 8-bit integers, respectively. Often more interesting view changes involve recursively defined proof functions, and I present several of such cases in the rest of this section.

When implementing an array subscripting operation of O(1)-time, we need a proof function to change the view of one array into the views of two adjacently located arrays and another proof function to do precisely the opposite. Formally speaking, we need to construct the following two proof functions array_v_split and array_v_unsplit:

```
extern
prfun array_v_split
  {a:t@ype} {n,i:nat | i <= n} {l:addr} {ofs:int} (
  pfmul: MUL (i, sizeof(a), ofs), pfarr: array_v (a, n, l)
) : (array_v (a, i, l), array_v (a, n-i, l+ofs))

extern
prfun array_v_unsplit
  {a:t@ype} {n1,n2:nat} {l:addr} {ofs:int} (
  pfmul: MUL (n1, sizeof(a), ofs)
, pf1arr: array_v (a, n1, l), pf2arr: array_v (a, n2, l+ofs)
) : array_v (a, n1+n2, l)
```

An implementation of array_v_split is given as follows:

```
implement
array_v_split {a} (pfmul, pfarr) = let
  prfun split
    {n,i:nat | i <= n} {l:addr} {ofs:int} .<i>. (
    pfmul: MUL (i, sizeof(a), ofs), pfarr: array_v (a, n, l)
  ) : (array_v (a, i, l), array_v (a, n-i, l+ofs)) =
    sif i > 0 then let
      prval MULind (pf1mul) = pfmul
      prval array_v_cons (pf1at, pf1arr) = pfarr
      prval (pf1res1, pf1res2) = split (pf1mul, pf1arr)
    in
      (array_v_cons (pf1at, pf1res1), pf1res2)
    end else let
      prval MULbas () = pfmul in (array_v_nil (), pfarr)
    end // end of [sif]
in
  split (pfmul, pfarr)
end // end of [array_v_split]
```

Clearly, the proof function split directly encodes a proof based on mathematical induction. In addition, an implementation of array_v_unsplit is given as follows:

```
implement
array_v_unsplit {a}
  (pfmul, pf1arr, pf2arr) = let
  prfun unsplit
    {n1,n2:nat} {l:addr} {ofs:int} .<n1>. (
    pfmul: MUL (n1, sizeof(a), ofs)
  , pf1arr: array_v (a, n1, l)
  , pf2arr: array_v (a, n2, l+ofs)
  ) : array_v (a, n1+n2, l) =
    sif n1 > 0 then let
```

```
      prval MULind (pf1mul) = pfmul
      prval array_v_cons (pf1at, pf1arr) = pf1arr
      prval pfres = unsplit (pf1mul, pf1arr, pf2arr)
    in
      array_v_cons (pf1at, pfres)
    end else let
      prval MULbas () = pfmul
      prval array_v_nil () = pf1arr
    in
      pf2arr
    end // end of [sif]
in
  unsplit (pfmul, pf1arr, pf2arr)
end // end of [array_v_unsplit]
```

The proof function unsplit also directly encodes a proof based on mathematical induction.

Let us now see an even more interesting proof function for performing view change. The interface of the proof function array_v_takeout is given as follows:

```
extern
prfun array_v_takeout
  {a:t@ype} {n,i:nat | i < n} {l:addr} {ofs:int} (
  pfmul: MUL (i, sizeof(a), ofs), pfarr: array_v (a, n, l)
) : (a @ l+ofs, a @ l+ofs -<lin,prf> array_v (a, n, l))
```

Note that the following type is for a linear proof function that takes a proof of an at-view to return a proof of an array-view:

```
a @ l+ofs -<lin,prf> array_v (a, n, l)
```

As such a function essentially represents an array with one missing cell, we may simply say that array_v_takeout turns the view of an array into an at-view (for one cell) and a view for the rest of the array. By making use of array_v_takeout, we can give another implementation of arrget:

```
implement{a}
arrget (pf | p, i) = x where {
  val tsz = int1_of_size1 (sizeof<a>)
  val (pfmul | ofs) = i imul2 tsz
  prval (pf1, fpf2) = array_v_takeout {a} (pfmul, pf)
  val x = ptr_get1<a> (pf1 | p+ofs)
  prval () = pf := fpf2 (pf1) // putting the cell and the rest together
} // end of [arrget]
```

The proof function array_v_takeout can be implemented as follows:

```
implement
array_v_takeout
  {a} (pfmul, pfarr) = let
  prfun takeout
    {n,i:nat | i < n} {l:addr} {ofs:int} .<i>. (
    pfmul: MUL (i, sizeof(a), ofs), pfarr: array_v (a, n, l)
  ) : (a @ l+ofs, a @ l+ofs -<lin,prf> array_v (a, n, l)) = let
    prval array_v_cons (pf1at, pf1arr) = pfarr
  in
    sif i > 0 then let
      prval MULind (pf1mul) = pfmul
      prval (pfres, fpfres) = takeout (pf1mul, pf1arr)
    in
      (pfres, llam (pfres) => array_v_cons (pf1at, fpfres (pfres)))
    end else let
```

```
      prval MULbas () = pfmul
    in
      (pf1at, llam (pf1at) => array_v_cons (pf1at, pf1arr))
    end // end of [sif]
  end // end of [takeout]
in
  takeout (pfmul, pfarr)
end // end of [array_v_takeout]
```

Note that llam is a keyword for forming linear functons. Once a linear function is applied, it is consumed and the resource in it, if not reclaimed, moves into the result it returns.

The proof functions presented so far for view changes are all manipulating array-views. The following one is different in this regard as it combines two views for singly-linked list segments into one:

```
prfun slseg_v_unsplit
  {a:t@ype} {n1,n2:nat} {l1,l2,l3:addr} (
  pf1lst: slseg_v (a, n1, l1, l2), pf2lst: slseg_v (a, n2, l2, l3)
) : slseg_v (a, n1+n2, l1, l3)
```

The type of slseg_v_unsplit essentially states that a list segment from L1 to L2 that is of length N1 and another list segment from L2 to L3 that is of length N2 can be thought of as a list segment from L1 to L3 that is of length N1+N2. The following implementation of slseg_v_unsplit is largely parallel to that of array_v_unsplit:

```
implement
slseg_v_unsplit {a}
  (pf1lst, pf2lst) = let
  prfun unsplit
    {n1,n2:nat} {l1,l2,l3:addr} .<n1>. (
    pf1lst: slseg_v (a, n1, l1, l2), pf2lst: slseg_v (a, n2, l2, l3)
  ) : slseg_v (a, n1+n2, l1, l3) =
    sif n1 > 0 then let
      prval slseg_v_cons (pf1at, pf1lst) = pf1lst
    in
      slseg_v_cons (pf1at, unsplit (pf1lst, pf2lst))
    end else let
      prval slseg_v_nil () = pf1lst in pf2lst
    end // end of [sif]
in
  unsplit (pf1lst, pf2lst)
end // end of [slseg_v_unsplit]
```

The reader may find it interesting to give an implementation of sllst_ptr_append by making use of slseg_v_unsplit.

Please find on-line the files array.dats[4] and sllst.dats[5], which contains the code employed for illustration in this section.

## Example: Quicksort

Quicksort is a commonly employed sorting algorithm in practice. Given an array of n elements for some n > 0 and an ordering on these elements, the algorithm chooses one element in a more or less random fashion and then uses the chosen element as a pivot to shuffle the rest of the array into two parts separated by the pivot such that one part consists of all the elements that are less than or equal to the pivot (according to the given ordering) and the other part consists of the complement, that is, all the elements that are greater than the pivot; then the algorithm is applied recursively to each part unless it is empty. It is straightforward to see that the array is sorted after the algorithm terminates. In terms of time-complexity, quicksort is quadratic in the worst case and log-linear on average. Also, quicksort is not a stable sorting algorithm in the sense that the order of two equal elements may change after sorting.

The following function npivot returns the index of the element chosen to be the pivot:

```
fun{a:t@ype}
npivot {n:pos} {l:addr} (
  pf: !array_v (a, n, l) | p: ptr l, n: int (n), cmp: cmp (a)
) : natLt (n) = n/2
```

For simplicity, the index of the pivot for an array of size n is always n/2 (where integer division is used). Often a more elaborate method is to choose the index among 0, n/2 and n-1 such that the element stored at that index is between the elements stored at the other two indexes. Another possibility is to choose the index of the pivot based on a pseudo random number generator.

