Decision Procedures

An Algorithmic Point of View

Equalities and Uninterpreted Functions

D. Kroening O. Strichman

ETH/Technion

Version 1.0, 2007

Part III

Equalities and Uninterpreted Functions

Outline

1. Introduction to Equality Logic
   - Definition, complexity
2. Reducing uninterpreted functions to Equality Logic
3. Using uninterpreted functions in proofs
4. Simplifications

- A Boolean combination of Equalities and Propositions

$$x_1 = x_2 \wedge (x_2 = x_3 \vee \neg((x_1 = x_3) \wedge b \wedge x_1 = 2))$$

- We always push negations inside (NNF):

$$x_1 = x_2 \wedge (x_2 = x_3 \vee ((x_1 \neq x_3) \wedge \neg b \wedge x_1 \neq 2))$$

$$
\begin{array}{lcl}
\mathit{formula} & : & \mathit{formula} \lor \mathit{formula} \\
 & | & \neg \mathit{formula} \\
 & | & \mathit{atom} \\
 & & \\
\mathit{atom} & : & \mathit{term\text{-}variable} = \mathit{term\text{-}variable} \\
 & | & \mathit{term\text{-}variable} = \mathit{constant} \\
 & | & \mathit{Boolean\text{-}variable}
\end{array}
$$

- The *term-variables* are defined over some (possible infinite) domain. The constants are from the same domain.
- The set of Boolean variables is always separate from the set of term variables

- Allows more natural description of systems, although technically it is as expressible as Propositional Logic.
- Obviously NP-hard.
- In fact, it is in NP, and hence NP-complete, for reasons we shall see later.

Equality logic with uninterpreted functions

$$\begin{array}{lcl}
\textit{formula} & : & \textit{formula} \vee \textit{formula} \\
 & | & \neg\textit{formula} \\
 & | & \textit{atom} \\
 & & \\
\textit{atom} & : & \textit{term} = \textit{term} \\
 & | & \textit{Boolean-variable} \\
 & & \\
\textit{term} & : & \textit{term-variable} \\
 & | & \textit{function}\,(\,\text{list of } \textit{terms}\,)
\end{array}$$

The *term-variables* are defined over some (possible infinite) domain.
Constants are functions with an empty list of terms.

- Every function is a mapping from a domain to a range.
- Example: the '+' function over the naturals $\mathbb{N}$ is a mapping from $\langle \mathbb{N} \times \mathbb{N} \rangle$ to $\mathbb{N}$.

- Suppose we replace '$+$' by an uninterpreted binary function $f(a, b)$
- Example:

$$x_1 + x_2 = x_3 + x_4 \quad \text{is replaced by} \quad f(x_1, x_2) = f(x_3, x_4)$$

- We lost the 'semantics' of '$+$', as f can represent any binary function.

- 'Loosing the semantics' means that f is not restricted by any axioms or rules of inference.
- But f is still a function!

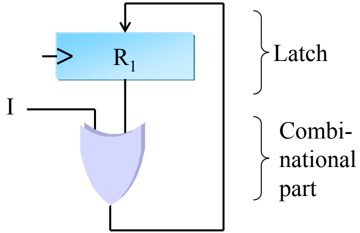
- The most general axiom for any function is functional consistency.
- Example: if $x = y$, then $f(x) = f(y)$ for any function f.

- Functional consistency axiom schema:

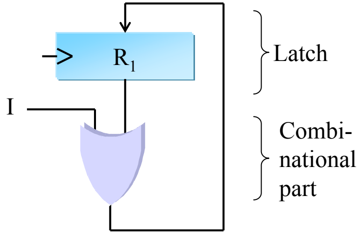
$$x_1 = x'_1 \wedge \ldots \wedge x_n = x'_n \quad \Longrightarrow \quad f(x_1, \ldots, x_n) = f(x'_1, \ldots, x'_n)$$

- Sometimes, functional consistency is all that is needed for a proof.

- Circuits consist of combinational gates and latches (registers)



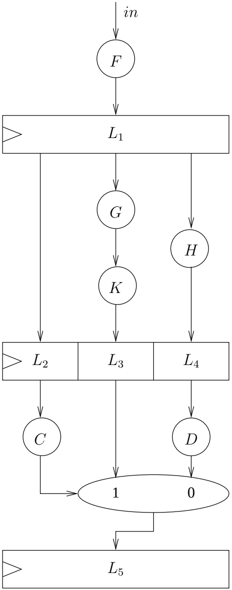
- Circuits consist of combinational gates and latches (registers)



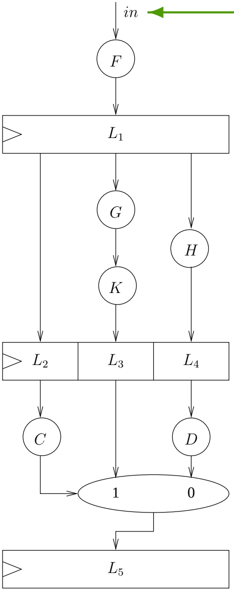
- The combinational gates can be modeled using functions
- The latches can be modeled with variables

