Decision Procedures

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):
  \[ x_1 = x_2 \land (x_2 = x_3 \lor ((x_1 \neq x_3) \lor \neg b \lor x_1 \neq 2)) \]

Syntax of Equality Logic

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

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

- 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

```
formula : formula _ formula
| ¬formula
| atom

atom : term = term
| Boolean-variable

term : term-variable
| function ( list of terms )
```

- 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, N, is a mapping from N x N to N.

Uninterpreted Functions

- 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!
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:
  \[
  x_1 = x_1' \land \ldots \land x_n = x_n' \implies f(x_1, \ldots, x_n) = f(x_1', \ldots, x_n')
  \]
- 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

\[
\begin{align*}
  f(x, y) &:= x \lor y \\
  R'_1 &= f(R_1, I)
\end{align*}
\]
Example: Circuit Transformations

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

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 circuit only uses one of \( L_3 \) or \( L_4 \), never both, so can remove one of the latches.
Example: Circuit Transformations

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

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

- 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 \( \phi^{\text{UF}} \) with uninterpreted functions
- For each function in \( \phi^{\text{UF}} \):
  - Number function instances
  - Replace each (top level) function instance with a new variable
  - Add functional consistency constraint for every pair of instances of the same function.

Suppose we want to check validity of:

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

- Number function instances
- Replace the function instances: \( x_1 \neq x_2 \lor f_1 = f_2 \lor f_1 \neq f_3 \)
- Add functional consistency constraints:

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

Transforming UFs to Equality Logic using Bryant’s reduction

- Given: a formula \( \phi^{\text{UF}} \) with uninterpreted functions
- For each function in \( \phi^{\text{UF}} \):
  - Number function instances
  - Replace each (top level) function instance with a new variable
  - Add functional consistency constraint for every pair of instances of the same function.

\[
F^{\text{UF}}(F(x)) = 0
\]

\[
F_1^{\text{UF}}(F(x)) = 0
\]

\[
F_2^{\text{UF}} = 0
\]

\[
\begin{align*}
& \text{case: } f_i = x \land f_i = 0 \lor (f_i \neq x \land f_i = 0) \\
& \text{true: } f_i
\end{align*}
\]
### Transforming UFs to Equality Logic using Bryant’s reduction

- Case expression is a "macro" or abbreviation for what to substitute for based on assumptions about the relationships among the arguments $x_i$ to the function occurrences $F_i$.

From $\varphi$, produces:

\[
\begin{aligned}
\text{case: } & x_1 = x_i : f_1 \\
& x_2 = x_i : f_2 \\
& \vdots \\
& x_{i-1} = x_i : f_{i-1} \\
\text{true : } f_i
\end{aligned}
\]

\[
\begin{aligned}
q[F_i^*=f_j] \lor (x_1 = x_i \land q[F_i^*=f_j]) \\
\lor (x_1 \neq x_i \land x_2 = x_i \land q[F_i^*=f_j]) \\
\lor (x_1 \neq x_i \land x_2 \neq x_i \land x_3 = x_i \land q[F_i^*=f_j]) \\
\lor q[F_i^*=f_j]
\end{aligned}
\]

### Example of Bryant’s reduction (multiple UFs)

\[
\begin{aligned}
a = b \rightarrow F(G(a)) = F(G(b)) \\
& G_1^* = \text{case: } a = b \rightarrow \begin{cases} 
G_1^* = G_2^* : f_1 \\
\text{true : } f_2
\end{cases}
\end{aligned}
\]

\[
\begin{aligned}
a = b & \rightarrow F_1^* = F_2^*, \text{ where } F_1^* = \begin{cases} 
\text{true : } f_i
\end{cases}
\text{ and } F_i^* = f_i \\
G_1^* & = \begin{cases} 
\text{case: } a = b : g_1 \\
\text{true : } g_2
\end{cases}
\text{ and } G_i^* = g_i
\end{aligned}
\]

\[
\begin{aligned}
(a = b \land g_1 = g_3) \land (a = b \rightarrow f_1 = f_2) \lor (a = b \land g_2 = g_3) \land (a = b \rightarrow f_1 = f_2) \\
\lor (a \neq b \land g_1 = g_3) \land (a = b \rightarrow f_1 = f_2) \\
\lor (a \neq b \land g_2 = g_3) \land (a = b \rightarrow f_1 = f_2)
\end{aligned}
\]

