УДК 004.05

Трансформация, спецификация и верификация программы вычисления числа элементов множества, представленного в виде битовой шкалы

Тумуров Э.Г. (Институт систем информатики СО РАН),
Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описывается трансформация программы memweight из библиотеки ОС Linux, устраняющая указатели. Далее программа трансформируется на язык предикатного программирования P. Для предикатной программы, полученной в результате серии упрощающих трансформаций, строится спецификация и проводится дедуктивная верификация. Для упрощения верификации в рамках спецификации строится модель внутреннего состояния исполняемой программы. Верификация программы memweight реализована в системе автоматического доказательства Why3.

Ключевые слова: *дедуктивная верификация, трансформации программ, автоматическое доказательство, функциональное программирование, предикатное программирование.*

1. Введение

В библиотеке ядра операционной системы (ОС) Linux имеется программа memweight на языке Си, вычисляющая число элементов множества, представленного в виде битовой шкалы. Код программы memweight приводится в Приложении 1. Программа более известна как программа вычисления *веса* Хэмминга. Она вычисляет число единиц в битовом представлении памяти. Параметрами программы memweight являются: ptr – указатель на байтовый массив, bytes – число байтов в массиве. В указанном массиве памяти нужно определить число единичных битов.

Функция memweight многократно вызывается в программе ядра ОС Linux. Поэтому высоки требования к ее эффективности. Используется сложный эффективный алгоритм, так называемая битовая магия [4] для 32- или 64-битных слов.

Дедуктивная верификация намного проще и быстрее для функциональных программ [12], чем для аналогичных императивных программ, потому что функциональная программа проще эквивалентной императивной программы. Этот факт отмечается разными исследователями. Преимущество по времени верификации для предикатных программ [3] оценивалось примерно в 5 раз. Для программы быстрой сортировки с двумя опорными элементами зафиксировано преимущество в 10 раз [11]. Отметим высокую сложность проведения формальной спецификации и дедуктивной верификации императивных программ.

Причина сложности императивных программ в том, что указатели, конструкции необходимые для оптимизации программ, существенно усложняют логику императивных программ. Для упрощения императивных программ применяются трансформации, устраняющие указатели в императивной программе [8]. Операции с указателями заменяются эквивалентными действиями без указателей. Далее к полученной программе применяются трансформации, превращающие ее в эквивалентную предикатную программу. Здесь проводится оформление аргументов и результатов функций и замена каждого цикла соответствующей рекурсивной программой.

Для предикатной программы memweight, полученной в результате трансформации, конструируются формальные спецификации в виде предусловия и постусловия для каждой функции, входящей в состав предикатной программы. Для упрощения верификации в рамках спецификации строится модель внутреннего состояния исполняемой программы. Модель экстрагируется в виде независимой теории из константной части программы. Данный метод позволяет успешно декомпозировать сложные доказательства.

Далее строятся формулы корректности предикатной программы memweight относительно спецификации применением системы правил [11]. Доказана корректность правил [1]. Совокупность формул корректности вместе с описаниями типов и переменных оформляется в виде набора теорий. Эти теории транслируются на язык спецификаций why3 [19]. Далее в системе дедуктивной верификации Why3 [19] реализуется процесс доказательства формул корректности.

В версии 1.3.1 системы Why3 в дополнении к возможности использования множества эффективных SMT-решателей реализована полноценная система интерактивного доказательства. Благодаря этому появилась возможность полностью реализовать доказательство в рамках системы Why3 без выхода в систему Coq [17], как это было ранее [6, 7]. Дополнительным эффективным методом является использование аппарата лемма-

функций [18] для доказательства сложной леммы, в частности, требующей доказательства по индукции. Здесь строится вспомогательная предикатная программа, спецификация которой совпадает с исходной леммой.

Во втором разделе настоящей работы дается краткое описание языка предикатного программирования. В третьем разделе описывается трансформация исходной программы memweight с получением эквивалентной предикатной программы. В следующем разделе описывается процесс спецификации предикатной программы. Особенности процесса дедуктивной верификации предикатной программы в системе Why3[19] описывается в шестом разделе. Исправлено значительное число ошибок спецификации. Далее обзор работ. В заключении суммируются итоги верификации. В Приложении 1 код исходной программы memweight. В Приложении 2 представлены полные теории, созданные для доказательства формул корректности на языке Why3.

2. Язык предикатного программирования

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [3] следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
pre <предусловие>
post <постусловие>
measure <выражение>
{ <оператор> }
```

Предусловие и постусловие являются формулами на языке исчисления предикатов. Они обязательны при дедуктивной верификации [5, 9, 10]. Мера задается только для рекурсивных программ и используется для верификации.

Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

