Лекция 6. Спецификация и верификация программ из нескольких функций

Цель лекции

Узнать особенности спецификации и верификации рекурсивных функций, а также функций, вызывающих другие функции. Сформулировать правила спецификации интерфейсов.

Содержание

Пример: быстрая сортировка

- Исходный код примера смотрите в директории `examples\qsort`.
- Шаги примера записаны в `Makefile`.
- `qsort_1.h` — заголовок функции `qsort` и спецификация функции. Она должна отсортировать по неубыванию данный массив из целых чисел.
- `qsort_2.c` — определение функции `qsort`. Быстрая сортировка делит сортируемый диапазон на 3 части: опорный элемент, все меньшие-или-равные-опорному элементы, все большие-или-равные-опорному элементы и сортирует каждый поддиапазон. Разделение на части выполняет функция `partition`.
- `part_3.h` — объявление функции `partition` и ее спецификация.

Методы Флойда при наличии вызовов функций

- Надо доказать полную коррек-ть вызывающей функции относительно **любой** реализации вызываемой функции, полностью корректной относительно своей специф-ции.
- Добавить условие в safety: на всех базовых путях из точки сечения к месту вызова функции должно быть выполнено предусловие вызываемой функции.
- Изменить определение предиката пути: если внутри базового пути встретился вызов функции, то в предикате пути, проходящем через него, помещается постусловие функции и дополнительные переменные для возвращаемых значений вызываемой функции.
- Важно, чтобы существовало хотя бы одно определение вызываемой функции, полностью корректной относительно ее спецификации. Иначе доказательство полной корректности вызывающей функции бесполезно.

Метод фундированных множеств для рекурсии

- Вычисление для рекурсивной функции — это дерево: каждый вызов функции порождает поддерево, непосредственные дети узлов - это последовательность конфигураций функции родительского узла.
- Классический метод фундированных множеств позволит доказать конечность количества детей у каждого узла. Осталось доказать, что количество узлов конечно. Или, что в дереве нет бесконечных путей. Путь — это цепочка рекурсивных вызовов.
- Вводим фундированное множество и оценочную функцию для узла дерева. У каждого дочернего узла оценочная функция должна быть меньше оценочной функции родительского узла. И значение оценочной функции должно принадлежать фундированному множеству. В роли индуктивных утверждений выступает предусловие ф-ции.

qsort: доказательство safety

- В ACSL фундированное множество для доказательства завершения рекурсии — неотрицательные целые числа и стандартное арифметическое отношение «меньше». Оценочная функция указывается в аннотации `decreases`.
- `make step1`. Аннотация `decreases` указана не у заголовка функции `qsort`, а у реализации функции, ведь могут быть и нерекурсивные реализации, а аннотации у заголовка будут относиться ко всем реализациям. Аннотации у заголовка и у реализации соединяются вместе.

qsort: доказательство перестановочности

- Многие условия верификации не доказываются. Добавляем `assert`ы, чтобы понять, почему они не доказываются.
- `make step2` — добавлены `assert`ы. Делаем вывод, что для доказательства условий верификации и `assert`ов нужны дополнительные леммы. Возможно, они будут полезны для доказательства условий верификации про разные Си-функции. Поэтому имеет смысл разместить леммы в отдельном файле и подключать его по необходимости.
- Размещаем леммы в файле `permut_min_theorems.h`. Можно не #includeить его, а запустить `frama-c` сразу с несколькими файлами. В этом случае еще полезна опция `-av-file-name`, задающая имя директории для сессии. Если этого не сделать, то оно будет выбрано по умолчанию как `whole_program`. Посмотреть все опции плагина AstraVer можно так: `frama-c -av-help`.

qsort: доказательство перестановочности

- `make step3` — запуск с несколькими файлами.
- Не доказывается `assert` о том, что после второго вызова `qsort` левый сегмент остался неотронутым. Почему? Смотрим в why3: после второго вызова состояние памяти поменялось и известно про нее лишь то, что записано в спецификации `qsort`. Там не записано, что память за пределами сортируемого отрезка не изменилась. Чтобы выразить такие свойства, есть аннотация функции `assigns`. У нее указываются lvalue, за пределами которых память сохраняется такой же, какой была до вызова функции. Что происходит с самими lvalue, записывается в аннотации `ensures`.