The function template array_ptr_exch for exchanging two elements in a given array is assgined the following interface:

```
extern
fun{a:t@ype}
array_ptr_exch {n:nat} {l:addr} (
  pf: !array_v (a, n, l) | p: ptr l, i: natLt n, j: natLt n
) : void // end of [array_ptr_exch]
```

I give no implementation of array_ptr_exch here as the reader should have no difficulty constructing one by now.

Given an array of elements, its size, an ordering on the elements and a pivot, the following function template split turns the array into two subarrays such that one subarray consists of all the elements in the array that are less than or equal to the pivot and the other subarray consists of the complement; it then returns the size of the first subarray plus proofs of the views of the two subarrays (as well as a proof for handling multiplication).

```
extern
fun{a:t@ype}
split {n:nat} {l:addr} (
  pf: array_v (a, n, l) | p: ptr l, n: int n, cmp: cmp a, piv: &a
) : [n1,n2:nat | n1+n2==n] [ofs:int] (
  MUL (n1, sizeof(a), ofs), array_v (a, n1, l), array_v (a, n2, l+ofs) | int n1
) // end of [split]
```

I postpone implementing split for the moment. Instead, let us first see an implementation of quicksort based on split:

```
fun{a:t@ype}
qsort {n:nat} {l:addr} .<n>. (
  pfarr: !array_v (a, n, l) | p: ptr l, n: int (n), cmp: cmp (a)
) : void =
  if n > 0 then let
    val tsz = int1_of_size1 (sizeof<a>)
    val npiv = npivot (pfarr | p, n, cmp)
    val () = array_ptr_exch (pfarr | p, 0, npiv) // move the pivot to the front
//
    val p1 = p+tsz
    prval array_v_cons (pfat, pf1arr) = pfarr
    val (pfmul, pf1arr_lte, pf1arr_gt | n1) = split (pf1arr | p1, n-1, cmp, !p)
// combining the pivot with the first segment
    prval pf1arr_lte = array_v_cons (pfat, pf1arr_lte)
// exchanging the pivot with the last element in the first segment
    val () = array_ptr_exch (pf1arr_lte | p, 0, n1)
// separating the pivot from all the elements in front of it
```

```
    prval (pf1arr_lte, pflast) = array_v_split {a} (pfmul, pf1arr_lte)
    val () = qsort (pf1arr_lte | p, n1, cmp) // recursive all to qsort
// combining the pivot with all the elements in front of it
    prval pf1arr_lte = array_v_unsplit {a} (pfmul, pf1arr_lte, pflast)
//
    val (pfmul_alt | ofs) = n1 imul2 tsz
    prval () = mul_isfun (pfmul, pfmul_alt)
    val () = qsort (pf1arr_gt | p1+ofs, n-n1-1, cmp) // recursive call to qsort
// combining the first and the second segments together
    prval () = pfarr := array_v_unsplit {a} (MULind (pfmul), pf1arr_lte, pf1arr_gt)
  in
    // nothing
  end else () // empty array
// end of [qsort]
```

The comments given in the implementation of qsort should make it rather clear how quicksort formally operates on a given array.

Let us now implement the function template split. Given an array, one common approach is to have two pointers pointing to the first and last elements of the array; the front pointer moves forward until an element that is not less than or equal to the pivot is encountered; the rear pointer moves backward until an element that is not greater than the pivot is encountered; the elements pointed to by the front and rear pointers are exchanged and the process is repeated; the process finishes at the moment when the front pointer either encounters or crosses over the rear one. As it is considerably involved to present an implementation based on this approach, I will use an alternative one, which I learned from the K&R book on the C programming language. This alternative approach starts with two pointers p1 and p2 pointing to the beginning of the given array and maintains the invariant that each element between p1 and p2 is greater than the pivot; first p2 moves forward until an element that is less than or equal to the pivot is encountered; then the elements stored at p1 and p2 are exchanged and both p1 and p2 move forward by one unit; the process repeats until p2 reaches the end of the array. For a slightly cleaner presentation, p1 is represented as a real pointer (p) in the implementation of the following inner function loop while p2 is represented as an integer (k):

```
implement{a}
split {n} (
  pfarr | p, n, cmp, piv
) = let
  fun loop
    {n1:nat}
    {k:nat | n1+k <= n}
    {l:addr} {ofs:int} .<n-n1-k>. (
    pfmul: MUL (n1, sizeof(a), ofs)
  , pf1arr: array_v (a, n1, l)
  , pf2arr: array_v (a, n-n1, l+ofs)
  | p: ptr (l+ofs), n: int n, n1: int n1, k: int k, cmp: cmp(a), piv: &a
  ) : [n1,n2:nat | n1+n2==n] [ofs:int] (
    MUL (n1, sizeof(a), ofs), array_v (a, n1, l), array_v (a, n2, l+ofs)
  | int (n1)
  ) = // [loop] is tail-recursive
    if n1+k < n then let
      val (pfat, fpf2 | pk) = array_ptr_takeout (pf2arr | p, k)
      val sgn = compare (!pk, piv, cmp)
      prval () = pf2arr := fpf2 (pfat)
    in
      if sgn > 0 then
        loop (pfmul, pf1arr, pf2arr | p, n, n1, k+1, cmp, piv)
      else let
        val () = array_ptr_exch (pf2arr | p, 0, k) // no work is done if k = 0
        prval array_v_cons (pfat, pf2arr) = pf2arr
        prval () = pf1arr := array_v_extend {a} (pfmul, pf1arr, pfat)
      in
```

```
        loop (MULind (pfmul), pf1arr, pf2arr | p+sizeof<a>, n, n1+1, k, cmp, piv)
      end (* end of [if] *)
    end else (
      pfmul, pf1arr, pf2arr | n1
    ) // end of [if]
in
  loop (MULbas (), array_v_nil (), pfarr | p, n, 0, 0, cmp, piv)
end // end of [split]
```

Note the proof function array_v_extend is given the following interface:

```
prfun array_v_extend
  {a:t@ype} {n:nat} {l:addr} {ofs:int} (
  pfmul: MUL (n, sizeof(a), ofs), pfarr: array_v (a, n, l), pfat: a @ l+ofs
) : array_v (a, n+1, l)
```

This proof function can be thought of as a special case of array_v_unsplit where the second array is a singleton, that is, it contains exactly one element.

Please find the entire code in this section plus some additional code for testing on-line[6].

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVIEWS/
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVIEWS/optview
3. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVIEWS/array.da
4. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVIEWS/array.da
5. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVIEWS/sllst.dat
6. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVIEWS/quickso

# Chapter 20. Dataviewtypes as Linear Datatypes

A dataviewtype can be thought of as a linear version of datatype. To a large extent, it is a combination of a datatype and a dataview. This programming feature is primarily introduced into ATS for the purpose of providing the kind convenience of pattern matching associated with datatypes while incurring no need for run-time garbage collection (GC). In a situation where GC must be reduced or even completely eliminated, dataviewtypes can often chosen as a replacement for datatypes. I now present in this chapter some commonly encountered dataviewtypes and their uses.

## Linear Optional Values

When an optional value is created, it is most likely to be used immediately and then discarded. If such a value is assigned a linear type, then the memory allocated for storing it can be efficiently reclaimed. The dataviewtype option_vt for linear optional values is declared as follows:

```
dataviewtye
option_vt (a:t@ype+, bool) =
  | Some_vt (a, true) of a | None_vt (a, false) of ()
// end of [option_vt]

viewtypedef
Option_vt (a:t@ype) = [b:bool] option_vt (a, b)
```

By the declaration, the dataviewtype option_vt is covariant in its first argument and there are two data constructors Some_vt and None_vt associated with it. In the following example, find_rightmost tries to find the rightmost element in a list that satisfies a given predicate:

```
fun{a:t@ype}
find_rightmost {n:nat} .<n>.
  (xs: list (a, n), P: (a) -<cloref> bool): Option_vt (a) =
  case+ xs of
  | list_cons (x, xs) => let
      val opt = find_rightmost (xs, P)
    in
      case opt of
      | ~None_vt () => if P (x) then Some_vt (x) else None_vt ()
      | _ => opt
    end // end of [list_cons]
  | list_nil () => None_vt ()
// end of [find_rightmost]
```

Note that the tilde symbol (~) in front of the pattern None_vt() indicates that the memory for the node that matches the pattern is freed before the body of the matched clause is evaluated. In this case, no memory is actually freed as None_vt is mapped to the null pointer. I will soon give more detailed explanation about freeing memory allocated for constructors associated with dataviewtypes.

