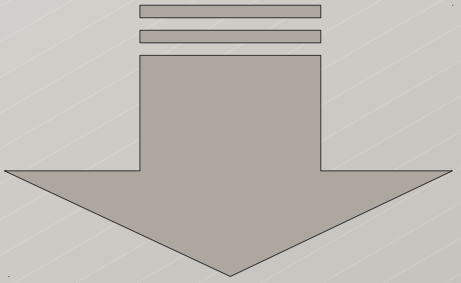
Моделирование памяти в инструментах дедуктивной верификации Frama-C/WP, Jessie и VCC

Алексей Хорошилов,
Михаил Мандрыкин,
ИСП РАН

Вороново, 28 ноября 2015 г.

Дедуктивная верификация


```
/*@ ensures \result == a || \result == b;
  @ ensures \result <= a && \result <= b;
  @*/
int min(int a, int b) {
   return a < b ? a : b;
}
```



$$a < b \land r = a \rightarrow (r = a \lor r = b) \quad \Rightarrow \checkmark$$

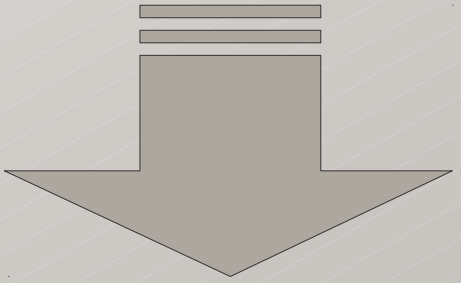
$$\neg(a < b) \land r = b \rightarrow (r = a \lor r = b) \quad \Rightarrow \checkmark$$

$$a < b \land r = a \rightarrow r \leq a \land r \leq b \quad \Rightarrow \checkmark$$

$$\neg(a < b) \land r = b \rightarrow r \leq a \land r \leq b \quad \Rightarrow \checkmark$$

Дедуктивная верификация + SMT-решатели


```
/*@ ensures \result == a || \result == b;
  @ ensures \result <= a && \result <= b;
  @*/
int min(int a, int b) {
   return a < b ? a : b;
}
```



$a < b \wedge r = a \wedge \neg(r = a \vee r = b)$ ⇒ `unsat` ⇒ ✔

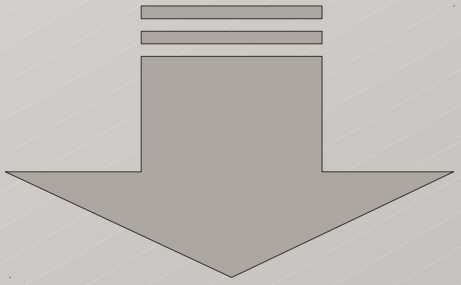
$\neg(a < b) \wedge r = b \wedge \neg(r = a \vee r = b)$ ⇒ `unsat` ⇒ ✔

$a < b \wedge r = a \wedge \neg(r \leq a \wedge r \leq b)$ ⇒ `unsat` ⇒ ✔

$\neg(a < b) \wedge r = b \wedge \neg(r \leq a \wedge r \leq b)$ ⇒ `unsat` ⇒ ✔

Работа с указателями

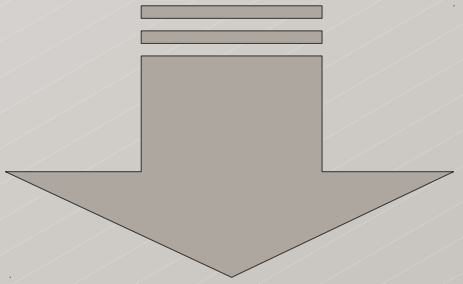

```
/*@ requires \valid(a) && \valid(b);
  @ assigns *a, *b;
  @ ensures *a == \old(*b) && *b == \old(*a);
  @*/
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```



$$t \neq M_0[a] \wedge t \neq M_0[b] \wedge t \neq a \wedge t \neq b \wedge$$
$$M_1 = M_0[t \leftarrow M_0[M_0[a]]] \wedge$$
$$M_2 = M_1[a \leftarrow M_1[M_1[b]]] \wedge$$
$$M_3 = M_2[b \leftarrow M_2[t]] \wedge$$
$$\neg(\boldsymbol{M_3[M_3[a]] = M_0[M_0[b]]})$$
$$\Rightarrow \texttt{unsat} \Rightarrow \checkmark$$

Работа с указателями


