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

Equality Logic

- A Boolean combination of Equalities and Propositions

$$x_1 = x_2 \land (x_2 = x_3 \lor \neg((x_1 = x_3) \land b \land x_1 = 2))$$

- We always push negations inside (NNF):

$$x_1 = x_2 \land (x_2 = x_3 \lor ((x_1 \neq x_3) \land \neg b \land x_1 \neq 2))$$

Syntax of Equality Logic

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

- 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

---

Expressiveness and complexity

- 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

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

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

Uninterpreted Functions

- 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 $\mathbb{N}$.

Uninterpreted Functions

- 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!

Uninterpreted Functions

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

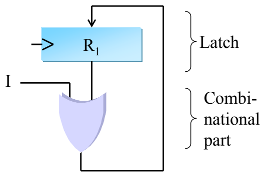
- Functional consistency axiom schema:

$$x_1 = x_1' \land \ldots \land x_n = x_n' \quad \Longrightarrow \quad f(x_1, \ldots, x_n) = f(x_1', \ldots, x_n')$$

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

Example: Circuit Transformations

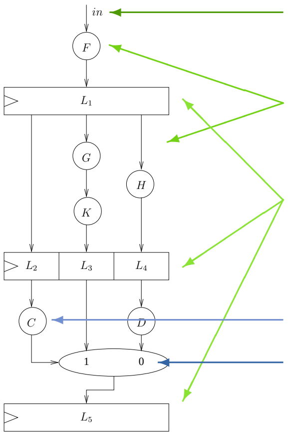
- Circuits consist of combinational gates and latches (registers)



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

$$\begin{array}{rcl} f(x, y) & := & x \lor y \\ R_1' & = & f(R_1, I) \end{array}$$

Example: Circuit Transformations



in: a primary input of the circuit

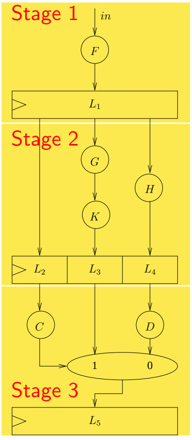
F, G, H, K, D: some functions over bit-vectors

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

C: a predicate over bit-vectors

a multiplexer (case-split)

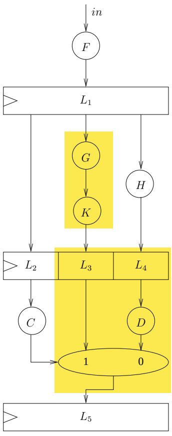
Example: Circuit Transformations



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

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

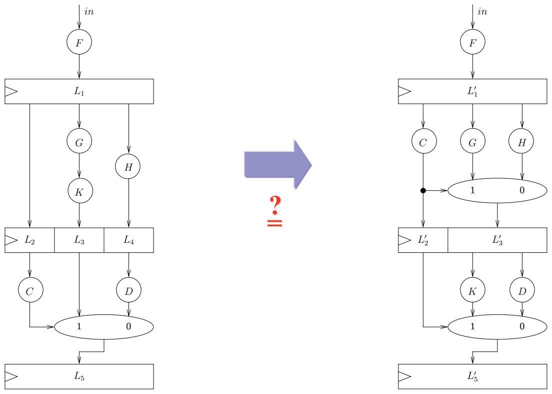
Example: Circuit Transformations



- 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
- Also note that the circuit only uses one of L_3 or L_4, never both

⇒ We can remove one of the latches

Example: Circuit Transformations



Example: Circuit Transformations

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

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

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

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

Transforming UFs to Equality Logic using Ackermann's reduction

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

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

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:

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

③ Add functional consistency constraints:

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

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

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) $\longrightarrow F_1(a) = F_2(b)$
  2. Replace each function instance F_i with an expression F_i^* $\longrightarrow F_1^* = F_2^*$

$$F_i^* := \left(\begin{array}{lll} \text{case} & x_1 = x_i & : f_1 \\ & x_2 = x_i & : f_2 \\ & \vdots & \\ & x_{i-1} = x_i & : f_{i-1} \\ & \text{true} & : f_i \end{array}\right) \longrightarrow f_1 = \left(\begin{array}{ll} \text{case} & a = b: f_1 \\ & \text{true} \ : f_2 \end{array}\right)$$

