Некоторые методики доказательства полной корректности при помощи методов Флойда в инструменте frama-c+jessie2 и солверах

Циклы.

loop invariant – индуктивное утверждение; точки сечения – перед проверкой условий циклов (не перед TEST).

loop variant – оценочная функция; точки сечения – перед проверкой условий циклов; фундированное множество – (Nat_0, <).

```
#include <limits.h>
/*@
    requires a >= 0;
    requires INT_MIN <= a + b <= INT_MAX;
    ensures \result == a + b;
*/
int sum(int a, int b) {
    int s = b;
    /*@
        loop invariant k <= a;
        loop invariant s == k + b;
        loop variant a - k;
    */
    for (int k = 0; k < a; ++k) {
        s ++;
    }
    return s;
}
```

```
#include <limits.h>
/*@
    requires a >= 0;
    requires INT_MIN <= a + b <= INT_MAX;
    ensures \result == a + b;
*/
int sum(int a, int b) {
    int s = b; int _a = a;
    /*@
        loop invariant a >= 0;
        loop invariant s == _a - a + b;
        loop variant a + 1;
    */
    while (--a >= 0) {
        s ++;
    }
    return s;
}
```

Рекурсия.

decreases – оценочная функция рекурсивной функции; фундированное множество – (Nat_0, <); выбирать дополнительные спецификации (ξ в методе Флойда) нельзя.

```
#include <limits.h>
/*@
    requires a >= 0;
    requires INT_MIN <= a + b <= INT_MAX;
    decreases a;
    ensures \result == a + b;
*/
int sum(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return sum(a - 1, b) + 1;
    }
}
```

Все аксиомы есть, но они не применяются для доказательства.

Солвер не рассчитан на доказательство теорем. Его задача – решать уравнения (искать модели, значения переменных). Вывод утверждения делается через поиск модели для отрицания утверждения (пришли к противоречию – значит доказали

истинность исходного утверждения). Почти все аксиомы задаются формулами с кванторами всеобщности. Их нельзя в таком виде применить для решения уравнения, но можно инстанцировать, т.е. перебирать разные значения подкванторных переменных. Если повезет, то в совокупности с другими формулами (и их инстансами) можно получить искомое противоречие. Солвер не может рандомно инстанцировать аксиомы, получится много бесполезных формул. Но можно посмотреть на то, что солвер не может доказать, и подсказать ему, какое еще инстанцирование ему нужно сделать. Для этого применяется метод E-matching: аксиома инстанцируется, если среди текущих термов есть такой терм, который унифицируется с частью формулы у аксиомы.

В примере ниже доказывается полная корректность функции, вычисляющей произведение своих аргументов. Чтобы солвер Alt-Ergo доказал отсутствие арифметического переполнения в выражении m + b в теле цикла, ему надо доказать общезначимость формулы

a >= 0 /\ b >= 0 /\ a*b <= INT_MAX /\ k <= a /\ m == k * b /\ k < a ==> m + b <= INT_MAX

Для этого ему нужно сделать:

m + b = k * b + b ?=? (k + 1) * b ?<=? a * b <= INT_MAX

Т.е. надо инстанцировать две аксиомы :

```
axiom Mul_distr_l :
   forall x:int, y:int, z:int. (x * (y + z)) = ((x * y) + (x * z))
axiom CompatOrderMult :
   forall x:int, y:int, z:int. x <= y -> zero <= z -> (x * z) <= (y * z)
```

Чтобы инстанцировать первую аксиому, среди термов должен появиться терм b * (k + 1) – он унифицируется с (x * (y + z)). Без нашей помощи солверу трудно догадаться о нем. Чтобы инстанцировать вторую аксиому, нужен терм (k + 1) * b и терм a * b. Они подойдут под термы (x * z) и (y * z) в формуле. Терм a * b уже есть в предусловии. Осталось добавить упоминание термов (k + 1) * b и b * (k + 1). Как это сделать? Изменить индуктивное утверждение (предусловие нельзя менять – это постановка

задачи) – добавить b * (k + 1) == (k + 1) * b; или (k + 1) * b == b * (k + 1);

```
#include <limits.h>

/*@
    requires a >= 0;
    requires b >= 0;
    requires a * b <= INT_MAX;
    ensures \result == a * b;
*/
int multiplication(int a, int b) {
    int m = 0;
    /*@
        loop invariant k <= a;
        loop invariant m == k * b;
        loop invariant b * (k + 1) == (k + 1) * b;
        loop variant a - k;
    */
    for (int k = 0; k < a; ++k) {
        m += b;
    }
    return m;
}
```

Как посмотреть доступные аксиомы? В Why3 IDE нажать на одно из условий верификации и заглянуть во вкладку Task.

