Teoria das funções não interpretadas, $\mathcal{EUF}$

- termos funcionais à teoria da igualdade

$$\varphi^{uf} := \varphi^{uf} \wedge \varphi^{uf} \mid \neg\varphi^{uf} \mid t = t$$
$$t := x|c|F(t_1, \ldots, t_n)$$

- apenas se exige a congruência funcional

$$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \rightarrow F(x_1, \ldots, x_n) = F(y_1, \ldots, y_n)$$

- Funções de uma dada teoria podem ser substituídas por funções não interpretadas simplificando as provas (de validade) embora não se preserve a equivalência. Temos

$$\models \varphi^{uf} \implies \models \varphi$$

- mas se $\not\models \varphi^{uf}$ nada se pode concluir.

Exemplo: equivalência de programas

```
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:

$out_1 = in \wedge$
$out_2 = out_1 * in \wedge$
$out_3 = out_2 * in$

$out'_1 = (in * in) * in$

Prove that both functions return the same value:

$$out_3 = out'_1$$

Static single assignment

1. Remove the variable declarations and `return` statements.
2. Unroll the `for` loop.
3. Replace the left-hand side variable in each assignment with a new auxiliary variables

4. Wherever a variable is read (referred to in an expression), replace it with the auxiliary variable that replaced it in the last place where it was assigned.

5. Conjoin all program statements.

No exemplo, dados os dois programas obtemos duas fórmulas φ_1 e φ'_1 e pretende-se que

$$\varphi_1 \wedge \varphi'_1 \implies out_3 = out'_1$$

Utilização de funções não interpretadas

Static single assignment (SSA) form:

$$\begin{array}{ll} out_1 = in \wedge & \\ out_2 = out_1 * in \wedge & out'_1 = (in * in) * in \\ out_3 = out_2 * in & \end{array}$$

With uninterpreted functions:

$$\begin{array}{ll} out_1 = in \wedge & \\ out_2 = F(out_1, in) \wedge & out'_1 = F(F(in, in), in) \\ out_3 = F(out_2, in) & \end{array}$$

A vantagem é que é mais fácil provar a validade de funções não interpretadas. No exemplo substituímos a multiplicação $*$ por uma função não interpretada F e obtemos φ_1^{uf} e $\varphi_1'^{uf}$.

Algoritmo de decisão para conjunções de igualdades com funções não interpretadas

Vamos apenas considerar funções unárias (mas o algoritmo aplica-se a funções de qualquer aridade)

Input: Uma conjunção de literais φ^{uf}
Output: Satisfazível ou Não Satisfazível

1. Construir classes de equivalência fechadas para a congruência funcional

   a) Se $t_1 = t_2 \in \varphi^{uf}$ colocar t_1 and t_2 na mesma classe de equivalência. Colocar todos os outros termos em classes de equivalência singulares.

   b) Se duas classes partilham um termo, unir as classes. Repetir até não haver mais classes para juntar.

   c) Calcular o fecho da congruência funcional: se t_1 and t_2 estão na mesma classe and se $F(t_1)$ and $F(t_2)$ são termos de φ^{uf} então juntar as classes de $F(t_1)$ and $F(t_2)$. Repetir até não haver mais classes para juntar.

2. Se existe uma desigualdade $t_1 \neq t_2$ em φ^{uf} tal que t_1 and t_2 pertencem à mesma classe de equivalência então retornar *Não Satisfazível*, senão retornar *Satisfazível*.

Ex. 21.1. *Seja* φ^{uf} *a conjunção*

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

Inicialmente as classes de equivalência são:

$$\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{F(x_3)\}, \{F(x_1)\}$$

Juntámos termos nas classe

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_3)\}, \{F(x_1)\}$$

Pelo fecho da congruência temos:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1), F(x_3)\}$$

Finalmente como $F(x_1) \neq F(x_3) \in \varphi^{uf}$*, retornamos* Não Satisfazível.

- Este algoritmo pode ser implementado eficientemente com **union-find**, com complexidade $O(n \log(n))$.
