Equalities and Uninterpreted Functions
Chapter 3

Decision Procedures
An Algorithmic Point of View

D.Kroening    O.Strichman
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

\[ \text{formula} : \quad \text{formula} \lor \text{formula} \]
\[ \quad \neg \text{formula} \]
\[ \quad \text{atom} \]

\[ \text{atom} : \quad \text{term-variable} = \text{term-variable} \]
\[ \quad \text{term-variable} = \text{constant} \]
\[ \quad \text{Boolean-variable} \]

- The \textit{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

\[
\text{formula} : \quad \text{formula} \lor \text{formula} \\
\quad \mid \neg \text{formula} \\
\quad \mid \text{atom}
\]

\[
\text{atom} : \quad \text{term} = \text{term} \\
\quad \mid \text{Boolean-variable}
\]

\[
\text{term} : \quad \text{term-variable} \\
\quad \mid \text{function} ( \text{list of terms} )
\]

The \textit{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:

$$x_1 + x_2 = x_3 + x_4$$ is replaced by $$f(x_1, x_2) = f(x_3, x_4)$$

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:

\[
x_1 = x'_1 \land \ldots \land x_n = x'_n \implies f(x_1, \ldots, x_n) = f(x'_1, \ldots, x'_n)
\]

Sometimes, functional consistency is all that is needed for a proof.
Example: Circuit Transformations

- Circuits consist of combinational gates and latches (registers)

\[ R_1' = f(R_1, I) \]

Decision Procedures – Equalities and Uninterpreted Functions
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

\[ f(x, y) := x \lor y \]
\[ R_1' = f(R_1, I) \]
in: a primary input of the circuit

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

L₁, . . . , L₅: latches (registers)
C: a predicate over bit-vectors

Decision Procedures – Equalities and Uninterpreted Functions
Example: Circuit Transformations

\[ \text{in} \quad \text{in}: \text{a primary input of the circuit} \]

\[ L_1 \]

\[ L_2 \quad L_3 \quad L_4 \]

\[ L_5 \]

\[ C \quad D \]

\[ F \quad G \quad H \quad K \]

\[ 1 \quad 0 \]
Example: Circuit Transformations

\[ \text{in}: \text{a primary input of the circuit} \]

\[ F, G, H, K, D: \text{some functions over bit-vectors} \]
Example: Circuit Transformations

\( \text{in} \): 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

\[ \text{in}: \text{a primary input of the circuit} \]

\[ F, G, H, K, D: \text{some functions over bit-vectors} \]

\[ L_1, \ldots, L_5: \text{latches (registers)} \]

\[ C: \text{a predicate over bit-vectors} \]

\[ \text{a multiplexer (case-split)} \]
A pipeline processes data in *stages*

Data is processed in parallel – as in an assembly line

**Formal model:**

\[
\begin{align*}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2) \ ? \ L_3 : l(L_4)
\end{align*}
\]
A pipeline processes data in *stages*

- Data is processed in parallel – as in an assembly line

**Formal model:**

\[
\begin{align*}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2) \ ? L_3 : l(L_4)
\end{align*}
\]
A pipeline processes data in stages
- Data is processed in parallel – as in an assembly line
- Formal model:

\[
\begin{align*}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2) \ ? \ L_3 : l(L_4)
\end{align*}
\]
Example: Circuit Transformations

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

\[
\begin{align*}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2) \ ? L_3 : l(L_4)
\end{align*}
\]
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

Decision Procedures – Equalities and Uninterpreted Functions
Example: Circuit Transformations

\[ L_1 = f(I) \]
\[ L_2 = L_1 \]
\[ L_3 = k(g(L_1)) \]
\[ L_4 = h(L_1) \]
\[ L_5 = c(L_2) \land L_3 \lor l(L_4) \]

\[ L_1' = f(I) \]
\[ L_2' = c(L_1') \]
\[ L_3' = c(L_1') \land g(L_1') \lor h(L_1') \]
\[ L_5' = L_2' \land k(L_3') \lor l(L_3') \]

\[ L_5 = L_5' \]
Example: Circuit Transformations

\[ L_1 = f(I) \]
\[ L_2 = L_1 \]
\[ L_3 = k(g(L_1)) \]
\[ L_4 = h(L_1) \]
\[ L_5 = c(L_2) \land L_3 : l(L_4) \]

\[ L'_1 = f(I) \]
\[ L'_2 = c(L'_1) \]
\[ L'_3 = c(L'_1) \land g(L'_1) : h(L'_1) \]
\[ L'_4 = L'_2 \land k(L'_3) : l(L'_3) \]

\[ L_5 \ ? \ L'_5 \]