```
<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> else <оператор2>
<имя программы>(<список аргументов>: <список результатов>)
<тип> <пробел> <список имен переменных>
```

Всякая переменная характеризуется *типом* – множеством допустимых значений. *Описание типа* **type** T(p) = D с возможными параметрами p связывает имя типа T с его

изображением D. Типы **bool**, **int**, **real** и **char** являются *примитивными*. Значением типа **array**(T_e, T_i) является *массив* с *элементами массива* типа T_e и *индексами* конечного типа T_i. Тип массива является предикатным типом, его значения (массивы) являются тотальными и однозначными предикатами.

Пусть E(x) – логическое выражение. Тип **subtype**(T x**:** E(x)) определяет *подтип* типа T при истинном предикате E(x), т.е. множество {x∈T | E(x)}. Определенный в языке P тип целых чисел **nat** представляется описанием:

type nat = **subtype**(**int** x: x ≥ 0) .

Допускаются подтипы, параметризуемые переменными. Примером является тип *диапазона* целых чисел:

type Diap(**nat** n) = **subtype**(**int** x: x ≥ 1 & x ≤ n) .

В языке P для изображения типа диапазона используется конструкция 1..n.

Описания типов переменных являются частью спецификации программы. Описание переменной T x есть утверждение x∈T, которое становится частью предусловия, если x – аргумент предикатной программы, или частью постусловия, если x – результат программы. При этом утверждение x∈T обычно не пишется в составе предусловия или постусловия, хотя предполагается.

В языке предикатного программирования P [3] нет указателей, серьезно усложняющих программу. Вместо указателей используются объекты алгебраических типов: списки и деревья. Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением *оптимизирующих трансформаций* [2]. Они определяют отличную от классической оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу.

Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- открытая подстановка программы на место ее вызова;
- кодирование объектов алгебраических типов (списков и деревьев) при помощи массивов и указателей.

3. Трансформации программы вычисления числа элементов множества

Трансформации направлены на упрощение исходной императивной программы. В итоге получается эквивалентная предикатная программа. Сначала проводится нормализация программы. Удаляются конструкции, ориентированные на трансляцию. Раскрываются умолчания для условий в условных операторах и заголовках циклов. Далее устраняются указатели. Вместо них в программе появляются явные обращения к элементам массивов. Наконец, циклы преобразуются в рекурсивные программы.

3.1. Описание программы memweight

Представим заголовок программы:

```
size_t memweight(const void *ptr, size_t bytes)
```

Дана область памяти, начало которой фиксируется указателем ptr. Размером области в байтах определяется вторым параметром bytes. Программа вычисляет ***вес*** (вес Хэмминга) области памяти – количество единичных битов в двоичном представлении области памяти. Программа более известна как программа вычисления веса Хэмминга.

Полный код программы memweight вместе со всеми вызываемыми функциями в исходном ненормализованном виде приведен в Приложении 1. Далее представлена часть программы после проведения этапа нормализации, на котором устраняются макросы, раскрываются умолчания для выражений нелогического типа в логических позициях и проводятся другие преобразования.

Ниже приводится код главной программы memweight. Этот код и его описание доступны также по ссылке https://elixir.bootlin.com/linux/v5.7.8/source/lib/memweight.c .

```
const INT_MAX = 0x7fffffff;
const BITS_PER_LONG = 64;

size_t memweight(const void *ptr, size_t bytes)
{
        size_t ret = 0;
        size_t longs;
        const unsigned char *bitmap = ptr;
        for (; bytes > 0 && ((unsigned long)bitmap) % sizeof(long) != 0;
                        bytes--, bitmap++)
                ret += hweight8(*bitmap);
        longs = bytes / sizeof(long);
        if (longs != 0) {
                assert(longs < INT_MAX / BITS_PER_LONG);
                ret += bitmap_weight((unsigned long *)bitmap,
                                longs * BITS_PER_LONG);
                bytes -= longs * sizeof(long);
                bitmap += longs * sizeof(long);
        }
        for (; bytes > 0; bytes--, bitmap++)
                ret += hweight8(*bitmap);
        return ret;
}
```

В теле главной программы memweight используются две функции: hweight8 для подсчета числа единиц в байте и bitmap_weight для подсчета единиц в массиве слов размером 32 или 64 бита в зависимости от архитектуры. Функция bitmap_weight значительно более эффективна в сравнении с побайтовой обработкой через hweight8.

Пословная адресация памяти в ряде архитектур ЭВМ возможна лишь с границы слова, т.е. когда адрес начала памяти кратен 4 для 32-битных слов (8 для 64-битных слов). Поэтому функция bitmap_weight может быть запущена лишь с границы слова. С учетом этого исходная память, заданная параметрами memweight, разделяется на три части: короткий байтовый массив до первой границы слова, словный массив и короткий байтовый массив за последним словом (см. Рис.1 и 2). Словный массив находится внутри исходного байтового массива.

Итак, алгоритм программы memweight следующий. Сначала последовательность байтов до границы слова обрабатывается вызовами hweight8. Далее запускается bitmap_weight. Оставшиеся в конце байты обрабатываются в цикле через hweight8.

Оператор **assert**, поставленный вместо оператора BUG_ON, должен превратиться в условие корректности.

Функция bitmap_weight имеет два параметра: src – указатель на массив слов, nbits – число битов шкалы, представляющей множество. Функция bitmap_weight является интерфейсной в библиотеке ОС Linux и может быть вызвана пользователем независимо. Значение nbits может быть не кратно 8. Код функции приведен ниже.

```
BITMAP_LAST_WORD_MASK(nbits) = (~0UL >> (-(nbits) & (BITS_PER_LONG - 1)));

int bitmap_weight(const unsigned long *src, unsigned int nbits)
{
        if (nbits <= BITS_PER_LONG && nbits > 0)
                return hweight_long(*src & BITMAP_LAST_WORD_MASK(nbits));
        return __bitmap_weight(src, nbits);
}
```

В функции bitmap_weight вызывается hweight_long в случае, когда массив состоит из одного слова. В общем случае реализуется вызов __bitmap_weight с теми же параметрами.

Функция __bitmap_weight аналогична bitmap_weight. Каждое слово обрабатывается вызовом функции hweight_long. Код и описание функции доступны по адресу: https://elixir.bootlin.com/linux/v5.7.8/source/lib/bitmap.c#L333

```
int __bitmap_weight(const unsigned long *bitmap, unsigned int bits)
{
        unsigned int k, lim = bits/BITS_PER_LONG;
        int w = 0;
        for (k = 0; k < lim; k++)
                w += hweight_long(bitmap[k]);
        if (bits % BITS_PER_LONG != 0)
                w += hweight_long(bitmap[k] & BITMAP_LAST_WORD_MASK(bits));
        return w;
}
```

Программа hweight_long определяет число единиц в слове w, являющимся параметром. В зависимости от архитектуры вызывается hweight32(w) или hweight64(w). В этих функциях применяются нетривиальные алгоритмы битовой магии [4]. Ниже пример одной из таких функций.