- Para estender o algoritmo de decisão a fórmulas sem quantificadores genéricas (não só conjunções de literais) pode-se usar o algoritmo DPLL(T) ou variantes (*lazy*); ou algoritmos que reduzem toda a fórmula φ^{uf} a uma fórmula proposicional equisatisfazível (como será visto a seguir) (*eager*).

Redução de funções não interpretadas a lógica equacional (Ackermann)

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.

1. First number the function instances:

$$x_1 \neq x_2 \vee F_1(x_1) = F_2(x_2) \vee F_1(x_1) \neq F_3(x_3)$$

2. Replace each function with a new variable:

$$x_1 \neq x_2 \vee f_1 = f_2 \vee f_1 \neq f_3$$

3. Add functional consistency constraints:

$$\left(\begin{array}{ll} (x_1 = x_2 \rightarrow f_1 = f_2) & \wedge \\ (x_1 = x_3 \rightarrow f_1 = f_3) & \wedge \\ (x_2 = x_3 \rightarrow f_2 = f_3) & \end{array} \right) \rightarrow$$

$$((x_1 \neq x_2) \vee (f_1 = f_2) \vee (f_1 \neq f_3))$$

Redução de Ackermann

Input:	Um fórmula de $\mathcal{EUF}$ φ^{uf} and m instâncias de uma função F (unária)
Output:	Uma fórmula da lógica equacional φ^{E} tal que φ^{E} é válida se and só se φ^{uf} é válida

1. Atribuir índices às instâncias da função F, F_i. Sejam $arg(F_i)$ o seus argumentos.
2. Seja $flat^E = \mathcal{T}(\varphi^{uf})$, onde $\mathcal{T}$ é uma função que substitui cada instância F_i por uma variável nova f_i. Se houver funções imbricadas, fica a variável da função mais externa.
3. Seja
$$FC^E := \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} \mathcal{T}(arg(F_i)) = \mathcal{T}(arg(F_j)) \implies f_i = f_j$$
4. Seja $\varphi^E := FC^E \implies flat^E$

Exemplo: programas equivalentes (cont)

With numbered uninterpreted functions:

$$\begin{array}{ll} out_1 = in \wedge & \\ out_2 = F_1(out_1, in) \wedge & out'_1 = F_4(F_3(in, in), in) \\ out_3 = F_2(out_2, in) & \end{array}$$

Ackermann's reduction:

$$\begin{array}{lll} & out_1 = in \wedge & \\ \varphi_a^E : & out_2 = f_1 \wedge & \varphi_b^E : out'_1 = f_4 \\ & out_3 = f_2 & \end{array}$$

The verification condition:

$$\left[\left(\begin{array}{lll} (out_1 = out_2 & \rightarrow f_1 = f_2) & \wedge \\ (out_1 = in & \rightarrow f_1 = f_3) & \wedge \\ (out_1 = f_3 & \rightarrow f_1 = f_4) & \wedge \\ (out_2 = in & \rightarrow f_2 = f_3) & \wedge \\ (out_2 = f_3 & \rightarrow f_2 = f_3) & \wedge \\ (in = f_3 & \rightarrow f_3 = f_4) & \end{array} \right) \wedge \varphi_a^E \wedge \varphi_b^E \right] \longrightarrow out_3 = out'_1$$

Exemplo

Considera a fórmula φ dada por

$$x_1 = x_2 \implies F(F(G(x_1)))) = F(F(G(x_2)))$$

Considera as seguintes variávies proposicionais g_1, g_2, f_1, f_2, f_3, e f_4

$$x_1 = x_2 \implies \underbrace{F(\overbrace{G(x_1)}^{g_1})}_{f_2} = \underbrace{F(\overbrace{G(x_2)}^{g_2})}_{f_4}$$

então $flat^E : x_1 = x_2 \implies f_2 = f_4$ e FC^E é

$$\begin{aligned}
x_1 = x_2 &\implies g_1 = g_2 \\
g_1 = f_1 &\implies f_1 = f_2 \\
g_2 = f_3 &\implies f_3 = f_4 \\
g_1 = g_2 &\implies f_1 = f_3 \\
g_1 = f_3 &\implies f_1 = f_4 \\
f_1 = g_2 &\implies f_2 = f_3 \\
f_1 = f_3 &\implies f_2 = f_4 \\
g_2 = f_3 &\implies f_1 = f_4
\end{aligned}$$

Então temos $\varphi^E = FC^E \implies flat^E$.