Усложнение инварианта цикла не всегда годится => assert

Чтобы подсказать солверу, как надо инстанцировать кванторы всеобщности, мы дополнили инвариант цикла. Но это привело к усложнению всех условий верификации. Можно ли добавить нужные термы «локально», не затрагивая всех условий верификации? Можно – для этого есть конструкция assert. Она добавит формулу из assert в посылку условий верификации для путей, проходящих через этот assert, не изменяя индуктивное утверждение. В качестве платы за это, будет сгенерированы еще условия верификации, доказывающее выполнение формулы у assert на всех базовых путях к этому assert.

```
#include <limits.h>

/*@
    requires a >= 0;
    requires b >= 0;
    requires a * b <= INT_MAX;
    ensures \result == a * b;
*/
int multiplication(int a, int b) {
    int m = 0;
    /*@
        loop invariant k <= a;
        loop invariant m == k * b;
        loop variant a - k;
    */
    for (int k = 0; k < a; ++k) {
        //@ assert b * (k + 1) == (k + 1) * b;
        m += b;
```

```
    }
    return m;
}
```

Когда приходится повторять доказательства одного и того же утверждения...

Докажите полную корректность следующих функций относительно их спецификаций:

```
/*@
    requires n == 2 * m;
    ensures (n % 2) == 0;
    ensures (n & 1) == 0;
    ensures (n / 2) == m;
    ensures (n >> 1) == m;
*/
void p1(unsigned n, unsigned m) {
}

/*@
    ensures \result == n % 2;
*/
unsigned even(unsigned n) {
    return n & 1;
}
```

Кажется всё очень просто – циклов нет. Запускаем солвер и видим, что не доказывается много утверждений. Смотрим на аксиомы, пишем доказательство утверждений «на бумажке», оформляем шаги доказательства при помощи assert. Пришлось использовать дополнительную переменную n_m_2, чтобы записать в нее выражение n % 2, т.к. инструмент странно определяет тип этого выражения и не дает его использовать в некоторых выражениях. Переменная объявлена как ghost, что означает, что она используется только для целей верификации и не влияет на реализацию функции (на языке Си).

```
/*@
    requires n == 2 * m;
    ensures (n % 2) == 0;
    ensures (n & 1) == 0;
    ensures (n / 2) == m;
    ensures (n >> 1) == m;
*/
void p1(unsigned n, unsigned m) {
    //@ assert (n >> 1) == (n / 2);

    //@ ghost unsigned n_m_2 = n % 2;
    //@ assert n_m_2 == 0 || n_m_2 == 1;
    //@ assert n_m_2 == n_m_2 % 2;
    //@ assert n_m_2 < 1 <==> n_m_2 % 2 < 1;
    // nth n 0 == 0 <==> nth (n % 2) == 0;
    //@ assert n >= 0 && n_m_2 < 1 <==> n_m_2 >= 0 && n_m_2 % 2 < 1;
    //@ assert \forall unsigned k; (k == 0 || k == 1) ==> (k & 1) == k;
```

```
    //@ assert (n_m_2 & 1) == n_m_2;
    //@ assert (n_m_2 & 1) == (n & 1);
    //@ assert (n & 1) == n_m_2;

    //@ assert n == (n >> 1) * 2 + (n & 1);
}

/*@
    ensures \result == n % 2;
*/
unsigned even(unsigned n) {
    //@ ghost unsigned n_m_2 = n % 2;
    //@ assert n_m_2 == 0 || n_m_2 == 1;
    //@ assert n_m_2 == n_m_2 % 2;
    //@ assert n_m_2 < 1 <==> n_m_2 % 2 < 1;
    // nth n 0 == 0 <==> nth (n % 2) == 0;
    //@ assert n >= 0 && n_m_2 < 1 <==> n_m_2 >= 0 && n_m_2 % 2 < 1;
    //@ assert \forall unsigned k; (k == 0 || k == 1) ==> (k & 1) == k;
    //@ assert (n_m_2 & 1) == n_m_2;
    //@ assert (n_m_2 & 1) == (n & 1);
    //@ assert (n & 1) == n_m_2;
    return n & 1;
}
```

В доказательстве полной корректности функций пришлось повторить один и тот же набор assert'ов для доказательства утверждения n % 2 == n & 1 для n типа unsigned. Такие доказательства нужно уметь отделить от функций и иметь доказательство одного утверждения в одном экземпляре, не повторять доказательство в разных функциях. Для этого есть *леммы*.

