Лекция 14. Инструмент AstraVer: приведение типов указателей, noembed, итоги

Цель лекции

Обсудить некоторые проблемы построения модели Си-программ и их решения, реализованные в инструменте AstraVer

Типы для примеров

Далее будут использоваться следующие типы (из практического задания):

```
typedef struct {
    int a, b;
} Key;
typedef struct {
    int c, d;
} Value;
typedef struct {
    Key key;
    Value value;
} Item;
```

Содержание

Иерархии типов указателей в Си

- `void *` – чем не аналог типа-предка для любого типа-указателя?
- еще примеры «наследования» типов в Си? (первым полем структуры является другая структура)

Модель Си-программы с наследованием

- Когда надо применять правила моделирования типа как потомка? Вспомните эти правила из предыдущей лекции.
- Решение AstraVer: только если в исходном коде или спецификации встречается операция приведения
- NB! Приведение между `void *` и остальными типами указателей может быть неявным!
- Если в условии верификации нет ничего про subtag нужных типов или генерируются условия верификации про приведение к `char *`, значит, возможно, в коде не хватает выражения с нужным приведением типа!

Еще пример неявного приведения типа

Изучите модель программы, которую сгенерирует AstraVer:

```
//@ ensures \result == 0;
int main(void) {
    int m[10];
    int n;
    //@ assert &n != &m[0];
    return 0;
}
```

Операция взятия адреса и массивы

- Определение массива превращается в malloc и в конце блока кода, где объявлена переменная, в free
- Переменная, от которой берется адрес, превращается в массив размера 1 (а он в malloc)
- Если адрес от переменной не берется, то она не превращается в массив (в этом превращении нет необходимости)

Корректность приведения типов указателей

- когда приведение `void *` к типу `T *` корректно? (последующее разыменование полученного указателя будет корректно)
- когда это приведение и не потребует переинтерпретации памяти?

Аналог instanceof в AstraVer

- Чтобы доказать корректность приведения `void *p` к некоторому типу `T *`, надо доказать, что `p` на самом деле указывает на `T` (или его потока) – в Java это называется instanceof
- AstraVer поддерживает такой синтаксис в спецификациях, означающий instanceof: `\typeof(p) <: \type(T)`

`void *` в стандартной библиотеке

- `void *` часто используется в стандартной библиотеке языка Си (как тип формального параметра функции, чтобы ее можно было применять к различным типам указателей)
- Для некоторых функций стандартной библиотеки AstraVer содержит шаблонные спецификации: надо сказать, какой динамический тип указателя, тогда будет сгенерирована спецификация с этим типом указателя

Содержание

Component-as-array: тип поля - скалярный

- Вспомним идею о том, что изменение одного поля структуры не влияет на остальные поля структуры. Иначе говоря, указатели на разные поля одной структуры не являются синонимами. Это используется для разделения модели памяти на независимые части.
- Для типа `Value` будут построены такие типы и переменные:
  - `type Value`
  - `constant cFieldMemory: memory (pointer Value) int`
  - `constant dFieldMemory: memory (pointer Value) int`
- Зачем тут pointer? (две отдельные переменные типа `Value *`, изменяем значение по одному указателю, другой указатель не трогали, но значение по второму указателю тоже должно измениться - как это промоделировать без pointer?)

Component-as-array: нескалярный тип поля

- Теперь строим переменные и типы для `Item`. Вот так правильно?
  - `type Item`
  - `type Key`
  - `type Value`
  - `constant keyFieldMemory: memory (pointer Item) Key`
  - `constant valueFieldMemory: memory (pointer Item) Value`
- Если да, то как промоделировать `item->value.c`? Вот так правильно? `(select cFieldMemory (select valueFieldMemory item))` Откуда взять там `pointer`?