Exerc. 21.1. *Na compilação de programas é necessário fazer transformação de programas (do original até ao program executável). Por exemplo a instrução:*

$$z = (x_1 + y_1) * (x_2 + y_2)$$

Pode compilar para

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

A condição de verificação que garante a correção é

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

Obtem uma fórmula da lógica da igualdade da condição abstraindo a uma fórmula com funções não interpretadas e aplicando a redução de Ackermann. $\diamond$

Algoritmo "eager" para lógica equacional

Vamos ver como construir uma fórmula da lógica proposicional equisatisfazível a uma fórmula da lógica equacional sem quantificadores. O resolutor SAT é assim chamado apenas uma vez. O algoritmo apresentado não será muito eficiente mas poderá ser optimizado de modo a executar em tempo polinomial e obter-se uma fórmula proposicional com um tamanho cúbico no número de variáveis da fórmula equacional.

Conjuntos de literais de igualdades e de desigualdades

Supondo que temos uma fórmula da lógica equacional φ^E (sem constantes) com operações Booleanas mas em NNF (forma normal negativa).

- Seja $E_=$ o conjunto de literais positivos de φ^E
- Seja $E_\neq$ o conjunto de literais negativos de φ^E

Sendo φ^E

$$\begin{array}{l}(x_1 \neq x_2 \vee y_1 \neq y_2 \vee f_1 = f_2) \wedge \\ (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ (u_1 = f_1 \vee u_2 = f_2 \vee z = g_1) \wedge z \neq g_2\end{array}$$

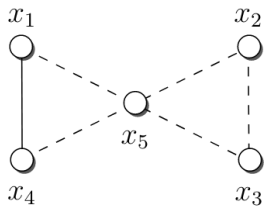
Temos

$$\begin{array}{rcl} E_= & = & \{f_1 = f_2, g_1 = g_2, u_1 = f_1, u_2 = f_2, z = g_1\} \\ E_\neq & = & \{x_1 \neq x_2, y_1 \neq y_2, u_1 \neq f_1, u_2 \neq f_2, z \neq g_2\} \end{array}$$

Grafo equacional (de igualdades)

Dada uma fórmula equacional φ^E em NNF, o *grafo das igualdades (ou equacional)* de φ^E, $G^E(\varphi^E)$ é um grafo (não dirigido) $(V, E_=, E_\neq)$ onde os vértices V são as variáveis em φ^E, as arestas de $E_=$ correspondem aos predicados no conjunto de igualdades e as arestas de $E_\neq$ aos predicados no conjunto de desigualdades.

Por exemplo, para $E_= = \{x_1 = x_5, x_2 = x_3, x_2 = x_5, x_4 = x_5\}$ e $E_\neq = \{x_1 \neq x_4\}$ temos



Como no caso das conjunções de literais, graficamente representamos com linha tracejada as arestas que correspondem a igualdades e a cheio as das desigualdades.

- O grafo equacional $G^E(\varphi^E)$ é uma abstração de φ^E
- Representa na realidade todas as fórmulas que têm os mesmos literais que φ^E
- Como não considera as conectivas Booleanas pode representar tanto fórmulas satisfazíveis como não satisfazíveis
- Por exemplo $x_1 = x_2 \wedge x_1 \neq x_2$ e $x_1 = x_2 \vee x_1 \neq x_2$ são representadas pelo mesmo grafo.

Caminhos de igualdades e de desigualdades

- Um *caminho de igualdades* em G^E é um caminho só com arestas de $E_=$. Se existe um caminho de igualdades entre x e y dizemos que $x =^* y$, para $x, y \in V$.
- Um *caminho de desigualdades* em G^E é um caminho com arestas em $E_=$ e uma aresta de $E_\neq$. Se exite um caminho de desigualdades entre x e y dizemos que $x \neq^* y$ $x, y \in V$.
- Qualquer desses caminhos é *simples* se não tiver ciclos.
- Se $x =^* y$ pode acontecer que x e y tenham que ter o mesmo valor mas não é necessário (porque não temos a estrutura Booleana da fórmula).
