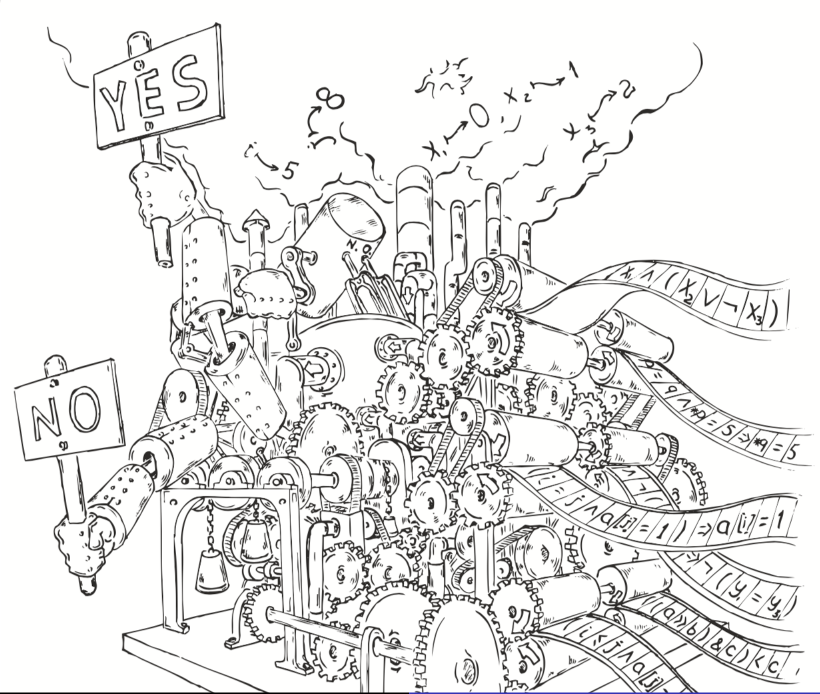

YES
NO

SAT/SMT solvers
4. Equalities and Uninterpreted Functions

Roman Kholin

Lalambda

Tbilisi, 2023

Definitions

Equality logic

An equality logic formula is defined by the following grammar:

- $formula : formula \wedge formula \mid \neg formula \mid (formula) \mid atom$
- $atom : term = term$
- $term : identifier \mid constant$

where the identifier s are variables defined over a single infinite domain such as the Reals or Integers. Constants are elements from the same domain as the identifiers.

Removing the constants

Theorem

Given an equality logic formula φ^E, there is an algorithm that generates an equisatisfiable formula $\varphi^{E'}$ without constants, in polynomial time.

1. $\varphi^{E'} := \varphi^{E}$.
2. In $\varphi^{E'}$, replace each constant c_i, $1 \le i \le n$, with a new variable C_{c_i}.
3. For each pair of constants c_i, c_j such that $1 \le i < j \le n$, add the constraint $C_{c_i} \neq C_{c_j}$ to $\varphi^{E'}$.

Definitions

Equality logic with Uninterpreted Functions (EUF)

An equality logic formula with uninterpreted functions and uninterpreted predicates is defined by the following grammar:

- $formula : formula \wedge formula \mid \neg formula \mid (formula) \mid atom$
- $atom : term = term \mid predicate\text{-}symbol$ (list of terms)
- $term : identifier \mid predicate\text{-}symbol$ (list of terms)

$$F(x) = F(G(y)) \vee x + 1 = y$$
$$F(x) = F(G(y)) \vee PLUS(x, 1) = y$$

Functional consistency

Instances of the same function return the same value if given equal arguments
$\models \varphi^{UF} \Rightarrow \models \varphi$

Equivalence of programs

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

(a)

```
int power3_new(int in)
{
  int out_b;

  out_b = (in * in) * in;

  return out_b;
}
```

(b)

Equivalence of Programs

```
int mul3(struct list *in)
{
 int i, out_a;
 struct list *a;
 a = in;
 out_a = in -> data;
 for (i = 0; i < 2; i++) {
   a = a -> n;
   out_a= out_a * a -> data;
 }
 return out_a;
}
```

(a)

```
int mul3_new(struct list *in)
{
  int out_b;

  out_b =
    in -> data *
    in -> n -> data *
    in -> n -> n -> data;

  return out_b;
}
```

(b)

Equivalence of Programs

```
int mul3(struct list *in)
{
 int i, out_a;
 struct list *a;
 a = in;
 out_a = in -> data;
 for (i = 0; i < 2; i++) {
   a = a -> n;
   out_a= out_a * a -> data;
 }
 return out_a;
}
```

(a)

```
int mul3_new(struct list *in)
{
  int out_b;

  out_b =
    in -> data *
    in -> n -> data *
    in -> n -> n -> data;

  return out_b;
}
```