```
/*@ requires \valid(a) && \valid(b);
  @ assigns *a, *b;
  @ ensures *a == \old(*b) && *b == \old(*a);
  @*/
void swap(int *a, int *b) {
   char c = 0;
   int t = *a;
   *a = *b;
   *b = t;
}
```



$$t \neq M_0[a] \wedge t \neq M_0[b] \wedge \ldots \wedge c \neq t \wedge c \neq M_0[a] \wedge c \neq M_0[b] \wedge$$

$$M_1 = M_0[c \leftarrow M_0[c][8{:}31) 0_8] \wedge$$

$$M_2 = M_1[t \leftarrow M_1[M_1[a]]] \wedge$$

$$M_3 = M_2[a \leftarrow M_2[M_2[b]]] \wedge$$

$$M_4 = M_3[b \leftarrow M_3[t]] \wedge$$

$$\neg(M_4[M_4[a]] = M_0[M_0[b]])$$

$\Rightarrow$ `unsat` $\Rightarrow$ ✔

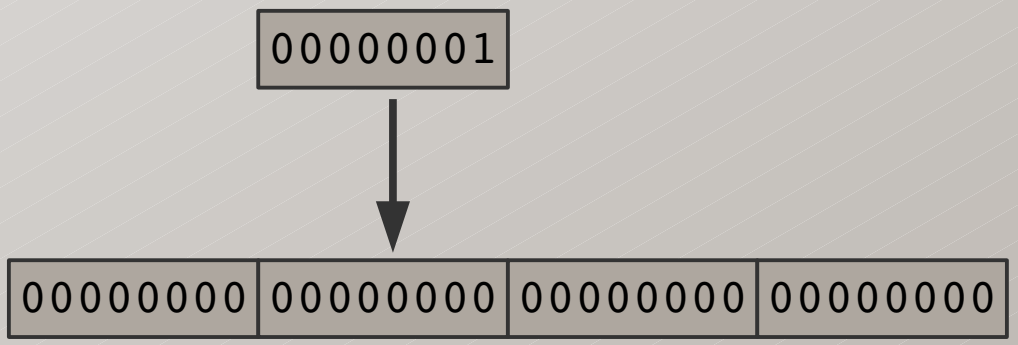
Моделирование битовыми векторами и математическими целыми

0001010010010001 Отсутствие дополнительных ограничений (допустимы побитовые операции, переполнения и произвольные преобразования/приведения типов, в т. ч. указателей)	Низкая производительность SMT-решателей Несовместимость с семантикой языков спецификации (в частности, ACSL использует математические целые)
ℤ Более высокая производительность SMT-решателей Хорошая совместимость с семантикой языков спецификации	Дополнительные ограничения на побитовые операции, переполнения и преобразование/приведение типов, в т. ч. указателей

Типизированная модель памяти

00010100

$$M_{i+1} = M_i[p \leftarrow M_i[p][0:7] \frown 1_8 \frown M_i[p][16:31]]$$



ℤ

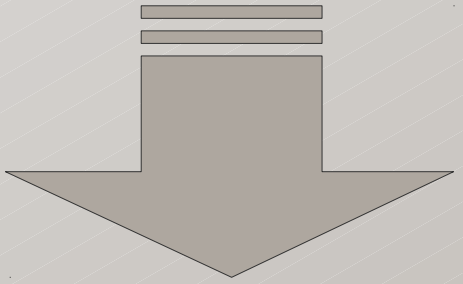
$$I_{i+1} = \text{?..}$$

Типизированная модель памяти

	`int p;` … `p`	`char p;` … `p`	`int *p;` … `p, *p`
00010100	$M_i[p]$	$M_i[p][0{:}7]$	$M_i[p], M_i[M_i[p]]$
$\mathbb{Z}$	$I_i[p]$	$C_j[p]$	$P_k[p], I_i[P_k[p]]$

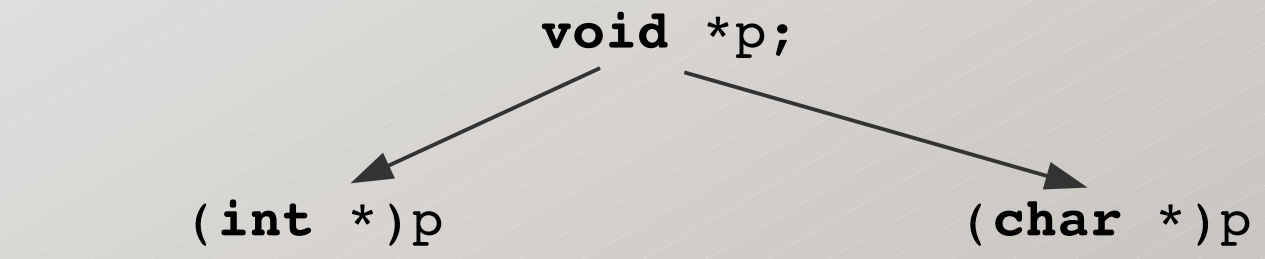
Типизированная модель памяти