- Equivalence in this case holds \textit{regardless of the actual functions}
- Conclusion: can be decided using \textit{Equality Logic and Uninterpreted Functions}
Transforming UFs to Equality Logic using Ackermann’s reduction

- Given: a formula \( \varphi^{UF} \) with uninterpreted functions
- For each function in \( \varphi^{UF} \):
  1. Number function instances (from the inside out)
     \[ F_2(F_1(x)) = 0 \]
Transforming UFs to Equality Logic using Ackermann’s reduction

- Given: a formula $\varphi^{UF}$ with uninterpreted functions
- For each function in $\varphi^{UF}$:
  1. Number function instances (from the inside out)
     $\quad \rightarrow \quad F_2(F_1(x)) = 0$
  2. Replace each function instance with a new variable
     $\quad \rightarrow \quad f_2 = 0$
Given: a formula $\varphi^{UF}$ with uninterpreted functions

For each function in $\varphi^{UF}$:

1. Number function instances (from the inside out)

   $\quad \rightarrow F_2(F_1(x)) = 0$

2. Replace each function instance with a new variable

   $\quad \rightarrow f_2 = 0$

3. Add functional consistency constraint to $\varphi^{UF}$ for every pair of instances of the same function.

   $\quad \rightarrow ((x = f_1) \rightarrow (f_2 = f_1)) \rightarrow f_2 = 0$
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:

\[ x_1 \neq x_2 \lor F_1(x_1) = F_2(x_2) \lor F_1(x_1) \neq F_3(x_3) \]
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:
\[ x_1 \neq x_2 \lor f_1 = f_2 \lor f_1 \neq f_3 \]
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:

\[
\left( (x_1 = x_2 \rightarrow f_1 = f_2) \land (x_1 = x_3 \rightarrow f_1 = f_3) \land (x_2 = x_3 \rightarrow f_2 = f_3) \right) \rightarrow
\]

Decision Procedures – Equalities and Uninterpreted Functions
Given: a formula $\varphi^{UF}$ with uninterpreted functions

For each function in $\varphi^{UF}$:

1. Number function instances (from the inside out) $\rightarrow F_1(a) = F_2(b)$
Transforming UFs to Equality Logic using Bryant’s reduction

- Given: a formula $\varphi^{UF}$ with uninterpreted functions
- For each function in $\varphi^{UF}$:
  1. Number function instances (from the inside out) $\quad \rightarrow F_1(a) = F_2(b)$
  2. Replace each function instance $F_i$ with an expression $F_i^*$ $\quad \rightarrow F_1^* = F_2^*$
Given: a formula $\varphi^{UF}$ with uninterpreted functions

For each function in $\varphi^{UF}$:

1. Number function instances (from the inside out)

   $F_1(a) = F_2(b)$

2. Replace each function instance $F_i$ with an expression $F_i^*$

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

   $f_1 = \begin{cases} 
   \text{case } a = b : f_1 \\
   \text{true } : f_2 
   \end{cases}$
Example of Bryant’s reduction

- 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:
  \[ a = b \rightarrow F_1(G_1(a) = F_2(G_2(b)) \]
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

\[ F_1^* = f_1 \]
\[ F_2^* = \left( \begin{array}{c} \text{case} \quad G_1^* = G_2^* : f_1 \\ \text{true} : f_2 \end{array} \right) \]
\[ G_1^* = g_1 \]
\[ G_2^* = \left( \begin{array}{c} \text{case} \quad a = b : g_1 \end{array} \right) \]
Using uninterpreted functions in proofs