- Para $x \neq^* y$ pode ser que x e y tenham que ter valores diferentes
- No exemplo temos que $x_1 =^* x_4$ e $x_1 \neq^* x_4$ mas isso pode não ser inconsistente
- Um *ciclo contraditório* é um ciclo em G^E que tem exactamente uma aresta de $E_\neq$
- Para qualquer $x, y \in V$ num ciclo contraditório tem-se que $x =^* y$ e $x \neq^* y$ (Verifica!).
- A conjunção dos literais correspondentes ao ciclo é não satisfazível.
- No exemplo x_1, x_2, x_4 é um ciclo contraditório

Vamos ver que podemos simplificar as fórmulas se houver literais que não participam em ciclos contraditórios (simples).

Simplificação de fórmulas

Algorithm 11.4.1: SIMPLIFY-EQUALITY-FORMULA

Input: An equality formula φ^{E}
Output: An equality formula $\varphi^{\mathrm{E}\prime}$ equisatisfiable with φ^{E}, with size less than or equal to the length of φ^{E}

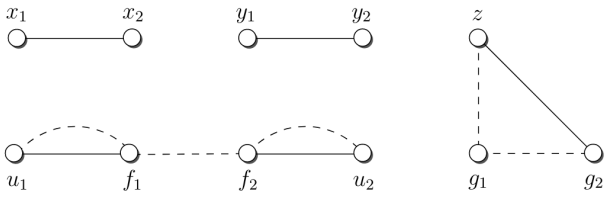
1. Let $\varphi^{\mathrm{E}\prime} := \varphi^{\mathrm{E}}$.
2. Construct the equality graph $G^{\mathrm{E}}(\varphi^{\mathrm{E}\prime})$.
3. Replace each pure literal in $\varphi^{\mathrm{E}\prime}$ whose corresponding edge is not part of a simple contradictory cycle with TRUE.
4. Simplify $\varphi^{\mathrm{E}\prime}$ with respect to the Boolean constants TRUE and FALSE (e.g., replace TRUE $\vee\, \phi$ with TRUE, and FALSE $\wedge\, \phi$ with FALSE).
5. If any rewriting has occurred in the previous two steps, go to step 2.
6. Return $\varphi^{\mathrm{E}\prime}$.

Exemplo

Para

$$\varphi^{E} :=(x_1 \neq x_2 \vee y_1 \neq y_2 \vee f_1 = f_2) \wedge (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge (u_1 = f_1 \vee u_2 = f_2 \vee z = g_1) \wedge z \neq g_2$$

o grafo G^E é



Sendo que $f_1 = f_2$, $x_1 \neq x_2$ e $y_1 \neq y_2$ não fazem parte de ciclos contraditórios, pelo que podem ser substituídos por true.

$$\varphi'^{E} :=(\mathsf{true} \vee \mathsf{true} \vee \mathsf{true}) \wedge (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge (u_1 = f_1 \vee u_2 = f_2 \vee z = g_1) \wedge z \neq g_2$$

Simplificando vem

$$\varphi'^{E} :=(u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge (u_1 = f_1 \vee u_2 = f_2 \vee z = g_1) \wedge z \neq g_2$$

E neste caso se calcularmos o grafo vemos que não podemos simplificar mais. Contudo se os ciclos contraditórios desaparecerem podemos concluir que a fórmula é satisfazível (e só usando simplificação).

Redução para fórmula proposicional (*sparse method*, Bryant et al)

Grafo equacional não polar

Sendo φ^E uma fórmula da lógica equacional, um *grafo equacional não polar* de φ^E, $G^E_{NP}(\varphi^E)$ é um grafo (V, E) onde V corresponde às variáveis de φ^E e as arestas em E a $At(\varphi^E)$, i.e., a todas as fórmulas atómicas (igualdades) de φ^E.

- Notar que $x_1 \neq x_2$ é apenas uma abreviatura de $\neg x_1 = x_2$, logo em G^E_{NP} só o predicado $x_1 = x_2$ estará representado en E.
- Assim em vez de literais apenas consideramos as igualdades (omitindo a sua polaridade).

