Programming IS Logic IS Math

Nathan Jarus

December 8, 2016

Propositions As Types

- Standard types correspond to pretty simple statements
  - For example: `int` corresponds to "This is an integer" and `5` is a proof of that statement
- But, there is no reason we can't imagine more complex types with more interesting statements
  - `int a[5]` says "This array has 5 elements, each of which is an integer"

Propositions As Types

- Standard types correspond to pretty simple statements
  - For example: `int` corresponds to "This is an integer" and `5` is a proof of that statement
- But, there is no reason we can't imagine more complex types with more interesting statements
  - `int a[5]` says "This array has 5 elements, each of which is an integer"
- The *Curry-Howard Correspondence* says
  - Types correspond to logical statements
  - Values correspond to proofs of those statements

Connecting Statements: AND

- We know that if we have two proven statements A and B, we can prove "A and B"
- How would we encode this idea in a type?

Connecting Statements: AND

- We know that if we have two proven statements A and B, we can prove "A and B"
- How would we encode this idea in a type?

```
template<class A, class B>
struct Pair {
  A first;
  B second;
};
```

- `Pair` says "If you have an element of type `A` and an element of type `B`, you can construct an element of type `Pair<A,B>`."
  - In `Pair<int,char> p = {7,'?'}`, `p` is a proof of the claim "`int` and `char`"

Connecting Statements: IF-THEN

- We want something that says "If you have an `A`, then you can get a `B`." What programming concept is this?

Connecting Statements: IF-THEN

- We want something that says "If you have an `A`, then you can get a `B`." What programming concept is this?

```
B implication(A a);
```

- If you have `a` of type `A`, then `implication(a)` will give you something of type `B`.
- In logical terms, `implication` transforms a proof of `A` into a proof of `B`.

Connecting Statements: OR

- The last logical connector we're missing is OR. How would we represent one of these?
- We want something we can construct given either something of type `A` or of type `B`.

```
const bool LEFT=false;
const bool RIGHT=true;

template<class A, class B>
struct Either {
  bool side;
  union {
    A a;
    B b;
  } item;
};
```

```
Either<int,char> e =
  {LEFT,{.a = 5}};

switch(e.side) {
  case LEFT:
    cout << "I'm an int! "
      << e.item.a << endl;
    break;
  case RIGHT:
    cout << "I'm a char! "
      << e.item.b << endl;
    break;
}
```

How neat is that?

- So this correspondence between logic and programming actually helps us discover new ideas for programming!
- You can use this to, for instance, return either a result or an error from a function.
- Returning an Either forces you to consider both options–no forgetting to check if you got an error!

Algebra

- Suppose `Bob = {1,2,3}` and `Frog = {☺,☹}`,
  - How many `Pair<Bob, Frog>` values are there?
  - How many `Either<Bob, Frog>` values are there?

Algebra

- Suppose `Bob = {1,2,3}` and `Frog = {☺,☹}`,
  - How many `Pair<Bob, Frog>` values are there?
  - How many `Either<Bob, Frog>` values are there?
- If there are x values of type `A` and y values of type `B`,
  - `Pair<A,B>` has $x * y$ values, so it is a *product*, `A × B`
  - `Either<A,B>` has $x + y$ values, so it is a *sum*, `A + B`
- Based on these ideas, you can make an algebra of types!