$$f(x,y) := x \vee y$$
$$R_1' = f(R_1, I)$$

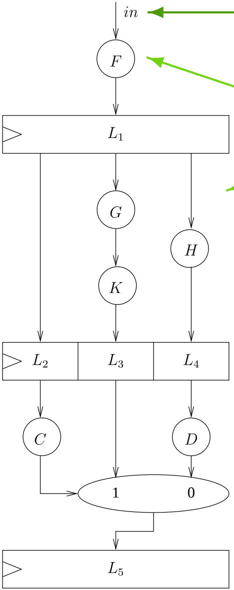
Example: Circuit Transformations



Example: Circuit Transformations

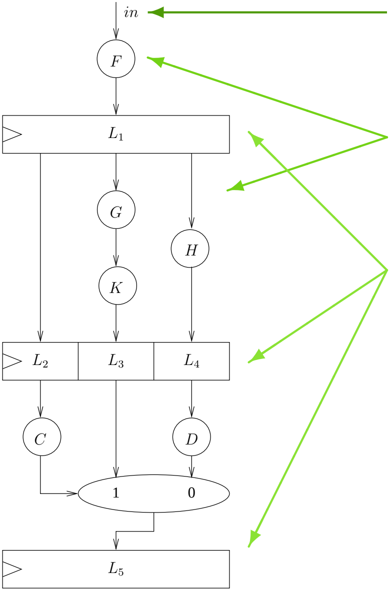


in: a primary input of the circuit



in: a primary input of the circuit

F, G, H, K, D: some functions over bit-vectors

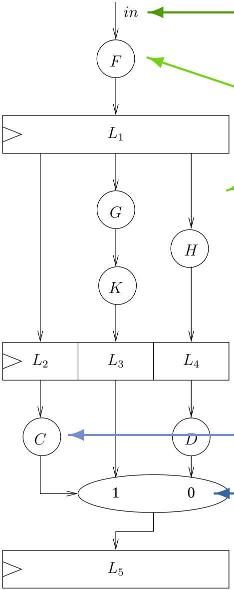


in: a primary input of the circuit

F, G, H, K, D: some functions over bit-vectors

$L_1, \ldots, L_5$: latches (registers)

Example: Circuit Transformations



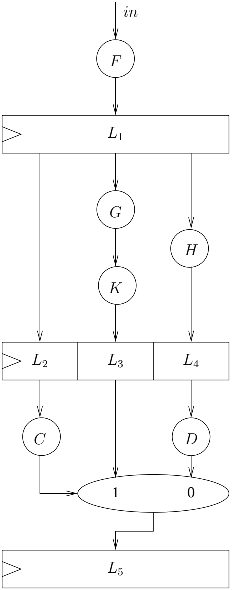
in: a primary input of the circuit

F, G, H, K, D: some functions over bit-vectors

$L_1, \ldots, L_5$: latches (registers)

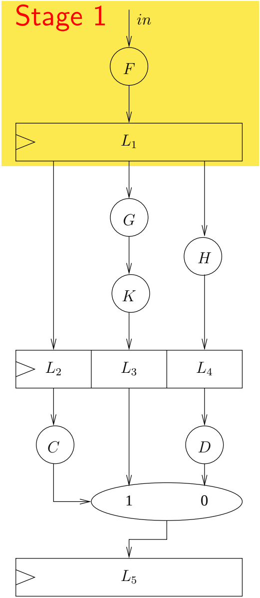
C: a predicate over bit-vectors

a multiplexer (case-split)



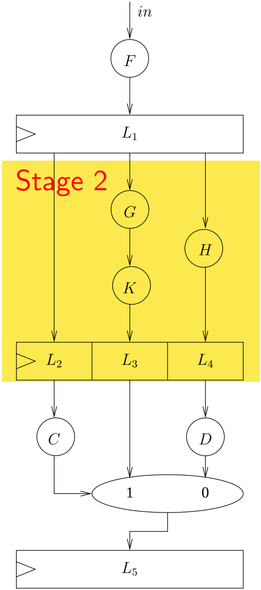
- A pipeline processes data in *stages*
- Data is processed in parallel – as in an assembly line
- Formal model:

$$
\begin{aligned}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2)\,?\,L_3 : l(L_4)
\end{aligned}
$$



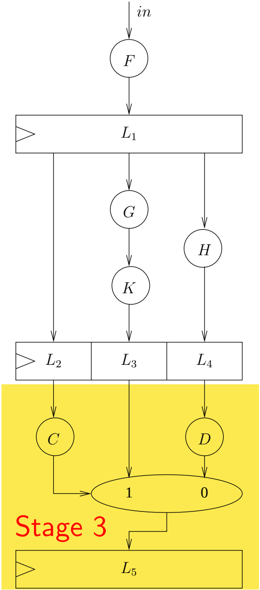
- A pipeline processes data in *stages*
- Data is processed in parallel – as in an assembly line
- Formal model:

$$\begin{array}{rcl} L_1 & = & f(I) \\ L_2 & = & L_1 \\ L_3 & = & k(g(L_1)) \\ L_4 & = & h(L_1) \\ L_5 & = & c(L_2)\,?\,L_3\ :\ l(L_4) \end{array}$$



- A pipeline processes data in *stages*
- Data is processed in parallel – as in an assembly line
- Formal model:

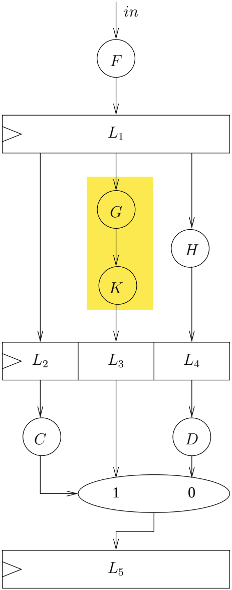
$$
\begin{aligned}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2) \, ? \, L_3 \; : \; l(L_4)
\end{aligned}
$$



