Notes & Problem Set 2

DPL Seminar, Summer 2001
Handout 4

Konstantine Arkoudas

July 12, 2001
Numbers in Athena

Athena does not have numbers (either integers or reals) built-in as primitive syntactic categories. However, both integer and real numerals can be introduced as constant terms, and everything that can be done with numbers in a typical programming language can then also be done in Athena just as easily. In fact, treating numerals as constant terms has some added advantages: they can be directly matched or unified, they can appear inside propositions, they can be quantified over, etc.

Integers

Integer numerals are typically introduced as follows:

>(domain Number)

New domain Number introduced.

>(use-numerals (0 ...) Number)

Numerals in the range (0 ...) can now be used as terms of Number.

The name Number is of course arbitrary; any other name would do, provided it is not already used to denote another domain. The use-numerals directive is essentially equivalent in effect to the following infinite sequence of symbol declarations:

>(declare 0 Number)

New symbol 0 declared.

>(declare 1 Number)

New symbol 1 declared.

>(declare 2 Number)

New symbol 2 declared.

... 

It instructs Athena to treat every numeral in that range as a constant term of sort Number. Accordingly, if we tell Athena to print out the sort of, say 75, we get the following:

>(print (join "\n" (sort-of 75) "\n"))

Number

Unit: 0

There is no limit to the size of numerals one might enter:

>(exists ?n (= ?n 1219843983246723726329234823672390237237682223940404))

Proposition: (exists ?n:Number

(= ?n 1219843983246723726329234823672390237237682223940404))

The usual numeric operations of addition, multiplication, etc., can be introduced with the following directive:
>(define-numeric-operations)

The following binary functions have been introduced:
plus, minus, times, div, mod, num-equal?, less?, and greater?.
These functions can be applied to any numerals in the (0 ... ) range.

This directive augments the top-level Athena environment with the listed functions, which perform
numeric operations on numerals (terms of sort Number) in the expected manner. For instance:

>(plus 10 47)
Term: 57
>(define (cube n) (times n (times n n)))
Function cube defined.
>(cube 10)
Term: 1000
>(cube "foo")
Error, top level, 1.34: Wrong kind of value given as first argument
to times---a term was expected, but the given argument was a list: ['f,'o,'o].

>(less? 23 754)
Term: true
If one prefers different names for these operations, they can be renamed, since Athena is higher-order:
>(define + plus)
Function + defined.
>(+ 20 30)
Term: 50
These operations do not have infinite precision. Overflow or underflow will occur if the relevant size
bounds (which are left unspecified in the language definition) are exceeded:

>(define (fact n)
  (check ((num-equal? n 0) 1)
    (else (times n (fact (minus n 1)))))
Function fact defined.
>(fact 10)
Term: 3628800
>(fact 20)
Error, top level, 3.17: Call to times resulted in overflow.

As introduced by the foregoing directives, the subtraction function minus will be partial. That is,
if its first argument is less than the second (in the customary ordering), then the numeral 0 will be
returned by default.
\texttt{\textbackslash{(\text{minus 3 1})}}

\texttt{Term: 2}

\texttt{\textbackslash{(\text{minus 3 200})}}

\texttt{Term: 0}

However, if we have already introduced a unary function symbol which we intend to denote the inverse operation on the numeric domain (viewed as a group, with 0 acting as the identity element), then we can instruct Athena to take this into account as illustrated below. First, suppose that we intend to model the inverse function on \texttt{Number} (i.e., unary minus) by the symbol \texttt{`}: 

\texttt{\textbackslash{(\text{domain Number})}}

New domain \texttt{Number} introduced.

\texttt{\textbackslash{(\text{declare } ` \rightarrow (\text{Number} \text{ Number})\texttt{)}}}

New symbol \texttt{`} declared.

\texttt{\textbackslash{(\text{use-numerals (0 \ldots) Number})}}

Numerals in the range (0 \ldots) can now be used as terms of \texttt{Number}.

