Proof Reconstruction: Translating Proofs

Progress Presentation

Alejandro Gómez-Londoño

Advisor - Andrés Sicard-Ramírez

EAFIT University

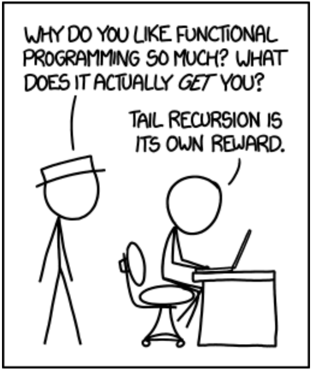
October 2, 2015

Introduction

Proof assistant

ATP

Functional Programming [1]



---

[1] Randall Munroe, https://xkcd.com/1270

Functional Programming

Features

- First class functions.	(They have weird function stuff)
- Higher-order functions.	(They are complicated)
- Purity.	(There are no variables)
- No arbitrary side effects.	(Also no easy way to print)
- Recursion.	(No loops!)
- Pattern matching.	(This is actually a good thing)
- Type inference.	(Type errors everywhere)

Functional Programming

Expressiveness

Functional Programming

Expressiveness

C++:

```
int mult(int a, int b,int ab, int aX,int bX,int abX){
  int sum;
  for (i=0; i<aX; i++){
    for (j=0; j<bX; j++){
      sum=0;
      for (k=0; k<abX; k++){
        sum += a[i][k] * b[k][j];
      }
      ab[i][j]=sum;
    }
  }
}
```

Functional Programming

Expressiveness

Matlab:

```
A * B
```

Haskell:

```
mmult :: Num a => [[a]] -> [[a]] -> [[a]]
mmult a b = do ar <- a
               bc <- transpose b
               sum $ zipWith (*) ar bc
```

Functional Programming

Expressiveness

```
factorial :: Int -> Int
factorial n = foldl (*) 1 [1..n]

fib = [Int]
fib = 0:1:zipWith (+) fib (tail fib)

main :: IO ()
main = scotty 80 $ do
  get "/" $ html "<h1> Hello world </h1>"
  get "/:name" $ do
    name <- param "name"
    html $ mconcat ["<h1>Hello, ", name, "</h1>"]
```

Functional Programming

Type systems

> *"In programming languages, a type system is a collection of rules that assign a property called type to various constructs a computer program consists of."* [2]

---

[2] Wikipedia contributors, *Type system*. September 28, 2015.

Functional Programming

Type systems

```
Int a
a :: Int

int[]
[Int]

(String) "SOME BAD JOKE"
"SOME OTHER BAD JOKE" :: String

Int fact(Int a, Int b)
fact :: Int -> Int -> Int
```

Functional Programming

Type systems (Soundness)

C++

```cpp
int foo(int a, int b){
...
}
```

Haskell:

```haskell
foo :: Int -> Int -> Int
foo ...
```

Functional Programming

Type systems (Soundness)

C++:

```
int foo(int a, int b){
  System.Weapons.Launch.Nuke()
  return a + b;
}
```

Haskell:

```
foo :: Int -> Int -> Int
foo a b = a + b --No bomb! see!
```

Functional Programming [3]

Type systems + Expressiveness (+ some more complicated stuff)

---

[3] Randall Munroe, https://xkcd.com/1312

Functional Programming

Type systems + Expressiveness (+ some more complicated stuff)

- Programs are in general safer.
- Less lines of code more work done!
- Reusable code.
- Quick prototyping.
- Easier parallelism.

Dependently typed functional programming languages

That name is really long! that should mean something bad is about to happen!

Wat?

Dependently typed functional programming languages

"Dependent types allow types to be predicated on values, meaning that some aspects of a program's behaviour can be specified precisely in the type." [4]

```
xs : List Bool
[]
[true,false]

xs' : Vect Bool 2
[]            -- no!
[true]        -- no!
[true,false] -- yes!
```

---

[4] Idris contributors, http://www.idris-lang.org. September 28 2015.

Dependently typed functional programming languages

Haskell:

```
tail :: [a] -> [a]
tail []     = []
tail (x:xs) = xs

head :: [a] -> a
-- head [] = ??
head (x:xs) = x
```

Agda:

```
tail :   ∀ {A} → List A → List A
tail []     = []
tail (x:xs) = xs

head : ∀ {A} → List A → A
-- head []  = ??
head (x:xs) = x
```

Dependently typed functional programming languages

Curry–Howard correspondence

> *"The Curry–Howard isomorphism, tells us that in order to prove any mathematical theorem, all we have to do is construct a certain type which reflects the nature of that theorem, then find a value that has that type."* [5]

Proofs as programs!

---

[5] Wikibooks contributors, *Haskell/The Curry–Howard isomorphism*. September 28 2015.

