A Tutorial on Programming Features in ATS

Hongwei Xi

hwxi AT cs DOT bu DOT edu

A Tutorial on Programming Features in ATS:
by Hongwei Xi

Copyright © 2010-201? Hongwei Xi

All rights are reserved. Permission is granted to print this document for personal use.

Table of Contents

Preface

This tutorial covers a variety of issues that a programmer typically encounters when programming in ATS. It is written mostly in the style of learn-by-examples. Although it is possible to learn programming in ATS by studying the tutorial (if the reader is familiar with ML and C), I consider the book Introduction to Programming in ATS[1] a far more approriate resource for someone to acquire a view of ATS in a coherent and systematic manner. Of course, there will also be considerable amount of overlapping between these two books. The primary purpose of the tutorial is to bring more insights into a rich set of programming features in ATS and also demonstrate through concrete examples that these features can be made of effective use in the construction of high-quality programs.

Notes

1. http://www.ats-lang.org/DOCUMENT/INTPROGINATS/HTML/book1.html

Chapter 1. Syntax-Coloring for ATS code

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

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

Please find an example of ATS code on-line[1] that involves all of the syntax-coloring mentioned above.

Notes

1. http://www.ats-lang.org/DOCUMENT/TUTORIALATS/CODE/fact_dats.html

Chapter 2. Filename Extensions

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

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

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

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

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

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

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

Chapter 3. File Inclusion inside ATS Code

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

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

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

A common use of file inclusion is to keep some constants, flags or parameters being defined consistently across a set of files. For instance, the file prelude/params.hats[1] serves such a purpose. File inclusion can also be used to emulate (in a limited but rather useful manner) functors supported in languages such as SML and OCaml.

Notes

1. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/params.hats

Chapter 4. Fixity Declarations

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

Each identifier in ATS can be assigned one of the following fixities: *prefix*, *infix* and *postfix*. The fixity declarations for many commonly used identifiers can be found in prelude/fixity.ats[1]. Often, the name *operator* is used to refer to an identifier that is assigned a fixity. For instance, the following syntax declares that + and - are infix operators of a precedence value equal to 50:

```
infixl 50 + -
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
nonfix iadd fadd padd uadd
```

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

Notes

1. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/fixity.ats

Chapter 5. The Program Entry Point: mainats

There are two special functions of the name main_void and main_argc_argv that are given the following interfaces:

```
fun main_void (): void = "mainats"
overload main with main_void

fun main_argc_argv {n:int | n >= 1}
  (argc: int n, argv: &(@[string][n])): void = "mainats"
overload main with main_argc_argv
```

The symbol main is overloaded with both of these functions. In addition, the global name mainats is assigned to both of them. When a function in ATS is translated into one in C, the global name of the function, if ever assgined, is used to refer to its translation in C.

The interface for main_argc_argv indicates that the function takes as its arguments an integer *argc* greater than 0 and an array *argv* of size *argc* in which each element is a string, and returns no value. The syntax argv: &(@[string][n] means that argv is a call-by-reference argument. If we followed the like of syntax in C++, then this would be written as something like &argv: @[string][n].

To turn ATS source code into an executable, the function is required to be present in the C code translated from the ATS code (as it is called within the main function in C). Normally, this means that either main_void or main_argc_argv needs to be implemented in the ATS code (that is to be turned into an executable). However, in certain situations, it may make sense to implement mainats in C directly. Note that the interface for mainats in C is:

```
extern ats_void_type mainats (ats_int_type, ats_ptr_type) ;
```

where ats_void_type, ats_int_type and ats_ptr_type are just aliases for void, int and void*, respectively.

As an example, the following ATS program echos onto the standard output the given command line:

```
implement
main (argc, argv) = let
  fun loop {n,i:nat | i <= n} // [loop] is tail-recursive
    (i: int i, argc: int n, argv: &(@[string][n])): void =
    if i < argc then begin
      if i > 0 then print (' '); print argv.[i]; loop (i+1, argc, argv)
    end // end of [if]
  // end of [loop]
in
  loop (0, argc, argv); print_newline ()
end // end of [main]
```

If mainats needs to be implemented in C, the proof function main_dummy should be implemented as follows:

```
implement main_dummy () = ()
```

This implementation is solely for telling the ATS compiler that mainats is expected to be implemented in C directly so that the compiler can generate proper code to handle the situation. As an example, I present as follows a typical scenario in GTK+ programming, where the function gtk_init needs to be called to modify the arguments passed from the given command line:

```
//
// some function implemented in ATS
//
extern fun main_work // implemented elsewhere
  {n:pos} (argc: int n, argv: &(@[string][n])): void = "main_work"
// end of [main_work]

%{^
// %{^ : the embedded C code is placed at the top
extern ats_void_type mainats (ats_int_type, ats_ptr_type) ;
%} // end of [%{^]

implement main_dummy () = () // indicating [mainats] being implemented in C

%{$
// %{$ : the embedded C code is placed at the bottom
ats_void_type
mainats (
  ats_int_type argc, ats_ptr_type argv
) {
  gtk_init ((int*)&argc, (char ***)&argv) ; main_work (argc, argv) ; return ;
} /* end of [mainats] */

%} // end of [%{$]
```

Chapter 6. Tail-Recursive Call Optimization

Tail-recursion is of crucial importance in functional programming as loops in imperative programming are often implemented as tail-recursive functions.