As another example, the following function template list_optcons tries to construct a new list with its head element extracted from a given optional value:

```
fn{a:t@ype}
list_optcons {b:bool} {n:nat} (
  opt: option_vt (a, b), xs: list (a, n)
) : list (a, n+int_of_bool(b)) =
  case+ opt of
  | ~Some_vt (x) => list_cons (x, xs) | ~None_vt () => xs
// end of [list_optcons]
```

The symbol int_of_bool stands for a built-in static function in ATS that maps true and false to 1 and 0, respectively. What is special here is that the first argument of list_optcons, which is linear, is consumed after a call to list_optcons returns and the memory it occupies is reclaimed.

## Linear Lists

A linear list is essentially the same as a singly-linked list depicted by the dataview sllst_v. However, memory allocation and deallocation of list nodes that were not handled previously are handled this time. The following declaration introduces a dataviewtype list_vt, which forms a boxed type (of the sort viewtype) when applied to a type and an integer:

```
dataviewtype
list_vt (a:t@ype+, int) =
  | {n:nat}
    list_vt_cons (a, n+1) of (a, list_vt (a, n))
  | list_vt_nil (a, 0) of ()
// end of [list_vt]
```

Assume that a data constructor named *foo* is associated with a dataviewtype. Then there is a viewtype construtor of the name *foo_unfold* that takes n addresses to form a viewtype, where n is the arity of *foo*. For instance, there is a viewtype constructor list_vt_cons_unfold that takes two address L0 and L1 to form a viewtype list_vt_cons_unfold(L0, L1). This viewtype is for a list node created by a call to list_vt_cons such that the two arguments of list_vt_cons are located at L0 and L1 while the proofs for the at-views associated with L0 and L1 are put in the store for currently available proofs.

Given a type T and an integer I, the viewtype list_vt(T, I) is for linear lists of length I in which each element is assigned the type T. The following function template length computes the length of a given linear list:

```
fn{a:t@ype}
length {n:nat}
  (xs: !list_vt (a, n)): int n = let
  fun loop
    {i,j:nat | i+j==n} .<i>.
    (xs: !list_vt (a, i), j: int j): int (n) =
    case+ xs of
    | list_vt_cons (_, !p_xs1) => let
        val n = loop (!p_xs1, j+1); val () = fold@ (xs) in n
      end // end of [list_vt_cons]
    | list_vt_nil () => (fold@ (xs); j)
  // end of [loop]
in
  loop (xs, 0)
end // end of [length]
```

The interface of length indicates that length<T> returns an integer equal to I when applied to a list of the type list_vt(T, I), where T and I are a type and an integer, respectively. Note that the symbol ! in front of the type of a function argument indicates that the argument is call-by-value and it is preserved after a call to the function.

What is particularly interesting here is the way in which pattern matching on a value of a dataviewtype works. In the body of the inner function loop, the type of xs changes to list_vt_cons_unfold(L1, L2) for some addresses L1 and L2 when it matches the pattern list_vt_cons(_, !p_xs1), and p_xs1 is bound to a value of the type ptr(L2), and a proof of the at-view a@L1 and another proof of the at-view list_vt(a,n-1)@L2 are automatically put into the store for the currently availble proofs. Note that the symbol ! in front of the variable p_xs1 indicates that p_xs1 is bound to the pointer to the tail of the list referred to by xs1 (rather than the tail itself). In order to change the type of xs back to the type list_vt(a, n), we can apply fold@ to xs and this application implicitly consumes a proof of the at-view a@L1 and another proof of the at-view list_vt(a, n-1)@L2. Note that fold@ is a keyword in ATS, and an application of fold@ is treated as a proof and it is erased after typechecking. In the case where xs matches the pattern list_vt_nil(), the type of xs changes into list_vt_nil() while there is no proof added to the store for the currently available proofs, and the type of xs restores to list_vt(a, 0) when fold@ is applied to it.

Let us now see an example involving a linear list being freed manually:

```
fun{a:t@ype}
list_vt_free
  {n:nat} .<n>. (xs: list_vt (a, n)): void =
  case+ xs of
  | ~list_vt_cons (x, xs1) => list_vt_free (xs1) // [x] can be replaced with [_]
  | ~list_vt_nil () => ()
// end of [list_vt_free]
```

In the case where xs matches the pattern list_vt_cons(x, xs1), the names x and xs1 are bound to the head and the tail of the list referred to by xs, respectively, and the type of xs changes to list_vt_cons(L1, L2) for some addresses while a proof of the at-view a@L1 and another proof of the at-view list_vt(a, n-1)?!@L2 are put into the store for currently available proofs. Note that the symbol ?! indicates that the tail of the list, which is linear, is already taken out (as it is now referred by xs1). The special symbol ~ in front of the pattern list_vt_cons(x, xs1) indicates that the list node referred to by xs after xs matches the pattern is freed immediately.

It is also possible to use the special function free@ to explicitly free a node (also called a skeleton) left in a linear variable after the variable matches a pattern formed with a constructor associated with a dataviewtypes. For instance, the following code gives another implementation of list_vt_free:

```
fun{a:t@ype}
list_vt_free
  {n:nat} .<n>. (xs: list_vt (a, n)): void =
  case+ xs of
  | list_vt_cons (x, xs1) => (free@ {a}{0} (xs); list_vt_free (xs1))
  | list_vt_nil () => free@ {a} (xs)
// end of [list_vt_free]
```

As using free@ is a bit tricky in practice, I present more details as follows. First, let us note that the constructors list_vt_nil and list_vt_cons associated with list_vt are assigned the following types:

```
list_vt_nil : // one quantifier
  {a:t@ype} () -> list_vt (a, 0)
list_vt_cons : // two quantifiers
  {a:t@ype} {n:nat} (a, list_vt (a, n)) -> list_vt (a, n+1)
```

If free@ is applied to a node of the type list_vt_nil(), it needs one static argument. which is a type, to instantiate the quantifier in the type of the constructor list_vt_nil. If free@ is applied to a node of the type list_vt_cons_unfold(L1, L2), then it needs two static arguments, which are a type and an integer, to instantiate the two quantifiers in the type of the constructor list_vt_cons. In the case where the type of xs is list_vt_cons_unfold(L1, L2), typechecking the call free@ {a}{0} (xs) implicitly consumes a proof of the at-view a?@L1 and another proof of the at-view list_vt(a, 0)?. As there is no difference between list_vt(T, 0)? and list_vt(T, I)? for any T and I, the static argument 0 is supplied here. As a matter of fact, any natural number can be used in place of 0 as the second static argument of free@.

The next example I present is a function template that turns a linear list into its reverse:

```
fn{a:t@ype}
reverse {n:nat}
  (xs: list_vt (a, n)): list_vt (a, n) = let
  fun revapp
    {i,j:nat | i+j==n} .<i>.
    (xs: list_vt (a, i), ys: list_vt (a, j)): list_vt (a, n) =
    case+ xs of
    | list_vt_cons (_, !p_xs1) => let
        val xs1 = !p_xs1; val () = !p_xs1 := ys; val () = fold@ (xs)
      in
        revapp (xs1, xs)
      end // end of [list_vt_cons]
    | ~list_vt_nil () => ys
  // end of [revapp]
in
  revapp (xs, list_vt_nil)
end // end of [reverse]
```

This implementation of list reversal directly corresponds to the one presented previously that is based the dataview slseg_v (for singly-linked list segments). Comparing the two implementations, we can see that the above one is significantly simplified at the level of types. For instance, there is no explicit mentioning of pointers in the types assigned to reverse and revapp.

