Proving Non-Deterministic Computations in Agda

S. Libby

In collaboration with S. Antoy, M. Hanus

Portland State University

September 13, 2016

Why Dependant Types?

- prove correctness of programs
- `(a b : Int)` $\rightarrow$ `a + b` $\equiv$ `b + a`
- Curry Howard Isomorphism
  - true if we can write a function with this type
- all programs must terminate
  - `loop = loop` has any type

Non-Determinism in Agda

- Dependant types are hard
    - Common problem: prove `a + b` $\equiv$ `b + a`
- some problems are easier to state with non-determinism
- middle ground between proof assistants and SMT solvers

Functional Logic Programming

- FLP combines functional and logic programming
- langauge: Curry
- non-determinism introduced with `?` operator
- `coin = 0 ? 1`
  - `coin` could have the value of either `0` or `1`
- programs are evaluated via Narrowing

Dependant Types

- Types depend of values
- In fact there is no distinction between types/values/kinds
- A type is anything whose type is **`Set`**.
  `Tree a : Set`
- A value is anything whose type is a type.
  `leaf 1 : Tree Int`

Two Types of Non-Determinism

- Set of Values
- Planned Choices

Set of Values

- Create a tree of possible values
- **`data ND a : Set where`**
- **`Val a : ND A`**
- **`a ?? b : ND A`** $\rightarrow$ **`ND A`** $\rightarrow$ **`ND A`**
  - analog in Curry: **`? :: a`** $\rightarrow$ **`a`** $\rightarrow$ **`a`**

Set of Values

- `mapND : {A B : Set} → (A → B) → ND A → ND B`
  - apply deterministic function to non-deterministic argument
  - also known as a functor
- `with-nd-arg : {A B : Set} → (A → ND B) → ND A → ND B`
  - apply non-deterministic function to non-deterministic argument
  - separates non-determinism in functions and arguments

Example: Sorted List

- How can we use non-determinism in Agda?
- How about proving something non-deterministic?
- Condition for Sorting:
  `sort xs` $\in$ **`permute xs`**

Example: Sorted List

- Curry insert:

```
ndinsert x [] = [x]
ndinsert x (y : ys) =
    (x : y : ys) ? (y : ndinsert x ys)
```

- Curry permutation:

```
ndperm [] = []
ndperm (x : xs) = insert x (ndperm xs)
```

Example: Sorted List

- Agda insert:

```
ndinsert x [] = Val [x]
ndinsert x (y :: ys) =
   (Val (x :: y :: ys)) ??
   (mapND ((_ :: _) y) ndinsert x ys)
```

- Agda permutation:

```
ndperm [] = Val []
ndperm (x :: xs) =
    with-nd-arg (ndinsert x) (ndperm xs)
```

Example: Sorted List

- ```
  insert x [] = [x]
  insert x (y :: ys) =
      if x < y
          then (x :: y :: ys)
          else (y :: insert x xs)
  ```
- ```
  sort [] = []
  sort (x :: xs) = insert x (sort xs)
  ```

Example: Sorted List

- Equality in Agda:

```
data _≡_ {A : Set} (x : A) :
         A → Set where
    refl : x ≡ x
```

Example: Sorted List

- non-deterministic equality:

```
data _ ∈ _ {A : Set} (x : A) :
        (y : ND A) → Set where
    ndrefl : x ∈ (Val x)
    left : (l : ND A) → (r : ND A)
        → x ∈ l → x ∈ (l ?? r)
    right : (l : ND A) → (r : ND A)
        → x ∈ r → x ∈ (l ?? r)
```

Example: Sorted List

- Example

```
coin : ND ℕ
coin = Val 0 ??  Val 1

0inCoin : 0 ∈ coin
0inCoin = left (Val 0) (Val 1) ndrefl
```

Example: Sorted List

- `insert=ndinsert : ...`
  $\rightarrow$ `(insert x xs)` $\in$ `(ndinsert x xs)`
  - Either we insert `x` at the front of `xs` or somewhere else in `xs`
  - This is the definition of `ndinsert`
- `sortTheorem : ...`
  $\rightarrow$ `sort xs` $\in$ `ndperm xs`
  - reduces to
    `insert=ndinsert xs (sort xs))`

Two Types of Non-Determinism

- Set of Values
- Planned Choices

Planned Choices

- Abstract over non-determinism
- Curry: `f = x ? y`
- Agda:
  ```
  f ch = if (choose ch) then x else y
  ```

Planned Choices

- `data Choice : Set`
  - implementation not important
- `choose : Choice` $\rightarrow \mathbb{B}$
  - select which branch to take
- `lchoice : Choice` $\rightarrow$ `Choice`
- `rchoice : Choice` $\rightarrow$ `Choice`
  - produce independent choices for non-deterministic subexpressions

Example: Double Coin

- deterministic function
  `double x = x + x`

- non-deterministic argument
  `coin ch = if choose ch then 0 else 1`

- predicate
  `even 0 = tt`
  `even (suc 0) = ff`
  `even (suc (suc n))` $\equiv$ `even n`

- Can we prove `even (double coin)` $\equiv$ `tt?`

Example: Double Coin

- We can prove that
  $\forall$ (x : $\mathbb{N}$) $\rightarrow$ even (double x) $\equiv$ tt
- ```
  even-double zero = refl
  even-double (suc x)
      rewrite +suc x x |
      even-double x = refl
  ```
  ```

Example: Double Coin

- finally we abstract over the non-determinism to get our theorem
- ```
  even-double-coin : ∀ (ch : Choice)
      → even (double (coin ch)) ≡ tt
  even-double-coin ch =
      even-double (coin ch)
  ```
- since `ch` is a parameter this must be true for every non-deterministic choice
- produces very short proofs
  ```

Summary

- dependant types prove properties about programs
- sometimes those properties are easier to express with non-determinism
- we demonstrated two methods for introducing non-determinism into Agda
  - Set of Values
  - Planned Choices

Future Work

- applying dependent types to Curry
  - How do we deal with partial functions?
  - How do we deal with laziness?
  - How do we deal with free variables?