```
unsigned long hweight64No(__u64 w)
{
        __u64 res = w - ((w >> 1) & 0x5555555555555555ul);
        res = (res & 0x3333333333333333ul) + ((res >> 2) & 0x3333333333333333ul);
        res = (res + (res >> 4)) & 0x0F0F0F0F0F0F0F0Ful;
        res = res + (res >> 8);
        res = res + (res >> 16);
        return (res + (res >> 32)) & 0x00000000000000FFul;
}
```

3.2. Устранение указателей

После этапа нормализации программы применяется набор трансформаций, устраняющих указатели в программе. Доступ по указателю заменяется явным обращением к элементу массива. Арифметические действия с указателем заменяется пересчетом индекса массива.

Указателю ptr функции memweight соответствует массив байтов. Другой указатель bitmap смотрит внутрь этого массива. Обычно при наличии нескольких переменных-указателей, смотрящих внутрь одного массива, одна из переменных становится массивом, а другие – индексами в этом массиве [6, 7]. Однако в данном случае используется более общая схема трансформации, когда все указатели заменяются индексами массива памяти программы. Причина этого – использование операции «%» остатка от деления в применении к указателю bitmap.

Вначале процесса трансформации определяются новые типы и глобальные переменные. Введем тип Mem для глобального массива байтов mem, определяющего всю память.

```
type Mem = array(char, size_t);
Mem mem;
```

Указатель ptr становится индексом в массиве mem типа size_t, соответствующего размеру адресуемой памяти для текущей архитектуры ЭВМ. Вместо указателя bitmap введем переменную p (типа size_t), определяющую индекс элемента в массиве mem, который соответствует указателю bitmap.

Таким образом, программа memweight вычисляет число битов, содержащихся в вырезке mem[ptr..ptr+bytes-1]. Необходимым дополнительным требованием должно быть условие того, что данный массив не выходит за границы максимально адресуемой памяти:

```
ptr+bytes <= SIZE_MAX
```

Указатель (**unsigned long** *)bitmap есть указатель на массив слов. Этот указатель получен операцией приведения типа (**type cast)**. Данный массив слов определим глобальной переменной memL:

```
type MemL = array(unsigned long, size_t);
MemL memL;
```

Массив memL является частью массива mem. Элемент memL[0] начинается с ближайшей границы слова в массиве mem.

Ниже трансформированная программа memweight. Удален первый параметр вызова bitmap_weight, поскольку параметр – массив memL является глобальной переменной.

```
size_t memweight(size_t ptr, bytes)
{
        size_t ret = 0;
        size_t longs;
        size_t p = ptr;
        for (; bytes > 0 && p % sizeof(long) != 0;
                        bytes--, p++)
                ret += hweight8(p);
        longs = bytes / sizeof(long);
        if (longs != 0) {
                assert(longs < INT_MAX / BITS_PER_LONG);
                ret += bitmap_weight(longs * BITS_PER_LONG);
                bytes -= longs * sizeof(long);
                p += longs * sizeof(long);
        }
        for (; bytes > 0; bytes--, p++)
                ret += hweight8(p);
        return ret;
}
```

Далее представлены трансформации функций bitmap_weight и __bitmap_weight. Первый параметр опускается. Вхождение *src в bitmap_weight заменяется на memL.

```
int bitmap_weight(unsigned int nbits)
{
        if (nbits <= BITS_PER_LONG && nbits > 0)
                return hweight_long(memL[0] & BITMAP_LAST_WORD_MASK(nbits));
        return __bitmap_weight(nbits);
}
```

Вхождения указателя bitmap в функции __bitmap_weight заменяются на memL.

```
int __bitmap_weight(unsigned int bits)
{
        unsigned int k, lim = bits/BITS_PER_LONG;
        int w = 0;
        for (k = 0; k < lim; k++)
                w += hweight_long(memL[k]);
        if (bits % BITS_PER_LONG != 0)
                w += hweight_long(memL[k] & BITMAP_LAST_WORD_MASK(bits));
        return w;
}
```

3.3. Трансформация в предикатную программу

Программа, полученная устранением указателей, остается в рамках языка Си. Ее, в принципе, можно верифицировать традиционным методом Хоара.

Преобразование данной программы на язык предикатного программирования P является несложным и в принципе может быть проведено автоматически.

Преобразуем циклы в рекурсивные программы. Имеется три цикла:

```
for (; bytes > 0 && p % sizeof(long) != 0; bytes--, p++)
        ret += hweight8(mem[p]);

for (; bytes > 0; bytes--, p++)
        ret += hweight8(mem[p]);

for (k = 0; k < lim; k++)
        w += hweight_long(memL[k]);
```

Построим для них рекурсивные программы.