The following implementation of list append makes use of the feature of call-by-reference:

```
fn{a:t@ype}
append {m,n:nat} (
  xs: list_vt (a, m), ys: list_vt (a, n)
) : list_vt (a, m+n) = let
  fun loop {m,n:nat} .<m>. // [loop] is tail-recursive
    (xs: &list_vt (a, m) >> list_vt (a, m+n), ys: list_vt (a, n)): void =
    case+ xs of
    | list_vt_cons (_, !p_xs1) => let
        val () = loop (!p_xs1, ys) in fold@ (xs)
      end // end of [list_vt_cons]
    | ~list_vt_nil () => xs := ys // [xs] is a left-value
  // end of [loop]
  var xs: List_vt (a) = xs // creating a left-value for [xs]
  val () = loop (xs, ys)
in
  xs
end // end of [append]
```

As the call fold@(xs) in the body of the function loop is erased after typechecking, loop is a tail-recursive function. Therefore, append can be called on lists of any length without the concern of possible stack overflow. The type for the first argument of loop begins with the symbol &, which indicates that this argument is call-by-reference. The type of loop simply means that its first argument is changed from a list of length m into a list of length m+n while its second argument is consumed. Again, this implementation of list append essentially corresponds to the one presented previously that is based on the dataview slseg_v. Comparing these two, we can easily see that the above one is much simpler and cleaner, demonstrating concretely some advantage of dataviewtypes over dataviews.

Lastly in this section, I mention a closely related issue involving (functional) list construction and tail-recursion. Following is a typical implementation of functioal list concatenation:

```
fun{a:t@ype}
append1 {m,n:nat}
  (xs: list (a, m), ys: list (a, n)): list (a, m+n) =
  case+ xs of
  | list_cons (x, xs) => list_cons (x, append1 (xs, ys))
  | list_nil () => ys
// end of [append1]
```

Clearly, append1 is not tail-recursive, which means that it may cause stack overflow at run-time if its first argument is very long (e.g., containing 1 million elements). There is, however, a direct and type-safe way in ATS to implement functional list concatenation in a tail-recursive manner, thus eliminating the concern of potential stack overflow. For instance, the following implementation of append2 returns the concatenation of two given lists while being tail-recursive:

```
fun{a:t@ype}
append2 {m,n:nat} (
  xs: list (a, m)
, ys: list (a, n)
) : list (a, m+n) = let
  fun loop
    {m,n:nat} .<m>. (
    xs: list (a, m)
  , ys: list (a, n)
  , res: &(List a)? >> list (a, m+n)
  ) :<> void = begin case+ xs of
    | list_cons (x, xs) => let
        val () = (
          res := list_cons {a}{0} (x, ?) // a partially initialized list
        ) // end of [val]
        val+ list_cons (_, !p) = res // [p] points to the tail of the list
        val () = loop (xs, ys, !p)
      in
        fold@ res // this is a no-op at run-time
      end // end of [list_cons]
    | list_nil () => (res := ys)
  end // end of [loop]
  var res: List a // uninitialized variable
  val () = loop (xs, ys, res)
in
  res
end // end of [append2]
```

During typechecking, the expression list_cons {a}{0} (x, ?), is assigned the (linear) type list_cons(L1, L2) for some addresses L1 and L2 while a proof of the at-view a@L1 and another proof of the at-view list(a, 0)?@L2 are put into the store for the currently available proofs. Note that the special symbol ? here simply indicates that the tail of the newly constructed list is uninitialized. A partially initialized list of the type list_cons(L1, L2) is guaranteed to match the pattern list_cons(_, !p), yielding a bindng between p and the (possibly uninitialized) tail of the list. When fold@ is called on a variable of the type list_cons(L1, L2), it changes the type of the variable to list(T, N+1) by consuming a proof of the view T@L1 and another proof of the view list(T, N), where T and N are a type and an integer, respectively.

In summary, dataviewtypes can largely retain the convenience of pattern matching associated with datatypes while requiring no GC support at run-time. Compared to dataviews, dataviewtypes are less general. However, if a dataviewtype can be employed to solve a problem, then the solution is often significantly simpler and cleaner than an alternative one based a dataview.

## Example: Mergesort on Linear Lists

When mergesort is employed to sort an array of elements, it requires additional memory proportionate to the size of the array in order to move the elements around, which is considered a significant weakness of mergesort. However, mergesort does not have this requirement when it operates on a linear list. I present as follows an implementation of mergesort on linear lists that can readily rival its counterpart in C in terms of time-efficiency as well as memory-efficiency. The invariants captured in this implementation and the easiness to capture them should provide strong evidence to ATS being a programming language capable of enforcing great precision in practical programming.

First, let us introduce a type definition and an interface for a function template (for comparing elements in a list to be sorted):

```
typedef cmp (a:t@ype) = (&a, &a) -> int

extern
fun{a:t@ype} compare (x: &a, y: &a, cmp: cmp (a)): int
```

The interface for mergesort is given as follows:

```
extern
fun{a:t@ype}
mergesort {n:nat}
  (xs: list_vt (a, n), cmp: cmp a): list_vt (a, n)
// end of [mergesort]
```

The first argument of mergesort is a linear list (to be sorted) and the second one a function for comparing the elements in the linear list. Clearly, the interface of mergesort indicates that mergesort consumes its first argument and then returns a linear list that is of same length as its first argument. As is to become clear, the returned linear list is constructed with the nodes of the consumed one. In particular, the implementation of mergesort given here does not involve any memory allocation or deallocation.

The function template for merging two sorted lists into one is given as follows:

```
fun{a:t@ype}
merge // tail-rec
  {m,n:nat} .<m+n>. (
  xs: list_vt (a, m)
, ys: list_vt (a, n)
, res: &List_vt(a)? >> list_vt (a, m+n)
, cmp: cmp a
) : void =
  case+ xs of
  | list_vt_cons (!p_x, !p_xs1) => (
    case+ ys of
    | list_vt_cons (!p_y, !p_ys1) => let
        val sgn = compare<a> (!p_x, !p_y, cmp)
      in
        if sgn <= 0 then let // stable sorting
          val () = res := xs
          val xs1 = !p_xs1
          val () = fold@ (ys)
          val () = merge (xs1, ys, !p_xs1, cmp)
        in
          fold@ (res)
        end else let
          val () = res := ys
          val ys1 = !p_ys1
          val () = fold@ (xs)
          val () = merge (xs, ys1, !p_ys1, cmp)
```

```
        in
          fold@ (res)
        end // end of [if]
      end (* end of [list_vt_cons] *)
    | ~list_vt_nil () => (fold@ (xs); res := xs)
    ) // end of [list_vt_cons]
  | ~list_vt_nil () => (res := ys)
// end of [merge]
```

Unlike the one given in a previous functional implementation, this implementation of merge is tail-recursive and thus is guaranteed to be translated into a loop in C by the ATS compiler. This means that the concern of merge being unable to handle very long lists (e.g., containg 1 million elements) due to potential stack overflow is completely eliminated.

The next function template is for splitting a given linear lists into two:

```
fun{a:t@ype}
split {n,k:nat | k <= n} .<n-k>. (
  xs: &list_vt (a, n) >> list_vt (a, n-k), nk: int (n-k)
) : list_vt (a, k) =
  if nk > 0 then let
    val+ list_vt_cons (_, !p_xs1) = xs
    val res = split (!p_xs1, nk-1); val () = fold@ (xs)
  in
    res
  end else let
    val res = xs; val () = xs := list_vt_nil () in res
  end // end of [if]
// end of [split]
```

Note that the implementation of split is also tail-recursive.

The following function template msort takes a linear list, its length and a comparsion function, and it returns a sorted version of the given linear list:

```
fun{a:t@ype}
msort {n:nat} .<n>. (
  xs: list_vt (a, n), n: int n, cmp: cmp(a)
) : list_vt (a, n) =
  if n >= 2 then let
    val n2 = n / 2
    val n3 = n - n2
    var xs = xs // a left-value for [xs]
    val ys = split {n,n/2} (xs(*cbr*), n3) // xs: call-by-ref
    val xs = msort (xs, n3, cmp)
    val ys = msort (ys, n2, cmp)
    var res: List_vt (a)
    val () = merge (xs, ys, res(*cbr*), cmp) // xs: call-by-ref
  in
    res
  end else xs
// end of [msort]
```

The second argument of msort is passed so that the length of the list being sorted does not have to be computed directly by traversing the list when each recursive call to msort is made.