```
/*@ requires \valid(a) && \valid(b);
  @ assigns *a, *b;
  @ ensures *a == \old(*b) && *b == \old(*a);
  @*/
void swap(int *a, int *b) {
   char c = 0;
   int t = *a;
   *a = *b;
   *b = t;
}
```



$$t \neq P_0[a] \wedge t \neq P_0[b] \wedge$$
$$C_1 = C_0[c \leftarrow 0] \wedge$$
$$I_1 = I_0[t \leftarrow I_0[P_0[a]]] \wedge$$
$$I_2 = I_1[a \leftarrow I_1[P_0[b]]] \wedge$$
$$I_3 = I_2[b \leftarrow I_2[t]] \wedge$$
$$\neg(I_3[P_0[a]] = I_0[P_0[b]])$$
$$\Rightarrow \texttt{unsat} \Rightarrow \checkmark$$

Побитовая и типизированная модели памяти



00010100

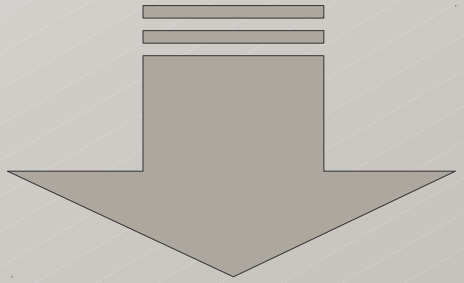
$M_i[p]$ $M_i[p][0:7]$

$\mathbb{N}$ $I_i[p]$ $C_j[p]$

Проблема типизированной модели памяти


```
{
    …
    int a = 0;
    a = 0;
    char *c = (char *)&a;
    *c = 1;
    //@ assert a == 0;
    …
}
```

✘

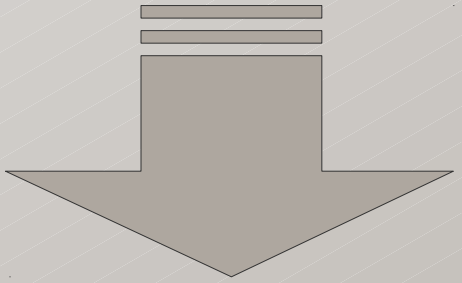


$$I_1 = I_0[a \leftarrow 0] \wedge$$
$$P_1 = P_0[c \leftarrow a] \wedge$$
$$C_1 = C_0[P_1[c] \leftarrow 1] \wedge$$
$$\neg(I_1[a] = 0)$$

⇒ unsat ⇒ ✔

Проблема типизированной модели памяти


```
union u { int a; char b } u;
…
{
    …
    u.a = 0;
    u.b = 1;
    //@ assert u.a == 0;   ✘
    …
}
```



$$I_1 = I_0[u \leftarrow 0] \wedge$$
$$C_1 = C_0[u \leftarrow 1] \wedge$$
$$\neg(I_1[u] = 0)$$
$$\Rightarrow \texttt{unsat} \Rightarrow \checkmark$$

Ограничения типизированной модели памяти

Ограничения на допустимые приведения типов указателей

Возможные решения:

Нет приведения типов указателей

Только префиксные приведения типов

Префиксные приведения типов и переинтерпретация

Ограничения типизированной модели памяти

Ограничения на использование объединений (**`union`**s)

Возможные решения:

Нет объединений

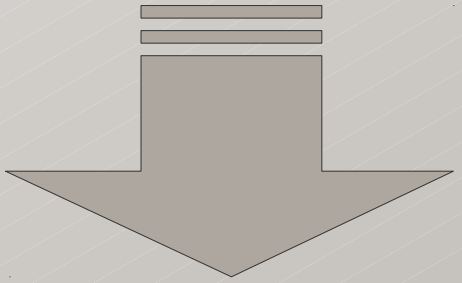
Только вариантные и различаемые объединения

Вариантные и различаемые объединения
с переинтерпретацией

*variant types, discriminated unions

Моделирование структур


