Лекция 7. WhyML, автоматизация построения условий верификации

Цель лекции

Познакомиться с языком WhyML и научиться использовать его для автоматизации дедуктивной верификации.

Содержание

Общая характеристика

- Инфраструктура Why3 умеет генерировать условия верификации (`why3 ide file`).
- *WhyML* – функциональный язык со средствами формальной спецификации и дедуктивной верификации.
- Обычно выполняет роль промежуточного языка между языком программирования и средствами верификации.

Теории и модули

- Теории – набор определений и явных целей доказательства
- Модули – набор определений и моделей программ, для моделей программ надо доказать полную корректность, т.е. сгенерировать условия верификации
- Синтаксис теории: `theory Name ... end`
- Синтаксис модуля: `module Name ... end`
- Файл `*.why` не может содержать модули, с модулями файл должен называться `*.mlw`

Типы данных

- Целые числа: `use import int.Int`
- Пустой тип `unit`
- Декартово произведение `type T = (T1,T2)`
- Структуры, поля которых можно делать изменяемыми – единственный способ реализовать переменные и операцию присваивания `type S = { mutable v: T }`
- Поле структуры – это еще и функция, поэтому его имя входит в глобальную область видимости
- Типы могут быть полиморфными `type array 't`

Операции и выражения

- Любое вычисление – это выражение
- `e1 ; e2` – последовательность вычислений выражений
- `s.v <- e` – присваивание в изменяемое поле `v` структуры `s`
- `()` – пустое вычисление (выражение типа `unit`)
- `(expr,expr)` – кортеж
- `{v = 3}` – новое значение типа структуры, у которого есть поле `v` (такой тип единственный)
- условие `if-then-else`, циклы `while`, `for`

Функции и локальные переменные

- Локальная переменная: `let var = expr in ...`
- Разбиение кортежа: `let (v1,v2) = expr in ...`
- Неанонимная функция – модель программы: `let function (arguments) : returntype = expr`
- Можно указать только заголовок функции, тогда вместо `let` надо написать `val`. Заголовок заканчивается перед символом равно.
- Функция нужна только для доказательства? После `let` надо написать `ghost`.
- Это лемма-функция? После `let` надо написать `lemma`.
- Нужна рекурсия? Надо приписать после `let` слово `rec`.

Пример модуля с функцией

```
module Factorial
use import int.Int
let rec fact_rec (n: int): int =
    if n = 0 then 1 else n * fact_rec (n - 1)
type intVar = { mutable v: int; }
let fact_loop (n: int): int =
    let i = { v = 1; } in
    let r = { v = 1; } in
        while i.v <= n do
            r.v <- r.v * i.v;
            i.v <- i.v + 1
        done;
        r.v
end
```

Исключения

- Исключения прерывают вычисление
- Исключения используются для возврата из функции до ее завершения
- Надо описать тип исключения (конструкция exception)
- try-блок, with-обработчик, raise-генерация исключения

Запись спецификации

- Предусловие и постусловие пишется между заголовком функции и символом равно.
- Предусловие состоит из 1 или более конструкций `requires {...}`
- Постусловие состоит из 1 или более конструкций `ensures {...}`
- Возвращаемое значение записывается как `result`.
- Несколько requires означает их конъюнкцию. Аналогично с ensures.
- Ни одного requires означает истину. Аналогично с ensures.

Пример спецификаций

```
module SqrtExample
use import int.Int
val sqrt (n: int) : int
    requires { n >= 0 }
    ensures { result * result <= n <
    (result + 1) * (result + 1) }
let foo (n: int) : int
    requires { n >= 0 }
    ensures { result >= 0 }
= sqrt n
end
```

Методы Флойда

- Точки сечения выбраны жестко: перед проверкой условий циклов
- Индуктивное утверждение записывается так: `invariant {...}` и пишется после слова `do`, если речь о цикле `while`
- Несколько `invariant` возможно, это означает их конъюнкцию
- Фундированно множество выбрано жестко: целые числа и сравнение меньше
- Оценочная функция цикла записывается так: `variant {...}` и пишется там же, где `invariant`.
- Оценочная функция рекурсии записывается так же, но после предусловия.

Пример спецификации цикла

```
module MultLoop
use import int.Int
type intVar = { mutable v: int; }
let loop_test (a b: int): int
    requires { a >= 0 }
    ensures { result = a * b }
= let i = { v = 1; } in
  let r = { v = 0; } in
    while i.v <= a do
        invariant { i.v <= a + 1 }
        invariant { r.v = (i.v - 1) * b }
        variant { a + 1 - i.v }
        r.v <- r.v + b; i.v <- i.v + 1
    done; r.v  end
```

Содержание

Доказательство примера

- Перепишите блок-схему вычисления квадратного корня в виде функции на языке *WhyML*. Напишите спецификацию и всё необходимое для методов Флойда.
- Завершите доказательство полной корректности: укажите недостающие леммы, докажите леммы по индукции или через последовательность других лемм, используйте вместо лемм вызов ghost-функции.

assert

- Вместо написания отдельной ghost-функции с пустым телом для доказательства некоторого утверждения удобнее воспользоваться конструкцией assert и не писать эту функцию.
- Синтаксис такой: `assert { condition }`
- Это аналогично ghost-функции с пустым телом и постусловием `condition`. Предусловие собрано из всех условий, которые находятся перед assert.

Пример с assert

```
module Theorems

use import int.Int

let theorem (x: int)
    requires { x > 0 }
    ensures { exists t. t > 0 /\ x * t <> x }
=
    assert { forall u. u * 2 = u + u };
    ()

end
```

ghost-переменные

- Если надо доказывать много утверждений про квантор существования, может быть проще завести переменную только для целей доказательства и нужным образом ее инициализировать или даже изменять прямо в функции. Такие переменные называются ghost-переменными.
- Синтаксис такой: `let ghost var = ... in ...`

Содержание

Мотивация

- Для каждого типа изменяемой переменной приходится писать тип с единственным mutable полем, причем имя поля должно быть уникально, это неудобно
- Можно решить проблему с уникальностью имени поля, сделав тип полиморфным, т.е. имеющим типовой параметр
- тем самым, уникальность будет обеспечивать значение типового параметра

Полиморфный тип

Это может выглядеть примерно так:

```
type s (* only small letters! *)
type ref 't = { mutable contents: 't; }
let prog (x: int) (sv: s) =
    let y = { contents = 0 } in
        (* type of y is ref int *)
    let z = { contents = sv } in
        (* type of z is ref s *)
        ()
```

ref.Ref

- В стандартной библиотеке why3 есть модуль `ref.Ref`.
- Он экспортирует такой же тип и функции для удобной работы с изменяемыми переменными.
- `let y = ref 0 in` – создание переменной
- `y := !y + 1` – чтение переменной через восклицательный знак, изменение как в паскале