

Chapter 5

Programming with Dependent Types

The primary purpose of introducing dependent types into programming is to greatly enhance the precision in using types to capture program invariants. 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 the boolean value *b*, it is often referred to as a singleton type. 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 the integer value *i*. Many other examples of dependent types are to be introduced gradually when this chapter unfolds. In particular, a means for the programmer to declare dependent datatype constructors is to be presented in Section 5.5.

5.1 Statics

The statics of ATS is a simply typed language. The types and terms in this language are referred to as sorts and static terms, respectively. Some of the base sorts are given as follows:

- The sort *bool* is for static terms of boolean values.
- The sort *int* is for static terms of integer values.
- The sort *type* is for static terms representing types of size equal to one word.
- The sort *t@ype* is for static terms representing types of unspecified size.

There are predicative and impredicative sorts: *bool* and *int* are predicative while *type* and *t@ype* are impredicative. If a static term is assigned a predicative sort, then then the term is often referred to as a type index.

There are various constants in the statics. In Figure 5.1, some commonly used static constant functions are listed together with their sorts. The symbols given inside parentheses are the names that refer to these constants in the concrete syntax. Many more static constants can be found in the following file:

```
$ATSHOME/prelude/basic_sta.sats.
```

\sim	(\sim)	:	$(int) \rightarrow int$
+	(+)	:	$(int, int) \rightarrow int$
-	(-)	:	$(int, int) \rightarrow int$
\times	(*)	:	$(int, int) \rightarrow int$
\div	(/)	:	$(int, int) \rightarrow int$
>	(>)	:	$(int, int) \rightarrow bool$
\geq	(>=)	:	$(int, int) \rightarrow bool$
<	(<)	:	$(int, int) \rightarrow bool$
\leq	(<=)	:	$(int, int) \rightarrow bool$
=	(==)	:	$(int, int) \rightarrow bool$
\neq	(<>)	:	$(int, int) \rightarrow bool$
\vee	()	:	$(bool, bool) \rightarrow bool$
\wedge	(&&)	:	$(bool, bool) \rightarrow bool$

Figure 5.1: Some commonly used static constants and their sorts

The names for static constant functions can be overloaded, and an overloaded name is resolved based on the arity information as well as the information on the sorts of the arguments.

A subset sort is a sort restricted by a predicate. For instance, *nat* is a subset sort defined as $\{a : int \mid a \geq 0\}$. In the concrete syntax, this is done as follows:

```
sortdef nat = {a: int | a >= 0 }
```

where *sortdef* is a keyword in ATS for introducing a sort definition. It is important to not confuse sorts with subset sorts. The latter can only be used to classify quantified static variables. For instance, the following type (written in the concrete syntax) can be assigned to a function that tests whether two given natural numbers are equal:

```
{i,j:nat} (int (i), int (j)) -> bool (i == j)
```

This type is essentially treated like syntactic sugar for the following one (also written in the concrete syntax):

```
{i,j:int | i >= 0; j >= 0} (int (i), int (j)) -> bool (i == j)
```

5.2 Common Arithmetic and Comparison Functions

Some commonly used arithmetic and comparison functions are listed in Figure 5.2 together with the dependent types assigned to them. In practice, overloaded names are often used to refer to these functions. For instance, in the following function definition, *>* and *-* are resolved into *igt* and *isub*, respectively:

```
// [mul_int_int]: the integer multiplication function
fun fact {n:nat} (x: int n): int =
  if x > 0 then mul_int_int (x, fact (x - 1)) else 1
```

$$\begin{aligned}
ineg & : \forall i : int. (int(i)) \rightarrow int(\sim i) \\
iadd & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow int(i_1 + i_2) \\
isub & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow int(i_1 - i_2) \\
imul & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow int(i_1 \times i_2) \\
idiv & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow int(i_1 \div i_2) \\
igt & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow bool(i_1 > i_2) \\
igte & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow bool(i_1 \geq i_2) \\
ilt & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow bool(i_1 < i_2) \\
ilte & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow bool(i_1 \leq i_2) \\
ieq & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow bool(i_1 = i_2) \\
ineq & : \forall i_1 : int. \forall i_2 : int. (int(i_1), int(i_2)) \rightarrow bool(i_1 \neq i_2)
\end{aligned}$$

