

Формальные методы в программной инженерии

Система верификации Why3

магистерский курс, 2021

Шелехов Владимир Иванович,
зав. лаб. Системного программирования ИСИ, к.т.н.,
Институт Систем Информатики, Новосибирск,
Новосибирский Государственный Университет,
vshel@iis.nsk.su, р.т. (383) 330-27-21, ИСИ СО РАН, к.269

15.10.21

ФМЗ

Лекция 1

Архитектура Why3. Язык WhyML

Команды интерактивного доказательства

<http://why3.lri.fr/manual.pdf>

Платформа Why3

Команды Why3

Типы, термы, выражения языка WhyML

Модули, предикаты, функции

Спецификации

Примеры

Команды интерактивного доказательства

Платформа Why3

Язык программирования и спецификаций **WhyML**

Графический интерфейс платформы **Why3**.

Язык команд для запуска и управления режимами **Why3**.

<http://why3.lri.fr/> - сайт **Why3**

Внешние инструменты автоматического доказательства.

SMT-решатели: Alt-Ergo, CVC3, CVC4, Simplify, veriT, Yices, Z3;

Другие автоматические инструменты: Beagle, E prover, Gappa, Metis, Metitarski, Princess, Psyche, SPASS, Vampire.

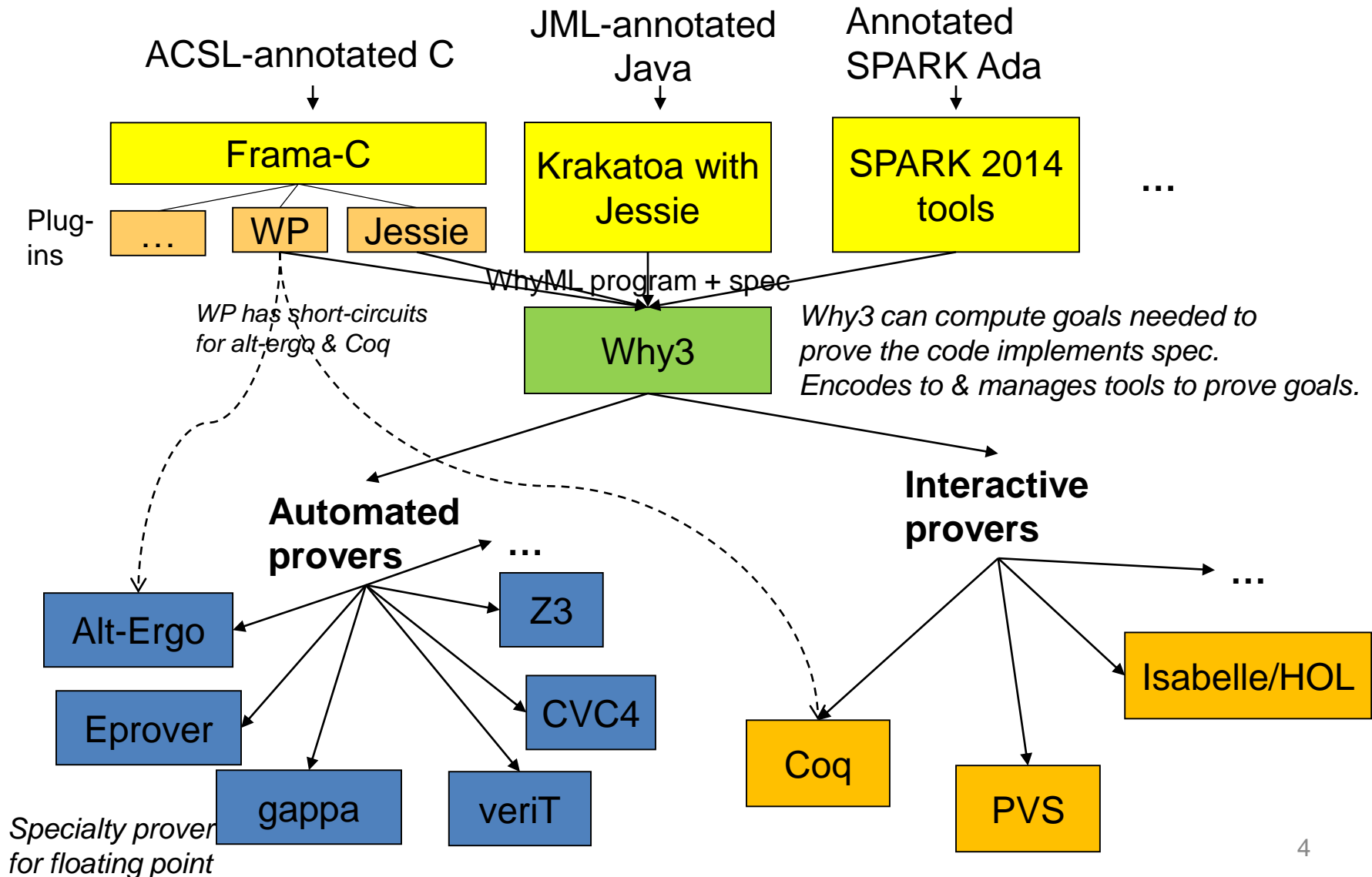
Инструменты интерактивного доказательства:

Coq, PVS, Isabelle/HOL.

Команды доказательства (трансформации)

Стандартная библиотека теорий для поддержки доказательства

Применение Why3



Командный интерфейс Why3

Запуск инструмента **Why3** с терминала:

```
why3 ide hello.mlw
```

Файл с расширением «**mlw**» -

- программа (или теория) на языке **WhyML**

Директория **.config**

```
why3 config --detect-provers
```

```
why3 --list-provers
```

Why3 proof session

The screenshot displays the Why3 IDE interface during a proof session. The left pane shows a tree of goals for the file `./maxsum_solution.mlw`. The tree is organized as follows:

- ./maxsum_solution.mlw
 - Kadane
 - VC maximum_subarray [VC for maxim
 - split_vc
 - 0 [loop invariant init]
 - 1 [loop invariant init]
 - 2 [loop invariant init]
 - 3 [loop invariant init]
 - 4 [index in array bounds]
 - 5 [loop invariant preservation]
 - 6 [loop invariant preservation]
 - 7 [loop invariant preservation]
 - 8 [loop invariant preservation]
 - 9 [loop invariant preservation]
 - 10 [loop invariant preservation]
 - CVC4 1.5 (0.03)
 - Alt-Ergo 1.30 (0.01 (steps: 31))
 - 11 [loop invariant preservation]
 - 12 [loop invariant preservation]
 - 13 [loop invariant preservation]
 - 14 [loop invariant preservation]
 - 15 [loop invariant preservation]
 - 16 [loop invariant preservation]
 - 17 [loop invariant preservation]
 - 18 [loop invariant preservation]
 - 19 [loop invariant preservation]
 - 20 [loop invariant preservation]
 - 21 [postcondition]
 - 22 [postcondition]
 - 23 [out of loop bounds]

```
Task ./maxsum_solution.mlw
25
26 let maximum_subarray (a: array int): int
27 ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
28 ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h = result }
29 =
30 let maxsum = ref 0 in
31 let curmax = ref 0 in
32 let ghost lo = ref 0 in
33 let ghost hi = ref 0 in
34 let ghost cl = ref 0 in
35 for i = 0 to a.length - 1 do
36 invariant { forall l : int. 0 <= l <= i -> sum a l i <= !curmax }
37 invariant { 0 <= !cl <= i /\ sum a !cl i = !curmax }
38 invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !maxsum }
39 invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !maxsum }
40   curmax += a[i];
41   if !curmax < 0 then begin curmax := 0; cl := i+1 end;
42   if !curmax > !maxsum then begin maxsum := !curmax; lo := !cl; hi := i+1 end
43 done;
44 !maxsum
45
46 end
47
```

0/0/0

Messages | Log | Edited proof | Prover output | Counterexample

Messages

Структура языка WhyML

WhyML – язык функционального программирования.

Язык формул why3 первого порядка без типизации переменными – это термы **term**

Операторы – это выражения **expr**

Файл с расширением «**mlw**» -- посл-ть модулей

Модуль: **module** <имя модуля> <описания> **end**

Описания типов, констант, предикатов, функций, индуктивных определений, аксиом, теорем, исключений.

