Dependent type Programming Paradigm

Introduction

A constant problem:

- Writing a correct computer program is hard and proving that a program is correct is even harder
- Dependent Types allow us to write programs and know they are correct before running them.
- dependent types: you can specify types that can check the value of your variables at compile time

Example:

Here is how you can declare a Vector that contains the values 1, 2, 3 :

```
val l1 = 1 :#: 2 :#: 3 :#: Vnil
```

This creates a variable l1 who's type signature specifies not only that it's a Vector that contains Ints, but also that it is a Vector of length 3. The compiler can use this information to catch errors. Let's use the vAdd method in Vector to perform a pairwise addition between two Vectors:

```
val l1 = 1 :#: 2 :#: 3 :#: VNil
val l2 = 1 :#: 2 :#: 3 :#: VNil
 val l3 = l1 vAdd l2
 // Result: l3 = 2 :#: 4 :#: 6 :#: VNil
```

Introduction

The example above works fine because the type system knows both Vectors have length 3. However, if we tried to vAdd two Vectors of different lengths, we'd get an error at compile time instead of having to wait until run time!

```
val l1 = 1 :#: 2 :#: 3 :#: VNil
val l2 = 1 :#: 2 :#: VNil

val l3 = l1 vAdd l2

// Result: a *compile* error because you can't pairwise add vectors
// of different lengths!
```

Note:

You can express almost anything with dependent types. A factorial function which only accepts natural numbers, a login function which doesn't accept empty strings, a remove Last function which only accepts non-empty arrays. And all this is checked before you run the program.

Introduction

A function has dependent type if the type of a function's result depends on the VALUE of its argument; this is not the same thing as a ParameterizedType. The second order lambda calculus possesses functions with dependent types.

What does it mean to be “correct”?

Depends on the application domain, but could mean one or more of:

- Functionally correct (e.g. arithmetic operations on a CPU)
- Resource safe (e.g. runs within memory bounds, no memory leaks, no accessing unallocated memory, no deadlock. . .)
- Secure (e.g. not allowing access to another user’s data)

What is type?

- In programming, types are a means of classifying values
- Exp: values 94, "thing", and [1,2,3,4,5] □ classified as an integer, a string, and a list of integers
- For a machine, types describe how bit patterns in memory are to be interpreted.
- For a compiler or interpreter, types help ensure that bit patterns are interpreted consistently when a program runs.
- For a programmer, types help name and organize concepts, aiding documentation and supporting interactive editing environments.

Introduction

In computer science and logic, a dependent type is a type whose definition depends on a value.

It is an overlapping feature of type theory and type systems.

Used to encode logic's quantifiers like "for all" and "there exists".

Dependent types may help reduce bugs by enabling the programmer to assign types that further restrain the set of possible implementations.

Quantifiers

A predicate becomes a proposition when we assign it fixed values. However, another way to make a predicate into a proposition is to quantify it. That is, the predicate is true (or false) for all possible values in the universe of discourse or for some value(s) in the universe of discourse. Such quantification can be done with two quantifiers: the universal quantifier and the existential quantifier.

Universal: Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing. The universal quantification of a predicate P(x) is the proposition "P(x) is true for all values of x in the universe of discourse" We use the notation ∀ for Universal quantifier

$$\forall x P(x)$$

which can be read "for all x"

Example:

Let P(x) be the predicate "x must take a discrete mathematics course" and let Q(x) be the predicate "x is a computer science student". The universe of discourse for both P(x) and Q(x) is all UNL students.

Express the statement "Every computer science student must take a discrete mathematics course".

$$\forall x(Q(x) \rightarrow P(x))$$

Express the statement "Everybody must take a discrete mathematics course or be a computer science student".

$$\forall x(Q(x) \vee P(x))$$

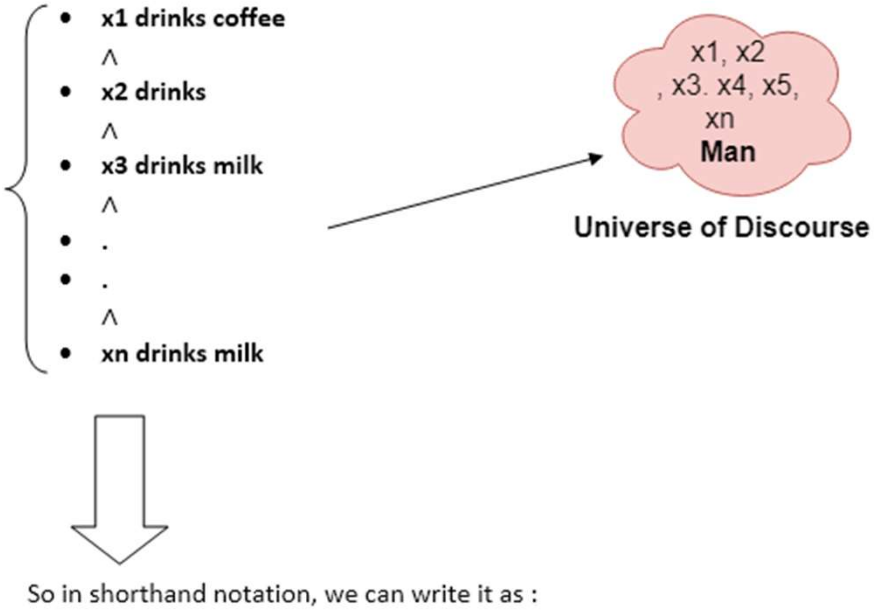
If x is a variable, then ∀x is read as:

For all x **For each x** **For every x.**

Quantifiers

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



∀x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifiers

The existential quantification of a predicate P(x) is the proposition "There exists an x in the universe of discourse such that P(x) is true." We use the notation

$$\exists x P(x)$$

which can be read "there exists an x"

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

There exists a 'x.'

For some 'x.'

For at least one 'x.'

Existential Quantifiers

Some boys are intelligent.

- **x1 is intelligent**

  V
- **x2 is intelligentV**
- **x3 is intelligent**

  V
- .
- .

  V
- **xn is intelligent**

Universe of Discourse

So in short-hand notation, we can write it as:

∃x: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

Examples

1. All birds fly.

In this question the predicate is "fly(bird)."

And since there are all birds who fly so it will be represented as follows.

$\forall x$ bird(x) $\rightarrow$fly(x).

2. Every man respects his parent.

In this question, the predicate is "respect(x, y)," where x=man, and y= parent.

Since there is every man so will use $\forall$, and it will be represented as follows:

$\forall x$ man(x) $\rightarrow$ respects (x, parent).

3. Some boys play cricket.

In this question, the predicate is "play(x, y)," where x= boys, and y= game. Since there are some boys so we will use $\exists$, and it will be represented as:

$\exists x$ boys(x) $\rightarrow$ play(x, cricket).