Figure 5.2: Some common arithmetic and comparison functions and their dependent types

Note that the symbol *int* is overloaded: it may represent a dependent type constructor (as in *int*(*n*)) or just a type (for integers). Many more common functions on integers can be found in the following file:

\$ATSHOME/prelude/SATS/integer.sats.

5.3 Constraint Solving

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

```
// this definition does not typecheck due to a nonlinear constraint
fun fact {n:nat}
  (x: int n): [r:nat] int r = if x > 0 then x * fact (x - 1) else 1
```

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

$$\forall n : nat. int(n) \rightarrow \exists r : nat. int(r)$$

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:

1. $\forall n : nat. n > 0 \supset n - 1 \geq 0$
2. $\forall n : nat. \forall r_1 : nat. n > 0 \supset n * r_1 \geq 0$
3. $\forall n : nat. n > 0 \supset 1 \geq 0$

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 (Dantzig and Eaves, 1973). 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 is given a means to verify them by constructing proofs in ATS explicitly. The issue of explicit proof construction is to be elaborated elsewhere.

5.4 A Simple Example: Dependent Types for Debugging

Given a non-negative integer x , the integer square root of x is the greatest integer i satisfying $i * i \leq x$. In Figure 5.3, an implementation of the integer square root function is given based on the method of binary search. 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), we now attempt to identify the cause for non-termination by trying to prove the termination of the function *search* through the use of dependent types.

```
fn 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
      // [div_int_int] is the integer division function
      if div_int_int (x, m) < m then search (x, l, m) else search (x, m, r)
    end // end of [if]
    | _ => l
  end // end of [search]
in
  search (x, 0, x)
end // end of [isqrt]
```

Figure 5.3: A buggy implementation of the integer square root function

The function *search* in Figure 5.3 is assigned the type $(int, int, int) \rightarrow int$, meaning that *search* takes three integers as its arguments and returns an integer as its result. On the other hand, the programmer may have thought that the function *search* should possess the following invariants (if implemented correctly):

1. $l * l \leq x < r * r$ must hold when $search(x, l, r)$ is called.
2. Assume $l * l \leq x < r * r$ for some integers x, l, r . If a recursive call $search(x, l', r')$ for some integers l' and r' is encountered in the body of $search(x, l, r)$, then $r' - l' < r - l$ must hold. This invariant implies that *search* is terminating.

```

fn isqrt {x:nat} (x: int x): int = let
  fun search {x,l,r:nat | l < r} .<r-1>.
    (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 = 1 + (diff / 2)
        in
          // [div_int_int] is the integer division function
          if div_int_int (x, m) < m then search (x, l, m) else search (x, m, r)
        end // end of [if]
      | _ => l
    end // end of [search]
in
  if x > 0 then search (x, 0, x) else 0
end // end of [isqrt]

```

Figure 5.4: A fixed dependently typed implementation corresponding to the one in Figure 5.3

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, the following dependent function type is assigned to *search* in Figure 5.4, which captures a weaker invariant stating that $l < r$ must hold when $search(x, l, r)$ is called:

$$search : \forall x : nat. \forall l : nat. \forall r : nat. (l < r) \supset \langle r - l \rangle \Rightarrow ((int(x), int(l), int(r)) \rightarrow int)$$

It should not be difficult to relate this type to the concrete syntax representing it in the code. Note that the term $\langle r - l \rangle$, which corresponds to the concrete syntax $.<r-1>.$, represents a termination metric needed for verifying the second invariant. More details on termination metrics are to be given later.

With *search* being assigned the above dependent function type, the implementation in Figure 5.3 needs to be modified in order to pass typechecking. The modifications can be readily identified by comparing the code in Figure 5.3 to the the code in Figure 5.4, and the reader is strongly encouraged to do so and then figure out the reason for these modifications.