Импорт, экспорт, клонирование

Вкрапления **императивных конструкций**: модифицируемые переменные, циклы **for** и **while**, указатели

Язык WhyML. Лексика

Основные лексемы: целые и вещественные константы, литеры (типа **char**), строковые константы, имена, операции

Имена:

Со **строчной** буквы - имена переменных, типов, предикатов, функций.

С **заглавной** буквы - имена теорий, модулей, теорем, целей, ...

С **апострофа** – аргументы типов.

Имя может завершаться апострофом

`qualifier ::= (uident ".")+`

`lqualid ::= qualifier? lident`

`uqualid ::= qualifier? uident`

Набор операций: префиксные, инфиксные.

4 приоритета инфиксных операций.

Префиксные сильнее инфиксных.

Язык WhyML. Типы

Простые встроенные типы: **int**, **real**, **char**, **bool**, **unit**.

Отображения (map), кортежи (tuple), алгебраические типы, простые диапазоны. Массивы

Типовые термины: простой тип

в позиции типа

<имя типа> <параметры>

type -> **type** --mapping type

(**type** (, **type**)⁺) --tuple type

Описание типа:

type <имя типа> <параметры> = <определение типа>

Записи, алгебраические типы, простые диапазоны.

Параметры типов -- с апострофом

Тупли (type1, type2, ...)

Тип диапазон **type** r = <**range** a b> a и b --константы

Язык WhyML. Записи

Описание типа записи:

```
type t = { a: int; b: bool }
```

Агрегат { a = 42; b = true } - выражение типа запись

Модифицируемое поле:

```
type t = { mutable a: int; b: bool }
```

Разрешается оператор присваивания x.a <- 42

Спецификация: **writes** { x.a }

Инвариант типа:

```
type t = { mutable a: int; b: bool }
```

invariant { b = true -> a >= 0 } должен быть истинным в начале и конце

by { a = 42; b = true } свидетельство непустоты

```
let f2 (x: t) = x.a <- x.a - 1; assert { x.a < old x.a }; x.a <- 0
```

Язык WhyML. Алгебраические типы

Описание алгебраического типа:

```
type option 'a = None | Some 'a
```

'a – параметр типа, None и Some – конструкторы.

Рекурсивный тип списка:

```
type list 'a = Nil | Cons 'a (list 'a)
```

'a – параметр типа, Nil и Cons – конструкторы.

Примеры функций со списками

```
type list a = Nil | Cons a (list a)
```

```
function append(l1:list a,l2:list a): list a =  
match l1 with  
| Nil → l2  
| Cons(x,l) → Cons(x, append(l,l2))  
end
```

```
function length(l:list a): int =  
match l with  
| Nil → 0  
| Cons(_,r) → 1 + length r  
end
```

```
function rev(l:list a): list a =  
match l with  
| Nil → Nil  
| Cons(x,r) → append(rev(r), Cons(x,Nil))  
end
```

Язык WhyML. Термы

- `term ::= целые / вещественные константы | "true" | "false"`
- `| имя переменной или константы`
- `| <операция или имя функции> term+` также возможно (+), ([]), ... в качестве префиксных
- `| <унарная префиксная> term`
- `| "(" term ")"` term в указанном контексте
- `| term "." lqualid` поле записи
- `| "{" term_field+ "}"` вида `field = value`
- `| "{ term "with" term_field+ "}"` модификация записи
- `| term "[" term "]" ""*` доступ к элементу коллекции (массиву)
- `| term "[" term "<-" term "]" ""*` модификация элемента коллекции
- `| term "[" term ".." term "]" ""*` вырезка коллекции
- `| term "[" term ".." "]" ""* | term "[" ".." term "]" ""*` справа и слева

Агрегат `{ a = 42; b = true }` - выражение типа запись

Коллекция: отображение, массив, список, последовательность

Язык WhyML. Другие термы

term ::=

- | "if" term "then" term "else" term условный терм
- | "match" term "with" term_case+ "end" разбор случаев; pattern matching
- | "let" pattern "=" term "in" term let-binding
- | "fun" param+ "->" term безымянная функция как значение
- | "{" (lqualid "=" pattern ";")+ "}" record pattern
- | <имя конструктора> pattern* конструктор
- | pattern "," pattern tuple pattern

term_case ::= "|" pattern "->" term

pattern ::= binder variable or "_"

binder ::= "_" | bound_var

bound_var ::= lident attribute*

Формулы why3

Часть конструкций `term` и `expr`

`formula ::= true | false | formula -> formula`

| `formula <-> formula`

| `formula ∧ formula`

| `formula && formula`

| `formula ∨ formula`

| `formula || formula`

| `not formula`

| `prefix-op term`

| `term infix-op term`

| `lqualid term+`

| `if formula then formula else formula`

| `let pattern = term in formula`

| `quantifier binders (, binders) . formula`

| `(formula)`

implication

equivalence

conjunction

asymmetric conj.

