2.12 Directives

In addition to expressions and deductions, the user can give various *directives* as input to Athena. These are commands that direct Athena to do something, typically to enter new information or adjust some setting about how it processes its input or how it displays its output. Most such directives have been mentioned already: **load** (page 22), **set-precedence** (page 32), **left-assoc**/**right-assoc** (page 33), **define** (page 40), **assert** (page 43), **clear-assumption-base**/**retract** (page 43), and **set-flag** auto-assert-dt-axioms (page 47). In this section we describe a couple of **set-flag** directives for controlling output, and the next section continues with the **overload** directive for adapting function symbols to have different meanings depending on context.

- **set-flag** print-var-sorts *s*, where *s* is either the string "on" or the string "off". The default value is "on". When set to "off", Athena will not print out the sorts of variables. Examples:

```
> set-flag print-var-sorts "off"

OK.

> (?x = ?y)

Term: (= ?x ?y)

> set-flag print-var-sorts "on"

OK.

> (?x = ?y)

Term: (= ?x:'T185 ?y:'T185)
```

- **set-flag** print-qvar-sorts *s*, where *s* is again either the string "on" or "off". When print-vars-sorts is turned off, variable sorts in the body of a quantified sentence are not printed, but variable occurrences that immediately follow a quantifier occurrence continue to have their sorts printed. The printing of even those sorts can be disabled by turning off the flag print-qvar-sorts:

```
> set-flag print-qvar-sorts "off"

OK.

> (forall ?x . exists ?y . ?y > ?x)
```

```
Sentence: (forall ?x
            (exists ?y
              (> ?y ?x)))
```

Turning off variable sort printing can simplify output significantly, especially when there are several long polymorphic sorts involved, but keeping it on can often provide useful information.

2.13 Overloading

Say we have introduced `Plus` as a binary function symbol intended to represent addition on the natural numbers:

```
declare Plus: [N N] -> N
```

While we could go ahead and use `Plus` in all of our subsequent proofs, it might be preferable, for enhanced readability, if we could use `+` instead of `Plus`, since `+` is traditionally understood to designate addition. However, `+` is already used in Athena to represent addition on real numbers, i.e., it is already declared at the top level as a binary function symbol that takes two real numbers and returns a real number. This means that we cannot redeclare `+` to take natural numbers instead. If we tried to do that we would get an error message:

```
> declare +: [N N] -> N

Warning, standard input:1:9: Duplicate symbol---the name + is
already used as a function symbol.
```

We could, of course, simply **`define`** `+` to be `Plus`, but then we would lose the original meaning of `+` as a binary function symbol on the real numbers. Alternatively, we could work inside a module, `M`, say, and declare `+` directly as a function symbol with the above signature, which would avoid any conflicts with the built-in declaration of `+`. But then when working outside the module we would have to qualify `+` with the module name, writing `(x M.+ y)` instead of `(x + y)`.

At the top level we can get around these difficulties by *overloading* `+` so that it can stand for `Plus` whenever that makes sense but revert to its original meaning at all other times:

```
> overload + Plus

OK.
```

We can now use `+` for both purposes: as an alias for `Plus`, to denote addition on natural numbers; and also to denote the original function, addition on real numbers. Which of these alternatives is chosen depends on the context. More specifically, it depends on the sorts of

the terms that we supply as arguments to +. If the arguments to + are natural numbers, then + is understood as Plus. If, on the other hand, the arguments are not natural numbers, then Athena infers that + is being used in its original capacity, to represent a function on real numbers:

```
1 > (?a + zero)
2
3 Term: (Plus ?a:N zero)
4
5 > (?a + 2.5)
6
7 Term: (a:Real + 2.5)
```

Here, on line 1, Athena interpreted + as Plus because, even though the variable ?a was not annotated, the second argument was zero, a natural number. Hence, the only alternative that was viable was to treat + as Plus. On line 5, by contrast, + could not possibly be understood as Plus, since the second argument was 2.5, of sort Real, and hence + was treated as it would have been normally treated prior to the overloading. If the arguments to + are completely unconstrained, in which case both interpretations are possible, then the most recently overloaded meaning takes precedence, in this case Plus:

```
> (?a + ?b)

Term: (Plus ?a:N ?b:N)
```

Essentially (if somewhat loosely), after a directive of the form **overload** *f g* has been issued, every time Athena encounters an application of the form (*f* ···) it tries to interpret it as (*g* ···). If that fails, then it interprets the application based on the original meaning of *f*.

