Implementing a variable-sized bit-vector theory for KeY

Olivier Borne

May 18, 2009

Current state

```
public boolean some(ip1, ip2) {
   return ip1 & netmask == ip2 & netmask;
}
```

- Applications for embedded systems and network do use such representation (e.g. Sun SPOT)
- There is no support for bitwise operations in KeY

develop a bitwise arithmetic

Bitwise representations of integers

- Explicit data-type *bv*:

operations (&, |, projection) are defined on *bv*

- Implicit as normal integers

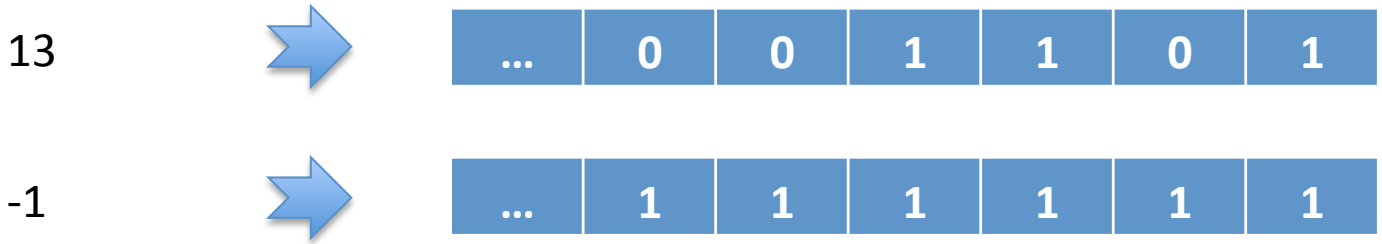
Operation are defined on *int*

Examples : 2 & 1, 4 | (2 & 0)

Bit-vector : fixed or variable size?

- Bounded model for integers (sign bit, overflow) :

- Unbounded : closer to natural numbers but need formalization



From now on : variable-sized bit-vectors

Projectors and functions

- bitProjector : $x{\downarrow}n$

$13{\downarrow}2$

…	0	0	0	1	1	0	1
	6	*5*	*4*	*3*	*2*	*1*	*0*

$$x{\downarrow}n = (x / 2^n) \% 2$$

- BitIntervalProjector : $x{\downarrow}(i, j)$

$13{\downarrow}(3,1)$

…	0	0	0	1	1	0	1
	6	*5*	*4*	*3*	*2*	*1*	*0*

$$x{\downarrow}(i,j) = (x/2^j) \% 2^{i-j+1}$$

Projectors and functions

- Composition : $a \otimes_n b$

Example : $13 \otimes_2 5$

…	0	1	1	0	1

$\otimes_2$

…	0	0	1	0	1

…	0	1	1	0	1

$\otimes_2$

…	0	0	1	0	1

…	0	1	1	0	1	0	1

$$13 \otimes_2 5 = 53$$

Bitwise functions

- Bitwise functions are defined with projectors

<u>*Example*</u> : \find((a & b) ↓ n))

- Exceptions are the shift-right and shift-left, with arithmetic definitions

  shift-right : a >> b := $a/2^b$

  shift-left : a << b := $a * 2^b$

Example

$$2 \,\&\, [(4|c) \,\&\, 2\,] = 2 \,\&\, c$$

$2 \,\&\, [2 \,\&\, (4|c)\,]$ (commute)

$= (2 \,\&\, 2) \,\&\, (4|c)$ (associate)

$= 2 \,\&\, (4|c)$ (identity)

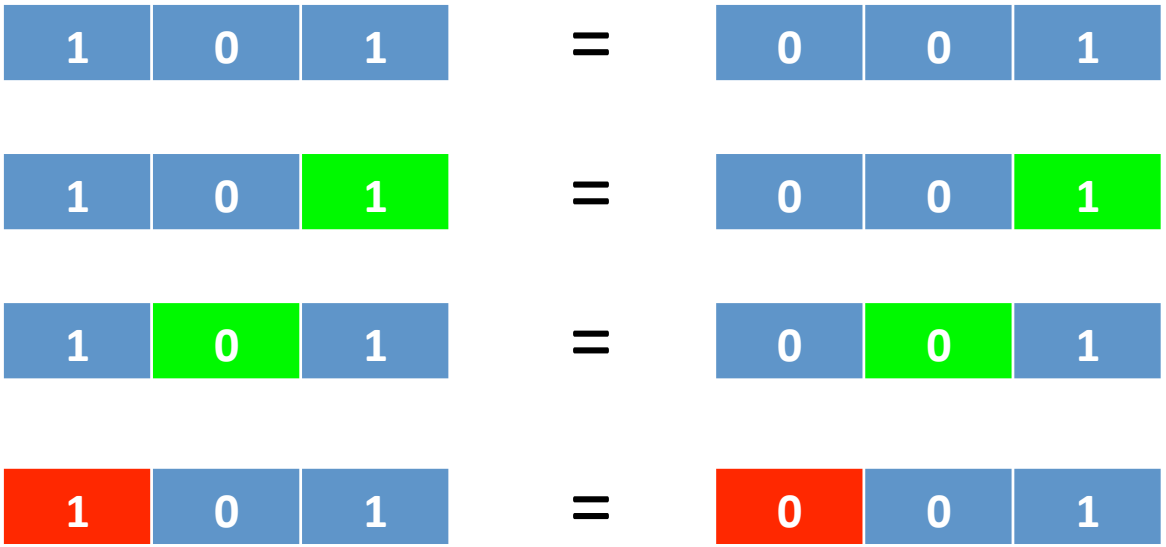
$= (2 \,\&\, 4) \mid (2 \,\&\, c)$ (distribute)