```
struct s { int a; int b; char c; } s;
…
{
    …
    s.a = 0;
    s.b = 1;
    //@ assert s.a == 0;  ✔
    …
}
```



$$I_1 = I_0[s \leftarrow 0] \wedge$$

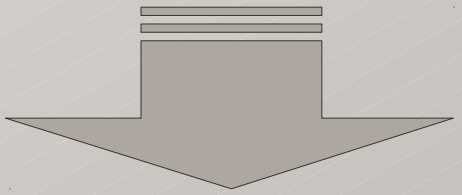
$$I_1 = I_0[s + 4 \leftarrow 1] \wedge$$

$$\neg(I_1[s] = 0)$$

$\Rightarrow$ unsat $\Rightarrow$ ✔

Префиксные приведения типа (1-ый случай)


```
struct prefix { int a; };
struct extended { struct prefix p; int a; } e;
…
{
    …
    e.p.a = 0;
    e.a = 0;
    ((struct prefix *) &e)->a = 1;
    //@ assert e.p.a == 0;  ✘
    …
}
```

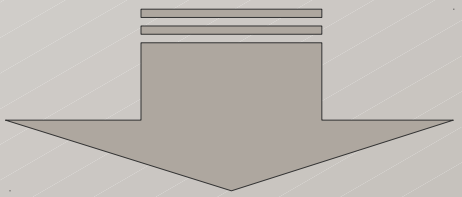


$$I_1 = I_0[e \leftarrow 0] \wedge$$
$$I_2 = I_1[e + 4 \leftarrow 0] \wedge$$
$$I_3 = I_2[e \leftarrow 1] \wedge$$
$$\neg(I_3[e] = 0)$$

$\Rightarrow$ sat $\Rightarrow$ ✘

Префиксные приведения типа (2-ой случай)


```
struct prefix { int a; };
struct extended { struct prefix p; int a; } e;
…
{
    …
    struct prefix *p = (struct prefix *)&e;
    ((struct extended *) p)->a = 1;
    //@ assert e.a == 1;  ✔
    …
}
```



$$P_1 = P_0[p \leftarrow \underline{e}] \wedge$$

$$I_2 = I_1[\underline{P[p]} + 4 \leftarrow 1] \wedge$$

$$\neg(I_2[e + 4] = 1)$$

$$\Rightarrow \texttt{unsat} \Rightarrow ✔$$

Префиксные приведения типа

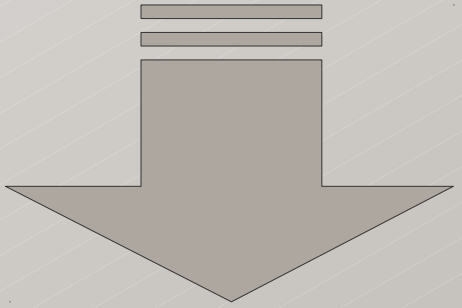

```
struct prefix { int a; } e;
struct extended { struct prefix p; int a; };
…
{
    …
    struct prefix *p = &e;
    ((struct extended *) p)->a = 1;
    …
}
```

Префиксные приведения типа


```
struct prefix { int a; } e;
struct extended { struct prefix p; int a; };
…
{
    …
    struct prefix *p = &e;
    ((struct extended *) p)->a = 1;
    …
}
```

Префиксные приведения типа