Выводы

- Чтобы повысить эффективность солверов, при моделировании программы можно применить подход `component-as-array`.
- Тем не менее, синонимы могут быть (два указателя на одно и то же поле). Поэтому при моделировании частей нескалярных типов не получается выразить их вложение в объемлющий тип. Каждое нескалярное поле структуры моделируется указателем на отдельную область памяти.
- Например, следующее утверждение (верное по стандарту языка Си) не докажется: `assert (void *)&key->a < (void *)&key->b;`

Иногда это может создать трудности

Почему assert не доказывается?

```
//@ requires \valid(items + (0 .. 1));
void prog(Item *items)
{
    items[0].value.c = 0;
    items[1].value.c = 1;
    //@ assert items[0].value.c == 0;
}
```

container_of

- assert не доказывается, потому что нет предусловия, что `&items[0].value` и `&items[1].value` не являются синонимами (они находятся в одном регионе, т.к. в одном регионе находятся `&items[0]` и `&items[1]`).
- У каждого item есть свой value. Это можно записать?
- `\forall Item *item1, *item2; item1 != item2 ==> &item1->value != &item2->value`

container_of (2)

Можно записать следующими способами:

- как предусловие функции, которой необходимо это утверждение (как доказать корректность вызова этой функции? В постусловии malloc() будет сгенерировано это утверждение – надо им воспользоваться)
- как аксиому (описать ее в axiomatic, не забыть про триггер квантора, если аксиома не будет применяться; в axiomatic придется добавить фиктивный символ и обратиться к нему в условии верификации, иначе axiomatic не будет включен в условие верификации)

Аксиомы двух видов

Аксиому можно записать двумя способами:

- ровно так, как было написано выше
- или с использованием аналога макроса container_of из исходного кода ядра Linux: `\forall Item *i; i == (Item *)((char *)&i->value - 8);`
- из этой аксиомы следует, что при равных указателях на value равны указатели i (или, если указатели i отличаются, то отличаются их указатели на value)
- в ядре Linux на Си используется ООП-стиль программирования; структура-потомок содержит структуру-предка как одно из полей
- макрос container_of по указателю на структуру-предка дает указатель на структуру-потомка

Пример с container_of

```
/*@ axiomatic ContainerOf {
    predicate id(integer x) = \true;
    axiom container_of{L}: \forall Item *i;
      i == (Item *)((char *)&i->value - 8); }*/
//@ requires \valid(items + (0 .. 1));
void prog(Item *items) {
    items[0].value.c = 0; items[1].value.c = 1;
    //@ assert id(0); // to add axiomatic
    //@ assert (char *)&items[0].value !=
    (char *)&items[1].value; // triggers for axiom
    //@ assert items[0].value.c == 0 ;
}
```

Содержание

Обращение к не первому полю

- Пусть `item` имеет тип `Item *`.
- Тогда чтение `item->value.c` будет моделироваться так: `(select cFieldMemory (select valueFieldMemory item))`.
- Типы `Item` и `Value` не находятся в отношении наследования, переменная `item` будет иметь тип `pointer Item`, переменная `valueFieldMemory` будет иметь тип `memory (pointer Item) (pointer Value)`, переменная `cFieldMemory` будет иметь тип `memory (pointer Value) int`.

Обращение к первому полю

- Чтение `item->key.a` можно моделировать так же (через два select и два memory), но можно учесть наследование и использовать более простое выражение: `(select aFieldMemory item)`. Это совпадает с чтением `((Key *)item)->a`.
- Иными словами, выражение `& item->key` можно моделировать так же, как и `(Key *)item`, а (из-за наследования) последнее выражение можно моделировать просто как `item`.
- Переменная `item` теперь будет иметь тип `pointer Key` (а не `pointer Item`, т.к. здесь нужно иметь возможность привести указатель на потомка к указателю на предка). Переменная `aFieldMemory` будет иметь тип `memory (pointer Key) int`.

Хитрость с адресной арифметикой

- Пусть `items` имеет тип `Item [2]`.
- Какова модель для `items[1].key.a`? Какова модель для `(& items->key)[1].a`?
- Они совпадают! `(items + 1)->a` Но оба ли исходных выражения корректны? Только первое (почему?). Как научить модель запрещать некорректное выражение?

Модели всё же немного отличаются

- Можно увидеть отличие моделей, если рассмотреть их вот так:
  - первое выражение: `((Key *)(items + 1))->a`
  - второе выражение: `((Key *) items + 1 )-> a`
- Чтобы запретить второе, можно добавить еще одно предусловие для операции обращения к полю от указателя, полученного сдвига другого указателя: что тип блока этих указателей должен быть ровно таким, где определено поле a.