(b)

$$
\begin{aligned}
&a0\_a = in0\_a && \wedge\\
&out0\_a = list\_data(in0\_a) && \wedge\\
&a1\_a = list\_n(a0\_a) && \wedge\\
&out1\_a = G(out0\_a, list\_data(a1\_a)) && \wedge\\
&a2\_a = list\_n(a1\_a) && \wedge\\
&out2\_a = G(out1\_a, list\_data(a2\_a))
\end{aligned}
$$

$$
\begin{aligned}
&out0\_b = G(G(list\_data(in0\_b),\\
&list\_data(list\_n(in0\_b)),\\
&list\_data(list\_n(list\_n(in0\_b)))))
\end{aligned}
$$

Congruence closure

1. Build congruence-closed equivalence classes.
   (a) Initially, put two terms t_1, t_2 (either variables or uninterpreted-function instances) in their own equivalence class if $(t_1 = t_2)$ is a predicate in φ^{UF}. All other variables form singleton equivalence classes.
   (b) Given two equivalence classes with a shared term, merge them. Repeat until there are no more classes to be merged.
   (c) Compute the *congruence closure*: given two terms t_i, t_j that are in the same class and that $F(t_i)$ and $F(t_j)$ are terms in φ^{UF} for some uninterpreted function F, merge the classes of $F(t_i)$ and $F(t_j)$. Repeat until there are no more such instances.
2. If there exists a disequality $t_i \neq t_j$ in φ^{UF} such that t_i and t_j are in the same equivalence class, return "Unsatisfiable". Otherwise return "Satisfiable".

Example

$$(x_1 = x_2) \land (x_2 = x_3) \land (x_4 = x_5) \land (x_5 \neq x_1) \land (F(x_1) \neq F(x_3))$$

Functional consistency is Not Enough

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

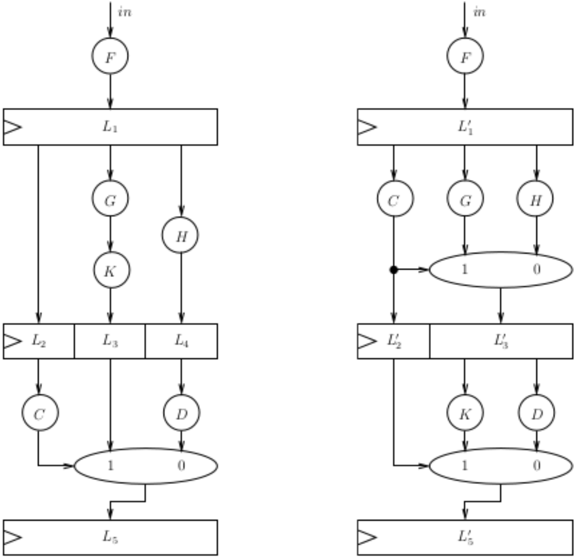
Functional consistency is Not Enough

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

Abstraction–refinement loop

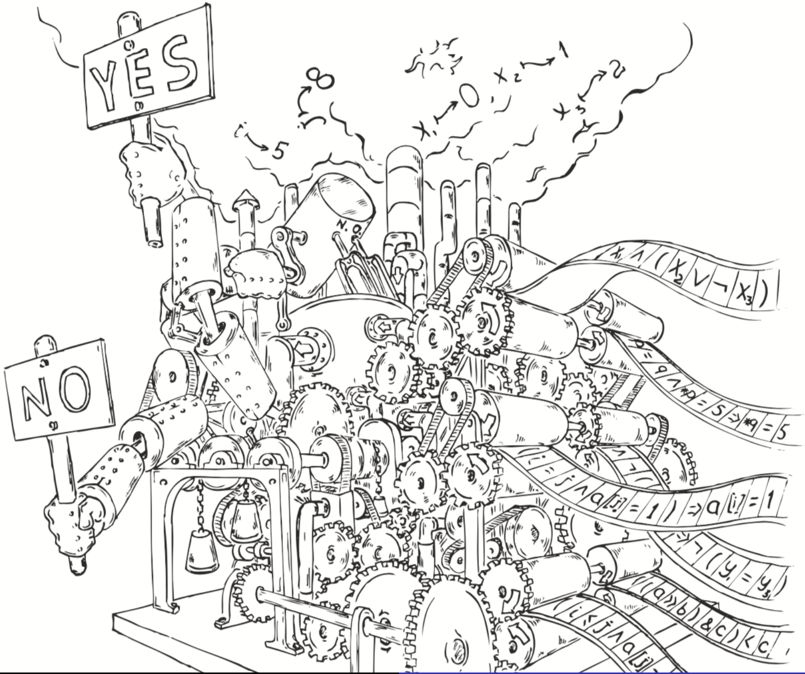
1. $\varphi' := T(\varphi)$ (T is an abstraction function)
2. If φ is valid then return «Valid»
3. If $\varphi' = \varphi$ then return «Not valid»
4. Refine φ by adding more constraints or by replacing uninterpreted functions with their original interpreted versions
5. Return to step 2

Examples



Examples

$$z = (x_1 + y_1) * (x_2 + y_2)$$
$$u_1 = x_1 + y_1; u_2 = x_2 + y_2; z = u_1 * u_2$$


YES
NO