Transformação lógica equacional para lógica proposicional

Dada φ^E o algoritmo gera duas proposições $e(\varphi^E)$ e $\mathcal{B}_{trans}$ tal que

$$\varphi^E \text{ é satisfazível} \iff e(\varphi^E) \wedge \mathcal{B}_{trans} \text{ é satisfazível}$$

- A fórmula $e(\varphi^E)$ é o *esqueleto proposicional* de φ^E, em que cada predicado $x_i = x_j$ (podemos supor sempre que $i \leq j$) é substituído por uma variável proposicional $e_{i,j}$
- A fórmula $\mathcal{B}_{trans}$ é uma conjunção de implicações, as *restrições transitivas*. Cada restrição está associada a um ciclo no grafo G^E_{NP}. Para um ciclo com n arestas $\mathcal{B}_{trans}$ proíbe uma valorização false a uma das arestas quando todas as outras têm o valor true.

Correção do algoritmo

- Se φ^E é satisfazível então $e(\varphi^E)$ também é satisfazível
- As restrições $\mathcal{B}_{trans}$ são suficientes para garantir que φ^E é satisfazível se $e(\varphi^E)$ for.

Sendo $\varphi^E := x_1 = x_2 \wedge (((x_2 = x_3) \wedge (x_1 \neq x_3)) \vee (x_1 \neq x_2))$ então

$$e(\varphi^E) := e_{1,2} \wedge (((e_{2,3} \wedge (\neg e_{1,3})) \vee (\neg e_{1,2}))$$

As fórmulas $x_1 = x_2$, $x_2 = x_3$ e $x_1 = x_3$ formam um ciclo em G^E_{NP} logo neste caso as restrições transitivas são apenas:

$$\begin{aligned}\mathcal{B}_{trans} :=&((e_{1,2} \wedge e_{2,3}) \implies e_{1,3}) \wedge \\ &(e_{1,2} \wedge e_{1,3}) \implies e_{2,3}) \wedge \\ &(e_{2,3} \wedge e_{1,3}) \implies e_{1,2}).\end{aligned}$$

Complexidade e optimizações

- O algoritmo apresentado pode ter uma complexidade exponencial porque pode ser exponencial o número de ciclos num grafo
- Uma *corda* num ciclo é qualquer aresta que liga dois vértices não adjacentes num ciclo
- Bryant et al provaram que

  > É suficiente adicionar restrições transitivas para ciclos simples sem cordas

- *Grafos cordais* são grafos em que nenhum ciclo de tamanho 4 ou mais é livre de cordas
- Qualquer grafo pode ser transformado num grafo cordal em tempo e espaço polinomial
- Dado que os únicos ciclos sem cordas são os triângulos se aplicarmos o algoritmo a esse grafos podemos fazê-lo em tempo polinomial e obter uma fórmula cujo tamanho é apenas cúbico em relação à original.

Algorithm 11.5.1: EQUALITY-LOGIC-TO-PROPOSITIONAL-LOGIC

Input: An equality formula φ^{E}
Output: A propositional formula equisatisfiable with φ^{E}

1. Construct a Boolean formula $e(\varphi^{\mathrm{E}})$ by replacing each atom of the form $x_i = x_j$ in φ^{E} with a Boolean variable $e_{i,j}$.
2. Construct the nonpolar equality graph $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$.
3. Make $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$ chordal.
4. $\mathcal{B}_{trans} :=$ TRUE.
5. For each triangle $(e_{i,j}, e_{j,k}, e_{i,k})$ in $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$,

$$\begin{aligned} \mathcal{B}_{trans} := \mathcal{B}_{trans} \wedge \\ (e_{i,j} \wedge e_{j,k} \implies e_{i,k}) \wedge \\ (e_{i,j} \wedge e_{i,k} \implies e_{j,k}) \wedge \\ (e_{i,k} \wedge e_{j,k} \implies e_{i,j}) \,. \end{aligned} \tag{11.42}$$

6. Return $e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{trans}$.

Teoria de *arrays*

Já vimos a seguinte axiomática para os *arrays*, em que o fragmento sem quantificadores é decidível:

$$\begin{aligned} & \mathcal{T}_E \\ \forall a, i, j. i = j \rightarrow & \; read(a, i) = read(a, j) \\ \forall a, i, j, v. i = j \rightarrow & \; read(write(a, i, v), j) = v \\ \forall a, i, j, v. \neg(i = j) \rightarrow & \; read(write(a, i, v), j) = read(a, j) \\ \forall a, b. (\forall i. read(a, i) = read(b, i)) \rightarrow & \; a = b \end{aligned}$$

Vamos representar um *array* a como uma *função* de um conjunto de índices duma teoria (tipo) T_I num conjunto de elementos duma (tipo) teoria T_E.