Example of Bryant's reduction

- Original formula:

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

- Number the instances:

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

- Replace each function application with an expression:

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

where

$$\begin{array}{lll} F_1^* & = & f_1 \\ F_2^* & = & \left(\begin{array}{lll} \text{case} & G_1^* = G_2^* & : f_1 \\ & \text{true} & : f_2 \end{array}\right) \\ G_1^* & = & g_1 \\ G_2^* & = & \left(\begin{array}{lll} \text{case} & a = b & : g_1 \\ & \text{true} & : g_2 \end{array}\right) \end{array}$$

Using uninterpreted functions in proofs

- 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.

Using uninterpreted functions in proofs

- 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'

Using uninterpreted functions in proofs

- 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?

Using uninterpreted functions in proofs

- (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

Example: Translation Validation

- 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:

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

Example: Translation Validation

- Claim: φ^{UF} is valid

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

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

$$((u_1 = f_1 \;\wedge\; u_2 = f_2 \;\wedge\; z = g_1) \quad \longrightarrow \quad z = g_2)$$

Uninterpreted functions: usability

- 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
$x + x$	$2x$

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:

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

- 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!

Example: equivalence of C programs (1/4)

```
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

From programs to equations

- 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.

Static Single Assignment (SSA) form

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

Example:

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

⟹

$\texttt{x}_1\texttt{=x}_0\texttt{+y}_0\texttt{;}$
$\texttt{x}_2\texttt{=x}_1\texttt{*2;}$
$\texttt{a}_1\texttt{[i}_0\texttt{]=100;}$

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

Example:

$\texttt{x}_1\texttt{=x}_0\texttt{+y}_0\texttt{;}$
$\texttt{x}_2\texttt{=x}_1\texttt{*2;}$
$\texttt{a}_1\texttt{[i}_0\texttt{]=100;}$

⟹

$$x_1 = x_0 + y_0 \quad \wedge$$
$$x_2 = x_1 * 2 \quad \wedge$$
$$a_1[i_0] = 100$$

SSA for bounded programs

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;
}
```

⟹

$\texttt{int main() \{}$
$\quad\texttt{int x, y, z;}$
$\quad\texttt{y}_1\texttt{=8;}$
$\quad\texttt{if(x}_0\texttt{)}$
$\qquad\texttt{y}_2\texttt{=y}_1\texttt{-1;}$
$\quad\texttt{else}$
$\qquad\texttt{y}_3\texttt{=y}_1\texttt{+1;}$
$\quad\texttt{y}_4\texttt{=}\phi\texttt{(y}_2\texttt{, y}_3\texttt{);}$
$\quad\texttt{z}_1\texttt{=y}_4\texttt{+1;}$
$\texttt{\}}$

⟹

$$y_1 = 8 \quad \wedge$$
$$y_2 = y_1 - 1 \quad \wedge$$
$$y_3 = y_1 + 1 \quad \wedge$$
$$y_4 = (x_0 \neq 0\, ?\, y_2 : y_3) \wedge$$
$$z_1 = y_4 + 1$$

SSA for bounded programs

What about loops?
→ We unwind them!

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

⟶

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

SSA for bounded programs

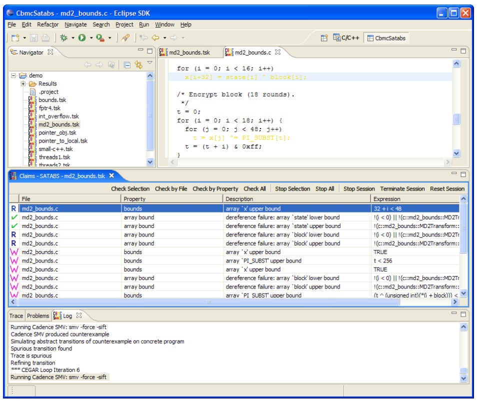
Some caveats:

- 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
  ⟶ Bound is a run-time guarantee
- Integrated into Eclipse
- Decision problem can be exported

SSA for bounded programs: CBMC



```
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;
}
```