```
bits1(bytes, p, ret0 : bytes', p', ret') {
        if (bytes > 0 && p % sizeof(long) != 0)
                bits1(bytes-1, p+1, ret0 + hweight8(mem[p]) : bytes', p', ret')
        else  { bytes' = bytes ||  p' = p ||  ret' = ret0 }
};

bits2(bytes, p, ret0 : ret') {
        if (bytes > 0)
                bits2(bytes-1, p+1, ret0 + hweight8(mem[p]) : ret')
        else ret' = ret0
};
```

```
weightL(k, w0, lim : k', w') {
        if (k < lim)
                weightL(k+1, w0 + hweight_long(memL[k])) : k', w')
        else { w' = w0 || k' = k }
}
```

Замена циклов на вызовы рекурсивных программ дает следующие программы:

```
size_t memweight(size_t ptr, bytes)
{
        bits1(bytes, ptr, 0 : bytes1, p, ret);
        size_t longs = bytes1 / sizeof(long);
        if (longs != 0) {
                assert(longs < INT_MAX / BITS_PER_LONG);
                ret1 = ret + bitmap_weight(longs * BITS_PER_LONG) ||
                bytes2 = bytes1 - longs * sizeof(long) ||
                p1 = p + longs * sizeof(long)
        } else { bytes2 = bytes1 || p1 = p || ret1 = ret }
        bits2(bytes2, p1, ret1 : ret2);
        return ret2;
}

int __bitmap_weight(unsigned int bits)
{
        unsigned int lim = bits/BITS_PER_LONG;
        weightL(0, 0 : k, int w);
        if (bits % BITS_PER_LONG != 0)
                w += hweight_long(memL[k] & BITMAP_LAST_WORD_MASK(bits));
        return w;
}
```

4. Спецификация предикатной программы

Предикатная программа memweight содержит набор глобальных типов, констант и переменных и включает набор функций memweight, bitmap_weight, __bitmap_weight, bits1, bits2 и weightL. Спецификации программы определяют предусловия и постусловия функций в дополнении к типам аргументов и результатов функций.

Сначала определим типы языка Си.

```
const int wsize;  // =sizeof(long); = 4 для 32 бит, =8 для 64 бит
type int_t = INT_MIN .. INT_MAX;
type uint_t = 0 ..  UINT_MAX;
type long_t = LONG_MIN ..  LONG_MAX;
type ulong_t = 0 .. ULONG_MAX;
type size_t = 0 ..  SIZE_MAX;
```

Значения констант для граничных значений типов зависят от архитектуры ЭВМ.

Определим тип Mem для глобального массива байтов mem, определяющего всю память.

```
type Mem = array(char, size_t);
Mem mem;
size_t   ptr, bytes;
```

Переменные ptr и bytes, параметры функции memweight, определены здесь как глобальные переменные. Перевычисляемые значения переменной bytes внутри функции memweight определены другими переменными.

Таким образом, исходная область памяти, в которой нужно определить число единичных битов, определяется вырезкой mem[ptr..ptr+bytes-1]. Эта область памяти разделяется на три части: короткий байтовый массив до первой границы слова, словный массив и короткий байтовый массив за последним словом. Словный массив находится внутри исходного байтового массива. Эквивалентом данного словного массива является массив memL.

Определим формализацию разделения на три части исходной области памяти, что позволит в дальнейшем существенно упростить доказательство формул корректности. Введем глобальные переменные:

```
size_t   ptrB, ptrU, ptrW, longW, bytesU, bytesW;
```

Здесь ptrB – ближайший к ptr индекс границы слова; ptrU = min(ptrB, ptr+bytes); ptrW – индекс байта в массиве mem за последним словом внутри вырезки mem[ptr..ptr+bytes-1]; longW – число слов внутри вырезки от позиции ptrU. Переменные bytesU и bytesW определяют оставшееся число байтов для позиций ptrU и ptrW в массиве mem. Обычно ptrU = ptrB. Однако если граница слова не достигается внутри вырезки mem[ptr..ptr+bytes-1], то ptrU < ptrB, и тогда ptrU соответствует концу исходной области памяти. Описанная модель представлена на Рис.1 и Рис.2.

Рис.1. Модель разбиения памяти. Общий случай: ptrU = ptrB

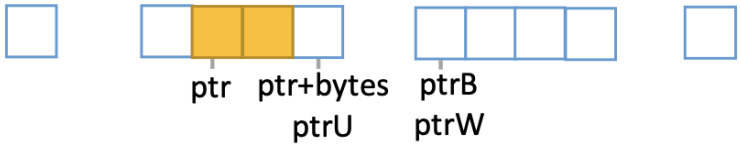

Рис.2. Модель разбиения памяти. Вырожденный случай: ptrU < ptrB

Формально, модель разбиения памяти представляется следующим набором аксиом.

```
formula upper(size_t  p) = p >= ptr & p % wsize = 0 & p – ptr < wsize;
axiom upper(ptrB);
axiom if ptr+bytes< ptrB then ptrU = ptr+bytes else ptrU = ptrB;
axiom bytesU = bytes – (ptrU – ptr);
axiom longW = bytesU / wsize ;
axiom ptrW = ptrU + longW * wsize;
axiom bytesW = bytes – (ptrW – ptr);
```

Представим функции bytew и memw, используемые далее в спецификациях.

```
formula bytew(char b : size_t);
formula memw(size_t  p, n : size_t);
```

Функция bytew определяет число единичных битов в байте b. Функция memw вычисляет число единичных битов вырезки mem[p..p+n–1].

Часть формул, используемых в спецификациях, вводятся без определений. Они будут определены на языке спецификаций why3 с использованием библиотечных функций. Формулы на языке P [3] не всегда адекватно выразимы в языке why3.

Функция memweight вычисляет число единичных битов, содержащихся в вырезке mem[ptr..ptr+bytes-1]. Ниже приводится код функции и постусловие. Отметим, что принадлежность результата ret типу size_t неявно считается частью постусловия.