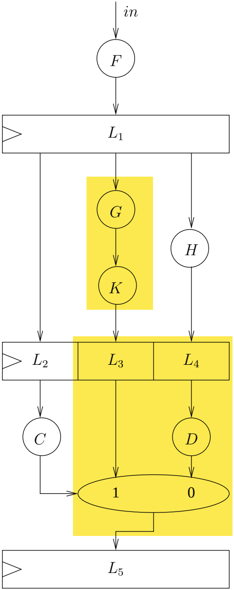
- A pipeline processes data in *stages*
- Data is processed in parallel – as in an assembly line
- Formal model:

$$
\begin{aligned}
L_1 &= f(I)\\
L_2 &= L_1\\
L_3 &= k(g(L_1))\\
L_4 &= h(L_1)\\
L_5 &= c(L_2)\,?\,L_3 \,:\, l(L_4)
\end{aligned}
$$

Example: Circuit Transformations



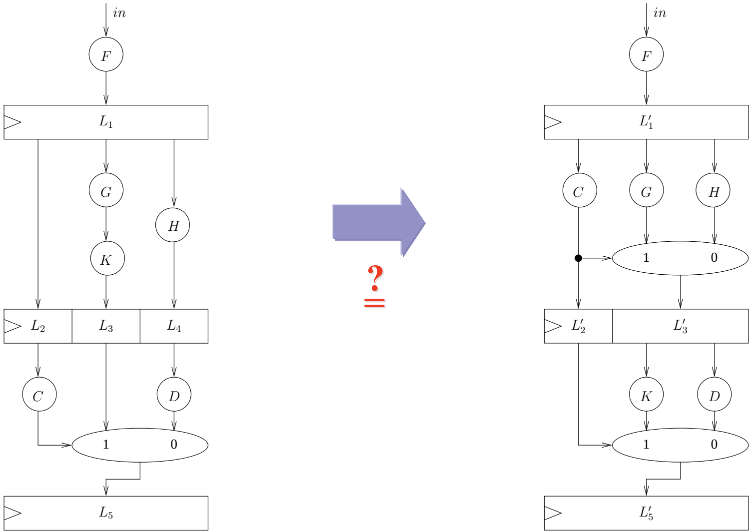
- The maximum clock frequency depends on the longest path between two latches
- Note that the output of g is used as input to k
- We want to speed up the design by postponing k to the third stage



- The maximum clock frequency depends on the longest path between two latches
- Note that the output of g is used as input to k
- We want to speed up the design by postponing k to the third stage
- Also note that the circuit only uses one of L_3 or L_4, never both

$\Rightarrow$ We can remove one of the latches

Example: Circuit Transformations



$$
\begin{aligned}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2)\,?\,L_3 \;:\; l(L_4)
\end{aligned}
$$

$$
\begin{aligned}
L'_1 &= f(I) \\
L'_2 &= c(L'_1) \\
L'_3 &= c(L'_1)\,?\,g(L'_1) \;:\; h(L'_1) \\
L'_5 &= L'_2\,?\,k(L'_3) \;:\; l(L'_3)
\end{aligned}
$$

$$
L_5 \stackrel{?}{=} L'_5
$$

$$\begin{aligned} L_1 &= f(I) \\ L_2 &= L_1 \\ L_3 &= k(g(L_1)) \\ L_4 &= h(L_1) \\ L_5 &= c(L_2)\,?\,L_3 \,:\, l(L_4) \end{aligned}$$

$$\begin{aligned} L'_1 &= f(I) \\ L'_2 &= c(L'_1) \\ L'_3 &= c(L'_1)\,?\,g(L'_1) \,:\, h(L'_1) \\ L'_5 &= L'_2\,?\,k(L'_3) \,:\, l(L'_3) \end{aligned}$$

$$L_5 \stackrel{?}{=} L'_5$$

- Equivalence in this case holds regardless of the actual functions
- Conclusion: can be decided using *Equality Logic and Uninterpreted Functions*

- Given: a formula φ^{UF} with uninterpreted functions
- For each function in φ^{UF}:

  1. Number function instances (from the inside out) $\longrightarrow$ $F_2(\,F_1(x)\,) = 0$

- Given: a formula φ^{UF} with uninterpreted functions
- For each function in φ^{UF}:

1. Number function instances (from the inside out) $\longrightarrow$ $\underbrace{F_2(\overbrace{F_1(x)}^{f_1})}_{f_2} = 0$

2. Replace each function instance with a new variable $\longrightarrow$ $f_2 = 0$

- Given: a formula φ^{UF} with uninterpreted functions
- For each function in φ^{UF}:

1. Number function instances (from the inside out) $\longrightarrow$ $\underbrace{F_2(\overbrace{F_1(x)}^{f_1})}_{f_2} = 0$
2. Replace each function instance with a new variable $\longrightarrow$ $f_2 = 0$
3. Add functional consistency constraint to φ^{UF} for every pair of instances of the same function. $\longrightarrow$ $((x = f_1) \longrightarrow (f_2 = f_1)) \longrightarrow f_2 = 0$

Suppose we want to check

$$x_1 \neq x_2 \vee F(x_1) = F(x_2) \vee F(x_1) \neq F(x_3)$$

for validity.

1. First number the function instances:

$$x_1 \neq x_2 \vee F_1(x_1) = F_2(x_2) \vee F_1(x_1) \neq F_3(x_3)$$

Ackermann's reduction: Example

Suppose we want to check

$$x_1 \neq x_2 \lor F(x_1) = F(x_2) \lor F(x_1) \neq F(x_3)$$

for validity.