- Uninterpreted functions give us the ability to represent an \textit{abstract} view of functions.
- It \textit{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 $\varphi_{UF}$:

1. Transform it into Equality Logic formula $\varphi^E$.
2. If $\varphi^E$ is unsatisfiable, return 'Unsatisfiable'.
3. Else return 'Don’t know'.
Question #1: is this useful?
Question #1: is this useful?
Question #2: can it be made complete in some cases?
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.
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")
(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:

\[
\begin{align*}
  u_1 &= x_1 + y_1; \\
  u_2 &= x_2 + y_2; \\
  z &= u_1 \cdot u_2;
\end{align*}
\]
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:

\[
\begin{align*}
u_1 &= x_1 + y_1; \\
u_2 &= x_2 + y_2; \\
z &= u_1 \cdot u_2;
\end{align*}
\]

- We need to prove that:

\[
\begin{align*}
(u_1 = x_1 + y_1 \land u_2 = x_2 + y_2 \land z = u_1 \cdot u_2) \\
\implies (z = (x_1 + y_1) \cdot (x_2 + y_2))
\end{align*}
\]
Claim: $\varphi^{UF}$ is valid

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

\[
\varphi^E = \left( (x_1 = x_2 \land y_1 = y_2 \rightarrow f_1 = f_2) \land (u_1 = f_1 \land u_2 = f_2 \rightarrow g_1 = g_2) \right) \rightarrow (u_1 = f_1 \land u_2 = f_2 \land z = g_1) \rightarrow z = g_2
\]
Uninterpreted functions: usability

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

  \[
  \begin{array}{ll}
  \text{Left} & \text{Right} \\
  x + x & 2x \\
  \end{array}
  \]
This is easy to prove:

\[(x_1 = x_2 \land y_1 = y_2) \rightarrow (x_1 + y_1 = x_2 + y_2)\]
This is easy to prove:

\[(x_1 = x_2 \land y_1 = y_2) \rightarrow (x_1 + y_1 = x_2 + y_2)\]

This requires *commutativity*:

\[(x_1 = x_2 \land y_1 = y_2) \rightarrow (x_1 + y_1 = y_2 + x_2)\]
This is easy to prove:

$$(x_1 = x_2 \land y_1 = y_2) \rightarrow (x_1 + y_1 = x_2 + y_2)$$

This requires **commutativity**:

$$(x_1 = x_2 \land y_1 = y_2) \rightarrow (x_1 + y_1 = y_2 + x_2)$$

Fix by adding:

$$(x_1 + y_1 = y_1 + x_1) \land (x_2 + y_2 = y_2 + x_2)$$
This is easy to prove:

\[(x_1 = x_2 \land y_1 = y_2) \rightarrow (x_1 + y_1 = x_2 + y_2)\]

This requires commutativity:

\[(x_1 = x_2 \land y_1 = y_2) \rightarrow (x_1 + y_1 = y_2 + x_2)\]

Fix by adding:

\[(x_1 + y_1 = y_1 + x_1) \land (x_2 + y_2 = y_2 + x_2)\]

What about other cases?
Use more rewriting rules!
Example: equivalence of C programs (1/4)

```c
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.
Static Single Assignment (SSA) form

- see compiler class

Idea: Rename variables such that each variable is assigned exactly once

Example:

\[
\begin{align*}
  x &= x + y; \\
  x &= x \times 2; \\
  a[i] &= 100;
\end{align*}
\]

\[
\begin{align*}
  x_1 &= x_0 + y_0; \\
  x_2 &= x_1 \times 2; \\
  a_1[i_0] &= 100;
\end{align*}
\]
Static Single Assignment (SSA) form

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

\[
\begin{align*}
x &= x + y; & \quad x_1 &= x_0 + y_0; \\
x &= x \times 2; & \quad x_2 &= x_1 \times 2; \\
a[i] &= 100; & \quad a_1[i_0] &= 100;
\end{align*}
\]

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

\[
\begin{align*}
x_1 &= x_0 + y_0; \\
x_2 &= x_1 \times 2; \\
a_1[i_0] &= 100;
\end{align*}
\]

\[
\begin{align*}
x_1 &= x_0 + y_0 \land \\
x_2 &= x_1 \times 2 \land \\
a_1[i_0] &= 100
\end{align*}
\]
What about if? Branches are handled using $\phi$-nodes.