```
struct prefix { int a; } e;
struct extended { struct prefix p; int a; };
…
{
    …
    struct prefix *p = &e;
    ((struct extended *) p)->a = 1;
    …
}
```



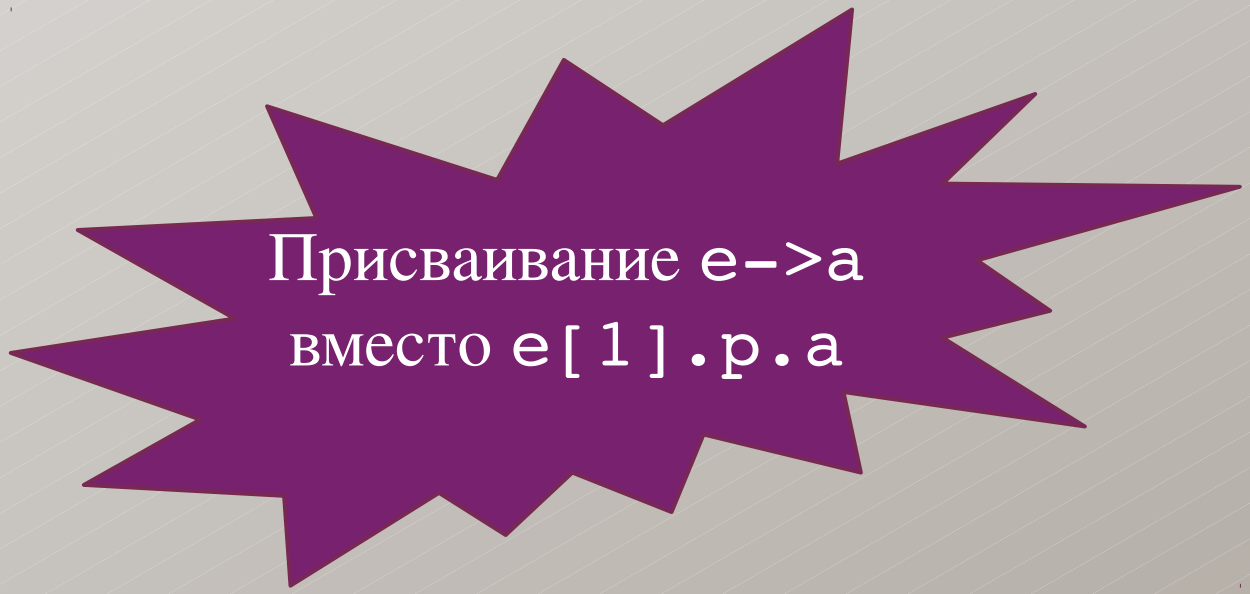
$$\preceq(t_{extended},\ t_{prefix}) \wedge$$
$$T_1 = T_0[e \leftarrow t_{prefix}] \wedge$$
$$P_1 = P_0[p \leftarrow e] \wedge$$
$$\neg\preceq(T_1[P_1[p]], t_{extended})$$
$$\Rightarrow \texttt{sat} \Rightarrow \times$$

Префиксные приведения типа


```
struct prefix { int a; };
struct extended { struct prefix p; int a; } e[2];
…
{
    …
    ((struct prefix *) e)[1]->a = 1;
    …
}
```

Префиксные приведения типа


```
struct prefix { int a; };
struct extended { struct prefix p; int a; } e[2];
…
{
    …
    ((struct prefix *) e)[1]->a = 1;
    …
}
```

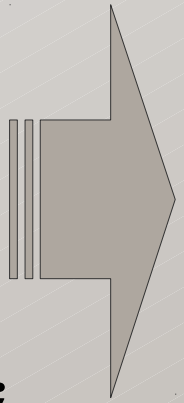


Нормализация и приведение к `void *`


```
void *p;
int a;



…
{
    …
    p = ((void *) &a);
    *((int *) p) = 1;
    …
}
```



```
struct voidP {} *p;
struct intP {
    struct voidP p;
    int intM
} a;
…
{
    …
    p = ((struct voidP *) &a);
    ((struct intP *) p)->intM = 1;
    …
}
```

Префиксные приведения типа

Поддержка большинства* случаев приведения типа указателей (по некоторым данным, до 99%
[Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. SIGPLAN Not., 38(5):232–244, 2003])

* – вместе с различаемыми объединениями

Кодирование тегов и отношения ≤ в формулах

Дополнительные условия корректности для приведений типов и арифметики указателей (в т.ч. взятие элемента массива)

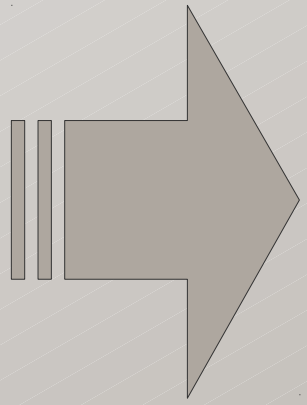
Вариантные объединения

Вариантное объединение – объединение, для которого все операции осуществляются над одним и тем же полем