Suppose that a function *foo* calls another function *bar*, that is, there is a call to *bar* appearing in the body of *foo*, where *foo* and *bar* may actually be the same function. If the return value of the call to *bar* also happens to be the return value of *foo*, then this call is often referred to as a *tail-call*. If *foo* and *bar* are the same, then this call is a (recursive) self tail-call. For instanace, there are two recursive calls in the body of the following defined function f91:

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

where the outer call to f91 is a tail-call while the inner one is not. If each self call in the body of a function is a tail-call, then this function is tail-recursive.

It is arguably that the single most important optimization performed by the ATS compiler is the translation of every self tail-call into a direct (local) jump. This optimization effectively turns every tail-recursive function into the equivalent of a loop, guaranteeing that a fixed amount of stack space is adequate for executing each call to the function.

Let us now see some examples that can help further illustrate the notion of tail-recursive call optimization. The following recursive function sum1 sums up all the natural numbers less than or equal to a given integer parameter n:

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

Clearly, sum1 is not tail-recursive as the only self call in its body is not a tail-call. The counterpart of sum1 in C can essentially be given as follows:

```
int sum1 (int n) {
  return (n > 0) ? n + sum1 (n-1) : 1 ;
} // end of [sum1]
```

When applied to an integer n, the following defined function sum2 also sums up all the natural numbers less than or equal to n:

```
fn sum2 (n: int): int = let
  fun loop (n: int, res: int): int =
    if n > 0 then loop (n-1, res + n) else res
  // end of [loop]
in
  loop (n, 0)
end // end of [sum2]
```

The inner function loop in the definition of sum2 is tail-recursive. The stack space consumed by loop is constant, that is, it is independent of the values of the arguments of loop. Essentially, the ATS compiler translates the definition of sum2 into the following C code:

```
int sum2_loop (int n, int res) {
  loop:
  if (n > 0) {
    res = res + n ; n = n - 1; goto loop;
  } else {
    // do nothing
  }
  return res ;
```

```
}

int sum2 (int n) { return sum2_loop (n, 0) ; }
```

Translating sum1 into sum2 is a fairly straightforward process. However, it can be highly involved, sometimes, to turn a non-tail-recursive implementation into an equivalent one that is tail-recursive.

Chapter 7. Mutual Tail-Recursion

Mutually tail-recursive functions are commonly encountered in practice. Assume that *foo* and *bar* are two mutually defined functions. In the body of either *foo* or *bar*, a tail-call to *foo* or *bar* is referred to as a mutually tail-recursive call. If every call to *foo* or *bar* in the bodies of *foo* and *bar* are tail-call, then *foo* and *bar* are mutually tail-recursive. Mutual recursion involving more functions can be defined similarly. As an example, the following two functions isEvn and isOdd are defined mutually recursively:

```
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 call to isOdd in the body of isEvn is a mutually tail-recursive call, and the call to isEvn in the body of isOdd is also a a mutually tail-recursive call. Therefore, isEvn and isOdd are mutually tail-recursive.

In order to turn mutually recursive tail-calls into jumps, the bodies of the involved mutually recursive functions need to be combined. The keyword fn* in ATS is introduced precisely for this purpose. For instance, let us replace fun with fn* in the code above:

```
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
```

Then the following C code is essentially what is generated from compiling these two mutually defined functions:

```
bool isEvnisOdd (int tag, int n) {

bool res ;

switch (tag) {
  0: goto isEvn ;
  1: goto isOdd ;
  default : exit (1) ;
}

isEvn: if (n > 0) { n = n - 1; goto isodd; } else { res = true; goto done; }
isOdd: if (n > 0) { n = n - 1; goto isEvn; } else { res = false; goto done; }

done: return res ;

} /* end of [isEvnisOdd] */

bool isEvn (int n) { return isEvnisOdd (0, n) ; }
bool isOdd (int n) { return isEvnisOdd (1, n) ; }
```

Note that mutually recursive functions can be combined in such a manner only if they all have the same return type. In the above case, both isEvn and isOdd have the same return type bool.

When translating C code involving embedded loops, we often encounter the need for mutual tail-recursion. For example, the following C code prints out some ordered pairs of digits:

```
int main (
  int argc, char *argv[]
) {
  int i, j ;
  for (i = 0; i <= 9; i += 1) {
    for (j = i; j <= 9; j += 1) {
```

```
      if (i < j) printf (", ") ; printf ("(%i, %i)", i, j) ;
    } /* for */
    printf ("
") ;
  } /* for */
  return 0 ;
}
```

A straightforward translation of the C code into ATS (in the style of functional programming) can be done as follows:

```
implement
main (argc, argv) = let
  fn* loop1
    {i:nat | i <= 10} (i: int i): void =
    if i <= 9 then loop2 (i, i) else ()
  // end of [loop1]

  and loop2
    {i,j:nat | i <= 9; j <= 10} (i: int i, j: int j): void =
    if j <= 9 then (
      if i < j then (
        print ", "; printf ("(%i, %i)", @(i, j)); loop2 (i, j+1)
      ) // end of [if]
    ) else (
      print_newline (); loop1 (i+1)
    ) // end of [if]
  // end of [loop2]
in
  loop1 (0)
end // end of [main]
```

Evidently, loop1 and loop2 are defined mutually tail-recursively. The use of the keyword fn* ensures that these two functions are translated into the equivalent of two loops in C, which require only a fixed amount of stack space to run.

Chapter 8. Metrics for Termination Verification

ATS provides a simple means for the programmer to verify the termination of recursively defined functions by supplying proper termination metrics. This is an indispensable feature for supporting the paradigm of programming with theorem-proving as proof functions, namely, functions representing proofs, must be proven to be terminating.

A termination metric is a tuple (M_1, ..., M_n) of natural numbers, where n >= 0. We use the standard well-founded lexicographical ordering on natural numbers to order such tuples.

Primitive Recursion

The kind of recursion in the following implementation of the factorial function is primitive recursion:

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

The special syntax .<n>. indicates that the metric supplied for verifying the termination of the defined function is a singleton tuple (n). In the definition of fact, the metric for the recursive call to fact is (n-1), which is strictly less than the original metric (n). Therefore, the defined function fact is terminating.

General Recursion

We implement as follows a function gcd that computes the greatest common divisor of two given positive integers:

```
//
// computing the greatest common divisors of two positive ints
//
fun gcd
  {m,n:int | m > 0; n > 0} .<m+n>.
  (m: int m, n: int n): [r:nat | 1 <= r; r <= min(m, n)] int r =
  if m > n then gcd (m - n, n)
  else if m < n then gcd (m, n - m)
  else m
// end of [gcd]
```

The syntax .<m+n>. indicates that the termination metric (m+n) is supplied to verify that the defined function gcd is terminating. In the definition of gcd, the termination metric for the first recursive call to gcd is (m-n)+n=m, which is strictly less than the original termination metri m+n (as n is positive); the termination metric for the second recursive call to gcd is m+(n-m)=n, which is also strictly less than the original termination metric m+n (as m is positive). Thus, gcd is a terminating function.

As another example, we implement as follows the Ackermann's function, which is famous for being recursive but not primitive recursive:

```
//
// [ack] implements the Ackermann's function
//
fun ack {m,n:nat} .<m, n>.
  (m: int m, n: int n): Nat =
  if m > 0 then
    if n > 0 then ack (m-1, ack (m, n-1)) else ack (m-1, 1)
  else n+1 // end of [if]
// end of [ack]
```

The syntax .<m, n>. indicates that the termination metric is a pair of natural numbers: (m, n). Note that the standard lexicographical ordering on natural numbers is employed to compare such metrics. To verify that ack is terminating, we need to solve the following constraints:

- (m-1, k) is less than (m, n) under the assumption m > 0, where k can be any natural number.
- (m, n-1) is less than (m, n) under the assumption m > 0 and n > 0.
- (m-1, 1) is less than (m, n) under the assumption m > 0.

As all of these constraints can be readily solved, we conclude that ack is a terminating funciton.

Mutual Recursion

When mutually recursive functions are to be verified, the termination metrics for these functions, which are tuples of natural numbers, must be of the same tuple length. We given a simple example as follows:

```
fun isEvn
  {n:nat} .<2*n+2>. (n: int n): bool =
  if n > 0 then ~(isOdd n) else true // end of [if]
// end of [isEvn]

and isOdd
  {n:nat} .<2*n+1>. (n: int n): bool =
  if n > 0 then isEvn (n-1) else false // end of [if]
// end of [isOdd]
```

Clearly, we can also verify the termination of these two functions by using the metrics .<n, 1>. and .<n, 0>. for isEvn and isOdd, respectively.

Termination Checking at Run-time

Suppose that foo and bar are declared as follows:

```
fun foo ():<> void and bar ():<> void
```

Moreover, suppose that the following implementation of foo is given in a file named `foo.dats`:

```
implement foo () = $Bar.bar ()
```

while the following implementation of bar is given in another file named `bar.dats` that is different from `foo.dats`:

```
implement bar () = $Foo.foo ()
```

Clearly, neither foo nor bar is terminating. In practice, it is difficult to resolve this issue of calling cycles among functions by solely relying on termination metrics. Instead, atscc can generate run-time code for detecting calling cycles among functions if the flag *-D_ATS_PROOFCHECK* is present. For instance, if foo.dats and bar.dats are compiled as follows:

```
atscc -D_ATS_PROOFCHECK foo.dats and bar.dats
```

then a run-time error is to be reported to indicate a calling cycle when either foo.dats or bar.dats is loaded dynamically.

Chapter 9. Higher-Order Functions

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

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

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

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

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

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

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

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

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

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

```
in
  loop (f, f1, x0)
end // end of [newton_raphson]
```

We can now readily implement square root function and the cubic root function based on newton_raphson:

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

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

Chapter 10. Parametric Polymorphism

Parametric polymorphism (or polymorphism for short) offers a flexible and effective approach to supporting code reuse. For instance, given a pair (v1, v2) where v1 and v2 are a boolean a character, respectively, the function swap_bool_char defined below returns a pair (v2, v1):

```
fun swap_bool_char (xy: (bool, char)): (char, bool) = (xy.1, xy.0)
```

Suppose that integer values need to be swapped as well. This leads to the implementation of the following function swap_int_int:

```
fun swap_int_int (xy: (int, int)): (int, int) = (xy.1, xy.0)
```

The code duplication between swap_bool_char and swap_int_int is obvious, and it can be easily avoided by implementing a function template as follows:

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

Now the functions swap_bool_char and swap_int_int can simply be replaced with swap<bool,char> and swap<int,int>, respectively. The function template swap cannot be compiled into binary object code directly as the sizes of type variables a and b are unknown: The special sort t@ype is for classifying types whose sizes are unspecified. If swap<T1,T2> is used for some types T1 and T2 of known sizes, then an instantiation of swap is created where type variables a and b are replaced with T1 and T2, respectively, and the instantiation is compiled into binary object code. For those know the feature of templates in C++, this should sound rather familiar.

In contrast to swap, swap_type_type is defined below as a polymorphic function (rather than a function template):

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

This function can be compiled into binary object code as the sizes of type variables a and b are known: The special sort type is for classifying types of size equal to exactly one word, that is, the size of a pointer. For example, the size of a string is one word, and the size of any declared datatype is also one word. Given strings str1 and str2, an application of swap_type_type to str1 and str2 can be written as follows:

```
swap_type_type {string,string} (str1, str2)
```

where the expression {string,string} is often referred to as a static argument. As in this case, most static arguments do not have to be supplied explicitly since they can be automatically inferred. However, such static arguments, if given, can often greatly enhance the quality and precision of the error messages reported in case of type-checking failure.

Chapter 11. Printf-like Functions

The printf function in C is variadic, that is, the arity of the function is indefinite. In ATS, there is a function of the same name that is essentially the counterpart of the printf function in C.

The following interface is assigned to printf in ATS:

```
fun printf {ts:types} (fmt: printf_c ts, arg: ts): void
```

We use printf_c for a type constructor that takes a list of types to form a type for format strings (in C). For instance, printf_c(char, double, int) is a type for format strings that require a character, a double, and an integer to be supplied. Given a character c, a double d and an integer i, @(c, d, i) is an argument of types (char, double, int), and the following expression is well-typed in ATS:

```
printf ("c = %c and d = %f and i = %i", @(c, d, i))
```

The type of the format string "c = %c and d = %f and i = %i" is computed to be printf_c(char, double, int) and then @(c, d, i) is checked to be of the type (char, double, int). Note that a format string must be a constant in order for its type to be computed during typechecking.

As an example, we present as follows a program that prints out a multiplication table for single digits:

```
#define N 9

implement
main () = let
  fun loop1
    {i:nat | i <= N}
    (i: int i): void =
    if i < N then loop2 (i+1, 0) else ()
  // end of [loop1]
  and loop2
    {i,j:nat | i <= N; j <= i}
    (i: int i, j: int j): void =
    if j < i then let
      val () = if (j > 0) then print ' '
      val () = printf ("%1d*%1d=%2.2d", @(j+1, i, (j+1) * i))
    in
      loop2 (i, j+1)
    end else let
      val () = print_newline () in loop1 (i)
    end // end of [if]
  // end of [loop2]
in
  loop1 (0)
end // end of [main]
```

This programs generates the following expected output:

```
1*1=01
1*2=02 2*2=04
1*3=03 2*3=06 3*3=09
1*4=04 2*4=08 3*4=12 4*4=16
1*5=05 2*5=10 3*5=15 4*5=20 5*5=25
1*6=06 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=07 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=08 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=09 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
```

Please find a few other functions declared in prelude/SATS/printf.sats[1] that are similar to printf.

Notes

1. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/printf.sats

Chapter 12. Functional Lists

Lists are by far the most commonly used data structure in functional programming. We say that a data structure is functional if it is heap-allocated and immutable and can only be freed through garbage collection (GC). In contrast, a data structure is said to be linear if it is either stack-allocated or heap-allocated and can be freed by the user as well as by the GC.

The datatype for functional lists in ATS is (essentially) declared as follows:

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

There are two data constructors associated with list: list_nil forms an empty list and list_cons for a list of a given head and tail. Given a type T and an integer I, the type list(T, I) is for lists of length I in which each element is of type T. Note that the sort t@ype indicates that the element type of a list can be unboxed (i.e., flat).

Often the following abbreviations are introduced for the list constructors so as to make the code involving list-processing less verbose:

```
#define nil list_nil
#define cons list_cons
#define :: list_cons // [::] is an infix operator
```

As an example of list creation, the following expression evaluates to a list consisting of integers 1, 2 and 3:

```
cons (1, cons (2, cons (3, nil ()))) // [nil ()] can be replaced with [nil]
```

Clearly, this kind of syntax is a bit unwieldy if longer lists need to be constructed. The following alternatives can also be used to create lists:

```
val xs = '[1, 2, 3] // the first character is quote (')
val xs = $lst (1, 2, 3) // this is equivalent to '[1, 2, 3]
val xs = $lst {Nat} (1, 2, 3) // [Nat] is given as the type for the list elements
```

The interfaces for various functions on lists can be found in prelude/SATS/list.sats[1].

Let us now see some list-processing code in ATS. The following program implements a function template that computes the length of a given list:

```
fun{a:t@ype}
length {n:nat} .<n>.
  (xs: list (a, n)): int n =
  case+ xs of _ :: xs => 1 + length xs | nil () => 0
// end of [length]
```

As this is not a tail-recursive implementation, the function length may have difficulty handling long lists (e.g., (e.g., those containing more than 1 million elements). A tail-recursive implementation of length that can handle lists of any length is given as follows:

```
fun{a:t@ype}
length {n:nat} .<>.
  (xs: list (a, n)): int n = let
  fun loop {i,j:nat} .<i>.
    (xs: list (a, i), j: int j): int (i+j) =
```

```
    case+ xs of _ :: xs => loop (xs, j+1) | nil () => j
  // end of [loop]
in
  loop (xs, 0)
end // end of [length]
```

Let us see another example. The following function append returns the concatenation of two given lists:

```
fun{a:t@ype}
append {m,n:nat} .<m>. (
  xs: list (a, m), ys: list (a, n)
) : list (a, m+n) =
  case+ xs of
  | cons (x, xs) => cons (x, append (xs, ys)) | nil () => ys
// end of [append]
```

This is not a tail-recursive implementation. As a consequence, append may have difficulty handling a case where its first argument is of a large length (e.g., 1 million). Can append be given a tail-recursive implementation in ATS? The answer is affirmative. For instance, a tail-recursive implementation of append is available in prelude/DATS/list.dats[2]. As the implementation makes use of linear types, it is to be explained elsewhere.

Notes

1. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/list.sats
2. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/DATS/list.dats

Chapter 13. Persistent Arrays

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

Given a viewtype VT, the type for arrays containing n values of viewtype VT is array(VT, n). Note that arrays in ATS are the same as those in C: There is no size information attached them. The interfaces for various functions on arrays can be found in prelude/SATS/array.sats[1].

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

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

fun{a:t@ype}
array_make_lst {n:nat}
  (asz: size_t n, xs: list (a, n)):<> array (a, n)
// end of [array_make_lst]
```

Applied to a size and an element, array_make_elt returns an array of the given size in which each cell is initialized with the given element. Applied to a size and a list of elements, array_make_lst returns an array of the given size in which each cell is initialized with the corresponding element in the given list.