Finally, mergesort can be implemented with a call to msort:

```
implement{a}
mergesort (xs, cmp) = msort (xs, length (xs), cmp)
```

Please find the entire code in this section plus some additional code for testing on-line[1].

## Linear Binary Search Trees

A binary search tree with respect to a given ordering is a binary tree such that the value stored in each node inside the tree is greater than or equal to those stored in the left child of the node and less than or equal to those stored in the right child of the node. Binary search trees are a common data structure for implementing finite maps.

A family of binary trees are said to be balanced if there is a fixed constant C (for the entire family) such that the ratio between the length of a longest path and the length of a shortest path is bounded by C for every tree in the family. For instance, common examples of balanced binary trees include AVL trees and red-black trees. Finite maps based on balanced binary search trees support guaranteed log-time insertion and deletion operations, that is, the time to complete such an operation is O(log(n)) in the worst case, where n is the size of the map.

In this section, I am to implement several basic operations on linear binary search trees, further illustrating some use of dataviewtypes. Let us first declare as follows a dataviewtype bstree_vt for linear binary (search) trees:

```
dataviewtype
bstree_vt
  (a:t@ype+, int) =
  | {n1,n2:nat}
    bstree_vt_cons (a, 1+n1+n2) of
      (bstree_vt (a, n1), a, bstree_vt (a, n2))
  | bstree_vt_nil (a, 0) of ()
// end of [bstree_vt]
```

Note that the integer index of bstree_vt captures the size information of a binary (search) tree. There are two constructors bstree_vt_cons and bstree_vt_nil associated with bstree_vt. It should be pointed out that the tree created by bstree_vt_nil is empty and thus not a leaf, which on the other hand is a node whose left and right children are both empty. As a simple example, the following function template size computes the size of a given tree:

```
fun{a:t@ype}
size {n:nat} .<n>. (
  t: !bstree_vt (a, n)
) : int (n) =
  case+ t of
  | bstree_vt_cons (!p_tl, _, !p_tr) => let
      val n = 1 + size (!p_tl) + size (!p_tr) in fold@ (t); n
    end // end of [bstree_vt_cons]
  | bstree_vt_nil () => (fold@ (t); 0)
// end of [size]
```

Assume that we have a binary search tree with repect to some ordering. If a predicate P on the elements stored in the tree possesses the property that P(x1) implies P(x2) whenever x1 is less than x2 (according to the ordering), then we can locate the least element in the tree that satisfies the predicate P by employing so-called binary search as is demonstrated in the following implementation of search:

```
fun{a:t@ype}
search {n:nat} .<n>. (
  t: !bstree_vt (a, n), P: (&a) -<cloref> bool
) : Option_vt (a) =
  case+ t of
  | bstree_vt_cons
```

```
      (!p_tl, !p_x, !p_tr) =>
      if P (!p_x) then let
        val res = search (!p_tl, P)
        val res = (
          case+ res of
          | ~None_vt () => Some_vt (!p_x) | _ => res
        ) : Option_vt (a)
      in
        fold@ (t); res
      end else let
        val res = search (!p_tr, P) in fold@ (t); res
      end // end of [if]
  | bstree_vt_nil () => (fold@ (t); None_vt ())
// end of [search]
```

Clearly, if the argument t of search ranges over a family of balanced trees, then the time-complexity of search is O(log(n)) (assuming that P is O(1)).

Let us next see an operation that inserts a given element into a binary search tree:

```
fun{a:t@ype}
insert {n:nat} .<n>. (
  t: bstree_vt (a, n), x0: &a, cmp: cmp(a)
) : bstree_vt (a, n+1) =
  case+ t of
  | bstree_vt_cons
      (!p_tl, !p_x, !p_tr) => let
      val sgn = compare<a> (x0, !p_x, cmp)
    in
      if sgn <= 0 then let
        val () = !p_tl := insert (!p_tl, x0, cmp)
      in
        fold@ (t); t
      end else let
        val () = !p_tr := insert (!p_tr, x0, cmp)
      in
        fold@ (t); t
      end (* end of [if] *)
    end // end of [bstree_vt_cons]
  | ~bstree_vt_nil () =>
      bstree_vt_cons (bstree_vt_nil, x0, bstree_vt_nil)
    // end of [bstree_vt_nil]
// end of [insert]
```

When inserting an element, the function template insert extends the given tree with a new leaf node containing the element, and this form of insertion is often referred to as leaf-insertion. There is another form of insertion often referred to as root-insertion, which always puts at the root position the new node containing the inserted element. The following function template insertRT is implemented to perform a standard root-insertion operation:

```
fun{a:t@ype}
insertRT {n:nat} .<n>. (
  t: bstree_vt (a, n), x0: &a, cmp: cmp(a)
) : bstree_vt (a, n+1) =
  case+ t of
  | bstree_vt_cons
      (!p_tl, !p_x, !p_tr) => let
      val sgn = compare<a> (x0, !p_x, cmp)
    in
      if sgn <= 0 then let
        val tl = insertRT (!p_tl, x0, cmp)
        val+ bstree_vt_cons (_, !p_tll, !p_tlr) = tl
        val () = !p_tl := !p_tlr
        val () = fold@ (t)
```

```
          val () = !p_tlr := t
        in
          fold@ (tl); tl
        end else let
          val tr = insertRT (!p_tr, x0, cmp)
          val+ bstree_vt_cons (!p_trl, _, !p_trr) = tr
          val () = !p_tr := !p_trl
          val () = fold@ (t)
          val () = !p_trl := t
        in
          fold@ (tr); tr
        end
      end // end of [bstree_vt_cons]
    | ~bstree_vt_nil () =>
        bstree_vt_cons (bstree_vt_nil, x0, bstree_vt_nil)
      // end of [bstree_vt_nil]
// end of [insertRT]
```

The code immediately following the first recursive call to insertRT performs a right tree rotation. Let us use T(tl, x, tr) for a tree such that its root node contains the element x and its left and right children are tl and tr, respectively. Then a right rotation turns T(T(tll, xl, tlr), x, tr) into T(tll, xl, T(tlr, x, tr)). The code immediately following the second recursive call to insertRT performs a left tree rotation, which turns T(tl, x, T(trl, xr, trr)) into T(T(tl, x, tlr), xr, trr).

To further illustrate tree rotations, I present as follows two function templates lrotate and rrotate, which implement the left and right tree rotations, respectively:

```
fn{a:t@ype}
lrotate
  {nl,nr:nat | nr > 0}
  {l_tl,l_x,l_tr:addr} (
  pf_tl: bstree_vt (a, nl) @ l_tl
, pf_x: a @ l_x
, pf_tr: bstree_vt (a, nr) @ l_tr
| t: bstree_vt_cons_unfold (l_tl, l_x, l_tr)
, p_tl: ptr l_tl
, p_tr: ptr l_tr
) : bstree_vt (a, 1+nl+nr) = let
  val tr = !p_tr
  val+ bstree_vt_cons (!p_trl, _, !p_trr) = tr
  val () = !p_tr := !p_trl
  val () = fold@ (t)
  val () = !p_trl := t
in
  fold@ (tr); tr
end // end of [lrotate]

fn{a:t@ype}
rrotate
  {nl,nr:nat | nl > 0}
  {l_tl,l_x,l_tr:addr} (
  pf_tl: bstree_vt (a, nl) @ l_tl
, pf_x: a @ l_x
, pf_tr: bstree_vt (a, nr) @ l_tr
| t: bstree_vt_cons_unfold (l_tl, l_x, l_tr)
, p_tl: ptr l_tl
, p_tr: ptr l_tr
) : bstree_vt (a, 1+nl+nr) = let
  val tl = !p_tl
  val+ bstree_vt_cons (!p_tll, x, !p_tlr) = tl
  val () = !p_tl := !p_tlr
  val () = fold@ (t)
  val () = !p_tlr := t
in
```

```
  fold@ (tl); tl
end // end of [rrotate]
```

Given three addresses L0, L1 and L2, the type bstree_vt_cons_unfold(L0, L1, l2) is for a tree node created by a call to bstree_vt_cons such that the three arguments of bstree_vt_cons are located at L0, L1 and L2, and the proofs for the at-views associated with L0, L1 and L2 are put in the store for the currently available proofs.

