# Лекция 7. Некоторые специальные методы доказательства в AstraVer

# Цель лекции

Показать ряд приемов доказательства, поддерживающих инструментом AstraVer, остающихся в рамках (автоматических) солверов. Основная идея — добавлять в код аннотации, которые нужны только для верификации. Такой подход называется `auto-active verification`.

# Содержание

# Методы Флойда для доказательства по индукции

- Ранее у нас оставалось недоказанными много лемм, которые можно доказать методом математической индукции (ММИ).
- Проблема в том, что солверы не могут применять ММИ.
- ММИ использовался при доказательстве теоремы корректности метода индуктивных утверждений.
- Пусть нам надо доказать некоторое утверждение ММИ. Если можно подобрать функцию на Си, полная корректность которой эквивалентна доказываемому утверждению и эту корректность мы доказали МИУ, то мы доказали исходное утверждение ММИ.

# Ghost-функции

Такие дополнительные функции оформляются в ghost-коде. Их спецификация пишется в комментариях вида `/@ ... @/`.
Пример: тут доказывается, что для любого `n` типа `int`, для которого выполнено `R(n)`, выполнено `E(n)`.

```
/*@ ghost /@ requires R(n);
              ensures E(n);
            @/
    void proof(int n) {
        ...
    }
*/
```

# Использование доказанных утверждений

Чтобы воспользоваться доказанным утверждением, нужно в определенном месте в коде на Си вызвать ghost-функцию. Поскольку она ghost, то это надо делать внутри ghost-кода: `/*@ ghost proof(a * b);*/` Для этого вызова будет сгенерировано safety-условие о выполнении `R(a * b);`, а в предикат пути, проходящий через этот вызов функции, будет вставлено `E(a * b);`

# Пример

- `examples\what_more\what_more.c` — функция подсчета количества вхождений. Мы верифицировали эту функцию, но оставили недоказанными одну лемму.
- `examples\auto-av\what_more_ghost.c` — лемма доказана при помощи ghost-функции.

# Ограничения ghost-функций

- Они – все еще Си-функции. Поэтому типами их аргументов должны быть Си-типы (в примере выше мы не можем доказать лемму для всех целочисленных `n`, а лишь для тех, которые принадлежат типу `int`).
- Ghost-функция не должна менять состояние памяти (иначе ее нельзя использовать как ghost-функцию, так как ее вызов изменит поведение верифицируемой функции). То есть она может работать только с одной меткой памяти. А у нас было много утверждений с двумя метками памяти.
- Чтобы воспользоваться доказанным утверждением, необходимо вызвать функцию. То есть это нельзя сделать в спецификационном коде, т.е. внутри assert, loop invariant.

# Лемма-функции

- Это попытка справиться с 3-м ограничением. Это ghost-функция, вначале спецификации которой есть ключевое слово `lemma`. Тогда, помимо прочего, генерируется аксиома с тем же утверждением, что и доказывает ghost-функция. Аксиома может быть автоматически применена солвером в разных местах.
- Вызов лемма-функции (как ghost-функции) тоже возможен.
- `examples\auto-av\what_more_lf.c` — доказательство леммы при помощи лемма-функции.

# Дублирование структур данных

- Чтобы частично справиться с проблемой единственности метки памяти у ghost-функции, можно попробовать дублировать структуру данных в ghost-переменную и передавать в ghost-функцию неё вместо метки памяти.
- `examples\auto-av\selsort.c` — доказательство леммы в сортировке выбором при помощи дублирования структуры данных.

# Содержание

# Почему солвер не справляется?

- большая цель теории (делаем Split, увеличиваем ограничение по времени на верификацию)
- большое условие верификации (много лишнего – как убрать лишнее?)
- неподходящие эвристики в солвере (попробовать другой солвер)
- неподходящие формулы, солвер «зацикливается» на казалось бы небольших условиях верификации

# Почему солвер зацикливается?

- Как солвер «доказывает» условия верификации (т.е. что из аксиом следует цель)? Строит отрицание цели, составляет его в конъюнкцию с аксиомами и пытается вывести противоречивый конъюнкт.
- Если аксиома – это выражение под квантором всеобщности, то надо «инстанцировать» аксиому, т.е. подставить какие-то термы вместо подкванторных переменных.
- Основная причина «зацикливания» солвера — это постоянное и безуспешное инстанцирование аксиом.

# Как помочь солверу?

- Уменьшить количество аксиом с квантором всеобщности
- Написать аксиомы по-другому, чтобы сработали эвристики подбора термов для подкванторных переменных

# Триггеры

- Это одна из эвристик подбор термов для подкванторных переменных
- *Триггер* аксиомы — это набор нетривиальных шаблонов термов, позволяющий получить термы для всех подкванторных переменных этой аксиомы. Тривиальный шаблон — это просто переменная.
- Пример: `\forall integer a, b, c; mul(a, sum(b, c)) == sum(mul(a, b), mul(a, c));` — для этой аксиомы триггером может быть шаблон `mul(X, sum(Y, Z))`.
- Триггер сравнивается со всеми термами (точнее, термами из E-графа), которые получил солвер на данный момент.
- В верификации сортировки выбором есть примеры использования триггеров (термы вводятся в assert и ghost-коде).

# Matching loop

- Неудачный триггер может привести к зацикливанию солвера.
- Пример: `forall integer x; h(x) ==> h(k(x))`. Если в качестве триггера выбрать `h(x)`, то как только он сработает (`x := E`), появится формула `h(E) ==> h(k(E))` и значит терм `h(k(E))`, который снова подходит для триггера и порождается `h(k(k(E)))` и так до бесконечности.
- Такая ситуация называется matching loop.

# Что можно сделать на уровне ACSL?

- Уменьшить число assert'ов, совмещая их последовательность в один предикат
- Составить лемму, в которой оставить только самые важные утверждения в посылке и цели
- Писать аксиомы такого вида, из которых солверы автоматически извлекут полезные триггеры. Синтаксис ACSL не позволяет задать триггер в аксиоме или лемме.
- Упростить утверждения при помощи ghost-переменных
- Разделить assert'ы по *behavior*'ам.

# Проект VerKer

- В ИСП РАН ведется дедуктивная верификация исходного кода ядер некоторых операционных систем.
- В проекте VerKer верифицированы некоторые функции, оперирующие с буферами (`string.h`).
- Репозиторий проекта — `https://github.com/evdenis/verker`.
- Обратите внимание на верификацию функции `sysfs_streq` — ghost-переменные там существенно упрощают условия верификации.

# Behavior

- Это синтаксическая конструкция, позволяющая дать имя части спецификации, а затем ссылаться на нее в доказательстве.
- Синтаксис см. в ACSL Language Reference.
- Можно пометить assert, loop invariant именем behavior (`for B: ...`), и тогда этот assert и loop invariant будут исключены из условий верификации для остальных behavior'ов.
- К сожалению, нельзя пометить вызов ghost-функции именем behavior'а.

# Проекции

Если никакие методы сокращения не работают или к тому есть технические препятствия, то можно пробовать генерировать специальные варианты верифицируемого кода для доказательства определенного условия верификации, из которых убрано все лишнее. Но тут никакой поддержки инструмент уже не окажет.