This is not overloading in the conventional sense of the term, because we do not directly declare + to have two distinct signatures, one of which expects two natural numbers as inputs and produces a natural number as output. Instead, we first introduce Plus, and then we essentially announce that + is to be used as an alias for Plus in any context in which it is sensible to do so. Thus, for instance, (+ zero zero) actually produces the term (Plus zero zero) as its output. So + here really is used simply as a synonym for Plus.

Note that after the overloading, + no longer denotes a function symbol. Rather, it denotes a binary procedure that does what was described above: it first tries to apply Plus to its two arguments, and if that fails, it then tries to apply to them *whatever was previously denoted by* +, which in this case is a function symbol.[28] This process can be iterated indefinitely. For instance, after first overloading + to represent Plus, we might later overload it even further, say, to represent ::, the reflexive constructor of the List datatype (see page 56):

28 Of course, + remains a function symbol in the current signature.

```
1  > overload + Plus
2
3  OK.
4
5  > (?a + ?b)
6
7  Term: (Plus ?a:N ?b:N)
8
9  > overload + ::
10
11 OK.
12
13 > (?a + ?b)
14
15 Term: (:: ?a:'T2
16           ?b:(List 'T2))
17
18 > (?a + zero)
19
20 Term: (Plus ?a:N zero)
21
22 > (?a + 3.14)
23
24 Term: (+ ?a:Real 3.14)
```

Here, after the second overloading on line 9, + denotes a binary procedure that takes two values v_1 and v_2 and tries to apply :: to them. If that fails, then it applies to v_1 and v_2 whatever was previously denoted by +, which in this case is the procedure that resulted from the first overloading, on line 1.

Multiple overloadings can be carried out in one fell swoop as follows:

```
> overload (+ Plus) (- Minus) (* Times) (/ Div)

OK.
```

The above is equivalent to the following four individual directives:

```
overload + Plus
overload - Minus
overload * Times
overload / Div
```

The **overload** directive is most useful when we are working exclusively at the top level or inside a single module. Across different modules there is rarely a need for explicit overloading, since the same function symbol can be freely declared in multiple modules with different signatures. That is, two distinct modules M and N are allowed to declare a function symbol of one and the same name, f. No conflict arises because the enclosing modules serve to disambiguate the symbols: in one case we are dealing with $M.f$ and in the other

with $N.f$. So we could, for instance, develop our theory of natural numbers inside a module named, say, `N`, and then directly introduce a function symbol `+` inside `N` to designate addition, which would altogether avoid the introduction of `Plus`. This is, in fact, the preferred approach when developing specifications and proofs in the large. The alternative we have described here, **`overload`**, does not involve modules and can be used in smaller-scale projects (though it can also come in handy sometimes inside modules).

We have seen that if f is a function symbol, then after a directive like

`overload` f g

is issued, f will no longer denote the symbol in question; it will instead denote a procedure (recall that function symbols and procedures are distinct types of values). This raises the question of how we can now retrieve the *symbol* f. For instance, consider the first time we overload `+`:

```
> overload + Plus

OK

> +

Procedure: +
```

How can we now get ahold of the actual function symbol `+`? (We might need the symbol itself for some purpose or other.) The newly defined procedure can still make terms with the symbol `+` at the top, so all we need to do is grab that symbol with the `root` procedure, though simple pattern matching would also work:

```
> (root (1 + 2))

Symbol: +
```

But probably the easiest way to obtain the function symbol after the corresponding name has been redefined via **`overload`** is to use the primitive procedure `string->symbol`, which takes a string and, assuming that the current symbol set contains a function symbol f of the same name as the given string, it returns f:

```
> (string->symbol "+")

Symbol: +
```

2.14 Programming

In this section we briefly survey some Athena features that are useful for programming.