Lazy, staged binary decision diagrams

Wouter Swierstra

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Programming with dependent types

Languages such as Agda, Coq, and Idris provide a single framework for *programming* and *proving*.

To prove a theorem corresponds to writing a program of with the correct type.

These proof terms can be complex and hard to write.

… but sometimes they can be computed!

Proof by reflection: example

```
data IsEven : Nat -> Set where
  Base : IsEven 0
  Step : IsEven n -> IsEven (Succ (Succ n))

easy : IsEven 4
easy = Step (Step Base)
```

Proof by reflection: example

```
data IsEven : Nat -> Set where
  Base : IsEven 0
  Step : IsEven n -> IsEven (Succ (Succ n))

easy : IsEven 4
easy = Step (Step Base)

harder : IsEven 1024
```

Proof by reflection: example

Instead, we can define a function that *computes* which numbers are even:

```
even? : Nat -> Bool
```

And a proof that this function is sound with respect to our previous notion of evenness:

```
sound : (n : Nat) -> even? n == True -> IsEven n
```

Universiteit Utrecht

Proof by reflection: example

Using this soundness result, it is trivial to prove large numbers are even:

```
harder : IsEven 1024
harder = sound 1024 refl
```

Of course, this example only works for *closed* terms but the technique extends to many different domains, including equations over rings, set membership, or any decidable property.

Application: Π-ware

Together with Joao Pizani Flor, we've been designing an embedded language for hardware description and verification, Π-ware.

```
data C : Nat -> Nat -> Set where
  ...
```

Using all of Agda's abstractions, we can define various combinators and *circuit generators*:

```
adder : (n : Nat) -> C (2 * n) (n + 1)
```

Verification: aims

Existing DSLs based on functional languages, such as Lava, use such combinators to *define* circuits…

… but verification is done by calling automated theorem provers on circuits of fixed size.

Can we not provide an inductive proof that the circuit generators are correct?

Verification: challenges

Unsurprisingly, such proofs involve lots of calculations over boolean expressions.

For small n, we can exhaustively test 2^n possible inputs.

And show that exhaustive testing implies proof.

Verification: challenges

Unsurprisingly, such proofs involve lots of calculations over boolean expressions.

For small n, we can exhaustively test 2^n possible inputs.

And show that exhaustive testing implies proof.

But this does not scale.

Smarter verification

The `industry standard' for efficient algorithms over such expressions is *binary decision diagrams* (BDDs).

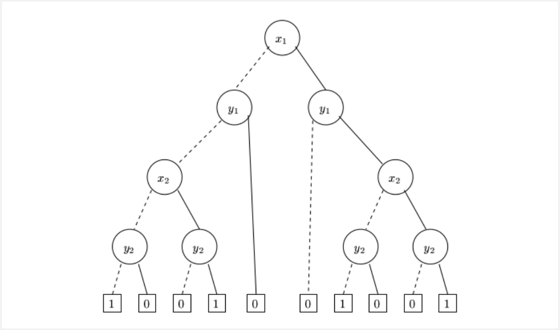
Can we use such BDDs to performe efficient proofs by reflection?

A first approximation...

We can begin by studying binary decision trees:

```
data Tree : Set where
  O : Tree
  I : Tree
  Node : Tree -> Var -> Tree -> Tree
```

Given such a tree and an assignment of values to variables, you can traverse the tree to evaluate the corresponding boolean expression.


x_1
y_1
y_1
x_2
x_2
y_2
y_2
y_2
y_2
1
0
0
1
0
0
1
0
0
1

Decision trees

It is easy enough to compute a decision tree corresponding to a boolean expression.

And prove they are equivalent.

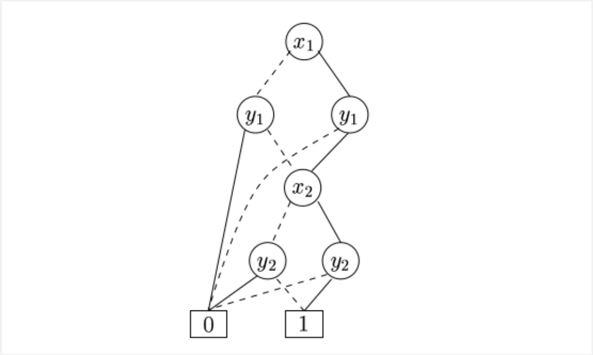
But what about real BDDs?

Reduced ordered binary decision diagrams

ROBDDs are much harder. The place the following additional constraints on binary decision trees:

- There is an ordering on variables that is respected by all paths from the root.
- No node has identical left and right subtrees
- Any two nodes with equal subtrees are equal

How can we represent such directed acyclic graphs in a functional setting?


x_1
y_1
y_1
x_2
y_2
y_2
0
1

Lazyness

Instead of working with pointers explicitly, we can rely on the host languages laziness to capture the desired sharing.

```
let t = Node (Node 1 y2 10) x2 (Node 0 y2 1)
in ... t ... t
```

Agda's metaprogramming framework lets you generate such programs from a boolean expression.

But to prove soundness, we need to reason about metaprograms…

Challenges & Caveats

Many algorithms computing with BDDs are inherently imperative, assigning unique names to nodes.

The usual algorithms to construct BDDs rely on hash tables to detect sharing, flattening out any structure.

This makes verification of the construction much harder.

We've started exploring this idea in Template Haskell, but don't have meaningful performance benchmarks yet.

The performance of Agda's compile-time evaluation and sharing are hard to predict. This may not be an improvement over brute-force testing.