\texttt{\textbackslash{(\text{define-numeric-operations } `)}}

The following binary functions have been introduced:
\texttt{plus, minus, times, div, mod, num-equal?, less?, and greater?}.

These functions can be applied to any numeric constants, where a "numeric constant" is either a numeral in the (0 \ldots) range, or a term of the form (\texttt{`} t'), where t is a numeric constant.

By passing the symbol \texttt{`} as an "argument" to \texttt{define-numeric-operations}, we instruct Athena to use \texttt{`} as an inverse operation, with 0 as the identity. We now have:

\texttt{\textbackslash{(\text{minus 2 5})}}

\texttt{Term: (\texttt{`} 3\texttt{)}}

\texttt{\textbackslash{(\text{plus (\texttt{`} 3\texttt{)}} (\texttt{`} 5\texttt{)))}}

\texttt{Term: (\texttt{`} 8\texttt{)}}

\texttt{\textbackslash{(\text{times (\texttt{`} 3\texttt{)}}
\texttt{\ (\texttt{`} (\texttt{`} 4\texttt{)))}}\texttt{)}}

\texttt{Term: 12}

\texttt{\textbackslash{(\text{plus 7 (\texttt{`} 7\texttt{)))}}

Term: 0}

Of course any other function symbol could be used in place of \texttt{`} so long as it was of the signature (\texttt{\rightarrow (\text{Number} \text{ Number})}). Athena will check to make sure that this is the case; it is important for sort checking, which can affect logical soundness.

Note that if we were only interested in the positive integers, we could have issued the following declarations:

\texttt{\textbackslash{(\text{domain PosNum}}

3
New domain PosNum introduced.

> (use-numerals (1 ...) PosNum)

Numerals in the range (1 ...) can now be used as terms of PosNum.
We then have:

> 23

Term: 23

> (print (join "\n" (sort-of 23) "\n"))

PosNum

Unit: 0

> 0

Error, top level, 1.1: Unbound identifier: 0.

In this case, 0 is not recognized. The previous numeric operations can be introduced here as well with a directive of the form (define-numeric-operations). Here minus will return 1 as the least element by default. Since the positive integers do not have an identity element in the usual interpretation, if we try to pass a unary function symbol of signature (-> (PosNum) PosNum) to define-numeric-operations, the latter will ignore it and will issue a relevant warning.

Reals

Real numerals can also be used in a similar manner:

> (domain Real)

New domain Real introduced.

> (use-numerals (0.0 ...) Real)

Numerals in the range (0.0 ...) can now be used as terms of Real.

> 3.14278

Term: 3.14278

> .9548458949583943464748317394765648309302387362754445763252373293229867645210

Term: .9548458949583943464748317394765648309302387352754445763252373293229867645210

> 23.

Term: 23.

> 55

Term: 55

> (define-numeric-operations)
The following binary functions have been introduced:
plus, minus, times, div, num-equal?, less?, and greater?.
These functions can be applied to any numerals in the (0.0 ...) range.

```
> (plus 1.3 7)
Term: 2.0
```

```
> (div 9.0 3)
Term: 3.0
```

Inverses and subtraction issues are handled as in the case of integers:

```
>(domain Real)
New domain Real introduced.
```

```
> (declare " (-> (Real) Real))
New symbol " declared.
```

```
> (use-numerals (0.0 ...) Real)
Numerals in the range (0.0 ...) can now be used as terms of Real.
```

```
> (define-numeric-operations ")
The following binary functions have been introduced:
plus, minus, times, div, num-equal?, less?, and greater?.
These functions can be applied to any numeric constants, where a "numeric constant" is
either a numeral in the (0.0 ...) range, or a term of the form (^ t),
where t is a numeric constant.
```

```
> (minus 2.0 5.0)
Term: (^ 3.0)
```

```
> (times (div 3.0 1) (^ 3.0))
Term: (^ 9.0)
```

We could also restrict attention to the positive reals with a directive of the form

```
(use-numerals (0.0 ...) Real).
```

This will affect inverse and subtraction issues just as it does in the case of integers.
Problems

Due: Thursday, July 19.

