MAU22C00 Lecture 12

John Stalker

Trinity College Dublin

Scope in programming languages versus scope in logic

The following is an unnecessarily complicated `C` function.

```
int factorial(int n) {
    int i, j, m;
    m = 1
    for(i = 1; i <= n; i++) {
        int j, k;
        /* Any reference to j here means the j within the scope of the
           loop. We have no way to refer to the j from the scope of
           the function. */
        m = i*m;
    }
    /* any reference to k here will generate a compiler error */
    return m;
}
```

The compiler makes sure you don't refer to variables out of the scope they're defined in and manages name conflicts. In logic you have to do this manually.

Elementary arithmetic

Our next topic is elementary arithmetic. Elementary means

- we only consider natural numbers, not integers, rational numbers, real or complex numbers, matrices, etc.
- we only make statements about numbers, not sets of numbers, sequences, functions, etc.

It's not *that* elementary though.

- we have available all of first order logic except predicates
- although we can't talk about sets or functions directly we can still write down Boolean expressions that a number or pair of numbers satisfy, so we can sort of define (some) sets or functions, just not name them.

A language for arithmetic

```
statement : bool_exp ;
bool_exp : ( ¬ bool_exp) | ( bool_exp b_operator bool_exp )
         | ( quantifier variable . bool_exp )
         | ( num_exp relation num_exp ) ;
         | [ ¬ bool_exp) | [ bool_exp b_operator bool_exp )
         | [ quantifier variable . bool_exp )
         | [ num_exp relation num_exp ) ;
         | { ¬ bool_exp} | { bool_exp b_operator bool_exp }
         | { quantifier variable . bool_exp }
         | { num_exp relation num_exp } ;
b_operator : ∧ | ∨  | ⊃ ;
quantifier : ∀ | ∃ ;
variable : letter | variable ! ;
variable : v | w | x | y | z ;
relation : = | ≤ ;
num_exp : 0 | variable | num_exp ' | ( num_exp a_operator num_exp ) ;
a_operator : + | · ;
```

Comments

We keep all of first order logic except predicates and parameters.

We add some arithmetic operators, for addition and multiplication, but not others, like subtraction, division and exponentiation.

We add some relations, $=$ and $\leq$, but not others, like $<$, $>$, $\geq$, or $\neq$.

The other relations are not needed because they don't add to the expressiveness of the language.

$(x < y)$ means the same as $[\neg(y \leq x)]$, $(x > y)$ means the same as $[\neg(x \leq y)]$, $(x \geq y)$ means the same as $(y \leq x)$, and $x \neq y$ means the same as $[\neg(x = y)]$.

Strictly speaking, we don't need both $=$ and $\leq$ either. $(x = y)$ means the same as $[(x \leq y) \wedge (y \leq x)]$ and $(x \leq y)$ means the same as $[\exists z.(x + z = y)]$.

This is a compromise between ease of writing and proving statements *within* the language and ease of writing and proving statements *about* the language.

More comments

The reason we don't have symbols for subtraction and division is different. First order logic does not work properly when you can name non-existent objects. So we can't allow expressions like $1/0$ and therefore can't allow x/y since y could be 0.

We also can't allow $0-1$. $0-1$ does exist, but not as a natural number.

Actually we don't allow 1 either! Instead we have 0 and the increment operator $'$, so we use $0'$ in place of 1.

So $2+2=4$ is $0''+0''=0''''$.

The reasons for this are mostly historical, but we don't want to make our theory of arithmetic dependent on decimal representation.

Still more comments

We have two types of expressions, Boolean and numerical.

Boolean operators combine Boolean expressions to produce Boolean expressions.

Arithmetic operators, $+$, $\cdot$ and $'$, combine numerical expressions to give numerical expressions.

The relations $=$ and $\leq$ combine numerical expressions to give Boolean expressions.

Variables are numerical expressions. There are no Boolean variables.

Roughly, Boolean expressions play the role in arithmetic that predicates did in first order logic, while numerical expressions play the role that parameters did.

Expressing more complicated ideas

We can build expressions expressing more complicated ideas.

"x is even" can be expressed as $\{\exists y.[x = (0'' \cdot y)]\}$ or $\{\exists y.[x = (y + y)]\}$

"x is odd" can be expressed as $(\exists y.\{x = [(0'' \cdot y) + 0']\})$ or $\{\exists y.[x = (y + y)']\}$ or $(\neg\{\exists y.[x = (0'' \cdot y)]\})$.

There are usually multiple "translations" of informal statements and the equivalence of these may require facts about arithmetic.

"$x = y - z$" can be expressed as $x + z = y$, so we don't really miss subtraction.

We can also express divisibility. "x is divisible by y" can be expressed as $\{\exists z.[(x \cdot z) = y]\}$.

We can express primality by translating the statement that x is prime if and only if is greater than 1 and has no divisors other than 1 and itself.

We can even express the fact that there are infinitely many primes.

See the notes for the corresponding statements, and other examples.

Expressing even more complicated ideas

Can we say that x is a power of 2?

We don't have a notation for exponentiation so we can't just write $\{\exists y.[(0''^{y}) = x]\}$.

If we assume that we know the theorem about unique factorisation we can say that a number which is not a power of 2 is divisible by some prime other than 2, which must be odd and greater than 1. Conversely, a number which is a power of 2 is divisible only by even numbers, except for 1.

"x is a power of 2" can therefore be expressed as
$(\neg\{\exists y.[\exists z.(x = \{y \cdot [(0'' \cdot z') + 0']\})]\})$.

There's a similar, but slightly different, translation in the notes.

This is a bit unsatisfying because "is a power of 2" is a simpler notion than the unique factorisation theorem.

The notes have a translation of "x is a Fibonacci number" as well.

Arithmetic sets

Sets whose membership condition is expressible in our language are called *arithmetic*, so the sets of even numbers, odd numbers, powers of 2, and Fibonacci numbers are all arithmetic.

Not every set of natural numbers is arithmetic.

We'll be able to prove this later. I'll also give an example, without a proof, of a non-arithmetic set.

Arithmetic sets are defined in terms of our language, but are not part of it.

Encoding

Arithmetic is of independent interest, but is also interesting because we can encode other subjects in it.

Given a language with b tokens we can label them 1 through b and associate to any list of tokens the natural number whose base $b+1$ representation is those digits in that order.

The language is then represented by a set of natural numbers.

This is one encoding method. There are others.

Some of them can cope with an infinite number of tokens, if it's not too infinite.

Are the sets obtained by encoding languages arithmetic?

No, in general, but yes, if the language has a phrase structure grammar.

This is quite complicated to prove. I won't even attempt it.

Ouroboros

We can encode any language in arithmetic, and encode languages with phrase structure grammars as arithmetic sets.

We have a phrase structure grammar for arithmetic, so we can encode arithmetic within arithmetic!

This isn't very useful for proving statements *within* arithmetic but it is a powerful technique for proving statements *about* arithmetic.