### Using uninterpreted functions in proofs

- Uninterpreted functions give us the ability to represent an abstract view $f^A$ of a function $f$.
- It over-approximates the concrete system.
  - $1 + 1 = 1$ is a contradiction
  - But $F(1, 1) = 1$ is satisfiable!
- Conclusion: If we prove $\varphi(f^A)$ is satisfiable, we have to be careful – we cannot conclude that $\varphi(f)$ is. But, if we prove $\varphi(f^A)$ is unsatisfiable, we can conclude that $\varphi(f)$ is also.

### 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 $\varphi^{UF}$:
  - Transform it into an Equality Logic formula $\varphi^{E}$ (using Ackerman’s or Bryant’s algorithms)
  - If $\varphi^{E}$ is unsatisfiable, return ‘Unsatisfiable’
  - 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?

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 \( \varphi \), where:
  \[ \varphi = (u_1 = x_1 + y_1 \land u_2 = x_2 + y_2 \land z = u_1 \cdot u_2) \rightarrow (z = (x_1 + y_1) \cdot (x_2 + y_2)) \]

Using uninterpreted functions in proofs

- (most common use) 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 use) Proving properties that do not rely on the exact functionality of some of the functions

Example: Translation Validation

- Claim: \( q^{UF} \) is valid
- We will prove this by reducing it to the Equality Logic formula:
  \[ q^{E} = \{(x_1 = x_2 \land y_1 = y_2 \rightarrow f_1 = f_2) \land (u_1 = f_1 \land u_2 = f_2 \rightarrow g_1 = g_2) \rightarrow (u_1 = f_1 \land u_2 = f_2 \land z = g_1) \rightarrow (z = g_2)\} \]
Uninterpreted functions: usability

- Good: each function on the left can be mapped to a function on the right with equivalent arguments
- Bad: almost all other cases
  - Example:
    | Left | Right |
    |------|-------|
    | x + x | 2x    |

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:

\[(x_1 = x_2 \land y_1 = y_3) \rightarrow (x_1 + y_1 = y_2 + x_2)\]

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;
}
```

```c
int power3 new(int in) {
    out = (in*in)*in;
    return out;
}
```

- These two functions return the same value regardless if boolean operator 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

- Idea: Rename variables such that each variable is assigned exactly once (see compilers notes)
- Example:

  \[
  x = x + y; \quad x_1 = x_0 + y_0; \\
  x = x * 2; \quad x_2 = x_1 * 2; \\
  a[i] = 100; \quad a_1[i_0] = 100; 
  \]

- Read assignments as equalities; generate constraints by simply conjoining these equalities

  \[
  (x_1 = x_0 + y_0) \land (x_2 = x_1 * 2) \land (a_1[i_0] = 100)
  \]

SSA for bounded programs

- What about loops?
- We unwind them!

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

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

SSA for bounded programs

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

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

```c
... main(...) {
  ... 
  while(cond){
    BODY;
    while(cond){
      BODY;
    }
  }
  ... 
  Remainder;
}
```

```c
\[(y_1 = 8) \land (y_2 = y_1 - 1) \land (y_3 = y_1 + 1) \land (y_4 = 0) \land (y_5 = y_2, y_6 = y_3) \land (z = y_1 + 1)\]
```

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
    - 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;
}
```