1. Consider the following four inference rules, collectively known as “quantifier negation”:

\[
\begin{align*}
\frac{\vdash \neg (\exists x) P}{\vdash (\exists x) \neg P} & \quad [\text{qn-1}] \\
\frac{\vdash \neg (\forall x) P}{\vdash (\forall x) \neg P} & \quad [\text{qn-2}] \\
\frac{\vdash \neg (\exists x) P}{\vdash (\forall x) \neg P} & \quad [\text{qn-3}] \\
\frac{\vdash \neg (\forall x) P}{\vdash (\exists x) \neg P} & \quad [\text{qn-4}]
\end{align*}
\]

Define four unary Athena methods qn-1, qn-2, qn-3, and qn-4 that implement these rules. Then write a unary method qn that takes a premise \( P \) and dispatches the appropriate quantifier negation method according to the form of \( P \).

Test these methods to convince yourself that qn-1 and qn-2 are inverses, in the sense that

\[
\beta \cup \{\neg (\forall x \ P)\} \vdash (\exists x (\neg P)) \quad \text{and} \quad (\text{exists x (not P)}) \vdash (\neg (\forall x \ P))
\]

and

\[
\beta \cup \{\neg (\exists x \ P)\} \vdash (\exists x (\neg P)) \quad \text{and} \quad (\text{exists x (not P)}) \vdash (\neg (\exists x \ P))
\]

for all assumption bases \( \beta \), Athena variables \( x \), and propositions \( P \); and likewise for qn-3 and qn-4.

2. Define a method un-curry that takes a premise of the form \( P_1 \Rightarrow (P_2 \Rightarrow P_3) \) and derives the conclusion \( (P_1 \land P_2) \Rightarrow P_3 \). Thus, schematically, the behavior of un-curry can be depicted as follows:

\[
\begin{align*}
\frac{\vdash P_1 \Rightarrow (P_2 \Rightarrow P_3)}{\vdash (P_1 \land P_2) \Rightarrow P_3} & \quad [\text{un-curry}]
\end{align*}
\]

Then define a method un-curry* that takes a premise of the form \( P_1 \Rightarrow \cdots \Rightarrow P_n \Rightarrow P_{n+1} \), \( n > 1 \), and derives the conclusion \( (P_1 \land \cdots \land P_n) \Rightarrow P_{n+1} \). Note: the name “uncurry” derives from identifying the connectives \( \land \) and \( \Rightarrow \) with the type constructors \( \times \) and \( \rightarrow \), respectively. In that light the uncurrying rule is viewed as transforming a curried functional type \( T_1 \rightarrow T_2 \rightarrow T_3 \) to the “uncurried” \( (T_1 \times T_2) \rightarrow T_3 \). (Bonus: write methods curry and curry* that proceed in the reverse direction.)

3. Consider an inference rule equiv-cong that takes a premise of the form \( P_1 \Leftrightarrow P_2 \) and derives the equivalence \( P \Leftrightarrow P'[P_2/P_1] \), for some arbitrary proposition \( P \), where we write \( P'[P_2/P_1] \) to denote the proposition obtained from \( P \) by replacing every occurrence of \( P_1 \) by \( P_2 \). Therefore, graphically, the rule may be depicted as

\[
\frac{\vdash P_1 \Leftrightarrow P_2}{\vdash P \Leftrightarrow P'[P_2/P_1]} & \quad [\text{equiv-cong}]
\]

Implement this rule as a binary method equiv-cong that takes the arbitrary proposition \( P \) as its first argument and the premise \( P_1 \Leftrightarrow P_2 \) as its second argument and deduces the conclusion \( P \Leftrightarrow P'[P_2/P_1] \).