```
union u { int a; char c } u1, u2;
…
{
    …
    u1.a = 1;
    u2.c = (char) u1.a;
    u1.a = u2.c;
    char *p = &u2.c;
    …
    u1.c = (char) u2.a;
    …
}
```

Вариантные объединения


```
union u { int a; char c } u1, u2;
…
{
    …
    //@ interpret u1 as .a;
    u1.a = 1;
    //@ interpret u2 as .c;
    u2.c = (char) u1.a;
    u1.a = u2.c;
    char *p = &u2.c;
    …
    u1.c = (char) u2.a;
    …
}
```



$$T_1 = T_0[u1 \leftarrow t_{intP}] \wedge$$
$$I_1 = I_0[u1 \leftarrow 1] \wedge$$
$$T_2 = T_1[u2 \leftarrow t_{charP}] \wedge$$
$$C_1 = C_0[u2 \leftarrow I_1[u1]] \wedge$$
$$I_2 = C_1[u1 \leftarrow C_1[u2]] \wedge$$
$$P_1 = P_0[p \leftarrow u2] \wedge$$
$$\neg(T_2[u2] = t_{charP})$$
$$\Rightarrow \texttt{sat} \Rightarrow \times$$

$$\neg\trianglelefteq(T_2[u2], t_{charP})$$

Модерируемые объединения

Модерируемое объединение – объединение, доступ к полям которого осуществляется только непосредственно через это объединение

```
union u { int a; char c } u1, u2;
…
{
    …
    u1.a = 1;
    u2.c = (char) u1.a;
    …
    int *p1 = (int *)&u1;
    char *p2 = &u2.c;
    …
}
```

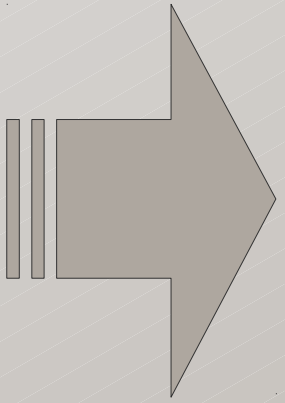
Различаемые объединения

Различаемое объединение – это модерируемое объединение, в котором чтение всегда осуществляется из последнего записанного поля

```
union u { int a; char c } u1, u2;
…
{
    …
    u1.a = 1;
    u2.c = (char) u1.a;
    …
    u1.a = u2.c;
    …
    u1.c = (char) u2.a;
    …
}
```

Различаемые объединения


```
union u { int a; char c } u1, u2;
…
{
    …
    //@ interpret u1 as .a;
    u1.a = 1;
    //@ reinterpret u1 as .c;
    u1.c = '\0';
    //@ assert u1.c == 0;
}
```

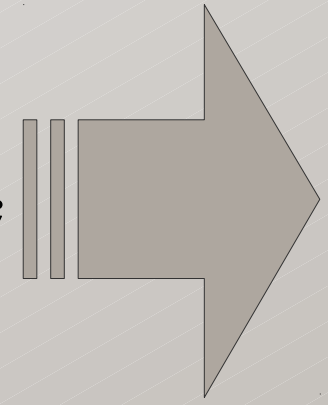


$$T_1 = T_0[u1 \leftarrow t_{u.a}] \wedge$$
$$I_1 = I_0[u1 \leftarrow 1] \wedge$$
$$T_2 = T_1[u1 \leftarrow t_{u.c}] \wedge$$
$$(\exists c \,.\, C_1 = C_0[u1 \leftarrow c]) \wedge$$
$$C_2 = C_1[u1 \leftarrow 0] \wedge$$
$$\neg(\boldsymbol{C_1[u1] = 0})$$

$\Rightarrow$ `unsat` $\Rightarrow$ ✔

Различаемые объединения