qsort: доказательство перестановочности

- Обращайте внимание на требования о том, какая память должна сохраняться неизменной после возврата из функции по сравнению с состоянием до вызова функции. Такие требования записываются при помощи аннотации `assigns`.
- `assigns \nothing;` — вся память не должна измениться.
- `make step4` — добавили аннотацию `assigns a[l..u]`.
- Куда следует добавить `assigns`: к спецификации заголовка или реализации? К спецификации заголовка, т.к. это требование должно быть справедливо для всех реализаций.
- Для доказательства некоторых условий потребовалось сделать один раз inline.

qsort: доказательство упорядоченности

- `make step5` — добавляем `assert`ы для доказательства упорядоченности. Не доказано лишь Postcondition-Assigns. То есть, что функция `qsort` в целом не меняет память вне `[l .. u]`. Это так? Да, если функция `partition` не меняет память вне этого диапазона. Не указали у нее аннотацию `assigns`, а она оказалась нужна.
- `make step6` — добавили `assigns` к функции `partition`. Доказана полная корректность функции `qsort` относительно всех реализаций `partition`, полностью корректных относительно ее спецификации.
- `make step7` — показываем, что существует определение функции `partition`, полностью корректное относительно ее спецификации. В цикле использовали аннотацию `loop assigns` — вся память за пределами указанных там lvalue не менялась между входом в цикл и текущей итерацией.

allocates, frees

- Нужно обращать внимание на требования о том, какую память не может освобождать и выделять функция.
- Такие требования записываются при помощи аннотаций `allocates` и `frees`.
- У аннотации `allocates` записываются lvalue типа указатель, вычисляемые в состоянии памяти Post, и функция обязана сохранить выделенной всю память, которая была выделена перед вызовом функции, за исключением, быть может, lvalue.
- Аннотация `frees` означает то же, но lvalue определяются в состоянии памяти Pre. Если есть несколько аннотаций `allocates`, `frees`, их области lvalue объединяются.

Выводы-1

- аннотация `decreases` для доказательства завершаемости рекурсивных функций.
- обращать внимание на требования о том, какую память не должна менять функция.
- аннотация `assigns` для спецификации таких требований.
- есть еще аннотация цикла `loop assigns`.
- аннотации `allocates` и `frees` для спецификации того, какую память не должна освободить или выделить функция.
- подробности синтаксиса и семантики этих аннотаций смотрите в документации по ACSL.

Содержание

Нужен assigns?

- Почему на шаге 5 в примере с `qsort` потребовался `assigns`? Почему до этого мы не сталкивались с такой аннотацией, но условия про сохранение памяти доказывались?
- Пример `examples_regions`. В файле `graph_1.c` размещена функция `insert_edge`, спецификация и реализация.
- Там же размещена функция `insert_double_edge`, которая вызывает функцию `insert_edge`.
- В спецификации `insert_edge` отсутствуют аннотации `assigns`, `allocates`, `frees`. То есть поведение функции дополнительно не ограничивается. По коду функции определяется, какие части памяти изменяются, а какие — сохраняются.

Уточнение assigns

- В этой функции не меняется память под саму переменную `*graph` и под массив `graph->vertices`. Но меняется элемент массива `graph->edges`. Поэтому 3-й `assert` в `insert_double_edge` не доказывается.
- Итак, в спецификации функции `insert_edge` надо добавить свойство, что изменился только один элемент массива `graph->edges`.
- `make step2` — первая идея: добавить аннотацию `ensures`
- `make step3` — но она не всегда работает
- `make step4` — правильная спецификация этого свойства (при помощи `assigns`) должна учитывать *регионы*.

Регион

Регион — это множество указательных переменных, которые могут быть синонимами, т.е. их значения могут принадлежать одному и тому же блоку. Это значит, что изменение памяти через один из указателей региона может повлиять на указатель из другого региона.

Если бы не было регионов, то везде пришлось бы писать сложные предикаты о том, какая часть памяти не должна меняться, и иметь другую модель памяти, не дающую эффективную верификацию не слишком тривиальных программ.

Определение регионов

Вычисление регионов — это статический межпроцедурный анализ. Он изложен в одной из статей в списке литературы курса.