Лемму надо записать, доказать и применить для доказательства условий верификации. Записать лемму просто:

```
/*@
    lemma eq_mod_bwand: \forall unsigned n; (n % 2) == (n & 1);
*/

/*@
    requires n == 2 * m;
    ensures (n % 2) == 0;
    ensures (n & 1) == 0;
    ensures (n / 2) == m;
    ensures (n >> 1) == m;
*/
void p1(unsigned n, unsigned m) {
    //@ assert (n >> 1) == (n / 2);
    //@ assert n == (n >> 1) * 2 + (n & 1);
}

/*@
    ensures \result == n % 2;
*/
unsigned even(unsigned n) {
    return n & 1;
}
```

После этого полная корректность обеих функций доказывается солвером Alt-Ergo быстро. Почему так получилось: при доказательстве условий верификации леммы воспринимаются как

аксиомы. Солвер увидел в условии верификации терм, который встречается в лемме, и поэтому смог нужным образом инстанцировать лемму.

Но Alt-Ergo должен доказать и саму лемму тоже, а сделать этго не может. Другие солверы тоже не могут ее доказать. Нам самим для доказательства утверждения леммы пришлось выписать немалый набор assert'ов. Хорошо, что у нас есть доказательство леммы. Нужно его правильно записать.

Лемма находится либо сама по себе, либо в аксиоматике – тогда к ней можно поместить дополнительные предикатные и функциональные символы и аксиомы с леммами. Первая попытка – записать assert'ы как предварительные леммы:

```
/*@ axiomatic EqModBwAndProof {
        logic integer n_m_2(unsigned n) = n % 2;
        lemma step_1: \forall unsigned n; n_m_2(n) == 0 || n_m_2(n) == 1;
        lemma step_2: \forall unsigned n; n_m_2(n) == n_m_2(n) % 2;
        lemma step_3: \forall unsigned n; n_m_2(n) < 1 <==> n_m_2(n) % 2 < 1;
        // nth n 0 == 0 <==> nth (n % 2) == 0;
        lemma step_4: \forall unsigned n; n >= 0 && n_m_2(n) < 1 <==> n_m_2(n) >= 0 &&
n_m_2(n) % 2 < 1;
        lemma step_5: \forall unsigned k; (k == 0 || k == 1) ==> (k & 1) == k;
        lemma step_6: \forall unsigned n, unsigned m; m == n_m_2(n) ==> (m & 1) ==
n_m_2(n);
        lemma step_7: \forall unsigned n, unsigned m; m == n_m_2(n) ==> (m & 1) == (n &
1);
        lemma eq_mod_bwand: \forall unsigned n; (n % 2) == (n & 1);
    }
*/

/*@
    requires n == 2 * m;
    ensures (n % 2) == 0;
    ensures (n & 1) == 0;
    ensures (n / 2) == m;
    ensures (n >> 1) == m;
*/
void p1(unsigned n, unsigned m) {
    //@ assert (n >> 1) == (n / 2);
    //@ assert n == (n >> 1) * 2 + (n & 1);
}

/*@
    ensures \result == n % 2;
*/
unsigned even(unsigned n) {
    return n & 1;
}
```

Солверы Alt-Ergo и CVC4 успешно доказали леммы со step_1 до step_6 и не доказали леммы step_7 и eq_mod_bwand. Первые леммы – достаточно простые, а последние две – сложные. Солверам нужно подсказать, из каких утверждений следует доказать step_7 и eq_mod_bwand. Для этого мы добавим ряд