disjunction

asymmetric disj.

negation

predicate application

conditional

local binding

quantifier `forall` | `exists`

parentheses

Язык WhyML. Выражения

`expr ::=` тот же набор конструкций, что и для термов, **плюс:**

- | `expr spec+` выр с добавленной спецификацией
- | `"match" expr "with" ("|" pattern "->" expr)+ "end"` выр по образцу
- | `qualifier? "begin" spec+ expr "end"` абстрактный блок
- | `expr ";" expr` последовательность выражений
- | `"let" pattern "=" expr "in" expr` выр с приставкой `let`
- | `"let" fun_defn "in" expr` локальная функция
- | `"let" "rec" fun_defn ("with" fun_defn)* "in" expr` рекурсивная функция
- | `"fun" param+ spec* "->" spec* expr` безымянная функция
- | `"while" expr "do" invariant* variant? expr "done"` цикл `loop`
- | `"for" lident "=" expr ("to" | "downto") expr "do" invariant* expr "done"`
- | `"for" pattern "in" expr "with" uident ("as" lident_nq)? "do" invariant*`
- | `("assert" | "assume" | "check") "{" term "}"` `assert`
- | `"raise" uqualid expr? | "raise" "(" uqualid expr? ")"` возбуждение исключения
- | `"try" expr "with" ("|" handler)+ "end"` выр с обработкой исключений
- | `"label" uident "in" expr` помеченное выражение

Язык WhyML. Функции. Предикаты

- | **function** function-decl (**with** logic-decl)
- | **predicate** predicate-decl (**with** logic-decl)
- | **inductive** inductive-decl (**with** inductive-decl)

function_decl ::= lident_nq attribute* type_param* ":" type ("=" term)*

predicate_decl ::= lident_nq attribute* type_param* ("=" formula)

| lident_nq attribute* type_param*

inductive_decl ::= lident_nq attribute* type_param* "=" "|" ? ind_case ("|" ind_case)*

ind_case ::= ident_nq attribute* ":" formula

tqualid ::= uident | ident ("." ident)* "." uident

type_param ::= "" lident | lqualid | "(" lident+ ":" type ")"

| "(" type ("," type)* ")" | "("

Индуктивные определения

module Order

type t

predicate (\leq) t t

axiom le_refl: forall x: t. $x \leq x$

axiom le_asym: forall x y: t. $x \leq y \rightarrow y \leq x \rightarrow x = y$

axiom le_trans: forall x y z: t. $x \leq y \rightarrow y \leq z \rightarrow x \leq z$

end

module SortedList

use list.List

clone export Order **with axiom** le_refl, **axiom** le_asym, **axiom** le_trans

inductive sorted (list t) =

| Sorted_nil: sorted Nil

| Sorted_one: **forall** x: t. sorted (Cons x Nil)

| Sorted_two: **forall** x y: t, l: list t. $x \leq y \rightarrow$ sorted (Cons y l) \rightarrow
sorted (Cons x (Cons y l))

end

Язык WhyML. Спецификации

spec ::= "requires" "{" term "}" предусловие
| "ensures" "{" term "}" постусловие
| "returns" "{" ("|" pattern "->" term)+ "}" постусловие возвращаемых значений
| "raises" "{" ("|" pattern "->" term)+ "}" постусловие исключений.
| "raises" "{" uqualid ("," uqualid)* "}" возбуждаемые исключения
| "reads" "{" lqualid ("," lqualid)* "}" внешние переменные по чтению
| "writes" "{" path ("," path)* "}" объекты памяти по записи
| "alias" "{" alias ("," alias)* "}" тождественные объекты памяти (aliases)
| variant подсказка для доказательства завершения
| "diverges" возможно не завершается
| ("reads" | "writes" | "alias") "{" "}" нет объектов указанного вида

path ::= lqualid ("." lqualid)* v.field1.field2
alias ::= path "with" path arg1 with result

invariant ::= "invariant" "{" term "}" инвариант цикла или инвариант типа
variant ::= "variant" "{" variant_term ("," variant_term)* "}" termination variant
variant_term ::= term ("with" lqualid)? терм + Well Formed порядок