- выделяется память – это новый регион
- указатели разных типов – из разных регионов
- указатели не сравниваются (отдельные два аргумента функции) – из разных регионов (но если функция вызывается с указателями из одного региона, то указатели-аргументы должны принадлежать одному региону)
- указатели сравниваются, присваиваются (в частности, как смещения в одном массиве) – значит их регионы надо объединить
- эти правила применяются как к Си коду, так и к спецификации, причем ко всему доступному коду

Зачем знать о регионах?

При верификации следует прогнозировать результат анализа и писать дополнительные утверждения про неизменность части региона. В `make step3` не обеспечивало точную передачу свойства неизменности при помощи `ensures`.
Данное свойство невозможно выразить в виде предиката, если в языке спецификации нет способа выразить регионы.
Поскольку разделение на регионы не относится к языку спецификации ACSL, а является одной из эвристик инструмента AstraVer, то предикат записать нельзя. Аннотация `assigns` избавлена от этой проблемы: все содержимое региона, кроме указанного в аннотации, сохраняется.

same block

- Полезное замечание: проверка `\base_addr(p1) != \base_addr(p2)` транслируется во внутренний предикат `same_block(p1, p2)`. Это не предикат ACSL, его нельзя написать напрямую в спецификации.
- Обращайте внимание на то, есть ли требования на разделение по блокам.
- `make step5` — не доказывается некоторые условия в функции `insert_edge_to_graphs`, т.к. все массивы `edges` в графах в массиве `graphs` находятся в одном регионе, значит, должно быть важно, что с разделением их по блокам. А про это ничего не требуется в предусловии.
- `make step6` — добавлено разделение по блокам. Теперь все условия доказываются.

Представление региона

- таблица значений — `map`, который используется для представления операции разыменования
- таблица блоков — `alloc_table`, который используется для представления блоков, т.е. определения предикатов `\offset_min`, `\offset_max`, `\valid`, `\freeable`, `\allocable`
- таблица тегов — `tag_table`, который используется для представления динамических типов объектов в блоке

Содержание

Интерфейсы

- Мы уже разделили верификацию программы на верификацию по отдельным функциям. У каждой функции есть спецификация и определение (реализация).
- Как сделать разделение, если есть не все реализации. Спецификации для всех функций есть. Задача та же — доказать полную корректность относительно всех реализаций вызываемых функций, полностью корректных относительно их спецификаций.
- Здесь нас поджидает одна особенность...

Пример

- Пример с прошлой лекции про создание и удаление графа. У нас будет библиотека работы с графами. Реализация библиотеки будет находиться отдельно от кода, использующего эту библиотеку.
- Исходный код примера: `examples\interfaces` — оформили как библиотеку. Написали модуль `test_3.c`, использующий библиотеку. Модуль кажется странным, т.к. в нем требуется доказать `assert \false;`, т.е. нет ли ошибки в спецификации библиотеки.

Пример (2)

- `make step1` — доказываем полную корректность модуля `test_3.c` относительно конкретной реализации библиотеки с графами. То есть вызываем `frama-c` с несколькими файлами. Конечно, `assert` не доказывается, ошибки нет.
- `make step2` — доказываем полную корректность модуля `test_3.c` без указания реализации библиотеки. И теперь `assert` почему-то стал доказываться... Видим, что поведение инструмента отличается, хотя спецификации не менялись.
- сравниваем код why3 и видим, что во втором случае AstraVer не видел реализации и не знал, что состояние памяти поменяется.

Автоматическое дополнение спецификации по заголовку

Если реализации функции нет, то регионы формируются по следующим правилам:

- Неконстантный указатель-аргумент $\rightarrow$ таблица значений региона меняется
- Константный указатель-аргумент $\rightarrow$ таблица значений региона не меняется
- Нет `allocates` или `frees` $\rightarrow$ `alloc_table` региона не меняется
- Есть `allocates v;` или `frees v;` $\rightarrow$ `alloc_table` региона переменной `v` меняется

Вопросы

- можно ли правильно описать спецификацию заголовка функции `graph_create` и `graph_destroy`?
- как писать спецификацию для библиотек, которые сами выбирают свои типы данных? (т.е. имея только заголовок функции, невозможно автоматически вычислить регионы)
- как понимается модульная верификация? Можно ли верифицировать функции по отдельности? Если можно, то в каких предположениях?