5.5 Dependent Datatypes

The feature of datatypes in ATS is directly adopted from ML. In ATS, there is even a means for the programmer to introduce dependent datatypes (or more precisely, dependent datatype constructors). For instance, the datatype constructor *list* in ATS is declared as follows:

```

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

```

The syntax indicates that *list* is a type constructor that forms a type $list(T, I)$ when applied to a type T (of sort $t@ype$) and an integer I . The sort $t@ype$ means that the size of T is unspecified. The plus sign following $t@ype$ states that *list* is covariant in its first argument, that is, $list(T_1, n)$ is a subtype of $list(T_2, n)$ if T_1 is a subtype of T_2 . There are two data constructors *list_nil* and *list_cons* associated with *list*, which are assigned the following types:

$$\begin{aligned} list_nil & : \forall a : t@ype. () \rightarrow list(a, 0) \\ list_cons & : \forall a : t@ype \forall n : int. (a, list(a, n)) \rightarrow list(a, n + 1) \end{aligned}$$

Given a type T and an integer I , it is clear that the type $list(T, I)$ is for lists of length I in which each element is of type T . The above declaration for *list* can also be given in a more succinct manner:

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

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

Figure 5.5: A tail-recursive implementation of the list length function

In Figure 5.5, an implementation of the list length function is given. The function template *list_length* can be instantiated with a type T (of unspecified size) to yield a function $list_length\langle T \rangle$ of the following function type:

$$\forall n : nat. list(T, n) \rightarrow int(n)$$

which clearly indicates that the value returned by the function $list_length\langle T \rangle$ is the length of its argument.

5.6 Pattern Matching

The feature of pattern matching in ATS is adopted from ML. However, there are some interesting issues with pattern matching that occur only in the presence of dependent datatypes (Xi, 2003).

5.6.1 Exhaustiveness

A function template is implemented as follows:

```
fun{a1,a2,b:t@type} list_zip_with {n:nat}
  (xs1: list (a1, n), xs2: list (a2, n), f: (a1, a2) -> b): list (b, n) =
  case+ (xs1, xs2) of
  | (list_cons (x1, xs1), list_cons (x2, xs2)) =>
    list_cons (f (x1, x2), list_zip_with (xs1, xs2, f))
  | (list_nil (), list_nil ()) => list_nil ()
```

Given two lists $v_{1,1}, \dots, v_{1,n}$ and $v_{2,1}, \dots, v_{2,n}$ where $v_{1,i}$ and $v_{2,i}$ are of types T_1 and T_2 for $1 \leq i \leq n$, the function call $list_zip_with\langle T_1, T_2 \rangle$ is expected to return a list v_1, \dots, v_n such that $v_i = f(v_{1,i}, v_{2,i})$ for $1 \leq i \leq n$. By the way, $list_zip_with$ is also often referred to as $list_map2$ in the literature. The use of the keyword `case+` indicates that the typechecker of ATS is able to verify the exhaustiveness of pattern matching in this example. If $xs1$ matches a non-empty list while $xs2$ matches an empty one, the typechecker essentially generates an assumption stating that $n = n_1 + 1$ and $n = 0$, where n_1 is a newly introduced variable ranging over natural numbers. As this is a contradictory assumption, the case is ruled out. Similarly, the case where $xs1$ matches an empty list while $xs2$ matches a non-empty one is also ruled out. As there are no other cases, the exhaustiveness of pattern matching is verified.

A slightly different implementation of $list_zip_with$ can be done as follows:

```
fun{a1,a2,b:t@type} list_zip_with {n:nat}
  (xs1: list (a1, n), xs2: list (a2, n), f: (a1, a2) -> b): list (b, n) =
  case+ xs1 of
  | list_cons (x1, xs1) => let
    val+ list_cons (x2, xs2) = xs2
  in
    list_cons (f (x1, x2), list_zip_with (xs1, xs2, f))
  end
  | list_nil () => list_nil ()
```

In this implementation, the keyword `val+` indicates that the pattern matching following it is exhaustive. Hence, the head and tail of $xs2$ can be extracted out without testing whether $xs2$ is empty.