O tipo do array a é

$$T_A = T_I \rightarrow T_E$$

Operações de leitura and escrita

Sendo $a \in T_A$ as operações básicas são:

Leitura $a[i]$ representa $read(a, i)$, isto é, um elemento de T_E que corresponde ao elemento de índice $i \in T_I$ de a

Escrita $a[i \leftarrow e]$ representa $write(a, i, e)$, isto é, $e \in T_E$ denota o valor escrito no elemento de índice i de a.

Lógica de arrays

Supomos que T_I é uma teoria em que o fragmento com quantificadores é decidível (p.e aritmética de Presburger).

Sendo t_I and t_E os termos de T_I and T_E and $id_a \in Var_{array}$ identificadores para os arrays, os termos de T_A são:

$$t_A := id_a \mid t_A[t_I \leftarrow t_E]$$

Estende-se os termos de t_E para incluir elementos de arrays:

$$t_E := t_A[term_I] \mid \cdots$$

Nas fórmulas, acrescenta-se a igualdade de termos de T_A, i.e.

$$\varphi := t_A = t_A \mid \cdots$$

Considera-se $a_1 = a_2$ uma abreviatura de $\forall i.a_1[i] = a_2[i]$. Os axiomas acima podem reescrever-se como:

$$\forall a_1 \in T_A.\forall a_2 \in T_A.\forall i \in T_I.\forall j \in T_I.(a_1 = a_2 \wedge i = j) \implies a_1[i] = a_2[j], \quad (1)$$

$$\forall a \in T_A.\forall e \in T_E.\forall i \in T_I.\forall j \in T_I.a[i \leftarrow e][j] = \begin{cases} e & i = j, \\ a[j] & \text{caso contrário} \end{cases} \quad (2)$$

$$\forall a_1 \in T_A.\forall a_2 \in T_A.(\forall i \in T_I.a_1[i] = a_2[i]) \implies a_1 = a_2. \quad (3)$$

Nota: nesta teoria os arrays têm dimensão não limitada. Na prática a dimensão dum array pode ser introduzida com fórmulas sobre inteiros.

Exemplo

Considera o seguinte triplo de Hoare

$$\{True\}\ \texttt{for}\ i \leftarrow 0\ \texttt{to}\ 99\ \texttt{do}\ a[i] \leftarrow 0\ \{\forall 0 \leq k < 100, a[k] = 0\}$$

Seja o invariante $\eta : \forall 0 \leq k < i,\ a[k] = 0$ and o seguinte *tableaux*:

$\{\mathsf{true}\}$
$\{0 \leq 99\}$
$\texttt{for}\ i \leftarrow 0\ \texttt{to}\ 99\ \texttt{do}$
$\{$
$\{(\forall 0 \leq k < i, a[k] = 0) \wedge 0 \leq i \wedge i \leq 99\}$
$\{\forall 0 \leq k < i + 1, a[i \leftarrow 0][k] = 0\}$ $\qquad cons_{tot}$
$a[i] \leftarrow 0$
$\{\forall 0 \leq k < i + 1, a[k] = 0\}$ $\qquad ass_{tot}$
$\}$
$\{\eta[100/i]\}$
$\{\forall 0 \leq k < 100, a[k] = 0\}$

Eliminação de termos com arrays usando funções não interpretadas

Temos em particular a seguinte condição de verificação a ser provada:

$$(\forall 0 \leq k < i, a[k] = 0) \implies \forall 0 \leq k < i + 1, a[i \leftarrow 0][k] = 0$$

Considerando a uma função vamos substituir as suas instâncias por funções não interpretadas. Em particular o axioma (1) é um caso particular da consistência funcional.