```c
int main() {
    int x, y, z;
    y=8;
    if(x)
        y--; 
    else 
        y++; 
    z=y+1;
}
```
SSA for bounded programs

What about if? Branches are handled using \( \phi \)-nodes.

```c
int main() {
    int x, y, z;
    y = 8;
    if (x)
        y--;  
    else
        y++;
    z = y + 1;
}
```

```c
int main() {
    int x, y, z;
    y_1 = 8;
    if (x_0)
        y_2 = y_1 - 1;
    else
        y_3 = y_1 + 1;
    y_4 = \phi(y_2, y_3);
    z_1 = y_4 + 1;
}
```
SSA for bounded programs

What about if? Branches are handled using $\phi$-nodes.

```c
int main() {
    int x, y, z;
    y = 8;
    if (x)
        y--;  // y = y - 1
    else
        y++;  // y = y + 1
    z = y + 1;
}
```

```c
int main() {
    int x, y, z;
    y = 8;
    if (x)  // x \neq 0
        y = y - 1;
    else
        y = y + 1;
    y = \phi(y, y);
    z = y + 1;
}
```

```
y_1 = 8 \land
y_2 = y_1 - 1 \land
y_3 = y_1 + 1 \land
y_4 = (x_0 \neq 0 ? y_2 : y_3) \land
z_1 = y_4 + 1
```
What about loops?
→ We **unwind** them!

```c
void f(...) {
    ...
    while(cond) {
        BODY;
    }
    ...
    Remainder;
}
```
What about loops?
→ We **unwind** them!

```c
void f(...) {
    ...
    if(\texttt{cond}) {
        \texttt{BODY};
        while(\texttt{cond}) {
            \texttt{BODY};
        }
    }
    ...
    \texttt{Remainder};
}
```
What about loops?
→ We unwind them!

```c
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 → Bound is a run-time guarantee
- Integrated into Eclipse
- Decision problem can be exported
SSA for bounded programs: CBMC

```c
for (i = 0; i < 16; i++)
    r(i-32) = state[i] ^ block[i];

/* Encrypt block (16 rounds). */

t = 0;
for (i = 0; i < 16; i++) {
    for (j = 0; j < 48; j++)
        v[i] = v[i] ^ P1_SUBST[c];
    t = (t + 1) & 0xff;
}
```

<table>
<thead>
<tr>
<th>File</th>
<th>Property</th>
<th>Description</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>md2_bounds.c</td>
<td>bounds</td>
<td>array 'x' upper bound</td>
<td>r + i &lt; 48</td>
</tr>
</tbody>
</table>
| md2_bounds.c | array    | dereference failure: array 'state' lower bound | 0 < 0 || (c
| md2_bounds.c | array    | dereference failure: array 'state' upper bound | (c
| md2_bounds.c | array    | dereference failure: array 'block' lower bound | 0 < 0 || (c
| md2_bounds.c | array    | dereference failure: array 'block' upper bound | (c
| md2_bounds.c | bounds   | array 'P1_SUBST upper bound     | t < 256    |
| md2_bounds.c | bounds   | array 'x' upper bound           | TRUE      |
| md2_bounds.c | array    | dereference failure: array 'block' lower bound | 0 < 0 || (c
| md2_bounds.c | array    | dereference failure: array 'block' upper bound | (c
| md2_bounds.c | bounds   | array 'P1_SUBST upper bound     | t < 256    |
| md2_bounds.c | bounds   | array 'x' upper bound           | TRUE      |
Example: equivalence of C programs (2/4)

```c
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;
}
```

Prove that both functions return the same value:
Example: equivalence of C programs (2/4)

```c
int power3(int in) {
    out = in;
    for(i=0; i<2; i++)
        out = out * in;
    return out;
}
```