```
memweight( : size_t ret)
post ret = memw(ptr, bytes)
{
        bits1(bytes, ptr, 0 : size_t bytes1, p1, ret1);
        size_t longs = bytes1 / wsize;
        if (longs != 0) {
                size_t ret2 = ret1 + bitmap_weight(longs * BITS_PER_LONG) ||
                size_t bytes2 = bytes1 - longs * wsize ||
                size_t p2 = p1 + longs * wsize
        } else { size_t  bytes2 = bytes1 || size_t p2 = p1 || size_t ret2 = ret1 } ;
        bits2(bytes2, p2, ret2 : ret)
}
```

Следует отметить, что ограничение ptr+bytes <= SIZE_MAX не может быть проверено при верификации. Оператор **assert** переписывается в виде:

```
theorem bytesU / wsize < INT_MAX / BITS_PER_LONG;
```

Однако данное условие недоказуемо.

В соответствии с определенной выше моделью массив memL является массивом слов длиной longW.

```
type MemL = array(ulong, longW);
MemL memL;
```

Массив memL тождественен вырезке mem[ptrU… ptrU + longW*wsize–1]. Элемент массива memL[k] эквивалентен вырезке mem[ptrU+k*wsize .. ptrU+k*wsize+7].

Ниже приведены коды программ bitmap_weight и __bitmap_weight вместе со спецификациями. В постусловии используется формула bitmw, определяющая число единичных битов в начальной части массива memL длиной nbits бит.

```
formula bitmw (nat nbits: nat);
formula BITMAP_LAST_WORD_MASK(nat nbits: nat) =
(~0UL >> (-(nbits) & (BITS_PER_LONG - 1)));

bitmap_weight(uint_t nbits : int_t ret)
pre nbits / BITS_PER_LONG <= longW
post ret = bitmw(nbits)
{
        if (nbits <= BITS_PER_LONG && nbits > 0)
                ret = hweight_long(memL[0] & BITMAP_LAST_WORD_MASK(nbits));
        else   ret = __bitmap_weight (nbits);
}

__bitmap_weight(uint_t bits: int_t ret)
pre nbits / BITS_PER_LONG <= longW
post ret = bitmw(bits)
{
        uint_t lim = bits/BITS_PER_LONG;
        weightL(0, 0, lim : uint_t k, int_t w);
        if (bits % BITS_PER_LONG != 0)
                w2 = w + hweight_long(memL[k] & BITMAP_LAST_WORD_MASK(bits));
        else w2 = w;
        ret = w2;
}
```

Далее представлены коды и спецификации функций bits1, bits2 и weightL, введенных взамен циклов исходной программы на языке Си. В процессе доказательства формул корректности функции bits1 предусловие было дополнено двумя конъюнктами.

```
bits1(size_t bytes0, p, ret0 : size_t bytes1, p1, ret)
pre ptr<=  p <= ptrU & p+bytes0 = ptr+bytes & ret0 = memw(ptr, p-ptr)
post ret1 = memw(ptr, p1-ptr) & p+bytes0 = p1+bytes1 & bytes1 = bytesU & p1 = ptrU
{
        if (bytes0 > 0 && p % wsize != 0)
                bits1(bytes0-1, p+1, ret0 + hweight8(mem[p]) : bytes1, p1, ret)
        else  { bytes1 = bytes0 ||  p1 = p ||  ret = ret0 }
};

bits2(size_t bytes, size_t p, size_t ret0 : size_t ret)
pre ret0 = memw(ptr, p-ptr)
post ret = ret0 + memw(p, bytes)
{
        if (bytes > 0)
                bits2(bytes-1, p+1, ret0 + hweight8(mem[p]) : ret)
        else ret = ret0
};
```

В спецификации функции weightL использована формула diapw, определяющая число единичных битов в вырезке memL[a .. b-1]. Предусловие функции weightL было уточнено в процессе доказательства формул корректности.

```
formula diapw (nat a, b: nat);

weightL(uint_t k, int_t w0, uint_t lim : uint_t k1, int_t w)
pre k <= lim < longW & w0 = diapw(0, k)
post w = diapw(0, lim) & k1=lim
{
        if (k < lim)
                weightL(k+1, w0 + hweight_long(memL[k])) : k1, w)
        else { w = w0 || k1 = k }
}

formula valL(nat k) = mem[k*wsize .. k*wsize+7];
axiom forall k=0..(longW-1). memL[k] = valL(k);
```

Количество единиц в битовом представлении слова w:

```
formula wordw (ulong w: nat);
```

Формула bitw определяет число бит в вырезке, начинающейся с mem[p] длиной bits битов. Точнее, это вырезка mem[p]...mem[p+bits/8] и плюс возможно в (bits % 8) битах следующего байта, если bits % 8 != 0.

```
formula bitw(size_t  p, uint_t  bits: int_t) =
        if bits >= 8 then memw(p, bits/8*8) + bitw8(p + bits/8*8, bits%8)
        else bitw8(p, bits)
```

Формула bitw8 определяет число битов в байте mem[p] среди первых bits битов.