1. First number the function instances:

$$x_1 \neq x_2 \lor F_1(x_1) = F_2(x_2) \lor F_1(x_1) \neq F_3(x_3)$$

2. Replace each function with a new variable:

$$x_1 \neq x_2 \lor f_1 = f_2 \lor f_1 \neq f_3$$

Suppose we want to check

$$x_1 \neq x_2 \vee F(x_1) = F(x_2) \vee F(x_1) \neq F(x_3)$$

for validity.

1. First number the function instances:

$$x_1 \neq x_2 \vee F_1(x_1) = F_2(x_2) \vee F_1(x_1) \neq F_3(x_3)$$

2. Replace each function with a new variable:

$$x_1 \neq x_2 \vee f_1 = f_2 \vee f_1 \neq f_3$$

3. Add functional consistency constraints:

$$\left(\begin{array}{ll} (x_1 = x_2 \rightarrow f_1 = f_2) & \wedge \\ (x_1 = x_3 \rightarrow f_1 = f_3) & \wedge \\ (x_2 = x_3 \rightarrow f_2 = f_3) & \end{array} \right) \rightarrow$$

$$((x_1 \neq x_2) \vee (f_1 = f_2) \vee (f_1 \neq f_3))$$

Transforming UFs to Equality Logic using Bryant's reduction

- Given: a formula φ^{UF} with uninterpreted functions
- For each function in φ^{UF}:

  1. Number function instances (from the inside out) $\longrightarrow$ $F_1(a) = F_2(b)$

- Given: a formula φ^{UF} with uninterpreted functions
- For each function in φ^{UF}:

1. Number function instances (from the inside out) $\longrightarrow$ $F_1(a) = F_2(b)$
2. Replace each function instance F_i with an expression F_i^* $\longrightarrow$ $F_1^* = F_2^*$

- Given: a formula φ^{UF} with uninterpreted functions
- For each function in φ^{UF}:

1. Number function instances (from the inside out) $\longrightarrow F_1(a) = F_2(b)$
2. Replace each function instance F_i with an expression F_i^* $\longrightarrow F_1^* = F_2^*$

$$F_i^* := \begin{pmatrix} \text{case} & x_1 = x_i & : f_1 \\ & x_2 = x_i & : f_2 \\ & \vdots & \\ & x_{i-1} = x_i & : f_{i-1} \\ & \text{true} & : f_i \end{pmatrix} \longrightarrow f_1 = \begin{pmatrix} \text{case} & a = b: f_1 \\ & \text{true} \ : f_2 \end{pmatrix}$$

- Original formula:

$$a = b \rightarrow F(G(a) = F(G(b))$$

Example of Bryant's reduction

- Original formula:

$$a = b \rightarrow F(G(a) = F(G(b))$$

- Number the instances:

$$a = b \rightarrow F_1(G_1(a) = F_2(G_2(b))$$

- Original formula:

$$a = b \rightarrow F(G(a) = F(G(b))$$

- Number the instances:

$$a = b \rightarrow F_1(G_1(a) = F_2(G_2(b))$$

- Replace each function application with an expression:

$$a = b \rightarrow F_1^* = F_2^*$$

where

$$\begin{array}{rcl} F_1^* & = & f_1 \\ F_2^* & = & \left(\begin{array}{lll} \text{case} & G_1^* = G_2^* & : f_1 \\ & \text{true} & : f_2 \end{array} \right) \end{array}$$

$$\begin{array}{rcl} G_1^* & = & g_1 \\ G_2^* & = & \left(\begin{array}{lll} \text{case} & a = b & : g_1 \\ & \text{true} & : g_2 \end{array} \right) \end{array}$$

- Uninterpreted functions give us the ability to represent an *abstract* view of functions.
- It over-approximates the concrete system.

  $1 + 1 = 1$ is a contradiction

  But

  $F(1, 1) = 1$ is satisfiable!

- Uninterpreted functions give us the ability to represent an *abstract* view of functions.
- It over-approximates the concrete system.

  $1 + 1 = 1$ is a contradiction

  But

  $F(1, 1) = 1$ is satisfiable!
- Conclusion: unless we are careful, we can give wrong answers, and this way, loose soundness.

- In general, a sound but incomplete method is more useful than an unsound but complete method.

- A sound but incomplete algorithm for deciding a formula with uninterpreted functions φ^{UF}:
  1. Transform it into Equality Logic formula φ^{E}
  2. If φ^{E} is unsatisfiable, return 'Unsatisfiable'
  3. Else return 'Don't know'

- Question #1: is this useful?

Using uninterpreted functions in proofs

- Question #1: is this useful?
- Question #2: can it be made complete in some cases?

- Question #1: is this useful?
- Question #2: can it be made complete in some cases?

- When the abstract view is sufficient for the proof, it enables (or at least simplifies) a mechanical proof.

- Question #1: is this useful?
- Question #2: can it be made complete in some cases?

- When the abstract view is sufficient for the proof, it enables (or at least simplifies) a mechanical proof.
- So when is the abstract view sufficient?

