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

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

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

- 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

\[
\text{formula} : \quad \text{formula} \lor \text{formula} \\
\quad | \quad \neg \text{formula} \\
\quad | \quad \text{atom} \\
\text{atom} : \quad \text{term} = \text{term} \\
\quad | \quad \text{Boolean-variable} \\
\text{term} : \quad \text{term-variable} \\
\quad | \quad \text{function} \left( \text{list of terms} \right)
\]

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:

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

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:

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

Sometimes, functional consistency is all that is needed for a proof.
Circuits consist of combinational gates and latches (registers)
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

\[ f(x, y) := x \lor y \]
\[ R_1' = f(R_1, I) \]
Example: Circuit Transformations

- \( F \): a primary input of the circuit
- \( L_1, L_2, L_3, L_4, L_5 \): latches (registers)
- \( G, H, K, D \): some functions over bit-vectors
- \( C \): a predicate over bit-vectors
- \( in \): a multiplexer (case-split)
Example: Circuit Transformations

\[ \text{in} \rightarrow F \rightarrow L_1 \rightarrow G \rightarrow H \rightarrow K \rightarrow L_2 \rightarrow L_3 \rightarrow L_4 \rightarrow C \rightarrow D \rightarrow 1 \text{ and } 0 \rightarrow L_5 \]

\text{in: a primary input of the circuit}
Example: Circuit Transformations

\(in\): a primary input of the circuit

\(F, G, H, K, D\): some functions over bit-vectors
Example: Circuit Transformations

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

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

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

- **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

- **D**: 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:

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

A pipeline processes data in stages.
Data is processed in parallel – as in an assembly line.

Formal model:

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

\[
\begin{align*}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2) ? L_3 : l(L_4)
\end{align*}
\]
The maximum clock frequency depends on the \textit{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

\[ \begin{array}{c}
in \\
F \\
\downarrow \\
L_1 \\
\downarrow \\
G \\
\downarrow \\
K \\
\downarrow \\
L_2 \quad L_3 \quad L_4 \\
\downarrow \\
C \quad D \\
\downarrow \\
1 \quad 0 \\
\end{array} \quad \begin{array}{c}
in \\
F \\
\downarrow \\
L_1' \\
\downarrow \\
C \\
\downarrow \\
1 \quad 0 \\
\downarrow \\
K \quad D \\
\downarrow \\
1 \quad 0 \\
\end{array} \]
Example: Circuit Transformations

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

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

\[ L_5 \equiv L'_5 \]
Example: Circuit Transformations

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

\[
\begin{align*}
L'_1 &= f(I) \\
L'_2 &= c(L'_1) \\
L'_3 &= c(L'_1) \land g(L'_1) : h(L'_1) \\
L'_4 &= L'_2 \land k(L'_3) : l(L'_3) \\
L'_5 &= L'_5
\end{align*}
\]

\[
L_5 \Rightarrow L'_5
\]

- Equivalence in this case holds regardless of the actual functions
- Conclusion: can be decided using **Equality Logic and Uninterpreted Functions**
Transforming UF 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)
      
      $F_2(F_1(x)) = 0$

D. Kroening, O. Strichman (ETH/Technion)
Decision Procedures
Version 1.0, 2007 17 / 47
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)
  \[ F_2(F_1(x)) = 0 \]
  2. Replace each function instance with a new variable
  \[ f_2 = 0 \]
- Given: a formula $\varphi^U$ with uninterpreted functions
- For each function in $\varphi^U$:
  1. Number function instances (from the inside out)
     
     \[ F_2(F_1(x)) = 0 \]
  2. Replace each function instance with a new variable
     
     \[ f_2 = 0 \]
  3. Add functional consistency constraint to $\varphi^U$ for every pair of instances of the same function.
     
     \[ ((x = f_1) \rightarrow (f_2 = f_1)) \rightarrow f_2 = 0 \]
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:

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

\[ x_1 \neq x_2 \lor F(x_1) = F(x_2) \lor F(x_1) \neq F(x_3) \]

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:

\[
\left( (x_1 = x_2 \rightarrow f_1 = f_2) \right) \wedge \\
\left( (x_1 = x_3 \rightarrow f_1 = f_3) \right) \wedge \\
\left( (x_2 = x_3 \rightarrow f_2 = f_3) \right) \rightarrow \\
((x_1 \neq x_2) \lor (f_1 = f_2) \lor (f_1 \neq f_3))
\]
Given: a formula $\varphi^{UF}$ with uninterpreted functions
For each function in $\varphi^{UF}$:

1. Number function instances (from the inside out)

\[ F_1(a) = F_2(b) \]
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) $\quad \rightarrow \quad F_1(a) = F_2(b)$
  2. Replace each function instance $F_i$ with an expression $F_i^*$ $\quad \rightarrow \quad F_1^* = F_2^*$
Transforming UF$s$ 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) $\rightarrow F_1(a) = F_2(b)$
  2. Replace each function instance $F_i$ with an expression $F_i^*$ $\rightarrow F_1^* = F_2^*$

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

$$f_1 = \begin{cases} 
  \text{case } a = b : f_1 \\
  \text{true} : f_2 
\end{cases}$$
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:**
  
  \[ a = b \to F(G(a)) = F(G(b)) \]

- **Number the instances:**
  
  \[ a = b \to F_1(G_1(a)) = F_2(G_2(b)) \]

- **Replace each function application with an expression:**
  
  \[ a = b \to F^*_1 = F^*_2 \]

  where

  \[ F^*_1 = f_1 \]
  \[ F^*_2 = \begin{cases} 
  G^*_1 = G^*_2 : f_1 \\
  \text{true} : f_2 
  \end{cases} \]
  \[ G^*_1 = g_1 \]
  \[ G^*_2 = \begin{cases} 
  \text{case} a = b : g_1 \\
  \text{true} : g_2 
  \end{cases} \]
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 $\varphi^{UF}$:

1. Transform it into Equality Logic formula $\varphi^E$
2. If $\varphi^E$ is unsatisfiable, return 'Unsatisfiable'
3. Else return 'Don’t know'
Using uninterpreted functions in proofs

- **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.
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:

\[ u_1 = x_1 + y_1; \]
\[ u_2 = x_2 + y_2; \]
\[ z = u_1 \cdot u_2; \]
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:

\[ u_1 = x_1 + y_1; \]
\[ u_2 = x_2 + y_2; \]
\[ z = u_1 \cdot u_2; \]

• We need to prove that:

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

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

$$\varphi^E = \left( (x_1 = x_2 \land y_1 = y_2 \implies f_1 = f_2) \land (u_1 = f_1 \land u_2 = f_2 \implies g_1 = g_2) \land ((u_1 = f_1 \land u_2 = f_2 \land z = g_1) \implies z = g_2) \right)$$
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:

  \[
  \begin{array}{c|c}
  \text{Left} & \text{Right} \\
  x + x & 2x \\
  \end{array}
  \]
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 \textit{commutativity}:

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

Fix by adding:

\[(x_1 + y_1 = y_1 + x_1) \land (x_2 + y_2 = y_2 + x_2)\]
Uninterpreted functions: usability

- 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!
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:

\[
\begin{align*}
\text{x} &= \text{x} + \text{y}; \\
\text{x}_1 &= \text{x}_0 + \text{y}_0; \\
\text{x} &= \text{x} \times 2; \\
\text{x}_2 &= \text{x}_1 \times 2; \\
\text{a}[i] &= 100; \\
\text{a}_1[i_0] &= 100;
\end{align*}
\]
Static Single Assignment (SSA) form

→ see compiler class

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

Example:

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

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

Read assignments as equalities

Generate constraints by simply conjoining these equalities

Example:

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

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

int main() {
    int x, y, z;
    y₁=8;
    if(x₀)
        y₂=y₁-1;
    else
        y₃=y₁+1;
    y₄=\phi(y₂, y₃);
    z₁=y₄+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;
    y1=8;
    if(x0)
        y2=y1-1;
    else
        y3=y1+1;
    y4=\phi(y2, y3);
    z1=y4+1;
}
```

```c
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
```
SSA for bounded programs

What about loops?
→ We unwind them!