Пример: isqrt

```
let isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧
    sqr(result) ≤ x < sqr(result + 1)
body
let ref res = 0 in let ref sum = 1 in
  while sum ≤ x do
    res <- res + 1;
    sum <- sum + 2 * res + 1
  done; res
```

Использование аксиом

Example: division

```
function div(real,real):real
axiom mul_div:
    forall x,y. y≠0 → div(x,y)*y = x
```

Example: factorial

```
function fact(int):int
axiom fact0: fact(0) = 1
axiom factn:
    forall n:int. n ≥ 1 → fact(n) = n * fact(n-1)
```

Спецификация факториала без аксиом

```
let function fact (n:int) : int  
requires { n ≥ 0 }  
variant { n }  
= if n=0 then 1 else n * fact(n-1)
```

generates the logic context

```
function fact int : int  
  
axiom f_body: forall n. n ≥ 0 →  
fact n = if n=0 then 1 else n * fact(n-1)
```

Язык WhyML. Декларации функций

let Definition – функция с параметрами, спецификацией и телом

val Declaration – функция с параметрами и спецификацией

let function Definition – чистая функция (без побочного эффекта), может использоваться в спецификации

let predicate Definition – чистая логическая функция. Может использоваться в спецификации как предикат

val function Declaration – чистая функция, может использоваться в спецификации

val predicate Declaration – чистая логическая функция. Может использоваться в спецификации как предикат

function Definition – функция why3. Может использоваться в невидимом коде

predicate Definition – предикат why3. Может использоваться в невидимом коде

let lemma – чистая лемма-функция (вместо док-ва по индукции)

Лемма-функции

- if a program function is *without side effects* and *terminating*:

let $f(x_1 : T_1, \dots, x_n : T_n) : \text{unit}$

requires Pre

variant var ,

ensures $Post$

body $Body$

then it is a proof of

$\forall x_1, \dots, x_n. Pre \rightarrow Post$

- Если f - рекурсивная, то доказательство сгенерированных формул корректности для рекурсивных вызовов помогает автоматически доказать леммы, обычно доказываемых по индукции

Клонирование модулей прод

```
let rec lemma exp_add (x: t) (n m: int)    лемма-функции
  requires { 0 <= n ∧ 0 <= m }
  variant { n }
  ensures { exp x (n + m) = mul (exp x n) (exp x m) }
= if n > 0 then exp_add x (n - 1) m
```

```
let rec lemma exp_mul (x: t) (n m: int)
  requires { 0 <= n ∧ 0 <= m }
  variant { m }
  ensures { exp x (n * m) = exp (exp x n) m }
= if m > 0 then exp_mul x n (m - 1)
```

```
module ExponentiationBySquaring
  use int.Int
  clone Exp with type t = int, val one = one, val mul = (*)
end
```

Система верификации Why3

**Команды интерактивного
доказательства
(Трансформации)**

Организация трансформаций Why3

Трансформация – команда, применяемая к текущей доказываемой формуле (цели). В результате получается другая формула или несколько других целей.

Имя трансформации. Параметры трансформации.

Перечень доступных трансформаций:

`why3 --list-transforms`

Запуск трансформаций из командной строки либо через раздел Tool в верхнем меню

Трансформации упорядочены по алфавиту

Why3 proof session

The screenshot displays the Why3 IDE interface. The left pane shows a tree of goals for the file `./maxsum_solution.mlw`. The tree is organized as follows:

- ./maxsum_solution.mlw
 - Kadane
 - VC maximum_subarray [VC for maxim
 - split_vc
 - 0 [loop invariant init]
 - 1 [loop invariant init]
 - 2 [loop invariant init]
 - 3 [loop invariant init]
 - 4 [index in array bounds]
 - 5 [loop invariant preservation]
 - 6 [loop invariant preservation]
 - 7 [loop invariant preservation]
 - 8 [loop invariant preservation]
 - 9 [loop invariant preservation]
 - 10 [loop invariant preservation]
 - CVC4 1.5 0.03
 - Alt-Ergo 1.30 0.01 (steps: 31)
 - 11 [loop invariant preservation]
 - 12 [loop invariant preservation]
 - 13 [loop invariant preservation]
 - 14 [loop invariant preservation]
 - 15 [loop invariant preservation]
 - 16 [loop invariant preservation]
 - 17 [loop invariant preservation]
 - 18 [loop invariant preservation]
 - 19 [loop invariant preservation]
 - 20 [loop invariant preservation]
 - 21 [postcondition]
 - 22 [postcondition]
 - 23 [out of loop bounds]

```
Task ./maxsum_solution.mlw
25
26 let maximum_subarray (a: array int): int
27 ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
28 ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h = result }
29 =
30 let maxsum = ref 0 in
31 let curmax = ref 0 in
32 let ghost lo = ref 0 in
33 let ghost hi = ref 0 in
34 let ghost cl = ref 0 in
35 for i = 0 to a.length - 1 do
36 invariant { forall l : int. 0 <= l <= i -> sum a l i <= !curmax }
37 invariant { 0 <= !cl <= i /\ sum a !cl i = !curmax }
38 invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !maxsum }
39 invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !maxsum }
40   curmax += a[i];
41   if !curmax < 0 then begin curmax := 0; cl := i+1 end;
42   if !curmax > !maxsum then begin maxsum := !curmax; lo := !cl; hi := i+1 end
43 done;
44 !maxsum
45
46 end
47
```

0/0/0

Messages | Log | Edited proof | Prover output | Counterexample

Messages

Приведение цели к нормальному виду

Нажать раздел **task** в верхнем правом меню.

Внизу – доказываемая формула (**цель**), выше - **контекст**

Контекст: определения типов, констант, аксиом, функций, предикатов, ... в проекции на цель.

Гипотезы (аксиомы): посылки + подкванторные переменные – появляются непосредственно выше после нормализации цели. **Имя гипотезы**

introduce_premises – полная нормализация цели

intros <имена переменных через запятую > – вынесение подкванторных переменных с назначением имен соответствующих констант

intros_n <целое> – вынесение указанного числа подкванторных переменных

Подстановка определения

Подстановка тела нерекурсивного определения на место вызова функции (предиката) с заменой вхождений формальных параметров на соответствующие фактические параметры

inline_goal – для всех вызовов только внутри цели

inline_all – везде, в цели и гипотезах

inline_trivial – подставить и удалить определения вида

function $f\ x_1 \dots x_n = g\ e_1 \dots e_k$

predicate $p\ x_1 \dots x_n = q\ e_1 \dots e_k$

e_i – простой или x_j , каждое $x_1 \dots x_n$ встречается не более 1 раза

unfold <имя определения> [**in** <имя гипотезы>] –

подстановка определения с данным именем в цели или указанной гипотезе

Декомпозиция гипотез

destruct <имя гипотезы> – устраняет конструкцию (операцию) верхнего уровня в указанной гипотезе.

Применяется к конструкциям вида:

- **false, true,**
- $\wedge, \vee, \rightarrow,$ **not,**
- **exists,**
- **if ... then ... else ...,**
- **match ... with ... end,**
- (не)равенство выражений одного типа.

destruct_rec <имя гипотезы> – многократное применение трансформации **destruct**

destruct_term <имя переменной> – декомпозиция по альтернативам (конструкциям) типа переменной

eliminate_if_term – вынос **if . then . else** на верхний уровень

split_premise_right – гипотезы в конъюнктивной форме заменяются несколькими гипотезами

split_premise_full – предварительно приводятся в конъюнктивную форму

Расщепление

split_goal_right – цель в виде конъюнкции формул заменяется набором целей для каждого конъюнкта исходной цели.

