Topics
- Representing Boolean functions with Binary Decision Diagrams
- Application to program verification

Verification Example

Do these functions produce identical results?
How could you find out?
How about exhaustive testing?

int abs(int x) {
    int mask = x >> 31;
    return (x ^ mask) + ~mask + 1;
}

int test_abs(int x) {
    return (x < 0) ? -x : x;
}

More Examples

int addXY(int x, int y) {
    return x + y;
}

int addYX(int x, int y) {
    return y + x;
}

int mulXY(int x, int y) {
    return x * y;
}

int mulYX(int x, int y) {
    return y * x;
}

How Can We Verify Programs?

Testing
- Exhaustive testing not generally feasible
- Currently, programs only tested over small fraction of possible cases

Formal Verification
- Mathematical “proof” that code is correct

Did Pythagoras show that $a^2 + b^2 = c^2$ by testing?
Bit-Level Program Verification

- View computer word as 32 separate bit values
- Each output becomes Boolean function of inputs

Extracting Boolean Representation

```c
int abs(int x) {
    int mask = x >> 31;
    return (x ^ mask) + ~mask + 1;
}
```

```c
int bitOr(int x, int y) {
    return ~(~x & ~y);
}
```

```c
int test_bitOr(int x, int y) {
    return x | y;
}
```

Do these functions produce identical results?

```c
int test_bitOr(int x, int y) {
    return x | y;
}
```

```c
int test_bitOr(int x, int y) {
    return x | y;
}
```

Straight-Line Evaluation

```
x
y
v1 = ~x
v2 = ~y
v3 = v1 & v2
v4 = ~v3
v5 = x | y
t = v4 == v5
```

Tabular Function Representation

- List every possible function value

Complexity

- Function with \( n \) variables

Algebraic Function Representation

```c
int abs(int x) {
    int mask = x >> 31;
    return (x ^ mask) + ~mask + 1;
}
```

```c
int bitOr(int x, int y) {
    return ~(~x & ~y);
}
```

```c
int test_bitOr(int x, int y) {
    return x | y;
}
```

Do these functions produce identical results?

```
x
y
v1 = ~x
v2 = ~y
v3 = v1 & v2
v4 = ~v3
v5 = x | y
t = v4 == v5
```

Complexity

- Representation
- Determining properties of function
  - E.g., deciding whether two expressions are equivalent
**Tree Representation**

Truth Table

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value

**Ordered Binary Decision Diagrams**

Initial Tree

Reduced Graph

$(x_1 + x_3) \cdot x_3$

Canonical representation of Boolean function

- Two functions equivalent if and only if graphs isomorphic
  - Can be tested in linear time
- Desirable property: simplest form is canonical.

**Example Functions**

**Constants**
- $0$: Unique unsatisfiable function
- $1$: Unique tautology

**Variable**
- Treat variable as function

**Typical Function**
- $(x_1 + x_2) \cdot x_4$
- No vertex labeled $x_3$
  - Independent of $x_3$
- Many subgraphs shared

**Odd Parity**

**More Complex Functions**

**Functions**
- Add 4-bit words $a$ and $b$
- Get 4-bit sum $S$
- Carry output bit $Cout$

**Shared Representation**
- Graph with multiple roots
- 31 nodes for 4-bit adder
- 571 nodes for 64-bit adder
- Linear growth!
### Symbolic Execution (3-bit word size)

#### x

- $v_1 = \bar{x}$
- $v_2 = \bar{y}$

#### y

- $v_3 = v_1 \land v_2$
- $v_4 = \bar{v}_3$
- $v_5 = x \lor y$
- $t = v_4 == v_5$

### Counterexample Generation

#### int bitOr(int x, int y)

```c
int bitOr(int x, int y) {
    return ~(~x & ~y);
}
```

#### int bitXor(int x, int y)

```c
int bitXor(int x, int y) {
    return x ^ y;
}
```

Find values of $x$ & $y$ for which these programs produce different results.
Performance: Good

```c
int addXY(int x, int y)
{
    return x+y;
}
```

<table>
<thead>
<tr>
<th>Word Size</th>
<th>Add-4</th>
<th>Multiplication-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>21</td>
<td>155</td>
</tr>
<tr>
<td>16</td>
<td>41</td>
<td>14560</td>
</tr>
</tbody>
</table>

---

Performance: Bad

```c
int addYX(int x, int y)
{
    return y+x;
}
```

---

Why Is Multiplication Slow?

Multiplication function intractable for BDDs
- Exponential growth, regardless of variable ordering

```
int factorK(int x, int y)
{
    int K = XXXX...X;
    int rangeOK =
        1 < x && x <= y;
    int factorOK =
        x*y == K;
    return
        !(rangeOK && factorOK);
}
```

---

What if Multiplication were Easy?

```c
int one(int x, int y)
{
    return 1;
}
```
Dealing with Conditionals

During Evaluation, Keep Track of:
- Current Context: Under what condition would code be evaluated
- Definedness (for each variable)
  - Has it been assigned a value

```plaintext
int abs(int x)
{
    int r;
    if (x < 0)
        r = -x;
    else
        r = x;
    return r;
}
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Defined</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t1 = x&lt;0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>v1 = -x</td>
<td>t1</td>
<td>0</td>
</tr>
<tr>
<td>r = v1</td>
<td>t1</td>
<td>t1?v1:0</td>
</tr>
<tr>
<td>r = x</td>
<td>!t1</td>
<td>1</td>
</tr>
<tr>
<td>v2 = r</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Dealing with Loops

Unrolled
```plaintext
int ilog2(unsigned x)
{
    int r = 31;
    if (x) {
        r++; x >>= 1;
    } else return r;
    if (x) {
        r++; x >>= 1;
    } else return r;
    . . .
    if (x) {
        r++; x >>= 1;
    } else return r;
    error();
}
```

Unroll
- Turn into bounded sequence of conditionals
  - Default limit = 33
- Signal runtime error if don't complete within limit

Evaluation

Strengths
- Provides 100% guarantee of correctness
- Performance very good for simple arithmetic functions

Weaknesses
- Important integer functions have exponential blowup
- Not practical for programs that build and operate on large data structures

Some History

Origins
- Lee 1959, Akers 1976
  - Idea of representing Boolean function as BDD
- Hopcroft, Fortune, Schmidt 1978
  - Recognized that ordered BDDs were like finite state machines
  - Polynomial algorithm for equivalence
- Bryant 1986
  - Proposed as useful data structure + efficient algorithms
- McMillan 1987
  - Developed symbolic model checking
  - Method for verifying complex sequential systems
- Bryant 1991
  - Proved that multiplication has exponential BDD
  - No matter how variables are ordered