5.6.2 Sequentiality

In ATS, pattern matching is performed sequentially at run-time. In other words, a clause is selected only if the given value matches the pattern associated with this clause but the value does not match the patterns associated with the clauses ahead of it. Naturally, one might expect that the following implementation of $list_zip_with$ also typechecks:

```
fun{a1,a2,b:t@type} list_zip_with {n:nat}
  (xs1: list (a1, n), xs2: list (a2, n), f: (a1, a2) -> b): list (b, n) =
  case+ (xs1, xs2) of
```

```

| (list_cons (x1, xs1), list_cons (x2, xs2)) =>
  list_cons (f (x1, x2), list_zip_with (xs1, xs2, f))
| (_, _) => list_nil ()

```

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 assuming that the given value does not match the pattern associated with the first clause. The second clause can be modified as follows:

```

| (_, _) =>> list_nil ()

```

The use of `=>>` (in place of `=>`) indicates to the typechecker that this clause needs to be typechecked under the assumption that the given value does not match the pattern associated with each previous clause. Hence, when the modified second clause is typechecked, it can be assumed that the value that matches the pattern `(_, _)` must match one of the following:

$$(list_cons(-, -), list_nil()) \quad (list_nil(), list_cons(-, -)) \quad (list_cons(-, -), list_cons(-, -))$$

This assumption allows typechecking to succeed.

5.7 Program Termination Verification

A termination metric is a tuple of natural numbers $\langle m_1, \dots, m_n \rangle$, and the standard lexicographic ordering on natural numbers is used to order such tuples. In ATS, termination metrics can be supplied (by the programmer) for verifying whether recursively defined functions are terminating, and this feature plays a crucial role in supporting the paradigm of programming with theorem proving.

In the following example, a singleton metric $\langle n \rangle$ is supplied to verify that the recursive function *fact* is terminating, when n is the value of the argument of *fact*:

```

fun fact {n:nat} .<n>. (x: int n): int = if x > 0 then x * fact (x-1) else 1

```

The metric attached to the call *fact*($x - 1$) is $\langle n - 1 \rangle$, which is obviously less than $\langle n \rangle$.

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 <= 100 && j == 91) || (i > 100 && j == i-10)] int j =
  if x > 100 then x-10 else f91 (f91 (x+11))

```

It is clear from the dependent type assigned to *f91* that the function always returns 91 when applied to an integer less than or equal 100. The metric supplied to verify the termination of *f91* is $\langle \max(101 - i, 0) \rangle$, where i is the value of the argument of *f91*.

The following code implements the Ackermann function, which is well-known for being recursive but not primitive recursive:

```

fun ack {m,n:nat} .<m,n>. (x: int m, y: int n): [r:nat] int r =
  if x > 0 then
    if y > 0 then ack (x - 1, ack (x, y - 1)) else ack (x - 1, 1)
  else y + 1

```

The metric supplied for verifying the termination of *ack* is a pair $\langle m, n \rangle$, where m and n are the values of the arguments of *ack*.

In the following example, *isEven* and *isOdd* are defined mutually recursively:

```
fun isEven {n:nat} .<2*n>. (x: int n): bool =  
  if x > 0 then isOdd (x - 1) else true  
  
and isOdd {n:nat} .<2*n+1>. (x: int n): bool =  
  not (isEven x)
```

The metrics supplied for verifying the termination of *isEven* and *isOdd* are $\langle 2 * n \rangle$ and $\langle 2 * n + 1 \rangle$, respectively, when n is the value of the argument of *isEven* and also the value of the argument of *isOdd*. Clearly, if the metrics $\langle n, 0 \rangle$ and $\langle n, 1 \rangle$ are supplied for *isEven* 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 of the same length.