**SSA form:**

\[
\begin{align*}
\text{out}_1 &= \text{in}_0 \\
\text{out}_2 &= \text{out}_1 \cdot \text{in}_0 \\
\text{out}_3 &= \text{out}_2 \cdot \text{in}_0
\end{align*}
\]

Prove that both functions return the same value:

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

Example: equivalence of C programs (3/4)

**SSA form:**

\[
\begin{align*}
\text{out}_1 &= \text{in}_0 \\
\text{out}_2 &= \text{out}_1 \cdot \text{in}_0 \\
\text{out}_3 &= \text{out}_2 \cdot \text{in}_0
\end{align*}
\]

With uninterpreted functions:

\[
\begin{align*}
\text{out}_1 &= \text{in}_0 \\
\text{out}_2 &= F(\text{out}_1, \text{in}_0) \\
\text{out}_3 &= F(\text{out}_2, \text{in}_0)
\end{align*}
\]

Number the function applications:

\[
\begin{align*}
\text{out}_1 &= \text{in}_0 \\
\text{out}_2 &= F(\text{out}_1, \text{in}_0) \\
\text{out}_3 &= F(\text{out}_2, \text{in}_0)
\end{align*}
\]

Replace function applications with variables:

\[
\begin{align*}
\phi_1: & \text{out}_1 = \text{in}_0 \\
\phi_2: & \text{out}_1 = \text{f}_1 \\
\phi_3: & \text{out}_2 = \text{f}_2 \\
\phi_4: & \text{out}_3 = \text{f}_3
\end{align*}
\]

Verification condition:

\[
\begin{align*}
(\phi_1 \land \phi_2) \land (\phi_3 \land \phi_4) \rightarrow \phi
\end{align*}
\]

Example: equivalence of C programs (4/4)

```c
int power3_new (int in) {
    out = \text{in}^3;
    return out;
}
```

**SSA form:**

\[
\begin{align*}
\text{out}_1 &= \text{in}_0 \\
\text{out}_2 &= \text{out}_1 \cdot \text{in}_0 \\
\text{out}_3 &= \text{out}_2 \cdot \text{in}_0
\end{align*}
\]

With uninterpreted functions:

\[
\begin{align*}
\text{out}_1 &= \text{in}_0 \\
\text{out}_2 &= F(\text{out}_1, \text{in}_0) \\
\text{out}_3 &= F(\text{out}_2, \text{in}_0)
\end{align*}
\]

Number the function applications:

\[
\begin{align*}
\text{out}_1 &= \text{in}_0 \\
\text{out}_2 &= F(\text{out}_1, \text{in}_0) \\
\text{out}_3 &= F(\text{out}_2, \text{in}_0)
\end{align*}
\]

Replace function applications with variables:

\[
\begin{align*}
\phi_1: & \text{out}_1 = \text{in}_0 \\
\phi_2: & \text{out}_2 = \text{f}_1 \\
\phi_3: & \text{out}_3 = \text{f}_2
\end{align*}
\]

Verification condition:

\[
\begin{align*}
(\phi_1 \land \phi_2) \land (\phi_3 \land \phi_4) \rightarrow \phi
\end{align*}
\]
\[(x_1 = x_2 \land y_1 = y_2 \rightarrow f_1 = f_2) \land (u_1 = f_1 \land u_2 = f_2 \rightarrow g_1 = g_2)\]
\[\rightarrow (u_1 = f_1 \land u_2 = f_2 \land z = g_1) \rightarrow (z = g_2)\]

\[\phi \rightarrow \phi\]

\[x = x + y;\]
\[x = x \times 2;\]
\[a[i] = 100;\]
\[x_1 = x_0 + y_0;\]
\[x_2 = x_1 \times 2;\]
\[a_1[i_0] = 100;\]

Syntactic vs. Semantic splits

- Now we start looking at methods that split the search space instead. This is called **semantic splitting**.

- SAT is a very good engine for performing semantic splitting, due to its ability to guide the search, prune the search-space etc.