```c
void f(...) {
    ...
    while(cond) {
        BODY;
    }
    ...
    Remainder;
}
```
SSA for bounded programs

What about loops?
→ We **unwind** them!

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

\[
\text{out}_3 = \text{out'}_1
\]
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:

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

\[
out'_1 = (in \cdot in) \cdot in
\]

Prove that both functions return the same value:

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

Static single assignment (SSA) form:

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

\[out_1' = (in \ast in) \ast in\]

With uninterpreted functions:

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

\[out_1' = F(F(in, in), in)\]
Static single assignment (SSA) form:

\[
\begin{align*}
out_1 &= in \\
out_2 &= \underbrace{out_1 \ast in} \\
out_3 &= \underbrace{out_2 \ast in} \\
out_1' &= (in \ast in) \ast in
\end{align*}
\]

With uninterpreted functions:

\[
\begin{align*}
out_1 &= in \\
out_2 &= F(out_1, in) \\
out_3 &= F(out_2, in) \\
out_1' &= F(F(in, in), in)
\end{align*}
\]

With numbered uninterpreted functions:

\[
\begin{align*}
out_1 &= in \\
out_2 &= F_1(out_1, in) \\
out_3 &= F_2(out_2, in) \\
out_1' &= F_4(F_3(in, in), in)
\end{align*}
\]
Example: equivalence of C programs (4/4)

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:

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

Ackermann’s reduction:

\[
\begin{align*}
\varphi^E_a : \text{out}_1 &= \text{in} \\
\varphi^E_b : \text{out}_2 &= f_1 \\
\text{out}_3 &= f_2 \\
\varphi^E_b : \text{out}'_1 &= f_4
\end{align*}
\]
Example: equivalence of C programs (4/4)

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

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

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

\[
\begin{align*}
    \varphi^E_b : 
    out' &= f_4
\end{align*}
\]

The verification condition:
\[
\left[ \left( \left( \begin{array}{c}
    (out_1 = out_2 \rightarrow f_1 = f_2) \land \\
    (out_1 = in \rightarrow f_1 = f_3) \land \\
    (out_1 = f_3 \rightarrow f_1 = f_4) \land \\
    (out_2 = in \rightarrow f_2 = f_3) \land \\
    (out_2 = f_3 \rightarrow f_2 = f_3) \land \\
    (in = f_3 \rightarrow f_3 = f_4)
    \end{array} \right) \land \varphi^E_a \land \varphi^E_b \right] \rightarrow out_3 = out'_1
\]

D. Kroening, O. Strichman (ETH/Technion) Decision Procedures Version 1.0, 2007 38 / 47
Uninterpreted functions: simplifications

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

- Solution: try to \textit{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 $|= o_1 = o_2$.

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

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

4 function instances $\rightarrow$ 6 comparisons
Given $x_1 = x_1', x_2 = x_2', x_3 = x_3'$, prove $\models o_1 = o_2$.

$$o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5$$ \text{ Left} \\
$$o_2 = (x_1' + (b \cdot x_2')) \land b = x_3' + 5$$ \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).
Simplifications (2)

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

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

- Down to 4 comparisons!
Simplifications (2)

\[ o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \]  \hspace{1cm} \text{Left}

\[ o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \]  \hspace{1cm} \text{Right}

- Down to 4 comparisons!
- Another guess: equivalence only depends on \( f_1 = f_3 \) and \( f_2 = f_4 \)
- \textit{Pattern matching} may help here
Simplifications (3)

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

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

Match according to patterns ('signatures')

Down to 2 comparisons!

\[ f_1, f_3 \]

\[ f_2, f_4 \]
Simplifications (4)

\[ o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \] \hspace{2cm} \text{Left}

\[ o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \] \hspace{2cm} \text{Right}

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

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

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

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

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

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

Map \( F_1 \) to \( F_3 \):

\[
\begin{array}{c}
F \\
\downarrow \\
v
\end{array} \quad \begin{array}{c}
v \\
\downarrow \\
v
\end{array}
\]

Map \( F_2 \) to \( F_4 \):

\[
\begin{array}{c}
F \\
\downarrow \\
v
\end{array} \quad \begin{array}{c}
v \\
\downarrow \\
v
\end{array}
\]
The SSA example revisited (2)

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:

\[ \begin{align*}
    out_1 &= in \land \\
    \varphi^E_a : \quad out_2 &= f_1 \land \\
    out_3 &= f_2 \land \\
    \varphi^E_b : \quad out'_1 &= f_4
\end{align*} \]

The verification condition has shrunk:

\[ \left[ \left( (out_1 = in \rightarrow f_1 = f_3) \land (out_2 = f_3 \rightarrow f_2 = f_4) \right) \land \varphi^E_a \land \varphi^E_b \right] \rightarrow out_3 = out'_1 \]
With numbered uninterpreted functions:

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

Bryant’s reduction:

\[
\begin{align*}
\varphi^E_a : & \quad \text{out}_1 = \text{in} \land \\
& \quad \text{out}_2 = f_1 \land \\
& \quad \text{out}_3 = f_2 \\
\varphi^E_b : & \quad \text{out}'_1 = \begin{cases} 
\text{case} & \text{in} = \text{out}_1 : f_1 \\
\text{true} & \text{true} : f_3 
\end{cases} = \text{out}_2 : f_2 \\
\end{align*}
\]

The verification condition:

\[
(\varphi^E_a \land \varphi^E_b) \implies \text{out}_3 = \text{out}'_1
\]
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.