дополнительных лемм. Но кроме этого мы обернем подкванторные формулы в предикаты, чтобы солвер мог доказывать утверждения, не раскрывая предикаты в их формулы – это очень сильно помогает доказывать сложные утверждения.

```
/*@ axiomatic EqModBwAndProof {
        predicate prop_1(unsigned m) = m == 0 || m == 1;
        predicate prop_2(unsigned m) = m == m % 2;
        predicate prop_3(unsigned m) = (m < 1 <==> m % 2 < 1);

        // nth n 0 == 0 <==> nth (n % 2) == 0;
        predicate prop_4(unsigned n, unsigned m) = (n >= 0 && m < 1 <==> m >= 0 && m % 2
< 1);

        predicate prop_5(unsigned m) = (m & 1) == m;
        predicate prop_6(unsigned n, unsigned m) = (m & 1) == (n & 1);

        predicate eq_mod_bwand_lemma(unsigned n, unsigned m) = m == (n & 1);

        // m0: s1 => s5
        // m1: s1 && s2 && s3 && s4 && s5 => s6
        // m2: s1 && s2 && s3 && s4 && s5
        // m3: s5 && s6
        // m4: s5 && s6 => lemma
        // m5: lemma

        lemma m_0: \forall unsigned k; prop_1(k) ==> prop_5(k);
        lemma m_1: \forall unsigned n, unsigned m; m == n % 2 ==> prop_1(m) && prop_2(m)
&& prop_3(m) && prop_4(n, m) && prop_5(m) ==> prop_6(n, m);
        lemma m_2: \forall unsigned n, unsigned m; m == n % 2 ==> prop_1(m) && prop_2(m)
&& prop_3(m) && prop_4(n, m) && prop_5(m);
        lemma m_3: \forall unsigned n, unsigned m; m == n % 2 ==> prop_5(m) && prop_6(n,
m);
        lemma m_4: \forall unsigned n, unsigned m; m == n % 2 ==> prop_5(m) && prop_6(n,
m) ==> eq_mod_bwand_lemma(n, m);

        lemma m_5: \forall unsigned n, m; m == n % 2 ==> eq_mod_bwand_lemma(n, m);
    }
*/

/*@
    requires n == 2 * m;
    ensures (n % 2) == 0;
    ensures (n & 1) == 0;
    ensures (n / 2) == m;
    ensures (n >> 1) == m;
*/
void p1(unsigned n, unsigned m) {
    //@ assert (n >> 1) == (n / 2);
    //@ assert n == (n >> 1) * 2 + (n & 1);
}

/*@
    ensures \result == n % 2;
*/
unsigned even(unsigned n) {
    return n & 1;
}
```

Теперь солверы Alt-Ergo и CVC4 доказывают все леммы.

Правда, когда мы записали лемму внутри аксиоматика, она перестала вставляться в условия верификации для функций. Чтобы инструмент вставил содержимое аксиоматика в условие верификации, в нем должен появиться какой-нибудь предикатный

или функциональный символ из этого аксиоматика. Добавим его и в итоге получим полностью доказанную программу.

```
/*@ axiomatic EqModBwAndProof {
        predicate prop_1(unsigned m) = m == 0 || m == 1;
        predicate prop_2(unsigned m) = m == m % 2;
        predicate prop_3(unsigned m) = (m < 1 <==> m % 2 < 1);

        // nth n 0 == 0 <==> nth (n % 2) == 0;
        predicate prop_4(unsigned n, unsigned m) = (n >= 0 && m < 1 <==> m >= 0 && m % 2
< 1);

        predicate prop_5(unsigned m) = (m & 1) == m;
        predicate prop_6(unsigned n, unsigned m) = (m & 1) == (n & 1);

        predicate eq_mod_bwand_lemma(unsigned n, unsigned m) = m == (n & 1);

        // m0: s1 => s5
        // m1: s1 && s2 && s3 && s4 && s5 => s6
        // m2: s1 && s2 && s3 && s4 && s5
        // m3: s5 && s6
        // m4: s5 && s6 => lemma
        // m5: lemma

        lemma m_0: \forall unsigned k; prop_1(k) ==> prop_5(k);
        lemma m_1: \forall unsigned n, unsigned m; m == n % 2 ==> prop_1(m) && prop_2(m)
&& prop_3(m) && prop_4(n, m) && prop_5(m) ==> prop_6(n, m);
        lemma m_2: \forall unsigned n, unsigned m; m == n % 2 ==> prop_1(m) && prop_2(m)
&& prop_3(m) && prop_4(n, m) && prop_5(m);
        lemma m_3: \forall unsigned n, unsigned m; m == n % 2 ==> prop_5(m) && prop_6(n,
m);
        lemma m_4: \forall unsigned n, unsigned m; m == n % 2 ==> prop_5(m) && prop_6(n,
m) ==> eq_mod_bwand_lemma(n, m);

        lemma m_5: \forall unsigned n, m; m == n % 2 ==> eq_mod_bwand_lemma(n, m);
    }
*/

/*@
    requires n == 2 * m;
    ensures (n % 2) == 0;
    ensures (n & 1) == 0;
    ensures (n / 2) == m;
    ensures (n >> 1) == m;
*/
void p1(unsigned n, unsigned m) {
    //@ assert (n >> 1) == (n / 2);
    //@ ghost unsigned p = n % 2;
    //@ assert eq_mod_bwand_lemma(n, p);
    //@ assert n == (n >> 1) * 2 + (n & 1);
}

/*@
    ensures \result == n % 2;
*/
unsigned even(unsigned n) {
    //@ ghost unsigned p = n % 2;
    //@ assert eq_mod_bwand_lemma(n, p);
    return n & 1;
}
```