Ex. 21.2. *Por exemplo se T_E corresponde ao conjunto dos caracteres*

$$(i = j \wedge a[j] = "z") \implies a[i] = "z"$$

pode ser substituido por

$$(i = j \wedge F_a(j) = "z") \implies F_a(j) = "z"$$

que pode ser validada pelos algoritmos já dados.

Para substituir termos $a[i \leftarrow e]$ introduz-se uma nova variável $a' \in Var_{array}$ and substituí-se por duas fórmulas (*Regra de Escrita*):

- $a'[i] = e$
- $\forall j \neq i . a'[j] = a[j]$

Ex. 21.3. *A fórmula $a[i \leftarrow e][i] \geq e$ é transformada em*

$$a'[i] = e \implies a'[i] \geq e$$

. A fórmula $a[0] = 10 \implies a[1 \leftarrow 20][0] = 10$ seria transformada em:

$$(a[0] = 10 \wedge a'[1] = 20 \wedge (\forall j \neq 1 . a'[j] = a[j])) \implies a'[0] = 10.$$

Introduzindo F_a and $F_{a'}$

$$(F_a(0) = 10 \wedge F_{a'}(1) = 20 \wedge (\forall j \neq 1 . F_{a'}(j) = F_a(j))) \implies F_{a'}(0) = 10.$$

Propriedades de array

Contudo, por exemplo, se adicionarmos à aritmética de Presburger funções não interpretadas obtemos uma teoria não decidível.

Por isso é necessário restringir a classe de fórmulas a considerar.

Vamos considerar fórmulas que são combinações Booleanas de *propriedades de array*.

Definição 21.1 (Propriedade de array)**.** *É uma fórmula da forma*

$$\forall i_1 \cdots \forall i_k \in T_I . \varphi_I(i_1, \ldots, i_k) \implies \varphi_V(i_1, \ldots, i_k)$$

1. *φ_I é a guarda dos índices and segue a seguinte gramática*

$$\begin{aligned} \varphi_I &:= \varphi_I \wedge \varphi_I \mid \varphi_I \vee \varphi_I \mid t_i \leq t_i \mid t_i = t_i \\ t_i &:= i_1 \mid \cdots \mid i_k \mid t \\ t &:= n \in \mathbb{N} \mid n.id_i \mid t + t \end{aligned}$$

   Os termos t são expressões sobre inteiros and id_i é uma variável da teoria dos índices T_I diferente dos i_j.

2. *Os índices $i_1, \ldots, i_k$ só podem ocorrer em expressões da forma $a[i_j]$ em φ_V.*

- A extensionalidade
$$\forall i.a_1[i] = a_2[i]$$
é uma propriedade de array onde a guarda é true.
- A fórmula $a' = a[i \leftarrow 0]$ é substituída por duas fórmulas:
  - $a'[i] = 0$ que é uma propriedade de array and
  - $\forall j \neq i.a'[j] = a[j]$.
    Neste caso temos de substituir por
$$\forall j.((j \leq i - 1 \vee i + 1 \leq j) \implies a'[j] = a[j])$$
    que é uma propriedade de array.

Algoritmo de redução de arrays

Input:	Uma propriedade de array φ_A em NNF
Output:	Uma fórmula φ^{uf} das teorias T_I and T_E, and com funções não interpretadas.

1. Aplicar a regra de escrita para eliminar termos de escrita $a[i \leftarrow e]$.
2. Substituir todos os quantificadores existênciais $\exists i \in T_I.P(i)$ por $P(j)$, onde j é uma variável nova.
3. Substituir todos os quantificadores universais $\forall i \in T_I.P(i)$ por $\bigwedge_{i \in \mathcal{I}(\varphi)} P(i)$.