The function template insertRT for root-insertion can now be implemented as follows by making direct use of lrotate and rrotate:

```
fun{a:t@ype}
insertRT {n:nat} .<n>. (
  t: bstree_vt (a, n), x0: &a, cmp: cmp(a)
) : bstree_vt (a, n+1) =
  case+ t of
  | bstree_vt_cons
      (!p_tl, !p_x, !p_tr) => let
      val sgn = compare<a> (x0, !p_x, cmp)
    in
      if sgn <= 0 then let
        val () = !p_tl := insertRT (!p_tl, x0, cmp)
      in
        rrotate (view@(!p_tl), view@(!p_x), view@(!p_tr) | t, p_tl, p_tr)
      end else let
        val () = !p_tr := insertRT (!p_tr, x0, cmp)
      in
        lrotate (view@(!p_tl), view@(!p_x), view@(!p_tr) | t, p_tl, p_tr)
      end
    end // end of [bstree_vt_cons]
  | ~bstree_vt_nil () =>
      bstree_vt_cons (bstree_vt_nil, x0, bstree_vt_nil)
    // end of [bstree_vt_nil]
// end of [insertRT]
```

I would like to point out that neither insert nor insertRT is tail-recursive. While it is straightforward to give the former a tail-recursive implementation, there is no direct way to do the same to the latter. In order to implement root-insertion in a tail-recursive manner, we are in need of binary search trees with parental pointers (so as to allow each node to gain direct access to its parent), which can be done with dataviews but not with dataviewtypes.

Please find the entire code in this section plus some additional code for testing online[2].

## Transition from Datatypes to Dataviewtypes

Many programmers are likely to find it a rather involved task to write code manipulating values of dataviewtypes. When handling a complex data structure, I myself often try to first use a datatype to model the data structure and implement some functionalities of the data structure based the datatype. I then change the datatype into a corresponding dataviewtype and modify the implementation accordingly to make it work with the dataviewtype. I now present as follows an implementation of linear red-black trees that is directly based on a previous implementation of functional red-black trees, illustrating concretely a kind of gradual transition from datatypes to dataviewtypes that can greatly reduce the level of difficulty one may otherwise encounter in an attempt to program with dataviewtypes directly.

The following declaration of dataviewtype rbtree is identical to the previous declaration of datatype rbtree except the keyword dataviewtype being now used instead of the keyword datatype:

```
#define BLK 0; #define RED 1
sortdef clr = {c:int | 0 <= c; c <= 1}

dataviewtype
rbtree (
  a: t@ype, int(*c*), int(*bh*), int(*v*)
) = // element type, color, black height, violations
  | rbtree_nil (a, BLK, 0, 0) of ()
  | {c,cl,cr:clr} {bh:nat} {v:int}
      {c==BLK && v==0 || c == RED && v==cl+cr}
    rbtree_cons (a, c, bh+1-c, v) of (
      int c, rbtree0 (a, cl, bh), a, rbtree0 (a, cr, bh)
    ) // end of [rbtree_cons]
// end of [rbtree]

where rbtree0 (a:t@ype, c:int, bh:int) = rbtree (a, c, bh, 0)
```

At the first sight, the following function template insfix_l is greatly more involved that a previous version of the same name (for manipulating functional red-black trees):

```
fn{a:t@ype}
insfix_l // right rotation
  {cl,cr:clr}
  {bh:nat}
  {v:nat}
  {l_c,l_tl,l_x,l_tr:addr} (
  pf_c: int(BLK) @ l_c
, pf_tl: rbtree (a, cl, bh, v) @ l_tl
, pf_x: a @ l_x
, pf_tr: rbtree (a, cr, bh, 0) @ l_tr
| t: rbtree_cons_unfold (l_c, l_tl, l_x, l_tr)
, p_tl: ptr (l_tl)
) : [c:clr] rbtree0 (a, c, bh+1) = let
  #define B BLK
  #define R RED
  #define cons rbtree_cons
in
  case+ !p_tl of
  | cons (!p_cl as R, !p_tll as cons (!p_cll as R, _, _, _), _, !p_tlr) => let
//
      val () = !p_cll := B
      val () = fold@ (!p_tll)
//
      val tl = !p_tl
      val () = !p_tl := !p_tlr
      val () = fold@ (t)
//
      val () = !p_tlr := t
    in
      fold@ (tl); tl
    end // end of [cons (R, cons (R, ...), ...)]
  | cons (!p_cl as R, !p_tll, _, !p_tlr as cons (!p_clr as R, !p_tlrl, _, !p_tlrr)) =>
//
      val tl = !p_tl
      val () = !p_tl := !p_tlrr
      val () = fold@ (t)
      val () = !p_tlrr := t
//
      val tlr = !p_tlr
      val () = !p_tlr := !p_tlrl
      val () = !p_cl := B
      val () = fold@ (tl)
      val () = !p_tlrl := tl
//
```

```
    in
      fold@ (tlr); tlr
    end // end of [cons (R, ..., cons (R, ...))]
  | _ =>> (fold@ (t); t)
end // end of [insfix_l]
```

However, I would like to point out that the interface for the above insfix_l is a *direct* translation of the interface for the previous infix_l. In other words, the previously captured relation between a tree being rotated and the one obtained from applying infix_l to it also holds in the setting of linear red-black trees. The very same can be said about the following function template insfix_r, which is just a mirror image of insfix_l:

```
fn{a:t@ype}
insfix_r // left rotation
  {cl,cr:clr}
  {bh:nat}
  {v:nat}
  {l_c,l_tl,l_x,l_tr:addr} (
  pf_c: int(BLK) @ l_c
, pf_tl: rbtree (a, cl, bh, 0) @ l_tl
, pf_x: a @ l_x
, pf_tr: rbtree (a, cr, bh, v) @ l_tr
| t: rbtree_cons_unfold (l_c, l_tl, l_x, l_tr)
, p_tr: ptr (l_tr)
) : [c:clr] rbtree0 (a, c, bh+1) = let
  #define B BLK
  #define R RED
  #define cons rbtree_cons
in
  case+ !p_tr of
  | cons (!p_cr as R, !p_trl, _, !p_trr as cons (!p_crr as R, _, _, _)) => let
//
      val () = !p_crr := B
      val () = fold@ (!p_trr)
//
      val tr = !p_tr
      val () = !p_tr := !p_trl
      val () = fold@ (t)
//
      val () = !p_trl := t
    in
      fold@ (tr); tr
    end // end of [cons (R, ..., cons (R, ...))]
  | cons (!p_cr as R, !p_trl as cons (!p_crr as R, !p_trll, _, !p_trlr), _, !p_trr) =>
//
      val tr = !p_tr
      val () = !p_tr := !p_trll
      val () = fold@ (t)
      val () = !p_trll := t
//
      val trl = !p_trl
      val () = !p_trl := !p_trlr
      val () = !p_cr := B
      val () = fold@ (tr)
      val () = !p_trlr := tr
//
    in
      fold@ (trl); trl
    end // end of [cons (R, cons (R, ...), ...)]
  | _ =>> (fold@ (t); t)
end // end of [insfix_r]
```