- (common) Proving equivalence between:
  - Two versions of a hardware design (one with and one without a pipeline)
  - Source and target of a compiler ("Translation Validation")

- (common) Proving equivalence between:
  - Two versions of a hardware design (one with and one without a pipeline)
  - Source and target of a compiler ("Translation Validation")
- (rare) Proving properties that do not rely on the exact functionality of some of the functions

- Assume the source program has the statement

$$z = (x_1 + y_1) \cdot (x_2 + y_2);$$

which the compiler turned into:

$$\begin{aligned} u_1 &= x_1 + y_1; \\ u_2 &= x_2 + y_2; \\ z &= u_1 \cdot u_2; \end{aligned}$$

- Assume the source program has the statement

$$z = (x_1 + y_1) \cdot (x_2 + y_2);$$

which the compiler turned into:

$$\begin{aligned} u_1 &= x_1 + y_1; \\ u_2 &= x_2 + y_2; \\ z &= u_1 \cdot u_2; \end{aligned}$$

- We need to prove that:

$$\begin{array}{ll} & (u_1 = x_1 + y_1 \quad \wedge \quad u_2 = x_2 + y_2 \quad \wedge \quad z = u_1 \cdot u_2) \\ \longrightarrow & (z = (x_1 + y_1) \cdot (x_2 + y_2)) \end{array}$$

- Claim: φ^{UF} is valid

- We will prove this by reducing it to an Equality Logic formula

$$\varphi^{E} = \begin{array}{c} \left(\begin{array}{cccc} (x_1 = x_2 \wedge y_1 = y_2 & \longrightarrow & f_1 = f_2) & \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 & \longrightarrow & g_1 = g_2) & \end{array} \right) \longrightarrow \\ ((u_1 = f_1 \ \wedge \ u_2 = f_2 \ \wedge \ z = g_1) \quad \longrightarrow \quad z = g_2) \end{array}$$

Uninterpreted functions: usability

- Good: each function on the left can be mapped to a function on the right with equivalent arguments

- Good: each function on the left can be mapped to a function on the right with equivalent arguments

- Bad: almost all other cases

- Example:

Left	Right
$x + x$	$2x$

- This is easy to prove:

$$(x_1 = x_2 \wedge y_1 = y_2) \longrightarrow (x_1 + y_1 = x_2 + y_2)$$

Uninterpreted functions: usability

- This is easy to prove:

$$(x_1 = x_2 \wedge y_1 = y_2) \longrightarrow (x_1 + y_1 = x_2 + y_2)$$

- This requires commutativity:

$$(x_1 = x_2 \wedge y_1 = y_2) \longrightarrow (x_1 + y_1 = y_2 + x_2)$$

- This is easy to prove:

$$(x_1 = x_2 \wedge y_1 = y_2) \longrightarrow (x_1 + y_1 = x_2 + y_2)$$

- This requires commutativity:

$$(x_1 = x_2 \wedge y_1 = y_2) \longrightarrow (x_1 + y_1 = y_2 + x_2)$$

- Fix by adding:

$$(x_1 + y_1 = y_1 + x_1) \wedge (x_2 + y_2 = y_2 + x_2)$$

- This is easy to prove:

$$(x_1 = x_2 \wedge y_1 = y_2) \longrightarrow (x_1 + y_1 = x_2 + y_2)$$

- This requires commutativity:

$$(x_1 = x_2 \wedge y_1 = y_2) \longrightarrow (x_1 + y_1 = y_2 + x_2)$$

- Fix by adding:

$$(x_1 + y_1 = y_1 + x_1) \wedge (x_2 + y_2 = y_2 + x_2)$$

- What about *other cases*?
  Use more rewriting rules!

```
int power3(int in) {
    out = in;

    for(i=0; i<2; i++)
       out = out * in;

    return out;
}
```

```
int power3_new(int in) {
    out = (in*in)*in;
    return out;
}
```

- These two functions return the same value regardless if it is '*' or any other function.
- *Conclusion*: we can prove equivalence by replacing '*' with an uninterpreted function

- But first we need to know how to turn programs into equations.
- There are several options – we will see static single assignment for bounded programs.

- $\longrightarrow$ see compiler class
- Idea: Rename variables such that each variable is assigned exactly once

Example:

```
x=x+y;
x=x*2;
a[i]=100;
```

$\Rightarrow$

$x_1 = x_0 + y_0;$
$x_2 = x_1 * 2;$
$a_1[i_0] = 100;$

- $\longrightarrow$ see compiler class
- Idea: Rename variables such that each variable is assigned exactly once

Example:

```
x=x+y;
x=x*2;
a[i]=100;
```

$\Rightarrow$

x_1=x_0+y_0;
x_2=x_1*2;
a_1[i_0]=100;

- Read assignments as equalities
- Generate constraints by simply conjoining these equalities

Example:

x_1=x_0+y_0;
x_2=x_1*2;
a_1[i_0]=100;

$\Rightarrow$

$$\begin{array}{cc} x_1 = x_0 + y_0 & \wedge \\ x_2 = x_1 * 2 & \wedge \\ a_1[i_0] = 100 & \end{array}$$

What about `if`? Branches are handled using ϕ-nodes.

```
int main() {
  int x, y, z;

  y=8;

  if(x)
    y--;
  else
    y++;


  z=y+1;
}
```

What about `if`? Branches are handled using ϕ-nodes.

```
int main() {
  int x, y, z;

  y=8;

  if(x)
    y--;
  else
    y++;


  z=y+1;
}
```

```
int main() {
  int x, y, z;

  y1=8;

  if(x0)
    y2=y1-1;
  else
    y3=y1+1;

  y4=φ(y2, y3);

  z1=y4+1;
}
```

What about `if`? Branches are handled using ϕ-nodes.

```
int main() {
   int x, y, z;

   y=8;

   if(x)
     y--;
   else
     y++;


   z=y+1;
}
```

```
int main() {
   int x, y, z;

   y₁=8;

   if(x₀)
     y₂=y₁-1;
   else
     y₃=y₁+1;

   y₄=ϕ(y₂, y₃);

   z₁=y₄+1;
}
```

$$
\begin{aligned}
&y_1 = 8 && \wedge\\
&y_2 = y_1 - 1 && \wedge\\
&y_3 = y_1 + 1 && \wedge\\
&y_4 = \\
&(x_0 \neq 0\, ?\, y_2 : y_3) && \wedge\\
&z_1 = y_4 + 1
\end{aligned}
$$

What about loops?