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

Static single assignment (SSA) form:

\[
\begin{align*}
out_1 &= \text{in} \\
out_2 &= out_1 \times \text{in} \\
out_3 &= out_2 \times \text{in}
\end{align*}
\]

\[
out'_1 = (\text{in} \times \text{in}) \times \text{in}
\]

Prove that both functions return the same value:

\[
out_3 = out'_1
\]
Example: equivalence of C programs (3/4)

Static single assignment (SSA) form:
\[
\begin{align*}
    out_1 &= in \\
    out_2 &= out_1 \times in \\
    out_3 &= out_2 \times in \\
    out'_1 &= (in \times in) \times in
\end{align*}
\]

With uninterpreted functions:
\[
\begin{align*}
    out_1 &= in \\
    out_2 &= F(out_1, in) \\
    out_3 &= F(out_2, in) \\
    out'_1 &= F(F(in, in), in)
\end{align*}
\]
Example: equivalence of C programs (3/4)

Static single assignment (SSA) form:
\[
\begin{align*}
\text{out}_1 &= \text{in} \land \\
\text{out}_2 &= \text{out}_1 \ast \text{in} \land \\
\text{out}_3 &= \text{out}_2 \ast \text{in}
\end{align*}
\]
\[
\text{out}_1' = (\text{in} \ast \text{in}) \ast \text{in}
\]

With uninterpreted functions:
\[
\begin{align*}
\text{out}_1 &= \text{in} \land \\
\text{out}_2 &= F(\text{out}_1, \text{in}) \land \\
\text{out}_3 &= F(\text{out}_2, \text{in})
\end{align*}
\]
\[
\text{out}_1' = F(F(\text{in}, \text{in}), \text{in})
\]

With numbered uninterpreted functions:
\[
\begin{align*}
\text{out}_1 &= \text{in} \land \\
\text{out}_2 &= F_1(\text{out}_1, \text{in}) \land \\
\text{out}_3 &= F_2(\text{out}_2, \text{in})
\end{align*}
\]
\[
\text{out}_1' = F_4(F_3(\text{in}, \text{in}), \text{in})
\]
With numbered uninterpreted functions:

\[
\begin{align*}
out_1 &= in \\
out_2 &= F_1(out_1, in) \\
out_3 &= F_2(out_2, in) \\
out'_1 &= F_4(F_3(in, in), in)
\end{align*}
\]
With numbered uninterpreted functions:

\[
\begin{align*}
out_1 &= in \
out_2 &= F_1(out_1, in) \
out_3 &= F_2(out_2, in) \\
out'_1 &= F_4(F_3(in, in), in)
\end{align*}
\]

Ackermann’s reduction:

\[
\begin{align*}
\varphi^E_a : & \quad out_2 = f_1 \
& \quad out_3 = f_2 \\
\varphi^E_b : & \quad out'_1 = f_4
\end{align*}
\]
Example: equivalence of C programs (4/4)

With numbered uninterpreted functions:

\[
\begin{align*}
out_1 &= in \land \\
out_2 &= F_1(out_1, in) \land \\
out_3 &= F_2(out_2, in)
\end{align*}
\]

\[
out'_1 = F_4(F_3(in, in), in)
\]

Ackermann’s reduction:

\[
\begin{align*}
out_1 &= in \land \\
\varphi^E_a : \quad out_2 &= f_1 \land \\
out_3 &= f_2
\end{align*}
\]

\[
\varphi^E_b : \quad out'_1 = f_4
\]

The verification condition:

\[
\left[
\begin{align*}
(out_1 = out_2 \rightarrow f_1 = f_2) \land \\
(out_1 = in \rightarrow f_1 = f_3) \land \\
(out_1 = f_3 \rightarrow f_1 = f_4) \land \\
(out_2 = in \rightarrow f_2 = f_3) \land \\
(out_2 = f_3 \rightarrow f_2 = f_3) \land \\
(in = f_3 \rightarrow f_3 = f_4)
\end{align*}
\right] \land \varphi^E_a \land \varphi^E_b \rightarrow out_3 = out'_1
\]
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
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 \( o_1 = o_2 \).

\[
o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \tag{Left} \]

\[
o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \tag{Right} \]