Цель вида $A \ \&\& \ B$ заменяется целями A и $A \rightarrow B$

Цель вида $A \ \vee \ B \rightarrow C$ заменяется целями $A \rightarrow C$ и $B \rightarrow C$

Цель вида $A \ || \ B \rightarrow C$ заменяется на $A \rightarrow C$ и $(\text{not } A) \wedge B \rightarrow C$

Цели вида **if** , **match** ...

split_goal_full – предварительно преобразуется в конъюнктивную форму

split_vc – стандартная агрессивная стратегия, предварительно проводит нормализацию цели

case (<формула>) – расщепляет на две цели для истинного и ложного значений <формулы>

case (<формула>) **as** <имя гипотезы>

Расщепление при **destruct** и других трансформациях

Применение теорем

apply <имя гипотезы> – применение правила *modus ponens*.
Пусть: $h: f \rightarrow p$; $G: p$. Трансформация **apply** h заменяет исходную цель на $G: f$. Для подкванторных переменных в h определяется замена на подходящие термы.

apply <имя гипотезы> **with** <терм> – трансформация с подсказкой.

assert (<формула>) – вводит новую цель, с помощью которой доказывается исходная цель. Для цели $G: p$ трансформация **assert** (f) дает две цели $h: f$ и $G: f \rightarrow p$

assert (<формула>) **as** <имя гипотезы>

cut (<формула>) – как и **assert**, но с другим порядком генерируемых целей

Действия с кванторами

exists <терм> – подстановка в цели <терма> на место переменной под квантором существования

instantiate <ИМЯ ГИПОТЕЗЫ> <список термов через запятую>
– создается новая гипотеза с заменой кванторных переменных указанными термами.

inst_rem <ИМЯ ГИПОТЕЗЫ> <список термов через запятую> –
как **instantiate** с удалением исходной гипотезы

Дизъюнктивная форма. Замены

left – удаление правой части дизъюнктивной формы

right – удаление левой части дизъюнктивной формы

pose <ИМЯ> <терм> – введение новой гипотезы вида
<ИМЯ> = <терм>

remove <список имен гипотез> – удалить указанные гипотезы

replace <терм1> <терм2> [**in** <имя гипотезы>] – в цели или гипотезе <терм1> заменяется на <терм2>. Генерируется дополнительная цель: <терм1> = <терм2>

rewrite <имя гипотезы> - заключительная часть гипотезы имеет вид <терм1> = <терм2>. В цели <терм1> меняется на <терм2>. Посылки гипотезы дают новые цели. Для подстановки подкванторных переменных используется подсказка вида **with** <терм>

Доказательство по индукции

induction <имя переменной> – доказательство по индукции для текущей цели: $h : p1\ n, G : p\ n$.

Команда **induction** n дает две цели.

$h : p1\ n, \text{Init} : n \leq 0, G : p\ n$

$h : p1\ n, \text{Init} : n > 0, \text{hRect} : \text{forall } n1 : \text{int}. n1 < n \rightarrow p1\ n1 \rightarrow p\ n1, G : p\ n$

Предварительно следует свернуть зависимость от n

revert – собрать гипотезы в цель, обратная к `intros`

induction_ty_lex <имя> – для алгебраических типов

Особенности доказательства

Преимущества последней версии. Новые трансформации

Оператор `assert { ... }`

Лемма-функции вместо доказательства по индукции

Возможности полного доказательства в Why3.

Перевод доказательства в Coq

Список трансформаций отдельным файлом

Пример программы на языке WhyML

```
use int.Int (** three global mutable variables *)
val ref x : int
val ref y : int
val ref z : int
(** computes the maximum of `x` and `y` and stores the result in `z` *)
let compute_max_x_y_in_z ()
  requires { true }
  ensures { z = x  $\vee$  z = y }
  ensures { z >= x  $\wedge$  z >= y }
=
  if x <= y then z <- y else z <- x
  (** computes the half of `x` (very inefficiently) stores the result in `y` *)
let compute_half_of_x ()
  requires { x >= 0 }
  ensures { 2*y = x  $\vee$  2*y+1 = x }
=
  y <- 0;
  while 2*(y+1) <= x do
    invariant { 2*y <= x }
    y <- y+1
  done
```

Пример. Использование меток

Ability to refer to past values of variables

```
{ true }
```

```
let v = r in (r <- v + 42; v)  
{ r = r@old + 42 ^ result = r@old }
```

```
{ true }
```

```
let tmp = x in x <- y; y <- tmp  
{ x = y@old ^ y = x@old }
```

SUM revisited:

```
{  $y \geq 0$  } L:
```

```
while  $y > 0$  do
```

```
invariant {  $x + y = x@L + y@L$  }     $x <- x + 1; y <- y - 1$   
{  $x = x@old + y@old \wedge y = 0$  }
```