# Лекция 11. Фрейм функции. Язык спецификации ACSL.

# Цель лекции

Обсудить вопросы спецификации Си-программ.

# Содержание

# Мотивация

- Вы уже написали функцию, заменяющую одно значение на другое значение в массиве.
- Напишите еще одну функцию, чтобы проверить спецификацию первой функции. Функция принимает на вход массив (указатель на начало и размер) и 3 значения `v0`, `v1`, `v2`. Размер массива не должен быть меньше 2. Постусловие функции – что 0-й элемент массива становится равным `v0`, а 1-й элемент массива становится равным `v2`. Функция сначала присваивает в 0-й элемент массива значение `v0`, в 1-й элемент массива значение `v1`, затем вызывает первую функцию для части массива, с 1-го элемента до его конца.
- Почему одно из условий постусловия не доказывается?

# Фрейм функции

- В спецификации первой функции не сказано, что может делать функция за рамками переданного массива. Значит, она может делать что угодно.
- Необходимо думать о границах области памяти (рамке, фрейме, footprint), которую разрешено изменять функции.
- Добавьте соответствующее постусловие к первой функции.
- Проверьте, что если второй функции передать еще один массив и с ним вторая функция ничего не делает, то этот факт доказывается!

# Содержание

# Frama-C

- Система статического анализа Си-программ.
- Состоит из ядра и плагинов. Ядро – это фронтенд анализа, плагин – бекэнд анализа.
- Один из плагинов (AstraVer) – дедуктивная верификация.
- Плагины могут взаимодействовать друг с другом для компенсации недостатков друг друга. Пример: есть плагин, который сам выводит несложные инварианты цикла, и уже на этой основе другой плагин сообщает об ошибке. Всё это делается полностью автоматически.
- `frama-c -av file.c ...`

# Работа на уровне исходного кода

- Исходный код может быть снабжен аннотациями для более точного анализа. Для дедуктивной верификации аннотациями записывается спецификация, инварианты цикла и т.п.
- Причем пользователь Frama-C работает исключительно на уровне исходного кода (иначе будет тяжело автоматизировать комбинирование плагинов).
- На уровне исходного кода не доступна его модель на WhyML. Эта модель может иметь существенные особенности по сравнению с исходным Си-кодом, важные для верификации.

# Модель кода WhyML не доступна

- (-) Например, нельзя написать лемму, в которой используются функциональные символы из модели памяти.
- (+) Плагин *AstraVer* может самостоятельно строить модель программы, применяя различные оптимизации для более эффективной верификации.

# Содержание

# Что такое ACSL

- ACSL – ANSI/ISO C Specification Language.
- Frama-C использует подмножество языка ACSL.
- Исходный код дополняется *аннотациями*: комментариями `/*@ ... */`

# Аннотации ACSL для спецификации функции

Спецификация функции – это аннотация, которая расположена перед заголовком функции.

- requires expr; – предусловие (можно несколько requires или ни одного)
- ensures expr; – постусловие (можно несколько ensures или ни одного)
- decreases expr; – оценочная функция для рекурсивной функции (фундированное множество – то же, что в WhyML)
- Синтаксис выражений похож на WhyML, но он сделан более похожим на Си (синтаксис кванторов!).

# Типы данных

- `integer` – бесконечный целый тип (только для спецификаций)
- Все Си-шные типы тоже есть
- В операциях с `int` и `integer` происходит преобразование `int` к `integer`

# Модель памяти недоступна

- Типы-символы, функциональные и предикатные символы модели памяти для одного вида анализа может не подходить для другого вида анализа, поэтому язык ACSL старается включать только общие возможности разных моделей памяти
- Наша модель памяти недоступна (типы-символы, функциональные и предикатные символы, аксиомы и леммы)
- Поэтому в спецификациях для указателей надо использовать Си-шное разыменование, сложение, вычитание, сравнение.

# At, Old, метки памяти

- Есть конструкции `\old`, `\at`
- Метки памяти – это Си-метки + `Pre`, `Here`, `Post`
- Не путать `\at(*p,L)` и `*\at(p,L)`

# Спецификация фрейма функции

- Модель памяти недоступна, поэтому для фрейма ввели дополнительные части спецификации функции:
- assigns Locations; – валидная память за пределами Locations не меняет своего значения при выходе из функции
- allocates Locations; – валидная память за пределами Locations не меняет своего статуса аллоцированности при выходе из функции (если была освобождена, остается освобожденной; была выделенной, остается выделенной); Locations вычисляется в состоянии памяти при возврате из функции
- frees Locations; – то же, что allocates, но Locations вычисляется при входе в функцию