Dependently typed functional programming languages

Curry–Howard correspondence

Hand written:

$$\frac{x \qquad y \qquad x \wedge y \Rightarrow z}{z}$$

Agda:

```
proof : ∀ { X Y Z } → X → Y → ( X ∧ Y → Z) → Z
```

Dependently typed functional programming languages

Curry–Howard correspondence

Hand written proof:

$$\dfrac{\dfrac{x \qquad y}{x \wedge y} \qquad x \wedge y \Rightarrow z}{z}$$

Agda proof:

```
proof : ∀ { X Y Z } → X → Y → ( X ∧ Y → Z) → Z
proof x y f = f ( ∧ -intro x y )
```

Dependently typed functional programming languages

Languages as proof assistants

> *"proof assistants or interactive theorem provers are a software tools to assist with the development of formal proofs."* [6]

---

[6] Wikipedia contributors, *Proof assistant*. September 28 2015.

Automated reasoning

> *"Deals with the development of computer programs that show that some statement (the conjecture) is a logical consequence of a set of statements (the axioms and hypotheses)."* [7]

---

[7] Geoff Sutcliffe's, *What is Automated Theorem Proving?*. September 28 2015.

Automated reasoning
ATPs

Hand written proof:

$$\dfrac{\dfrac{x \qquad y}{x \wedge y} \qquad x \wedge y \Rightarrow z}{z}$$

TPTP problem:

```
fof(a_0,axiom,x).
fof(a_1,axiom,y).
fof(a_2,axiom, ((x & y) => z)).
fof(c_0,conjecture, z).
```

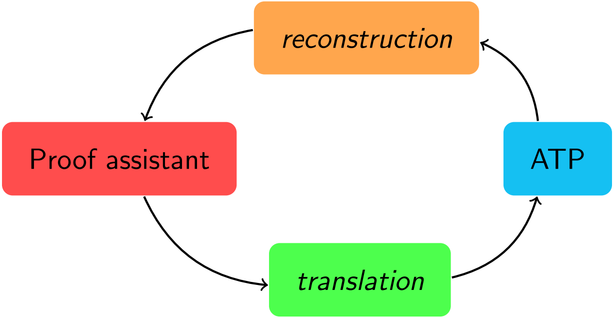
Automated reasoning
ATPs

Proof:

```
fof(a1, axiom, (x)).
fof(a2, axiom, (y)).
fof(a3, axiom, ((x & y) => z)).
fof(a4, conjecture, (z)).
fof(subgoal_0, plain, (z), inference(strip, [], [a4])).
fof(negate_0_0, plain, (~ z), inference(negate, [], [subgoal_0])).
fof(normalize_0_0, plain, (~ z),
    inference(canonicalize, [], [negate_0_0])).
fof(normalize_0_1, plain, (~ x | ~ y | z),
    inference(canonicalize, [], [a3])).
fof(normalize_0_2, plain, (x), inference(canonicalize, [], [a1])).
fof(normalize_0_3, plain, (y), inference(canonicalize, [], [a2])).
fof(normalize_0_4, plain, (z),
    inference(simplify, [],
              [normalize_0_1, normalize_0_2, normalize_0_3])).
fof(normalize_0_5, plain, ($false),
    inference(simplify, [], [normalize_0_0, normalize_0_4])).
cnf(refute_0_0, plain, ($false),
    inference(canonicalize, [], [normalize_0_5])).
```

Conclusion

Proof assistants + ATPs



Conclusion

Proof assistants + ATPs

Proof:

```
fof(a1, axiom, (x)).
fof(a2, axiom, (y)).
fof(a3, axiom, ((x & y) => z)).
fof(a4, conjecture, (z)).
fof(subgoal_0, plain, (z), inference(strip, [], [a4])).
fof(negate_0_0, plain, (~ z), inference(negate, [], [subgoal_0])).
fof(normalize_0_0, plain, (~ z),
    inference(canonicalize, [], [negate_0_0])).
fof(normalize_0_1, plain, (~ x | ~ y | z),
    inference(canonicalize, [], [a3])).
fof(normalize_0_2, plain, (x), inference(canonicalize, [], [a1])).
fof(normalize_0_3, plain, (y), inference(canonicalize, [], [a2])).
fof(normalize_0_4, plain, (z),
    inference(simplify, [],
              [normalize_0_1, normalize_0_2, normalize_0_3])).
fof(normalize_0_5, plain, ($false),
    inference(simplify, [], [normalize_0_0, normalize_0_4])).
cnf(refute_0_0, plain, ($false),
    inference(canonicalize, [], [normalize_0_5])).
```

Conclusion

Proof assistants + ATPs

Agda equivalent proof:

```
proof : ∀ {X Y Z} → X → Y → ( X ∧ Y → Z ) → Z
proof {_}{_}{Z} a1 a2 a3 = conclude0
  where
    norm03 = canon3 a2
    norm02 = canon2 a1
    norm01 = canon1 a3
    norm04 = simplify0 norm02 norm03 norm01
    negate0 : ¬ Z → ⊥
    negate0 negatate00 = refute00
      where
        norm00   = id negatate00
        norm05   = simplify1 norm00 norm04
        refute00 = canon4 norm05
    conclude0 = proofByContradiction negate0
```

Conclusion

Proof assistants + ATPs

Agda proof:

```
proof : ∀ { X Y Z } → X → Y → ( X ∧ Y → Z) → Z
proof x y f = f ( ∧-intro x y )
```