Как это сделано в AstraVer

- Чтобы увидеть, как эта идея реализована в аксиоматике памяти AstraVer, надо открыть файл `core.mlw`. Найти там модуль `Acc_offset_safe` и смотреть val-примитив `acc_offset`.

Но идея работает не всегда

Идея оказывается слишком жесткой, т.к. в таком коде возникает обращение к полю после сдвига и обращение безопасно, но оно тоже будет воспринято как некорректное (не докажется одно из условий верификации):

```
//@ requires \valid(items + (0 .. 1));
int prog(Item *items) {
    int offs = 1;
    return (& items->key + 1 - offs)->a;
}
```

Наконец, noembed

- Поэтому моделирование первого поля как предка некорректно.
- Чтобы это отменить, надо указать у первого поля атрибут noembed (тогда модель станет несколько сложнее, зато корректнее):

```
typedef struct {
    Key key __attribute__((noembed));
    Value value;
} Item;
```

Содержание

AstraVer на практике

- Цель проекта ИСП РАН – применять дедуктивную верификацию для исходного кода отдельных модулей ядра Linux.
- При этом оказалось, что:
  1. ядро написано под определенный набор расширений языка Си (gcc)
  2. ядро использует переинтерпретацию памяти
- За основу AstraVer был взят инструмент Jessie. AstraVer исправил часть модели памяти Jessie, добавил некоторые новые возможности (`http://astraver.linuxtesting.org/manuals/features/`). Но суть модели памяти не поменялась. Ограничения модели остались.
- Плюс проблема с тем, что модель памяти недоступна на уровне ACSL.

Новый инструмент

- Поэтому сейчас разрабатывается новый инструмент дедуктивной верификации.
- Он основан на интерактивном прувере Isabelle/HOL, чтобы можно было управлять процессом доказательства. Причем для спецификации требований и для доказательства условий верификации можно было бы использовать один и тот же язык.
- И не менее важное, что в новом инструменте будет поддерживаться настоящая модель памяти языка Си.

Другие проекты по дедуктивной верификации

- Есть игрушечные проекты, их много, но мы их не рассматриваем. Обычно они требуют особой подготовки пользователя и не все готовы тратить время на получение этой подготовки.
- VCC – проект Microsoft по дедуктивной верификации исходного кода гипервизора Hyper-V (исходники на Си). Архитектура инструмента та же, что у AstraVer, но используется другая модель памяти.
- SPARK – дедуктивная верификация Ada. Используется много где (например, при создании авионики, например, A350)

Формальные методы на практике

- Основная проблема формальных методов – это большой объем спецификаций (их надо написать), большие трудозатраты на формальную верификацию, специфические навыки по спецификации и по верификации.
- Опыт показывает, что одно только написание формальных спецификаций дает очень много:
  - это позволяет систематизировать требования к создаваемой программе
  - это требует продумывания поведения во всех ситуациях, даже в тех, о которых раньше не задумывались
- Иногда выписывание формальной спецификации приводит к пониманию того, что в дизайне системы есть ошибка, и эта ошибка исправляется до того, как система уже реализована!

Формальные методы на практике

- Еще одно направление в практике – это комбинация формальных спецификаций и легковесной верификации (формального тестирования). Формальная спецификация позволяет не пропустить важные тестовые случаи.
- Еще одно направление в практике – это многоуровневая формальная спецификация и верификация. То есть пишется сначала очень абстрактная спецификация, но описывающая свой уровень подробно. Затем абстрактные понятия уточняются и доказывается корректность уточнения.

Формальные методы в России

- Многоуровневые спецификации используются в следующих организациях:
  - РусБИТех – спецификация отдельных модулей безопасности в операционной системе AstraLinux, доказательство соответствия уровней (Event-B, Pro-B), тестирование реализаций и дедуктивная верификация (AstraVer)
  - МЦСТ – спецификация отдельных модулей безопасности в операционной системе Эльбрус
- По заказу ГосНИИАС разрабатывается ОСРВ с открытым кодом JetOS, удовлетворяющая стандарту ARINC653. ОС проектируется так, чтобы как можно быстрее пройти сертификацию.