- 4 function instances → 6 comparisons
Simplifications (1)

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

\[
o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \quad \text{Left} \quad f_1 \quad f_2
\]

\[
o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \quad \text{Right} \quad f_3 \quad f_4
\]

- 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).
Simplifications (2)

\[ o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \]

\( f_1 \)

\( f_2 \)

Left

\[ o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \]

\( f_3 \)

\( f_4 \)

Right

- Down to 4 comparisons!
Simplifications (2)

\[ o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \quad \text{Left} \]

\[ o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \quad \text{Right} \]

- 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)

\[ o_1 = \underbrace{(x_1 + (a \cdot x_2))}^{f_1} \land a = \underbrace{x_3 + 5}_{f_2} \quad \text{Left} \]

\[ o_2 = \underbrace{(x'_1 + (b \cdot x'_2))}^{f_3} \land b = \underbrace{x'_3 + 5}_{f_4} \quad \text{Right} \]

Match according to patterns ('signatures')

Down to 2 comparisons! \[ f_1, f_3 \]

\[ f_2, f_4 \]
Simplifications (4)

\[ o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \quad \text{Left} \]

\[ o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \quad \text{Right} \]

Substitute intermediate variables (in the example: \( a, b \))
Simplifications (4)

Left

\[ o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \]

Right

\[ o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \]

Substitute

intermediate variables (in the example: \( a, b \))
With numbered uninterpreted functions:

\[ \begin{align*}
out_1 &= \text{in} \land \\
out_2 &= F_1(out_1, \text{in}) \land \\
out_3 &= F_2(out_2, \text{in}) \\
\text{out}'_1 &= F_4(F_3(\text{in}, \text{in}), \text{in})
\end{align*} \]
The SSA example revisited (1)

With numbered uninterpreted functions:

\[
\begin{align*}
\text{out}_1 &= \text{in} \\
\text{out}_2 &= F_1(\text{out}_1, \text{in}) \\
\text{out}_3 &= F_2(\text{out}_2, \text{in}) \\
\text{out'}_1 &= F_4(F_3(\text{in}, \text{in}), \text{in})
\end{align*}
\]

Map $F_1$ to $F_3$:

Map $F_2$ to $F_4$:
The SSA example revisited (2)

With numbered uninterpreted functions:
\[
\begin{align*}
out_1 &= \text{in} \\
out_2 &= F_1(out_1, \text{in}) \\
out_3 &= F_2(out_2, \text{in}) \\
out'_1 &= F_4(F_3(\text{in}, \text{in}), \text{in})
\end{align*}
\]

Ackermann’s reduction:
\[
\begin{align*}
out_1 &= \text{in} \\
\varphi^E_a : out_2 &= f_1 \\
out_3 &= f_2 \\
\varphi^E_b : out'_1 &= f_4
\end{align*}
\]

The verification condition has shrunk:
\[
\left( (out_1 = \text{in} \rightarrow f_1 = f_3) \wedge (out_2 = f_3 \rightarrow f_2 = f_4) \wedge \varphi^E_a \wedge \varphi^E_b \right) \rightarrow out_3 = out'_1
\]
Same example with Bryant’s reduction

With numbered uninterpreted functions:
\[ out_1 = in \land \]
\[ out_2 = F_1(out_1, in) \land \]
\[ out_3 = F_2(out_2, in) \]
\[ out'_1 = F_4(F_3(in, in), in) \]

Bryant’s reduction:
\[ \varphi^E_a : \quad \begin{align*}
    out_1 &= in \land \\
    out_2 &= f_1 \land \\
    out_3 &= f_2 
\end{align*} \]
\[ \varphi^E_b : \quad \begin{align*}
    out'_1 &= \begin{cases}
        \text{case} & (\begin{cases}
            \text{case} & \text{true}
        \end{cases}
        \begin{align*}
            in &= out_1: f_1 \\
            in &= out_2: f_2 \\
        \end{align*}
        : f_4
    \end{cases}
    \end{cases}
\end{align*} \]

The verification condition:
\[ (\varphi^E_a \land \varphi^E_b) \rightarrow out_3 = out'_1 \]
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.