```
union u { int a; char c } u1, u2;
…
{
    …
    //@ interpret u1 as .a;
    int *p = &u1.a;
    u1.a = 1;
    //@ reinterpret u1 as .c;
    u1.c = '\0';
    //@ assert *p == 1;
}
```



$$T_1 = T_0[u1 \leftarrow t_{u.a}] \wedge$$
$$P_1 = P_0[p \leftarrow u1] \wedge$$
$$I_1 = I_0[u1 \leftarrow 1] \wedge$$
$$T_2 = T_1[u1 \leftarrow t_{u.c}] \wedge$$
$$(\exists c \,.\, C_1 = C_0[u1 \leftarrow c]) \wedge$$
$$C_2 = C_1[u1 \leftarrow 0] \wedge$$
$$\neg(I_1[P_1[p]] = 1)$$
$$\Rightarrow \texttt{unsat} \Rightarrow \checkmark$$

Различаемые объединения

Вместе с
префиксными приведениями типов
составляют до
99% всех
приведений типов указателей
в промышленном C-коде

[Jeremy Condit, Matthew Harren,
Scott McPeak, George C. Necula, and Westley
Weimer. CCured in the real world.
SIGPLAN Not., 38(5):232–244, 2003]

Кодирование тегов
в формулах

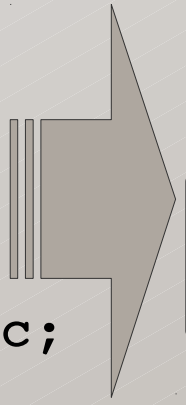
Дополнительные аннотации
(**`reinterpret`**)

Дополнительные обновления
памяти для **`reinterpret`**

Дополнительные условия корректности
для каждого обращения к полю
различаемого объединения

Переинтерпретация объединений


```
union u { int a; char c } u1, u2;
…
{
    …
    //@ interpret u1 as .a;
    u1.a = 1;
    //@ reinterpret u1.a as .c;
    //@ assert u1.c == 1;
}
```



$$T_1 = T_0[u1 \leftarrow t_{u.a}] \wedge$$
$$I_1 = I_0[u1 \leftarrow 1] \wedge$$
$$T_2 = T_1[u1 \leftarrow t_{u.c}] \wedge$$
$$C_1 = C_0[u1 \leftarrow I_1[u1] \bmod 256] \wedge$$
$$\neg(C_1[u1] = 1)$$

$\Rightarrow$ unsat $\Rightarrow$ ✔

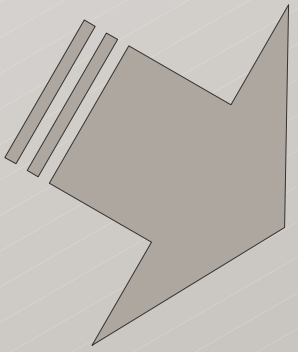
Переинтерпретация объединений

Поддержка практически всех случаев рабооты с объединениями

Сложные формулы для переинтерпретации памяти

Переинтерпретация указателей


```
…
{
    …
    int a;
    int *p = &a;
    a = 1;
    //@ reinterpret &a as char *;
    *((char *) a) = '\0';
    *p = 1;
}
```



$$P_1 = P_0[p \leftarrow a] \wedge$$
$$I_1 = I_0[a \leftarrow 1] \wedge$$
$$V_1 = V_0[a \leftarrow false] \wedge$$
$$V_2 = V_1[cast(T_0, a, char) \leftarrow false] \wedge$$
$$C_1 = C_0[cast(T_0, a, char) \leftarrow I_1[a] \bmod 256] \wedge$$
$$C_2 = C_1[cast(T_0, a, char) \leftarrow 0] \wedge$$
$$\neg(V_2[p] = true)$$

$\Rightarrow$ `sat` $\Rightarrow$ ✘

Переинтерпретация указателей

Поддержка практически всех случаев приведения типа указателей

Аксиоматизация функции *cast*

Дополнительные обновления теневого отображения *V*

Сложные формулы для переинтерпретации памяти

Спасибо за внимание

Исследование поддержано Министерством образования и науки РФ
(проект №RFMEFI60414X0051)

Михаил Мандрыкин <mandrykin@ispras.ru>,