Нам повезло, что лемму удалось доказать техниками в солверах. Чаще всего техник солверов не хватает. Тогда для доказательства лемм следует применять интерактивный прувер.

Си-функции доказывают assert'ы. Доказательство леммы получилось сложнее, чем исходная последовательность assert'ов.

Оказывается, можно воспользоваться уже упомянутым механизмом ghost не только для объявления дополнительных переменных, но и для вызова функций. Эти функции не должны влиять на реализацию (на языке Си). Однако их вызов будет подобен assert, причем тело этой функции может быть доказательством. В отличие от лемм в функции не будет кванторов, а это значит, что солверам будет существенно проще доказывать правильность этой функции, чем леммы.

```
//@ predicate eq_mod_bwand_lemma(unsigned n) = (n % 2) == (n & 1);

/*@
    ensures eq_mod_bwand_lemma(n);
*/
void eq_mod_bwand_lemma_proof(unsigned n) {
    unsigned n_m_2 = n % 2;

    //@ assert n_m_2 == 0 || n_m_2 == 1;

    //@ assert n_m_2 == n_m_2 % 2;

    //@ assert n_m_2 < 1 <==> n_m_2 % 2 < 1;

    // nth n 0 == 0 <==> nth (n % 2) == 0;
    //@ assert n >= 0 && n_m_2 < 1 <==> n_m_2 >= 0 && n_m_2 % 2 < 1;

    //@ assert \forall unsigned k; (k == 0 || k == 1) ==> (k & 1) == k;

    //@ assert (n_m_2 & 1) == n_m_2;

    //@ assert (n_m_2 & 1) == (n & 1);

    //@ assert (n & 1) == n_m_2;
}

/*@
    requires n == 2 * m;
    ensures (n % 2) == 0;
    ensures (n & 1) == 0;
    ensures (n / 2) == m;
    ensures (n >> 1) == m;
*/
void p1(unsigned n, unsigned m) {
    //@ assert (n >> 1) == (n / 2);
    //@ ghost eq_mod_bwand_lemma_proof(n);
    //@ assert n == (n >> 1) * 2 + (n & 1);
}

/*@
    ensures \result == n % 2;
*/
unsigned even(unsigned n) {
    //@ ghost eq_mod_bwand_lemma_proof(n);
    return n & 1;
}
```

Можно доказывать по индукции!!!

В функции “proof” можно использовать любые операторы языка Си, в частности, циклы и вызовы функций, даже рекурсивные вызовы. А это значит, что такие функции будут доказывать

утверждения по индукции, содержащейся в методах Флойда! Пример доказательства по индукции при помощи рекурсии:

```
/*@
    axiomatic arithm_sum {
        logic integer sum(integer a, integer b);
        axiom def1:
            \forall integer x; sum(x, x) == x;
        axiom def2:
            \forall integer x, y; sum(x, y + 1) == sum(x, y) + (y + 1);
    }
*/

/*@
    requires 0 < n < 1000;
    decreases n;
    ensures sum(1, n) == n * (n + 1) / 2;
*/
void proof(int n)
{
    if (n == 1) {
    } else {
        proof(n - 1);
    }
}
```

Минус функции в том, что ее нельзя вставлять в выражения и при ее вызове в ghost надо иметь значения аргументов, а типы аргументов должны быть Си-типами, не integer, что иногда мешает сформулировать некоторые утверждения для доказательства. Решить эти проблемы должны «лемма-функции», поддержка которых появится в следующих версиях frama-c+jessie2 (astraver).

Ghost-переменные помогают доказывать утверждения. Не надо бояться заводить ghost-переменные для уменьшения сложности доказываемых выражений и добавления похожих термов. Особенно это актуально при доказательстве утверждений с кванторами существования.