# Спецификация циклов

- Аннотация цикла записывается перед циклом. Точки сечения, фундированное множество – те же, что в WhyML.
- `loop invariant expr;` – индуктивное утверждение (может быть несколько или отсутствовать)
- `loop variant expr;` – оценочная функция (если ее нет, то она равна 0)
- `loop assigns Locations;` – фрейм цикла

# Символы

- Можно вводить свои символы!
- Типы-символы, функциональные и предикатные символы, аксиомы пишутся в аннотациях `axiomatic`
- Функциональный символ начинается со слова `logic`
- Предикатный символ начинается со слова `predicate`
- Аксиома начинается со слова `axiom`
- Полиморфные типы не поддерживаются

# Снова без модели памяти

- Модель памяти не доступна даже в axiomatic, это нужно учитывать.
- Если нужны метки памяти для функциональных и предикатных символов, их надо писать в фигурных скобках после имени символа, а потом использовать в конструкции `\at`.
- Меток может быть несколько – и это удобно! (для разыменования каждого указателя своя метка – и можно использовать такой предикатный символ в очень разном контексте, не создавая много однотипных предикатных символов)

# Но есть предикаты, входящие в язык

- `\valid` – валидность указателя
- `\offset_min`, `\offset_max`
- `\base_addr` (правда, AstraVer транслирует только равенство двух base_addr – это проверка, что два указателя находятся в одном блоке)
- `\allocable`, `\freeable`

# Аннотации для верификации

- `assert expr;`
- `ghost`-метки и локальные переменные (тип - только Си-шный), `ghost`-блоки (только с Си-шным кодом): это из-за того, что в Frama-C фронтенд кода принимает только Си-код (не ghost-аннотация анализируется фронтендом отдельно, поэтому с ней нет этих проблем)
- `ghost`-функции
- `lemma` – леммы

# Behavior

- Можно указать у assert, что он нужен для доказательства определенной аннотации ensures
- То же можно сделать с loop invariant
- Ensures надо поместить в behavior
- Синтаксис смотрите в документации по ACSL

# Лемма-функции

Пример:

```
/*@ ghost
      /@ lemma
            requires ....
            ensures ....
      @/
      void lemmafunction(....) {
        ....
      }
*/
```

# Построение теорий WhyML и их зависимости

- Каждый axiomatic и глобальная лемма становятся отдельной теорией.
- В теорию импортируются все теории, в которых объявляются символы, используемые в первой теории.
- Для более тонкого управления импортированием надо объявить символ и воспользоваться им в нужном месте.
- В условиях верификации используются все леммы, расположенные до верифицируемой функции.

# Триггеры

- В кванторах ACSL нет триггеров.
- Солвер будет выбирать триггер, исходя из квантора, каким он будет в WhyML! Квантор может сильно отличаться от его ACSL-варианта.

# Переносимость Си-программ и верификации

- В Си размеры типов выбирает платформа, а не язык.
- Frama-C фиксирует размеры типов во фронтенд анализе (опциями можно настраивать эти размеры). Дедуктивная верификация делается с этими размерами типов.
- Последовательность вычислений тоже фиксируется фронтендом Frama-C.

# Глобальные инварианты

- Это утверждения, которые выполнены всегда, когда программа находится в определенной точке программы
  - строгие - везде
  - слабые - при вызове и при возврате из каждой функции
- Инвариант глобальных переменных:

```
global invariant positive: size >= 0;
```

- Инвариант (каждой переменной) типа:

```
type invariant valid_array (Array *a) = a->size >= 0
&& \valid(a->data + (0 .. a->size - 1));
```

# Проблемы глобальных инвариантов

- Пусть у некоторого типа должен быть инвариант. У любой ли функции этот инвариант должен быть выполнен хотя бы в слабом смысле?
- Для функции-конструктора? (а что это в Си?)
- Для функции-деструктора? (а что это в Си?)
- Для вспомогательной функции, вызываемой из «публичной» функции? Любой такой вспомогательной функции?

# Подход AstraVer

- Надо точнее специфицировать, когда должен быть выполнен глобальный инвариант. Вводить дополнительные конструкции в язык спецификации?
- В AstraVer глобальные инварианты не вставляются автоматически в спецификации функций!
- Это просто предикаты, которые надо явно указать в спецификациях нужных функций.

# И это не все проблемы

- Пусть некоторый тип является частью другого типа. Причем не все корректные значения внутреннего типа являются допустимыми в рамках внешнего типа.
- Каков должен быть инвариант внутреннего типа? Как доказать, что модификация переменной внутреннего типа не нарушает инварианта переменной внешнего типа, если нет доступа до переменной внешнего типа?
- До сих пор нет лучшего ответа на этот вопрос.