→ We unwind them!

```
void f(...)  {
   ...
   while(cond) {
     BODY;
   }
   ...
   Remainder;
}
```

What about loops?

→ We unwind them!

```
void f(...)  {
   ...
   if(cond) {
     BODY;
     while(cond) {
       BODY;
     }
   }
   ...
   Remainder;
}
```

What about loops?

→ We unwind them!

```
void f(...)  {
   ...
   if(cond) {
     BODY;
     if(cond) {
       BODY;
       while(cond) {
         BODY;
       }
     }
   }
   ...
   Remainder;
}
```

Some caveats:

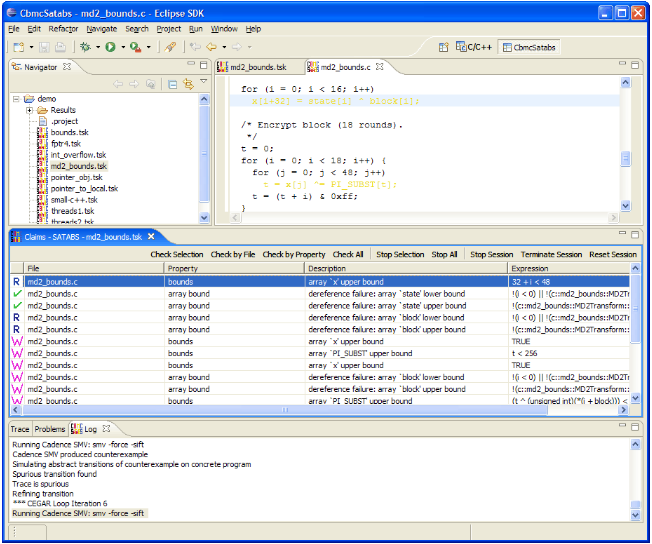
- Unwind how many times?
- Must preserve locality of variables declared inside loop

Some caveats:

- Unwind how many times?
- Must preserve locality of variables declared inside loop

There is a tool available that does this

- CBMC – C Bounded Model Checker
- Bound is verified using unwinding assertions
- Used frequently for embedded software
  $\longrightarrow$ Bound is a run-time guarantee
- Integrated into Eclipse
- Decision problem can be exported


CbmcSatabs - md2_bounds.c - Eclipse SDK
File Edit Refactor Navigate Search Project Run Window Help
C/C++
CbmcSatabs
Navigator
demo
Results
.project
bounds.tsk
fptr4.tsk
int_overflow.tsk
md2_bounds.tsk
pointer_obj.tsk
pointer_to_local.tsk
small-c++.tsk
threads1.tsk
md2_bounds.tsk
md2_bounds.c
for (i = 0; i < 16; i++)
x[i+32] = state[i] ^ block[i];
/* Encrypt block (18 rounds).
*/
t = 0;
for (i = 0; i < 18; i++) {
for (j = 0; j < 48; j++)
t = x[j] ^= PI_SUBST[t];
t = (t + i) & 0xff;
}
Claims - SATABS - md2_bounds.tsk
Check Selection Check by File Check by Property Check All Stop Selection Stop All Stop Session Terminate Session Reset Session
File Property Description Expression
md2_bounds.c bounds array `x' upper bound 32 + i < 48
md2_bounds.c array bound dereference failure: array `state' lower bound
md2_bounds.c array bound dereference failure: array `state' upper bound
md2_bounds.c array bound dereference failure: array `block' lower bound
md2_bounds.c array bound dereference failure: array `block' upper bound
md2_bounds.c bounds array `x' upper bound TRUE
md2_bounds.c bounds array `PI_SUBST' upper bound t < 256
md2_bounds.c bounds array `x' upper bound TRUE
md2_bounds.c array bound dereference failure: array `block' lower bound
md2_bounds.c array bound dereference failure: array `block' upper bound
Trace Problems Log
Running Cadence SMV: smv -force -sift
Cadence SMV produced counterexample
Simulating abstract transitions of counterexample on concrete program
Spurious transition found
Trace is spurious
Refining transition
*** CEGAR Loop Iteration 6
Running Cadence SMV: smv -force -sift

```
int power3(int in) {
    out = in;

    for(i=0; i<2; i++)
       out = out * in;

    return out;
}
```

```
int power3_new(int in) {
    out = (in*in)*in;
    return out;
}
```

```
int power3(int in) {
    out = in;

    for(i=0; i<2; i++)
       out = out * in;

    return out;
}
```

```
int power3_new(int in) {
    out = (in*in)*in;
    return out;
}
```