```
formula bitw8(nat p, nat bits: int_t);
```

5. Процесс дедуктивной верификации

Для предикатной программы, построены формулы корректности программ относительно их спецификаций. Спецификации, определяющие типы, глобальные константы и переменные, формулы предусловий и постусловий, а также формулы корректности оформляются в виде теорий на языке P [3]. Разделение на теории позволяет уменьшить сложность, локализовать вспомогательные леммы, выделить зависимости, а главное, ускорить доказательство через SMT-решатели.

Далее теории кодируются на языке спецификаций Why3 [19]. Отметим, что некоторые определения заимствуются из библиотеки системы Why3. Теории на языке функционального программирования Why3ML представлены в Приложении 2 и отражают состояние после завершения доказательства, включая дополнительно введенные леммы.

Доказательство формул корректности проводилось в системе Why3 версии 1.3.1 с SMT-решателями Z3 4.8.6, CVC3 2.4.1, CVC4 1.7.

Небольшая часть формул была доказана автоматически, ещё несколько формул были доказаны с помощью простых трансформаций introduce_premises, inline_goal.

Для доказательства сложного утверждения, в частности, требующего доказательства по индукции, оказалось полезным использование аппарата лемма-функций [18]. Здесь строится вспомогательная предикатная программа, спецификация которой совпадает с исходным утверждением.

Многие леммы были доказаны в интерактивном режиме, когда процесс доказательства реализуется пользователем последовательным набором команд (трансформаций): introduce_premises, inline_goal, unfold, destruct, rewrite, split_vc, case, apply, assert, instantiate, induction и других команд.

В целом, процесс доказательства лемм можно описать рекурсивной процедурой:

1. делается попытка доказать с помощью разных SMT решателей.
2. если попытка удачна, то ветвь доказана.
3. в противном случае формула разбивается на более простые части с помощью команд split, destruct и других. Далее каждая из этих ветвей обрабатывается с шага 1. Если же

формула уже достаточно простая, то анализируются выражения, и добавляются подсказки с помощью трансформаций assert и case, далее получившиеся ветви обрабатываются с шага 1.

4. в остальных случаях доказательство проводится в интерактивном режиме.

Если какое-то предположение доказывается и встречается при доказательстве разных лемм, оно обобщается и вводится как отдельная лемма для переиспользования.

Отметим трудность автоматического доказательства с помощью SMT-решателей утверждений, содержащие одновременно операции с битовыми векторами и целыми числами.

Верификация битовых функций «битовой магии» hweight8, hweight32 и hweight64 не проводилась. Их верификация описывается в работе [14].

6. Обзор работ

Простые алгоритмы реализации операций с битовыми векторами неэффективны. Разработке эффективных алгоритмов посвящено множество работ. Сверхэффективные методы реализации, использующие нетривиальные свойства разнообразных числовых представлений, получили название битовой магии [4]. Однако работ по дедуктивной верификации алгоритмов битовой магии крайне мало. Алгоритмы битовой магии на языке Why3ML и их верификация в системе Why3 [19] описываются в работе [14]. Разработана теория битовых векторов на языке спецификаций why3, которая позже была включена в стандартную библиотеку системы Why3.

В работе [14] верифицируется несколько алгоритмов, в том числе на языке SPARK. Алгоритмы вычисления числа битов (веса Хэмминга) для 8-, 32- и 64-битных векторов собраны в библиотеке [13]. Они аналогичны соответствующим библиотечным функциям ядра ОС Linux, используемым в настоящей работе. Детали битовой магии подробно описываются в работах [4, 14]. Кроме того, проведена нетривиальная подгонка к используемым SMT-решателям с рекомендациями, как учитывать эти особенности при спецификации и верификации в системе Why3.

В настоящей работе рассматривалась задача вычисления числа битов в последовательности байтов. При этом верификация алгоритмов вычисления числа батов для 8-, 32- и 64-битных векторов не проводилась.

Следует отметить, что в процессе дедуктивной верификации библиотечных программ ядра ОС Linux наряду с системой Why3[19] ранее применялась также система Coq [15]. Теперь стало возможным проводить доказательство полностью в системе Why3, поскольку,

начиная с версии 1.1.3, в ней появилась полноценная система интерактивного доказательства.

7. Заключение

Проведена обратная трансформация библиотечной программы memweight, вычисляющей число элементов множества, представленного в виде битовой шкалы. Построена модель программы (Разд. 4, рис. 1 и 2), используемая при разработке спецификаций предикатной программы. По программе и спецификациям построены формулы корректности, оформленных в виде набора теорий, перенесенных на язык спецификаций why3. Доказательство проводилось в системе Why3[19]. В процессе верификации уточнялись спецификации. Остались недоказанными две формулы корректности, в которых встречаются операции преобразования битовых векторов в целые числа.

В процессе доказательства формул корректности установлена недоказуемость условия, указанного в операторе **assert**. Это означает, что оператор **assert** мог бы сработать на сверхдлинной последовательности байтов. Однако в процессе длительной эксплуатации ОС Linux подобная ситуация ни разу не случилась. Нами была предложена упрощающая модификация программы memweight без оператора **assert**. Модификация передана разработчикам ядра ОС Linux.

Итоги верификации. Отмечена ошибка переполнения выражения ptr+bytes за границы типа size_t. Недоказуемо условие longs < INT_MAX / BITS_PER_LONG оператора **assert**. Других ошибок не обнаружено.

Список литературы

1. Доказательство правил корректности операторов предикатной программы. [Электронный ресурс]. URL:http://www.iis.nsk.su/persons/vshel/files/rules.zip . (дата обращения 12.08.2020).
2. Каблуков И.В., Шелехов В.И. Реализация оптимизирующих трансформаций в системе предикатного программирования. *Системная информатика*, № 11. Новосибирск, 2017. С. 21-48. Электрон. журн. 2018. http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf. (дата обращения 12.08.2020).
3. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 28с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf. (дата обращения 12.08.2020).
4. Обстоятельно о подсчёте единичных битов. 2016. [Электронный ресурс]. URL: https://habr.com/ru/post/276957/. (дата обращения 12.08.2020).

5. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования. *Программная инженерия*, 2011, № 2. С. 14-21.
6. Шелехов В.И. Верификация предикатной программы бинарного поиска объекта произвольного типа. Новосибирск, 2019. 26с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/fsearch2.pdf . (дата обращения 12.08.2020).
7. Шелехов В.И. Верификация предикатной программы пирамидальной сортировки — Новосибирск, 2019. 36с.
8. Шелехов В.И. Дедуктивная верификация программы конкатенации строк с применением обратной трансформации // Знания-Онтологии-Теории (ЗОНТ-19). — Новосибирск, 2019. — 19с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/logcflc1.pdf. (дата обращения 12.08.2020).
9. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
10. Шелехов В.И. Списки и строки в предикатном программировании. *Системная информатика*, ИСИ СО РАН, Новосибирск. Электрон. журн. 2014, №3. С. 25-43. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/Listcharing.pdf. (дата обращения 12.08.2020).
11. Шелехов В.И., Чушкин М.С. Верификация программы быстрой сортировки с двумя опорными элементами. *Научный сервис в сети Интернет*. М.: ИПМ им. М.В.Келдыша, 2018. 26с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf. (дата обращения 12.08.2020).
12. Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., Heiser, G.: Cogent: Verifying high-assurance file system implementations // Int. Conference on Architectural Support for Programming Languages and Operating Systems. 2016. pp. 175–188.
13. Counting bits in a bit vector. [Электронный ресурс]. URL: http://toccata.lri.fr/gallery/bitcount.en.html. (дата обращения 12.08.2020).
14. Dross C., Fumex C., Gerlach J., Marché C.. High-Level Functional Properties of Bit-Level Programs: Formal Specifications and Automated Proofs. Research Report RR-8821, Inria Saclay. 2015, pp.52.
15. Kennedy T.R. Using program transformations to improve program translation. *Massachusetts Institute of Technology*. AI Memo No.962, 1897. 38p.
16. Shelekhov V. I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, No. 7, P. 421–427.
17. The Coq Proof Assistant. [Электронный ресурс]. URL: https://coq.inria.fr/. (дата обращения 12.08.2020).
18. Volkov, G., Mandrykin, M., Efremov, D.: Lemma functions for Frama-C: C programs as proofs. In: Proceedings of the 2018 Ivannikov ISPRAS Open Conference (ISPRAS-2018). pp. 31–38 (2018).

19. Why 3. Where Programs Meet Provers. [Электронный ресурс]. URL:http://why3.lri.fr. (дата обращения 12.08.2020).

Приложение 1. Полный исходный текст программы memweight.c на языке Си