As can be expected, the following function template rbtree_insert is essentially a direct translation of the one of the same name for inserting an element into a functional red-black tree:

```
extern
fun{a:t@ype}
rbtree_insert
  {c:clr} {bh:nat} (
  t: rbtree0 (a, c, bh), x0: &a, cmp: cmp a
) : [bh1:nat] rbtree0 (a, BLK, bh1)

implement{a}
rbtree_insert
  (t, x0, cmp) = let
//
#define B BLK; #define R RED
#define nil rbtree_nil; #define cons rbtree_cons
//
fun ins
  {c:clr} {bh:nat} .<bh,c>. (
  t: rbtree0 (a, c, bh), x0: &a
) :<cloref1> [cl:clr; v:nat | v <= c] rbtree (a, cl, bh, v) =
  case+ t of
  | cons (
      !p_c, !p_tl, !p_x, !p_tr
    ) => let
      val sgn = compare (x0, !p_x, cmp)
    in
      if sgn < 0 then let
        val [cl,v:int] tl = ins (!p_tl, x0)
        val () = !p_tl := tl
      in
        if !p_c = B then
          insfix_l (view@(!p_c), view@(!p_tl), view@(!p_x), view@(!p_tr) | t, p_tl)
        else let
          val () = !p_c := R in fold@ {a}{..}{..}{cl} (t); t
        end // end of [if]
      end else if sgn > 0 then let
        val [cr,v:int] tr = ins (!p_tr, x0)
        val () = !p_tr := tr
      in
        if !p_c = B then
          insfix_r (view@(!p_c), view@(!p_tl), view@(!p_x), view@(!p_tr) | t, p_tr)
        else let
          val () = !p_c := R in fold@ {a}{..}{..}{cr} (t); t
        end // end of [if]
      end else (fold@ {a}{..}{..}{0} (t); t) // end of [if]
    end // end of [cons]
  | ~nil () => cons {..}{..}{..}{0} (R, nil, x0, nil)
// end of [ins]
val t = ins (t, x0)
//
in
//
case+ t of cons (!p_c as R, _, _, _) => (!p_c := B; fold@ (t); t) | _ =>> t
//
end // end of [rbtree_insert]
```

I literally implemented the above rbtree_insert by making a copy of the previous implementation of rbtree_insert for functional red-black trees and then properly modifying it to make it pass typechecking. Although this process of copying-and-modifying is difficult to be described formally, it is fairly straightforward to follow in practice as it is almost entirely guided by the error messages received during typechecking.

Please find the entire code in this section plus some additional code for testing online[3]. A challenging as well as rewarding exercise is for the reader to implement an operation to delete an element from a given linear red-black tree.

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVTYPES/merge
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVTYPES/bstree
3. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_DATAVTYPES/rbtree

# Chapter 21. Abstract Views and Viewtypes

I have so far given a presentation of views that solely focuses on at-views and the views built on top of at-views. This is largely due to at-views being the form of most widely used views in practice and also being the first form of views supported in ATS. However, other forms of views can be readily introduced into ATS abstractly. Even in a case where a view can be defined based on at-views (or other forms of views), one may still want to introduce it as an abstract view (accompanied with certain proof functions for performing view changes). Often what the programmer really needs is to figure out *conceptually* whether abstractly defined views and proof functions for manipulating them actually make sense. This is a bit like arguing whether a function is computable: There is rarely a need, if at all, to actually encode the function as a Turing-machine to prove its being computable. IMHO, learning proper use of abstract views and abstract viewtypes is a necessary step for one to take in order to employ linear types effectively in practice to deal with resource-related programming issues.

## Memory Allocation and Deallocation

The issue of memory allocation and deallocation is of paramount importance in systems programming, where garabage collection (GC) at run-time is most likely forbidden or only supported in a highly restricted manner. Handling memory management safely and efficiently is a long standing problem of great challenge in programming, and its novel solution in ATS is firmly rooted in the paradigm of programming with theorem-proving (PwTP).

The following function malloc_gc is available in ATS for memory allocation:

```
fun malloc_gc ()
  {n:nat} (n: size_t n)
  :<> [l:agz] (freebyte_gc_v (n, l), b0ytes n @ l | ptr l)
// end of [malloc_gc]
```

The sort agz is a subset sort defined for addresses that are not null:

```
sortdef agz = {a:addr | a > null} // [gz] for great-than-zero
```

The type b0ytes(n) is a shorthand for @[byte?][n], which is for an array of n uninitialized bytes. Therefore, the at-view b0ytes(n)@l is the same as the array-view array_v(byte?, n, l). The view freebyte_gc_v(n, l) stands for a form of capability allowing that the n bytes located at the address l be freed (or reclaimed) by the following function free_gc:

```
fun free_gc {n:nat} {l:addr}
  (pfgc: freebyte_gc_v (n, l), pfat: b0ytes n @ l | p: ptr l):<> void
// end of [free_gc]
```

Note that freebyte_gc_v is so far the first view we have encountered that is not built on top of any at-views.

In practice, it rather cumbersome to deal with bytes directly. Instead, the following two functions are more convenient for allocating and deallocating arrays:

```
fun{a:viewt@ype}
array_ptr_alloc {n:nat} (asz: size_t n)
  :<> [l:agz] (free_gc_v (a, n, l), array_v (a?, n, l) | ptr l)
// end of [array_ptr_alloc]

fun array_ptr_free
  {a:viewt@ype} {n:int} {l:addr} (
```

```
  pfgc: free_gc_v (a, n, l), pfarr: array_v (a?, n, l) | p: ptr l
) :<> void // end of [array_ptr_free]
```

Given a type T, an integer N and an address L, the view free_gc_v(T, N, L) means that the memory for the array located at L of N elements of the type T can be freed. In particular, the view freebyte_gc_v(N, L) is just free_gc_v(byte, N, L).

I now give a realistic and interesting example involving both array allocation and deallocation. The following two functions templates msort1 and msort2 perform mergesort on a given array:

```
typedef cmp (a:t@ype) = (&a, &a) -> int

extern
fun{a:t@ype}
msort1 {n:nat}
  (A: &(@[a][n]), n: size_t n, B: &(@[a?][n]), cmp: cmp(a)): void
// end of [msort1]

extern
fun{a:t@ype}
msort2 {n:nat}
  (A: &(@[a][n]), n: size_t n, B: &(@[a?][n]) >> @[a][n], cmp: cmp(a)): void
// end of [msort2]
```

It is well-known that merging two sorted segments of a given array requires additional space. When msort1 is called on arrays A and B, the array A is the one to be sorted and the array B is some kind of scratch area needed to perform merging (of sorted array segments). When a call to msort1 returns, the sorted version of A is still sotred in A. What msort2 does is similar but the sorted version of A is stored in B when a call to msort2 returns. As a good exercise, I suggest that the interested reader take the effort to give a mutually recursive implementation of msort1 and msort2. An implementation of mergesort based on msort1 can be readily given as follows:

```
extern
fun{a:t@ype}
mergesort {n:nat}
  (A: &(@[a][n]), n: size_t n, cmp: cmp(a)): void
// end of [mergesort]

implement{a}
mergesort (A, n, cmp) = let
  val (pfgc, pfat | p) = array_ptr_alloc<a> (n)
  val () = msort1 (A, n, !p, cmp)
  val () = array_ptr_free (pfgc, pfat | p)
in
  // nothing
end // end of [mergesort]
```

Clearly, an array is first allocated (to be used as a scratch area) and then deallocated after it is no longer needed.

The entire implementation of mergesort on arrays plus some testing code is available on-line[1].

## Simple Linear Objects

Objects in the physical world are conspicuously linear: They cannot be created from nothing or simply go vanished by turning into nothing. Thus, it is only natural to assign linear types to values that represent physical objects. I choose the name *simple linear object* here to refer to a linear value representing an object of some sort that does not contain built-in mechanism for supporting inheritance.

Let us first introduce a boxed abstract viewtype as follows for simple linear objects:

```
absviewtype sobjptr (a:viewt@ype+)
```

Given a viewtype VT, sobjptr(VT) is essentially meant for a pointer to some memory location L where a value of the viewtype VT is stored. The following function template sobjptr_new and function sobjptr_free are for creating and destroying (i.e., freeing) simple linear objects, respectively:

```
fun{a:viewt@ype} sobjptr_new (): sobjptr (a?)

fun sobjptr_free {a:viewt@ype} (x: sobjptr (a?)): void
```

The abstract viewtype sobjptr can be given the following definition:

```
assume
sobjptr (a:viewt@ype) = [l:addr] @{
  atview= a @ l, gcview= free_gc_v (a?, l), ptr= ptr l
} // end of [sobjptr]
```

Subsequently, sobjptr_new and sobjptr_free can be implemented as follows:

```
implement{a}
sobjptr_new () = let
  val (pfgc, pfat | p) = ptr_alloc<a> ()
in @{
  atview= pfat, gcview= pfgc, ptr= p
} end // end of [sobjptr_new]

implement
sobjptr_free {a} (pobj) =
  ptr_free {a} (pobj.gcview, pobj.atview | pobj.ptr)
// end of [sobjptr_free]
```