Static single assignment (SSA) form:

$$
\begin{aligned}
out_1 &= in \wedge \\
out_2 &= out_1 * in \wedge \\
out_3 &= out_2 * in
\end{aligned}
$$

$$out'_1 = (in * in) * in$$

Prove that both functions return the same value:

$$out_3 = out'_1$$

Static single assignment (SSA) form:

$$out_1 = in \wedge$$
$$out_2 = out_1 * in \wedge$$
$$out_3 = out_2 * in$$

$$out'_1 = (in * in) * in$$

With uninterpreted functions:

$$out_1 = in \wedge$$
$$out_2 = F(out_1, in) \wedge$$
$$out_3 = F(out_2, in)$$

$$out'_1 = F(F(in, in), in)$$

Static single assignment (SSA) form:

$$out_1 = in \land \qquad out'_1 = (in * in) * in$$
$$out_2 = out_1 * in \land$$
$$out_3 = out_2 * in$$

With uninterpreted functions:

$$out_1 = in \land \qquad out'_1 = F(F(in, in), in)$$
$$out_2 = F(out_1, in) \land$$
$$out_3 = F(out_2, in)$$

With numbered uninterpreted functions:

$$out_1 = in \land \qquad out'_1 = F_4(F_3(in, in), in)$$
$$out_2 = F_1(out_1, in) \land$$
$$out_3 = F_2(out_2, in)$$

With numbered uninterpreted functions:

$$out_1 = in \wedge$$
$$out_2 = F_1(out_1, in) \wedge \qquad out'_1 = F_4(F_3(in, in), in)$$
$$out_3 = F_2(out_2, in)$$

With numbered uninterpreted functions:

$$\begin{aligned} out_1 &= in \wedge \\ out_2 &= F_1(out_1, in) \wedge \\ out_3 &= F_2(out_2, in) \end{aligned} \qquad out'_1 = F_4(F_3(in, in), in)$$

Ackermann's reduction:

$$\varphi_a^E : \begin{aligned} out_1 &= in \wedge \\ out_2 &= f_1 \wedge \\ out_3 &= f_2 \end{aligned} \qquad \varphi_b^E : out'_1 = f_4$$

With numbered uninterpreted functions:

$$\begin{aligned} out_1 &= in \wedge \\ out_2 &= F_1(out_1, in) \wedge \\ out_3 &= F_2(out_2, in) \end{aligned} \qquad\qquad out'_1 = F_4(F_3(in, in), in)$$

Ackermann's reduction:

$$\varphi_a^E : \quad \begin{aligned} out_1 &= in \wedge \\ out_2 &= f_1 \wedge \\ out_3 &= f_2 \end{aligned} \qquad\qquad \varphi_b^E : \; out'_1 = f_4$$

The verification condition:

$$\left[\begin{pmatrix} (out_1 = out_2 \rightarrow f_1 = f_2) & \wedge \\ (out_1 = in \rightarrow f_1 = f_3) & \wedge \\ (out_1 = f_3 \rightarrow f_1 = f_4) & \wedge \\ (out_2 = in \rightarrow f_2 = f_3) & \wedge \\ (out_2 = f_3 \rightarrow f_2 = f_3) & \wedge \\ (in = f_3 \rightarrow f_3 = f_4) & \end{pmatrix} \wedge \varphi_a^E \wedge \varphi_b^E \right] \longrightarrow out_3 = out'_1$$

- Let n be the number of instances of $F()$
- Both reduction schemes require $O(n^2)$ comparisons
- This can be the *bottleneck* of the verification effort

Uninterpreted functions: simplifications

- Let n be the number of instances of $F()$
- Both reduction schemes require $O(n^2)$ comparisons
- This can be the *bottleneck* of the verification effort

- Solution: try to *guess* the pairing of functions
- Still sound: wrong guess can only make a valid formula invalid

- Given $x_1 = x'_1$, $x_2 = x'_2$, $x_3 = x'_3$, prove $\models o_1 = o_2$.

$$o_1 = (\underbrace{x_1 + (a \cdot x_2)}_{f_1}) \wedge a = \underbrace{x_3 + 5}_{f_2} \qquad \text{Left}$$

$$o_2 = (\underbrace{x'_1 + (b \cdot x'_2)}_{f_3}) \wedge b = \underbrace{x'_3 + 5}_{f_4} \qquad \text{Right}$$

- 4 function instances $\rightarrow$ 6 comparisons

Simplifications (1)

- Given $x_1 = x_1'$, $x_2 = x_2'$, $x_3 = x_3'$, prove $\models o_1 = o_2$.

$$o_1 = (\underbrace{x_1 + (a \cdot x_2)}_{f_1}) \land a = \underbrace{x_3 + 5}_{f_2} \qquad \text{Left}$$

$$o_2 = (\underbrace{x_1' + (b \cdot x_2')}_{f_3}) \land b = \underbrace{x_3' + 5}_{f_4} \qquad \text{Right}$$

- 4 function instances $\rightarrow$ 6 comparisons
- Guess: validity does not rely on $f_1 = f_2$ or on $f_3 = f_4$
- Idea: only enforce functional consistency of pairs (Left,Right).

$$o_1 = (\underbrace{x_1 + (a \cdot x_2)}_{f_1}) \wedge a = \underbrace{x_3 + 5}_{f_2} \qquad \text{Left}$$

$$o_2 = (\underbrace{x'_1 + (b \cdot x'_2)}_{f_3}) \wedge b = \underbrace{x'_3 + 5}_{f_4} \qquad \text{Right}$$

- Down to 4 comparisons!