```
#define KERNRELEASE "TEST"
#define BITS_PER_LONG 64
#define INT_MAX 0x7fffffff
#define __stringify_1(x...) #x
#define unlikely(x) __builtin_expect(!!(x), 0)
#define unreachable() do { __builtin_unreachable(); } while (0)
#define __stringify(x...) __stringify_1(x)
#define BUG() do { unreachable(); } while (0)
#define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while (0)
#define small_const_nbits(nbits) \
        (__builtin_constant_p(nbits) && (nbits) <= BITS_PER_LONG && (nbits) > 0)
#define BITMAP_FIRST_WORD_MASK(start) (~0UL << ((start) & (BITS_PER_LONG - 1)))
#define BITMAP_LAST_WORD_MASK(nbits) (~0UL >> (-(nbits) & (BITS_PER_LONG - 1)))
#define __always_inline inline
#define hweight8(w)  __arch_hweight8(w)
#define hweight32(w) __arch_hweight32(w)
#define hweight64(w) __arch_hweight64(w)

typedef unsigned long long __u64;

struct bug_entry {
 signed int bug_addr_disp;
 signed int file_disp;
 unsigned short line;
 unsigned short flags;
};

typedef unsigned long __kernel_ulong_t;
typedef __kernel_ulong_t __kernel_size_t;
typedef __kernel_size_t              size_t;
//-----------------------------------------------------------------------

#define CONFIG_ARCH_HAS_FAST_MULTIPLIER 1

/**
 * hweightN - returns the hamming weight of a N-bit word
 * @x: the word to weigh
 *
 * The Hamming Weight of a number is the total number of bits set in it.
```

```
 */

unsigned int __sw_hweight32(unsigned int w)
{
#ifdef CONFIG_ARCH_HAS_FAST_MULTIPLIER
        w -= (w >> 1) & 0x55555555;
        w =  (w & 0x33333333) + ((w >> 2) & 0x33333333);
        w =  (w + (w >> 4)) & 0x0f0f0f0f;
        return (w * 0x01010101) >> 24;
#else
        unsigned int res = w - ((w >> 1) & 0x55555555);
        res = (res & 0x33333333) + ((res >> 2) & 0x33333333);
        res = (res + (res >> 4)) & 0x0F0F0F0F;
        res = res + (res >> 8);
        return (res + (res >> 16)) & 0x000000FF;
#endif
}

unsigned long __sw_hweight64(__u64 w)
{
#ifdef CONFIG_ARCH_HAS_FAST_MULTIPLIER
        w -= (w >> 1) & 0x5555555555555555ul;
        w =  (w & 0x3333333333333333ul) + ((w >> 2) & 0x3333333333333333ul);
        w =  (w + (w >> 4)) & 0x0f0f0f0f0f0f0f0ful;
        return (w * 0x0101010101010101ul) >> 56;
#else
        __u64 res = w - ((w >> 1) & 0x5555555555555555ul);
        res = (res & 0x3333333333333333ul) + ((res >> 2) & 0x3333333333333333ul);
        res = (res + (res >> 4)) & 0x0F0F0F0F0F0F0F0Ful;
        res = res + (res >> 8);
        res = res + (res >> 16);
        return (res + (res >> 32)) & 0x00000000000000FFul;
#endif
}

static __always_inline unsigned long __arch_hweight64(__u64 w)
{
        unsigned long res;