Clearly, a simple object needs to be initialized before it is of any use. This can be done by calling the following function sobjptr_init:

```
extern
fun sobjptr_init {a:viewt@ype}
  (pobj: !sobjptr (a?) >> sobjptr (a), f: (&a? >> a) -> void): void
// end of [sobjptr_init]

implement
sobjptr_init
  (pobj, f) = let
  prval pfat = pobj.atview
  val () = f !(pobj.ptr)
  prval () = pobj.atview := pfat
in
  // nothing
end // end of [sobjptr_init]
```

As a simple object may contain resources, it needs to be cleared out before it is allowed to be freed. This can be done by calling the following function sobjptr_clear:

```
extern
fun sobjptr_clear
  {a:viewt@ype} (
  x: !sobjptr (a) >> sobjptr (a?), f: (&a >> a?) -> void
) : void // end of [sobjptr_clear]

implement
```

```
sobjptr_clear
  (pobj, f) = let
  prval pfat = pobj.atview
  val () = f !(pobj.ptr)
  prval () = pobj.atview := pfat
in
  // nothing
end // end of [sobjptr_clear]
```

Note that each type T (of the sort t@ype) is a subtype of T?, implying that sobjptr(T) is a subtype of sobjptr(T?) (as sobjptr is co-variant). Therefore, sobjptr_free can be called directly on a value of the type sobjptr(T) without need to call sobjptr_clear on the value first.

Let us now see a concrete example of simple linear object. Suppose that a timer (that is, stopwatch) is wanted to measure time (of some sort). Following is a natural interface for functions creating, destroying and manipulating timer objects:

```
absviewtype timerObj

fun timerObj_new (): timerObj
fun timerObj_free (x: timerObj): void
fun timerObj_start (x: !timerObj): void
fun timerObj_finish (x: !timerObj): void
fun timerObj_pause (x: !timerObj): void
fun timerObj_resume (x: !timerObj): void
fun timerObj_get_ntick (x: !timerObj): uint
fun timerObj_reset (x: !timerObj): void
```

The (flat) record type timer_struct is defined as follows to represent the state of a timer object:

```
typedef
timer_struct = @{
  started= bool // the timer has started
, running= bool // the timer is running
  // the tick number recorded
, ntick_beg= uint // when the timer was turned on the last time
, ntick_acc= uint // the number of accumulated ticks
} // end of [timer_struct]
```

The abstract viewtype timerObj can then be mapped to sobjptr(timer_struct):

```
assume timerObj = sobjptr (timer_struct)
```

The functions timerObj_new and timerObj_free can now be given the following implementation:

```
implement
timerObj_new () = let
  typedef T = timer_struct
  fn f (
    x: &T? >> T
  ) : void = {
    val () = x.started := false
    val () = x.running := false
    val () = x.ntick_beg := 0u // unsigned
    val () = x.ntick_acc := 0u // unsigned
  } // end of [f]
  val pobj = sobjptr_new<T> ()
in
  sobjptr_init {T} (pobj, f); pobj
end // end of [timerObj_new]
```

```
implement
timerObj_free (pobj) = sobjptr_free {timer_struct} (pobj)
```

For brevity, I omit the code implementing the other functions on timer objects, which the interested reader can find on-line[2] together with some additional testing code.

## Example: Implementing an Array-Based Circular Buffer

Array-based circular buffers (of fixed sizes) are of common use in practice. For instance, in a typical client/server model, a circular buffer can be employed to hold requests issued by multiple clients that are then processed by the server according to the first-in-first-out (FIFO) policy. In a case where each request needs to be given a priority (chosen from a fixed set), a circular buffer can be created for each priority to hold requests of that priority.

I first declare a linear abstract type (that is, abstract viewtype) as follows for values representing circular buffers:

```
absviewtype cbufObj (a:viewt@ype+, m:int, n: int)
```

Such values are considered simple linear objects (as inheritance is not an issue to be dealt with in this setting). Given a viewtype VT and two integers M and N, the viewtype cbufObj(VT, M, N) is for a given buffer of maximal capacity M that currently contains N elements of the type VT.

Some properties on the parameters of cbufObj can be captured by introducing the following proof function:

```
prfun cbufObj_param_lemma
  {a:viewt@ype} {m,n:int} (buf: !cbufObj (a, m, n)): [m>=n; n>=0] void
// end of [cbufObj_param_lemma]
```

The interface for the following two function templates indicates that they can be called to compute the capacity and current size of a buffer:

```
fun{a:viewt@ype}
cbufObj_get_cap
  {m,n:int} (
  buf: !cbufObj (a, m, n)
) : size_t (m) // end of [cbufObj_get_cap]

fun{a:viewt@ype}
cbufObj_get_size
  {m,n:int} (
  buf: !cbufObj (a, m, n)
) : size_t (n) // end of [cbufObj_get_size]
```

While it is straightforward to use cbufObj_get_cap and cbufObj_get_size to tell whether a buffer is currently empty or full, a direct approach is likely to be more efficient. The following two function templates check for the emptiness and fullness of a given circular buffer:

```
fun{a:viewt@ype}
cbufObj_is_empty
  {m,n:int} (buf: !cbufObj (a, m, n)): bool (n==0)

fun{a:viewt@ype}
cbufObj_is_full
  {m,n:int} (buf: !cbufObj (a, m, n)): bool (m==n)
```

The functions for creating and destroying circular buffers are named cbufObj_new and cbufObj_free, respectively:

```
fun{a:viewt@ype}
cbufObj_new {m:pos} (m: size_t m): cbufObj (a, m, 0)

fun cbufObj_free
  {a:viewt@ype} {m:int} (buf: cbufObj (a, m, 0)): void
// end of [cbufObj_free]
```

Note that a buffer can be freed only if it contains no elements as an element (of some viewtype) may contain resources. If elements in a buffer are of some (non-linear) type, then the following function can be called to clear out all the elements stored in the buffer:

```
fun cbufObj_clear_type
  {a:t@ype} {m:int} (
  buf: !cbufObj (a, m, n) >> cbufObj (a, m, 0)
) : void // end of [cbufObj_clear_type]
```

The next two functions are for inserting/removing an element into/from a given buffer, which are probably the most frequently used operations on buffers:

```
fun{a:viewt@ype}
cbufObj_insert
  {m,n:int | n < m} (
  buf: !cbufObj (a, m, n) >> cbufObj (a, m, n+1), x: a
) : void // end of [cbufObj_insert]

fun{a:viewt@ype}
cbufObj_remove
  {m,n:int | n > 0} (
  buf: !cbufObj (a, m, n) >> cbufObj (a, m, n-1)
) : a // end of [cbufObj_remove]
```

Please find on-line the file circbuf.sats[3] containing the entirety of the interface for functions creating, destroying and manipulating circular buffers.

There are many ways to implement the abstract type cbufObj and the functions declared in circbuf.sats[4]. In the on-line file circbuf.dats[5], I give an implementation that employs four pointers p_beg, p_end, p_frst and p_last to represent a circular buffer: p_beg and p_end are the starting and finishing addresses of the underline array, respectively, and p_frst and p_last are the starting addresses of the occupied and unoccupied segments (in the array), respectively. What is special about this implementation is its employing a style of programming that deliberately eschew the need for proof construction. While code written in this style is not guaranteed to be type-safe, the style can nonetheless be of great practical value in a setting where constructing formal proofs is deemed too great a requirement to be fully fulfilled. Anyone who tries to give a type-safe implementation for the functions declared in circbuf.sats[6] should likely find some genuine appreciation for this point.

In the on-line file circbuf2.dats[7], I give another implementation in which a circular buffer is represented as a pointer p_beg plus three integers m, n and f: p_beg points to the starting location of the underline array, m is the size of the array (that is, the capacity of the buffer), n is the number of elements currently stored in the buffer and f is the total number of elements that have so far been removed from the buffer. Again, proof construction is delibrately eschewed in this implementation.

## From linearity to non-linearity

## Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_ABSVTYPES/mergesc
2. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_ABSVTYPES/sobjptr.c
3. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_ABSVTYPES/circbuf.s
4. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_ABSVTYPES/circbuf.s
5. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_ABSVTYPES/circbuf.c
6. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_ABSVTYPES/circbuf.s
7. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/CODE/CHAPTER_ABSVTYPES/circbuf2

# Chapter 22. Summary