Simplifications (2)

$$o_1 = (\underbrace{x_1 + (a \cdot x_2)}_{f_1}) \wedge a = \underbrace{x_3 + 5}_{f_2}$$ Left

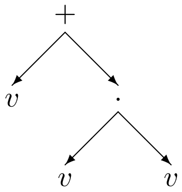
$$o_2 = (\underbrace{x'_1 + (b \cdot x'_2)}_{f_3}) \wedge b = \underbrace{x'_3 + 5}_{f_4}$$ Right

- Down to 4 comparisons!
- Another guess: equivalence only depends on $f_1 = f_3$ and $f_2 = f_4$
- *Pattern matching* may help here

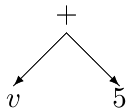
$$o_1 = (\underbrace{x_1 + (a \cdot x_2)}_{f_1}) \wedge a = \underbrace{x_3 + 5}_{f_2} \qquad \text{Left}$$

$$o_2 = (\underbrace{x'_1 + (b \cdot x'_2)}_{f_3}) \wedge b = \underbrace{x'_3 + 5}_{f_4} \qquad \text{Right}$$

Match according to patterns ('signatures')



f_1, f_3



f_2, f_4

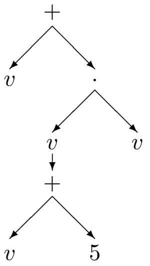
Down to 2 comparisons!

Simplifications (4)

$$o_1 = (\underbrace{x_1 + (a \cdot x_2)}_{f_1}) \ \wedge \ a = \underbrace{x_3 + 5}_{f_2} \qquad \text{Left}$$

$$o_2 = (\underbrace{x_1' + (b \cdot x_2')}_{f_3}) \ \wedge \ b = \underbrace{x_3' + 5}_{f_4} \qquad \text{Right}$$

Substitute intermediate variables (in the example: a, b)

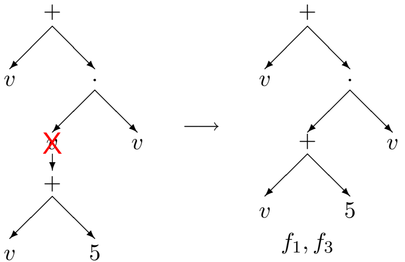


Simplifications (4)

$$o_1 = \underbrace{(x_1 + (a \cdot x_2))}_{f_1} \wedge a = \underbrace{x_3 + 5}_{f_2} \qquad \textcolor{red}{\text{Left}}$$

$$o_2 = \underbrace{(x'_1 + (b \cdot x'_2))}_{f_3} \wedge b = \underbrace{x'_3 + 5}_{f_4} \qquad \textcolor{red}{\text{Right}}$$

Substitute intermediate variables (in the example: a, b)



With numbered uninterpreted functions:

$$out_1 = in \wedge$$
$$out_2 = F_1(out_1, in) \wedge$$
$$out_3 = F_2(out_2, in)$$

$$out'_1 = F_4(F_3(in, in), in)$$

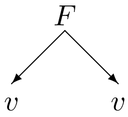
The SSA example revisited (1)

With numbered uninterpreted functions:

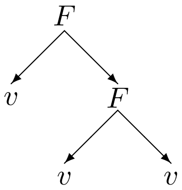
$$out_1 = in \land$$
$$out_2 = F_1(out_1, in) \land$$
$$out_3 = F_2(out_2, in)$$

$$out'_1 = F_4(F_3(in, in), in)$$

Map F_1 to F_3:



Map F_2 to F_4:



With numbered uninterpreted functions:

$$\begin{aligned} out_1 &= in \wedge \\ out_2 &= F_1(out_1, in) \wedge \\ out_3 &= F_2(out_2, in) \end{aligned} \qquad out'_1 = F_4(F_3(in, in), in)$$

Ackermann's reduction:

$$\varphi_a^E: \begin{array}{l} out_1 = in \wedge \\ out_2 = f_1 \wedge \\ out_3 = f_2 \end{array} \qquad \varphi_b^E: out'_1 = f_4$$

The verification condition has *shrunk*:

$$\left[\left(\begin{array}{ll} (out_1 = in \longrightarrow f_1 = f_3) & \wedge \\ (out_2 = f_3 \longrightarrow f_2 = f_4) & \end{array} \right) \wedge \varphi_a^E \wedge \varphi_b^E \right] \longrightarrow out_3 = out'_1$$

With numbered uninterpreted functions:

$$out_1 = in \wedge \qquad$$
$$out_2 = F_1(out_1, in) \wedge \qquad out'_1 = F_4(F_3(in, in), in)$$
$$out_3 = F_2(out_2, in)$$

Bryant's reduction:

$$\varphi_a^E : \begin{array}{l} out_1 = in \wedge \\ out_2 = f_1 \wedge \\ out_3 = f_2 \end{array} \qquad \varphi_b^E : out'_1 = \left(\begin{array}{ll} \text{case} & \left(\begin{array}{lll} \text{case} & in = out_1 : & f_1 \\ & \text{true} & : f_3 \end{array} \right) = out_2 : f_2 \\ & \text{true} \qquad\qquad\qquad\qquad : f_4 \end{array} \right)$$

The verification condition:

$$(\varphi_a^E \wedge \varphi_b^E) \longrightarrow out_3 = out'_1$$

1. It is expressible enough to state something interesting.
2. It is decidable and more efficiently solvable than richer logics, for example in which some functions are interpreted.
3. Models which rely on infinite-type variables are expressed more naturally in this logic in comparison with Propositional Logic.