For reading from and writing to an array, the function templates array_get_elt and array_set_elt can be used, respectively, which are assigned 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):<!ref> a

fun{a:t@ype}
array_set_elt_at {n:int}
  {i:nat | i < n} (A: array (a, n), i: size_t i, x: a):<!ref> void
```

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

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

```
fun{a:t@ype}
array_reverse {n:nat} (
  A: array (a, n), n: size_t (n)
) : void = let
  fun loop {i: nat | i <= n} .<n-i>.
    (A: array (a, n), n: size_t n, i: size_t i): void =
    if i < n/2 then let
      val tmp = A[i]
    in
      A[i] := A[n-1-i]; A[n-1-i] := tmp; loop (A, n, i+1)
    end else () // end of [if]
  // end of [loop]
in
  loop (A, n, 0)
end // end of [array_reverse]
```

If the test i < n/2 is changed to i <= n/2, a type-error is to be reported. Why? The reason is that A[n-1-i] becomes out-of-bounds array subscripting in the case where n and i both equal zero. Given that it is very unlikely to encounter a case where an array of size 0 is involved, a bug like this, if not detected early, can be buried so scarily deep!

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

```
fun{a:viewt@ype}
array_reverse {n:nat} (
  A: array (a, n), n: size_t (n)
) : void = let
  fun loop {i: nat | i <= n} .<n-i>.
    (A: array (a, n), n: size_t n, i: size_t i): void =
    if i < n/2 then let
      val () = array_exch (A, i, n-1-i) in loop (A, n, i+1)
    end else () // end of [if]
  // end of [loop]
in
  loop (A, n, 0)
end // end of [array_reverse]
```

The interface for the function template array_exch is given below:

```
fun{a:viewt@ype}
array_exch {n:nat}
  (A: array (a, n), i: sizeLt n, j: sizeLt n):<!ref> void
// end of [array_exch]
```

Note that array_exch can not be implemented in terms of array_get_elt_at and array_set_elt_at (unless some form of type-unsafe coding is empolyed). The curious reader can find its type-safe implementation in prelude/DATS/array.dats[2], which is based on a corresponding operation for linear arrays.

Notes

1. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/array.sats
2. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/DATS/array.dats

Chapter 14. Persistent References

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

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

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

The interfaces for various functions on references can be found in prelude/SATS/reference.sats[1].

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

```
fun{a:viewt@ype} ref_make_elt (x: a):<> ref a
```

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

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

Note that the symbol !ref indicates that these functions incur the so-called ref-effect when evaluated. Given a reference r and a value v, ref_get_elt(r) and ref_set_elt(r, v) can be written as !r and !r := v, respectively.

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

```
local
//
// [ref] is a shorthand for [ref_make_elt]
//
val count = ref<int> (0)

in // in of [local]

fun getNewName
  (prfx: string): string = let
  val n = !count
  val () = !count := n + 1
  val name = sprintf ("%s%i", @(prfx, n))
in
  string_of_strptr (name)
end // end of [getNewName]

end // end of [local]
```

The function getNewName is called to generate fresh names. As the integer content of the reference count is updated whenever a call to getNewName is made, each name returned by getNewName is guaranteed to have not generated before. Note that each string returned by sprinf is a linear one (of the type strptr) and the cast funtion string_of_strptr is called to turn it into a nonlinear one. There is no run-time cost associated with such a call as every call to a cast function is always a no-op at run-time.

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

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

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

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

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

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

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

Notes

1. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/reference.sats

Chapter 15. Call-by-Reference

The feature of call-by-reference in ATS is similar to the corresponding one in C++. What is special in ATS is the way in which this feature is handled by the type system. In general, if f is given a type of the following form for some viewtypes VT1 and VT2:

```
(..., &VT1 >> VT2, ...) -> ...
```

then a function call f(..., lval, ...) on some left-value lval of the viewtype VT1 is to change the viewtype of lval into VT2 upon its return. In the case where VT1 and VT2 are the same, &VT1 >> VT2 may simply be written as &VT1.

As an example, an implementation of the factorial function is given as follows that makes use of call-by-reference:

```
fun fact (x: int): int = let
  fun loop {l:addr} (x: int, res: &int): void =
    if x > 0 then (res := res * x; loop (x-1, res)) else ()
  var res: int = 1 // [res] is a variable!
in
  loop (x, res); res
end // end of [fact]
```

Note that if the line for introducing the variable res in the above implementation is replaced with the following one:

```
  val res: int = 1 // [res] is no longer a variable but a value!
```

then a type error is to be reported as res is no longer a left-value when it is passed as an argument to loop.

In functional programming, optional types are often used for error-handling. For instance, the following function divopt returns a value of the type Option(int) that either contains the result of the division or indicates a case of division-by-zero:

```
fun divopt
  (x: int, y: int): Option (int) =
  if y != 0 then Some (x/y) else None ()
// end of [divopt]
```

Given that a value of the form Some(v) is heap-allocated, the memory for storing it can only be reclaimed by garbage collection (GC). In other words, the memory is leaked if GC is not available. To address the issue of error-handing in the absence of GC, we can employ call-by-reference as is shown below:

```
fun diverr (
  x: int, y: int, err: &int
) : int =
  if y != 0 then x/y else (err := err+1; 0(*meaningless*))
// end of [diverr]
```

We can tell whether division-by-zero has occurred by comparing the values of err before and after a call to diverr. This style of error-handling is commonly seen in code written in languages like C.

Chapter 16. Lazy Evaluation

Though ATS is a language based on eager call-by-value evaluation, it also allows the programmer to perform lazy call-by-need evaluation. In ATS, there is a special language construct $delay that can be used to delay or suspend the evaluation of an expression (by forming a thunk) and a special function lazy_force that can be called to resume a suspended evaluation (represented by a thunk).

There is a special type constructor lazy of the sort (t@ype) => type in ATS, which forms a (boxed) type when applied to a type. On one hand, given an expression exp of type T, $delay(exp) forms a value of the type lazy(T) that represents the suspended evaluation of exp. On the other hand, given a value v of the type lazy(T) for some type T, lazy_force(v) resumes the suspended evaluation represented by v and returns a result of type T. The interface for the function template lazy_force is given as follows:

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

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

In prelude/SATS/lazy.sats[1], the following datatype stream_con and stream are declared mutually recursively for representing (lazy) streams:

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

Also, a number of common functions on streams are declared in prelude/SATS/lazy.sats[2] and implemented in prelude/DATS/lazy.dats[3].

The following code gives a standard implementation of the sieve of Eratosthenes. We first construct a stream of all the integers starting from 2 that are ordered ascendingly; we keep the first element of the stream and remove all of its multiples; we repeat this process on the rest of the stream recursively. The final stream then consists of all the prime numbers ordered ascendingly.

```
#define nil stream_nil
#define cons stream_cons
#define :: stream_cons

typedef N2 = [n:int | n >= 2] int n
val N2s: stream N2 = from 2 where {
  fun from (n: N2):<!laz> stream N2 = $delay (n :: from (n+1))
}

fun sieve
  (ns: stream N2):<!laz> stream N2 = let
  // [val-] means no warning message from the compiler
  val- n :: ns = !ns
in
  $delay (n :: sieve (stream_filter_cloref<N2> (ns, lam x => x nmod n > 0)))
end // end of [sieve]

val primes: stream N2 = sieve N2s

//
// Finding the nth prime where counting starts from 0
//
fn nprime {n: nat} (n: int n): N2 = stream_nth (primes, n)
```

The function template stream_filter_cloref is of the following interface:

```
fun{a:t@ype}
stream_filter_cloref
  (xs: stream a, p: a -<cloref,!laz> bool):<!laz> stream a
```

It is called to construct a stream consisting of all the elements in a given stream that satisfy a given predicate.

We give another example of lazy-evaluation as follows, which demonstrates an interesting approach to computing Fibonacci numbers:

```
val one = int64_of_int 1

val // the following values are defined mutually recursively
rec fibs_1: stream int64 = $delay (one :: fibs_2) // fib1, fib2, ...
and fibs_2: stream int64 = $delay (one :: fibs_3) // fib2, fib3, ...
and fibs_3: stream int64 = ( // fib3, fib4, ...
  stream_map2_fun<int64,int64><int64> (fibs_1, fibs_2, lam (x, y) => x + y)
)
// find the nth Fibonacci number
fn nfib {n:pos} (n: int n): int64 = stream_nth (fibs_1, n-1)
```

The function template stream_map2_fun is assigned the following interface:

```
fun{a1,a2:t@ype}{b:t@ype}
stream_map2_fun
  (xs1: stream a1, xs2: stream a2, f: (a1, a2) -<!laz> b):<!laz> stream b
```

Given two streams xs1 and xs2 and a binary function f, stream_map2_fun forms a stream such that the *n*th element in it, if it exists, equals f(x1, x2), where x1 and x2 are the *n*th elements in xs1 and xs2, respectively.

Notes

1. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/lazy.sats
2. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/SATS/lazy.sats
3. http://www.ats-lang.org/DOCUMENT/ANAIRIATS/prelude/DATS/lazy.dats

Chapter 17. Cast Functions

A cast function in ATS is equivalent to the identify function in terms of dynamic semantics. A call to such a function is evaluated at compile-time, and its argument is returned. For instance, we have the following commonly used cast functions:

```
castfn int1_of_int (x: int):<> [n:nat] int n
castfn string1_of_string (x: string):<> [n:nat] string n
```

Note that the keyword castfn is for introducing cast functions.

Let us now see a more interesting use of casting functions. The following declared function interface is intended for concatenating a list of lists:

```
extern fun{a:t@ype} list_concat (xss: List (List a)): List a
```

Assume that we would like to verify that the concatenation of a list of lists yields a list whose length equals the sum of the lengths of the lists in the given list of lists. This, for instance, can be done as follows by introducting a datatype constructor lstlst.

