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

- 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 expressive 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

\[
\begin{align*}
\text{formula} & : \text{formula} \lor \text{formula} \\
& \quad | \neg \text{formula} \\
\text{atom} & : \text{term} = \text{term} \\
& \quad | \text{Boolean-variable} \\
\text{term} & : \text{term-variable} \\
& \quad | \text{function (list of terms)}
\end{align*}
\]

The \textit{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 \(\mathbb{N}\).

- The most general axiom for any function is \textit{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).
  - 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

Stage 3

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

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
  \[\Rightarrow \text{We can remove one of the latches}\]

Example: Circuit Transformations

Stage 1

Stage 2

Stage 3

Example: Circuit Transformations

- Example: Circuit Transformations

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

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)
  \[F_2(F_1(x)) = 0\]
  \[f_2 = 0\]
  2. Replace each function instance with a new variable
  \[f_2 = 0\]
  3. Add functional consistency constraint to \(\varphi^{EF}\) for every pair of instances of the same function.

Ackermann’s reduction: Example

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.

- First number the function instances:
  \[x_1 \neq x_2 \vee F_1(x_1) = F_2(x_2) \vee F_1(x_1) \neq F_3(x_3)\]
- Replace each function with a new variable:
  \[x_1 \neq x_2 \vee f_1 = f_2 \vee f_1 \neq f_3\]
- Add functional consistency constraints:
  \[\begin{align*}
  &\left( (x_1 = x_2 \rightarrow f_1 = f_2) \land (x_1 = x_2 \rightarrow f_1 = f_3) \right) \\
  &\left( (x_2 = x_3 \rightarrow f_2 = f_3) \land (x_2 = x_3 \rightarrow f_2 = f_1) \right)
  \end{align*}\]

\[\begin{align*}
&\left( (x_1 \neq x_2) \lor (f_1 = f_2) \lor (f_1 \neq f_3) \right)
\end{align*}\]
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

- 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

- 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 $\phi_{UF}$:
    1. Transform it into Equality Logic formula $\phi^{EL}$
    2. If $\phi^{EL}$ is unsatisfiable, return ‘Unsatisfiable’
    3. Else return ‘Don’t know’

Transforming UFs to Equality Logic using Bryant’s reduction

- Given: a formula $\phi_{UF}$ with uninterpreted functions
- For each function in $\phi_{UF}$:
  1. Number function instances (from the inside out) $F_1(a) = F_2(b)$
  2. Replace each function instance $F_i$ with an expression $F^*_i$ $F^*_1 = F^*_2$

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
  
  \[
  \begin{align*}
  F^*_1 &= f_1 \\
  F^*_2 &= \begin{cases} 
  G^*_1 = G^*_2 : f_1 & \text{case } a = b \\
  \text{true : } f_2 & \text{true : } f_2
  \end{cases} \\
  G^*_1 &= g_1 \\
  G^*_2 &= \begin{cases} 
  a = b : g_1 \\
  \text{true : } g_2
  \end{cases}
  \end{align*}
  \]
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:
  \[
  u_1 = x_1 + y_1; \\
  u_2 = x_2 + y_2; \\
  z = u_1 \cdot u_2; \\
  \]

- We need to prove that:
  \[
  (u_1 = x_1 + y_1 \land u_2 = x_2 + y_2 \land z = u_1 \cdot u_2) \rightarrow (z = (x_1 + y_1) \cdot (x_2 + y_2))
  \]

Example: Translation Validation

- Claim: \( \varphi^{UF} \) is valid

- We will prove this by reducing it to an Equality Logic formula
  \[
  \varphi^E = \left\{ (x_1 = x_2 \land y_1 = y_2 \rightarrow f_1 = f_2) \land \\
  ((u_1 = f_1 \land u_2 = f_2 \land z = g_1) \rightarrow z = g_2) \right\}
  \]

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:

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

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!

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; 
}
```

- 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

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; 
}
```

- 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.

From programs to equations

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; 
}
```

- 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

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; 
}
```
SSA for bounded programs

What about loops?
→ We unwind them!

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

SSA for bounded programs: CBMC

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;
}
```

Static single assignment (SSA) form:
```
out_3 = in * in
out_2 = out_1 * in
out_1 = 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:
\[
\begin{align*}
\text{out}_1 &= \text{in} \\
\text{out}_2 &= \text{out}_1 \land \text{in} \\
\text{out}_3 &= \text{out}_2 \land \text{in}
\end{align*}
\]

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

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

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

Example: equivalence of C programs (4/4)

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

Ackermann’s reduction:
\[
\phi^E : \text{out}_2 = f_1 \\
\phi^E : \text{out'}_3 = f_4
\]

The verification condition:
\[
\begin{align*}
\left( \begin{array}{c}
\text{out}_1 = \text{out}_2 \rightarrow f_1 = f_2 \\
\text{out}_1 = \text{in} \rightarrow f_1 = f_3 \\
\text{out}_1 = f_3 \rightarrow f_1 = f_4 \\
\text{out}_2 = \text{in} \rightarrow f_2 = f_3 \\
\text{out}_2 = \text{out}_2 \rightarrow f_2 = f_4 \\
\text{out}_3 = f_2 \\
\text{out}_3 = f_3
\end{array} \right) \land \phi^E \land \phi^E \implies \text{out}_3 = \text{out'}_3
\end{align*}
\]

Simplifications (1)

- Given \( x_1 = x'_1, x_2 = x'_2, x_3 = x'_3 \), prove \( \models o_1 = o_2 \).
  \[
  o_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \quad \text{Left}
  \]
  \[
  o_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \quad \text{Right}
  \]
- 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)

\[
\begin{align*}
o_1 &= (x_1 + (a \cdot x_2)) \land a = x_3 + 5 & \text{Left} \\
o_2 &= (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 & \text{Right}
\end{align*}
\]

- 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)

\[
\begin{align*}
o_1 &= (x_1 + (a \cdot x_2)) \land a = x_3 + 5 & \text{Left} \\
o_2 &= (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 & \text{Right}
\end{align*}
\]

Match according to patterns (‘signatures’)

- Down to 2 comparisons!
Simplifications (4)

\[ a_1 = (x_1 + (a \cdot x_2)) \land a = x_3 + 5 \]  
\[ f_1 \]

Left

\[ a_2 = (x'_1 + (b \cdot x'_2)) \land b = x'_3 + 5 \]  
\[ f_2 \]

Right

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

The SSA example revisited (1)

With numbered uninterpreted functions:

out1 = in \land
out2 = F1(out1, in) \land
out3 = F2(out2, in)

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

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

The SSA example revisited (2)

With numbered uninterpreted functions:

out1 = in \land
out2 = F1(out1, in) \land
out3 = F2(out2, in)

Ackermann’s reduction:

\[ \varphi_E^* : \quad \text{out}_1 = f_3 \land \quad \text{out'}_1 = f_4 \]

\[ \text{out}_3 = f_2 \]

The verification condition has shrunk:

\[
\begin{aligned}
(\text{out}_1 = \text{in} \land \text{out}_2 = \text{f}_3) \land (\varphi_E^* \land \varphi_E^3) \quad \rightarrow \quad \text{out}_3 = \text{out'}_3
\end{aligned}
\]

So is Equality Logic with UFs interesting?

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