	File	Property	Description	Expression
R	md2_bounds.c	bounds	array `x' upper bound	32 + i < 48
✓	md2_bounds.c	array bound	dereference failure: array `state' lower bound	!(i < 0) || !(c::md2_bounds::MD2Tr
✓	md2_bounds.c	array bound	dereference failure: array `state' upper bound	!(c::md2_bounds::MD2Transform::
R	md2_bounds.c	array bound	dereference failure: array `block' lower bound	!(i < 0) || !(c::md2_bounds::MD2Tr
R	md2_bounds.c	array bound	dereference failure: array `block' upper bound	!(c::md2_bounds::MD2Transform::
W	md2_bounds.c	bounds	array `x' upper bound	TRUE
W	md2_bounds.c	bounds	array `PI_SUBST' upper bound	t < 256
W	md2_bounds.c	bounds	array `x' upper bound	TRUE
W	md2_bounds.c	array bound	dereference failure: array `block' lower bound	!(i < 0) || !(c::md2_bounds::MD2Tr
W	md2_bounds.c	array bound	dereference failure: array `block' upper bound	!(c::md2_bounds::MD2Transform::
W	md2_bounds.c	bounds	array `PI_SUBST' upper bound	(t ^ (unsigned int)(*(i + block))) <

```
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
```

Example: equivalence of C programs (2/4)

```
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:

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

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

Prove that both functions return the same value:

$$out_3 = out'_1$$

Example: equivalence of C programs (3/4)

Static single assignment (SSA) form:

$$\begin{array}{ll} out_1 = in \wedge & \\ out_2 = out_1 * in \wedge & out'_1 = (in * in) * in \\ out_3 = out_2 * in & \end{array}$$

With uninterpreted functions:

$$\begin{array}{ll} out_1 = in \wedge & \\ out_2 = F(out_1, in) \wedge & out'_1 = F(F(in, in), in) \\ out_3 = F(out_2, in) & \end{array}$$

With numbered uninterpreted functions:

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

Example: equivalence of C programs (4/4)

With numbered uninterpreted functions:

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

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:

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

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

Simplifications (1)

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

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

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

- 4 function instances → 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).

Simplifications (2)

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

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

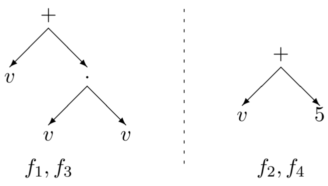
- Down to 4 comparisons!
- Another guess: equivalence only depends on $f_1 = f_3$ and $f_2 = f_4$
- *Pattern matching* may help here

Simplifications (3)

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

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

Match according to patterns ('signatures')



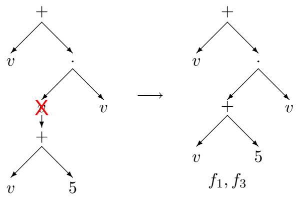
Down to 2 comparisons!

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)

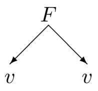


The SSA example revisited (1)

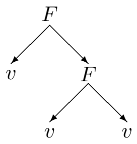
With numbered uninterpreted functions:

$$\begin{array}{l} out_1 = in \wedge \\ out_2 = F_1(out_1, in) \wedge \\ out_3 = F_2(out_2, in) \end{array} \qquad out_1' = F_4(F_3(in, in), in)$$

Map F_1 to F_3:



Map F_2 to F_4:



The SSA example revisited (2)

With numbered uninterpreted functions:

$$\begin{array}{l} out_1 = in \wedge \\ out_2 = F_1(out_1, in) \wedge \\ out_3 = F_2(out_2, in) \end{array} \qquad out_1' = F_4(F_3(in, in), in)$$

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*:

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

Same example with Bryant's reduction

With numbered uninterpreted functions:

$$\begin{array}{l} out_1 = in \wedge \\ out_2 = F_1(out_1, in) \wedge \\ out_3 = F_2(out_2, in) \end{array} \qquad out_1' = F_4(F_3(in, in), in)$$

Bryant's reduction:

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

The verification condition:

$$(\varphi_a^E \wedge \varphi_b^E) \longrightarrow out_3 = out_1'$$

So is Equality Logic with UFs interesting?

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.