$= 0 \mid (2 \,\&\, c)$ (literal)

$= 2 \,\&\, c$ (neutral element)

Bit-blasting

- Problematic : avoid the bit-blasting



- A solution is to use large chunks of bit-vectors

Bit-vector chunks algorithm

(based on a paper from David Cyrluk et al.)

$x \otimes_6 y \downarrow (0, 0) \otimes_5 z \downarrow (4, 4) \otimes_4 0 = x \otimes_6 z \otimes_1 x \downarrow (2, 2)$

Normal form with bit-vector chunks of the same size :

$x \otimes_6 y \downarrow (0, 0) \otimes_5 z \downarrow (4, 4) \otimes_4 \quad 0 \quad \otimes_1 0 =$

$x \otimes_6 z \downarrow (4, 4) \otimes_5 z \downarrow (3, 3) \otimes_4 z \downarrow (2, 0) \otimes_1 x \downarrow (2, 2)$

Solved by solving the conjunction of equations

Bit-vector chunks algorithm

Solve each equation :

csolve(x = x) = $\varnothing$

csolve(y↓(0,0) = z↓(4,4)) = $\{y=r^{(1)}_{2} \otimes_1 c^{(1)}_{[1]},\ z=r^{(2)}_{6} \otimes_1 c^{(1)}_{[1]} \otimes_4 a^{(1)}_{[4]}\}$

csolve(z↓(4,4) = z↓(3,3)) = $\{z = r^{(2)}_{6} \otimes_1 b^{(1)}_{[1]} \otimes_1 b^{(1)}_{[1]} \otimes_1 a^{(2)}_{[3]}\}$

Equations are rearranged for constant propagations and chunk size adjustments

$r^{(2)}_{6}$	$c^{(1)}_{[1]}$	$a^{(1)}_{[1]}{}'$	$a^{(1)}_{[3]}{}''$
$r^{(2)}_{6}$	$b^{(1)}_{[1]}$	$b^{(1)}_{[1]}$	$a^{(2)}_{[3]}$
$r^{(2)}_{6}$	$a^{(4)}_{[1]}{}'$	$a^{(4)}_{[1]}{}''$	$0_{[3]}$

Algorithm in KeY

- Canonizer : rewrite taclets
- Rewrite rule n°1 : $t\downarrow(i, j)\downarrow(k, l) = t\downarrow(k+j, l+j)$
- Rewrite rule n°2 : $t\downarrow(i, j) \otimes_{j-k} t\downarrow(j-1, k) = t\downarrow(i, k)$

- Taclet for rule n°1 :

```
\schemaVar int t, i, j, k, l;
\find(t ↓ (i, j) ↓ (k, l))
\replacewith(t ↓ (k+j, l+j))
```

Algorithm in KeY

- Bit-vector chunk splitting : combination applying equations and splitting taclet

- Taclet :

```
bitvector_split_composition {
  \find(var ↓ (high, low))
  \sameUpdateLevel
  \add(==> z >= low & z <= high)
    \replacewith( var ↓ (high, z)
                ⊗z-low var ↓ (z, low))
};
```

Mapping bitwise functions to composition

- There is still a gap between a bitwise formula and such equations to avoid bit-blasting
- Example for closing the gap : chunk decomposition using preconditions

There must be some knowledge about members of the equation to make a decomposition

- If/else approach (described in paper)

Summary & Future Work

- The extension of the fixe-sized algorithm seems very promising to cope with bitwise operations and can used into KeY

- Future work : concentrate on the automation of the algorithm and closing the gap using this chunk decomposition approach