Then define a binary method replace-equiv that takes an arbitrary premise \( P \) and a premise of the form \( P_1 \Leftrightarrow P_2 \) and derives the conclusion \( P'[P_2/P_1] \). (This is really easy once you have
Such a method is quite powerful and very convenient. For instance, suppose that $P$ is the proposition $A \land [C \Rightarrow (\neg B \lor D)]$ and that we want to replace the occurrence of $\neg B$ by $B$, where both $P$ and the equivalence $\neg B \Leftrightarrow B$ are in the assumption base. Then applying `replace-equiv` to $P$ and the said equivalence would derive the conclusion $A \land [C \Rightarrow (B \lor D)]$.

Note: You do not have to handle quantified propositions for this problem. You only have to cover atomic propositions and propositional combinations (negations, conjunctions, disjunctions, conditionals and biconditionals).

4. Consider the following abstract syntax for a simple functional programming language:

$$E ::= n \mid I \mid \mathtt{true} \mid \mathtt{false} \mid \langle E_1, E_2 \rangle \mid l(E) \mid r(E) \mid \lambda I. E \mid E_1 E_2$$

(1.1)

let $I = E_1$ in $E_2$ | fix $I. E$ | if $E_1$ then $E_2$ else $E_3$

where $n$ ranges over the natural numbers $0, 1, 2, \ldots$ and $I$ is an identifier. This abstract syntax can be represented by the following Athena structure:

(structure Exp
  (NumExp Num)
  (VarExp Id)
  (TrueExp)
  (FalseExp)
  (PairExp Exp Exp)
  (LeftExp Exp)
  (RightExp Exp)
  (LambdaExp Id Exp)
  (AppExp Exp Exp)
  (LetExp Id Exp Exp)
  (FixExp Id Exp)
  (CondExp Exp Exp Exp))

where we assume that the declaration (domain Num) and directive `use-numerals (0 \ldots) Num` have already been entered before the definition of `Exp`.

One possible concrete syntax that can be used for this language is the following (there are many other choices, of course):

$$E ::= n \mid I \mid \mathtt{true} \mid \mathtt{false} \mid (\mathtt{lambda} (I) E) \mid (E \ E) \mid (\mathtt{let} (I \ E) E) \mid \langle E \ E \rangle \mid (\mathtt{left} E) \mid (\mathtt{right} E) \mid \mathtt{if} \ E \ E \ E \mid (\mathtt{fix} \ I \ E)$$

(1.2)

where $n$ is a numeral and an identifier $I$ is any non-empty sequence of printable ASCII characters that:

- can serve as a constant term of sort $\mathtt{Id}$; and
- is neither a numeral nor a keyword (namely, `true`, `false`, `let`, `lambda`, `fix`, `left`, `right`, `if`, `(`, `)` `,<`, `>`)....

Definition 1.2 specifies an unambiguous LL(1) grammar that can be used as a concrete front end for the abstract grammar 1.1. In this problem, you will write a parser that reads a text file and produces a list of parse trees represented as terms of sort `Exp`.

This type of task is customarily broken into two parts: lexing first, parsing later. So first define a structure `Token` representing all possible lexical tokens, and then define a function `tokenize` that takes a string and returns a list of tokens (represented as terms of sort `Token`), getting rid of white space in the process. An appropriate structure for `Token` is the following:
(structure Token
TRUE
FALSE
LPAREN
RPAAREN
LET
FUN
FIX
LEFT
RIGHT
LBRACK
RBRACK
If
(Number Num)
(ID Ide))

Now write a function parse-tokens that takes a list of tokens tokens and:

- If any prefix of tokens constitutes a string of the above concrete grammar, parse-tokens returns (a) a term of sort Exp representing the shortest such prefix, and (b) the remaining suffix. (You can return (a) and (b) together in a two-element list.)
- Otherwise parse-tokens halts with a message indicating the cause of the parsing error.

Then write a function parse-string that takes a string, tokenizes it, and parses the resulting list of tokens using parse-tokens (hence the result of parse-string will be either a list of parse trees or an error message). Finally, write a function parse-file that takes the name of a text file, reads that file into a string, tokenizes that string, and then parses the resulting list of tokens. (Note that Athena has a top-level unary function read-file that takes the name of file, loads the file into a list of characters and returns that list.)