```
datatype lstlst (a:t@ype+, int, int) =
  | {m,t:nat} {n:nat}
    lstlst_cons (a, m+1, t+n) of (list (a, n), lstlst (a, m, t))
  | lstlst_nil (a, 0, 0) of ()
// end of [lstlst]

fun{a:t@ype} _concat {m,t:nat} .<m>.
  (xss: lstlst (a, m, t)):<> list (a, t) = case+ xss of
  | lstlst_cons (xs, xss) => list_append (xs, _concat<a> xss)
  | lstlst_nil () => list_nil ()
// end of [_concat]
```

Given a type T and integers I and J, the type lstlst(T, I, J) is for a list of lists such that the length of the list is I and each element in the list is a list of values of the type T and the sum of the lengths of these elements equals J. The function list_concat is the same as the function _concat in terms of dynamic semantics, and it can be implemented as follows:

```
implement{a}
list_concat (xss) =
  _concat<a> (lstlst_of_listlist xss) where {
  castfn lstlst_of_listlist
    {m:nat} .<m>. (xss: list (List a, m))
    :<> [t:nat] lstlst (a, m, t) = case+ xss of
    | list_cons (xs, xss) => lstlst_cons (xs, lstlst_of_listlist xss)
    | list_nil () => lstlst_nil ()
} // end of [list_concat]
```

Given lstlst_of_listlist being implemented as a casting function, the implementation of list_concat is equivalent to the following one in terms of dynamic semantics:

```
implement{a}
list_concat (xss) = _concat (xss) // this one does not typecheck
```

Chapter 18. Stack Allocation at Run-Time

In ATS, there is support for allocating memory at run-time in the stack frame of the calling function, and it is guaranteed by the type system of ATS that the memory thus allocated cannot be accessed once the calling function returns.

In the following contrived example, the implemented function name_of_month_1 allocates in its stack frame an array of size 12 that is initialized with the names of 12 months, and then returns the name of the *i*th month, where *i* is an integer between 1 and 12:

```
fn name_of_month_1
  {i:int | 1 <= i; i <= 12} (i: int i): string = let
  var !p_arr with pf_arr = @[string](
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
  ) // end of [var]
in
  p_arr->[i-1] // it can also be written as !p_arr[i-1]
end // end of [name_of_month_1]
```

The following syntax means that the starting address of the allocated array is stored in p_arr while the view of the array is stored in pf_arr:

```
  var !p_arr with pf_arr = @[string](
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
  ) // end of [var]
```

This allocated array is initialized with the strings representing the names of the 12 months: "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec".

A variant of the function name_of_month_1 is implemeneted as follows:

```
fn name_of_month_2
  {i:int | 1 <= i; i < 12}
  (i: int i): string = let
  var !p_arr with pf_arr = @[string][12]("")
  val () = p_arr->[0] := "Jan"
  val () = p_arr->[1] := "Feb"
  val () = p_arr->[2] := "Mar"
  val () = p_arr->[3] := "Apr"
  val () = p_arr->[4] := "May"
  val () = p_arr->[5] := "Jun"
  val () = p_arr->[6] := "Jul"
  val () = p_arr->[7] := "Aug"
  val () = p_arr->[8] := "Sep"
  val () = p_arr->[9] := "Oct"
  val () = p_arr->[10] := "Nov"
  val () = p_arr->[11] := "Dec"
in
  p_arr->[i-1]
end // end of [name_of_month_2]
```

The following syntax means that the function name_of_month_2 allocates a string array of size 12 in its stack frame and initializes the array with the empty string:

```
  var !p_arr with pf_arr = @[string][12]("")
```

The starting address and the view of the allocated array are stored in p_arr and pf_arr, respectively. If the following syntax is used:

```
  var !p_arr with pf_arr = @[string][12]()
```

then the allocated array is uninitialized, that is, the view of the proof pf_arr is [string?][12] @ p_arr (instead of [string][12] @ p_arr).

When higher-order functions are employed in systems programming, it is often desirable to form closures in the stack frame of the calling function so as to avoid the need for memory allocation on heap. In the following example, the implemented function print_month_name forms a closure in its stack frame, which is then passed to a higher-order function iforeach_array_ptr_clo:

```
fn print_month_names () = let
  var !p_arr with pf_arr = @[string](
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
  ) // end of [var]
//
  var !p_clo with pf_clo = @lam // this closure is stack-allocated
    (i: sizeLt 12, x: &string): void =<clo1> (if i > 0 then print ", "; print x)
  // end of [var]
  val () = array_ptr_iforeach_clo<string> (!p_arr, !p_clo, 12)
//
  val () = print_newline ()
in
  // empty
end // end of [print_month_names]
```

Note that the keyword @lam (instead of lam) is used here to indicate that the closure is constructed in the stack frame of the function print_month_names.