4. Substituir os termos de leitura ($a[i]$) por funções não interpretadas, obtendo-se φ^{uf}.

$\mathcal{I}(\varphi)$

O conjunto $\mathcal{I}(\varphi)$, onde φ é a fórmula corrente, contém:

1. Todas as expressões usadas como índices de arrays em φ excepto variáveis quantificadas

2. Todas as expressões usadas nas guardas de índice de φ excepto variáveis quantificadas
3. Se φ não contêm nenhuma expressão acima então $\mathcal{I}(\varphi) = \{0\}$

Exemplo

Consideremos então para $k, i \in \mathbb{N}_0$, a validade de

$$(\forall k.k < i \implies a[k] = 0) \implies (\forall k.k \leq i \implies a[i \leftarrow 0][k] = 0)$$

Para tal vamos considerar que não é satisfazível a sua negação.

$$(\forall k.k < i \implies a[k] = 0) \wedge (\exists k.k \leq i \wedge a[i \leftarrow 0][k] \neq 0)$$

Aplicando a regra da escrita

$$(\forall k.k < i \implies a[k] = 0) \wedge a'[i] = 0 \wedge (\forall j \neq i.a'[j] = a[j])$$
$$\wedge (\exists k.k \leq i \wedge a'[k] \neq 0)$$

Instanciamos k com k_1 para eliminar $\exists k$

$$(\forall k.k < i \implies a[k] = 0) \wedge a'[i] = 0 \wedge (\forall j \neq i.a'[j] = a[j])$$
$$\wedge k_1 \leq i \wedge a'[k_1] \neq 0$$

Temos $\mathcal{I} = \{i, k_1\}$. Então eliminamos os quantificadores universais:

$$(i < i \implies a[i] = 0) \wedge (k_1 < i \implies a[k_1] = 0) \wedge a'[i] = 0$$
$$\wedge (i \neq i \implies a'[i] = a[i])$$
$$\wedge (k_1 \neq i \implies a'[k_1] = a[k_1]) \wedge k_1 \leq i \wedge a'[k_1] \neq 0)$$

Simplificando vem

$$(k_1 < i \implies a[k_1] = 0) \wedge a'[i] = 0$$
$$\wedge (k_1 \neq i \implies a'[k_1] = a[k_1]) \wedge k_1 \leq i \wedge a'[k_1] \neq 0$$

Substituindo a and a' por funções não interpretadas vem

$$(k_1 < i \implies F_a(k_1) = 0) \wedge F_{a'}(i) = 0$$
$$\wedge (k_1 \neq i \implies F'_a(k_1) = F_a(k_1)) \wedge k_1 \leq i \wedge F'_a(k_1) \neq 0$$

Considerando os três casos $k_1 < i$, $k_1 = i$ and $k_1 > i$ podemos concluir que não é satisfazível.

Arrays em SMT-LIB/Z3

- para definir um array usase o tipo (*sort*) `Array`

```
A = Array('A', IntSort(), IntSort())
x, y = Consts('x y',IntSort())
solve(A[x] == x, Store(A, x, y) == A)
```

- $A[x]$ é definido por `Select(A,x)` (ou só `A[x]`)
- `Store(A,x,v)`, corresponde a $A[x \leftarrow v]$.
- `K(Sort,v)` corresponde a um array de tipo `Sort` em que todas as posições têm o valor v (array constante, que é usado para exibir uma solução).
- Para o exemplo de verificação dado antes temos

```
solve (Implies(ForAll([x],(Implies(x< y, A[x]==0))),
 ForAll([x],(Implies(x<= y, Store(A, y, 0)[x]==0)))))
```

Os arrays correspondem a funções que se podem representar por λ-termos : se $f : A \times B \to C$ então `Lambda [x,y]. f(x, y)` tem tipo `Array(A,B,C)`.

```
a[i]             # select array 'a' at index 'i'
                 # Select(a, i)
Store(a, i, v)   # update array 'a' with 'v' at index 'i'
                 # = Lambda(j, If(i == j, v, a[j]))
K(D, v)          # constant Array(D, R), where R is sort of 'v'.
                 # = Lambda(j, v)
Map(f, a)        # map function 'f' on values of 'a'
                 # = Lambda(j, f(a[j]))
Ext(a, b)        # Extensionality
                 # Implies(a[Ext(a, b)] == b[Ext(a, b)], a == b)
```

References

[BdM15] Nikolai Bjorner and Leonardo de Moura. *Z3 Theorem Prover*. Rise, Microsft, 2015.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Verlag, 2007.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures:An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.