        /*asm (ALTERNATIVE("call __sw_hweight64", POPCNT64, X86_FEATURE_POPCNT)
                        : "="REG_OUT (res)
                        : REG_IN (w));*/
  res = __sw_hweight64(w);

        return res;
}

static __always_inline unsigned int __arch_hweight32(unsigned int w)
```

```
{
        unsigned int res;

        /*asm (ALTERNATIVE("call __sw_hweight32", POPCNT32, X86_FEATURE_POPCNT)
                        : "="REG_OUT (res)
                        : REG_IN (w));*/

   res = __sw_hweight32(w);

        return res;
}

static inline unsigned int __arch_hweight16(unsigned int w)
{
        return __arch_hweight32(w & 0xffff);
}

static inline unsigned int __arch_hweight8(unsigned int w)
{
        return __arch_hweight32(w & 0xff);
}

static __always_inline unsigned long hweight_long(unsigned long w)
{
        return sizeof(w) == 4 ? hweight32(w) : hweight64(w);
}

int __bitmap_weight(const unsigned long *bitmap, unsigned int bits)
{
        unsigned int k, lim = bits/BITS_PER_LONG;
        int w = 0;

        for (k = 0; k < lim; k++)
                w += hweight_long(bitmap[k]);

        if (bits % BITS_PER_LONG)
                w += hweight_long(bitmap[k] & BITMAP_LAST_WORD_MASK(bits));

        return w;
}

static __always_inline int bitmap_weight(const unsigned long *src, unsigned int nbits)
{
        if (small_const_nbits(nbits))
                return hweight_long(*src & BITMAP_LAST_WORD_MASK(nbits));
        return __bitmap_weight(src, nbits);
}
```

```
size_t memweight(const void *ptr, size_t bytes)
{
        size_t ret = 0;
        size_t longs;
        const unsigned char *bitmap = ptr;

        for (; bytes > 0 && ((unsigned long)bitmap) % sizeof(long);
                        bytes--, bitmap++)
                ret += hweight8(*bitmap);

        longs = bytes / sizeof(long);
        if (longs) {
                BUG_ON(longs >= INT_MAX / BITS_PER_LONG);
                ret += bitmap_weight((unsigned long *)bitmap,
                                longs * BITS_PER_LONG);
                bytes -= longs * sizeof(long);
                bitmap += longs * sizeof(long);
        }
        /*
         * The reason that this last loop is distinct from the preceding
         * bitmap_weight() call is to compute 1-bits in the last region smaller
         * than sizeof(long) properly on big-endian systems.
         */
        for (; bytes > 0; bytes--, bitmap++)
                ret += hweight8(*bitmap);

        return ret;
}
```

Приложение 2. Теории на языке Why3ML

```
theory MemweightBase
    use int.Int, int.EuclideanDivision, int.NumOf
    use int.Sum as S
    use bool.Bool
    use array.Array, array.ArraySum, array.Init as ArrayInit
    use map.Const as MapConst
    use bv.BV_Gen as BV_Gen
    use bv.BV8 as BV8
    use bv.BV64 as BV64
    use bv.Pow2int

    constant wsize: int = 8 (* 8 для 64 бит, 4 для 32 бит *)
    constant isize: int = 4
    constant lsize: int = wsize
    constant size_max: int = pow2 (wsize * 8) - 1
    constant int_max: int = pow2 (isize * 8 - 1) - 1
    constant int_min: int = -int_max - 1
    constant uint_max: int = int_max * 2 + 1

    constant long_max: int = pow2 (lsize * 8) - 1
    constant long_min: int = -long_max - 1
    constant ulong_max: int = long_max * 2 + 1

    predicate int_t (v: int) = int_min <= v <= int_max
    predicate uint_t (v: int) = 0 <= v <= int_max
    predicate long_t (v: int) = long_min <= v <= long_max
    predicate ulong_t (v: int) = 0 <= v <= ulong_max
    predicate size_t (v: int) = 0 <= v <= size_max
    predicate char_t (v: int) = 0 <= v <= 255

    type mem_t = array int
    constant mem: mem_t
    constant ptr: int
    constant bytes: int
    axiom PtrType: size_t ptr
    axiom BytesType: size_t bytes
    axiom MemLimit: size_t (ptr + bytes)
    axiom Mem_array_char: forall i: int. size_t i -> char_t mem[i]
    axiom Mem_length: length mem = size_max + 1

    predicate upper (p: int) = p = ptr + (mod (wsize - (mod ptr wsize)) wsize)

    lemma UniUpper: forall x y: int. upper x /\ upper y -> x = y
    lemma PropUpper: forall x: int. upper x  -> (mod x wsize) = 0 /\ x - ptr < wsize
```

```
    constant ptrB: int
    axiom PtrB: upper ptrB

    constant ptrU: int
    axiom PtrU: if (ptr + bytes) < ptrB then ptrU = ptr + bytes else ptrU = ptrB
    constant bytesU: int
    axiom BytesU: bytesU = bytes - (ptrU - ptr)

    constant bits_per_long: int = wsize * 8
    axiom BytesU_limit: (div bytesU wsize) < (div int_max bits_per_long)

    constant longW: int
    axiom LongW: longW = div bytesU wsize
    lemma LongW_limit: longW * wsize <= bytesU
    constant ptrT: int
    axiom PtrW: ptrT = ptrU + longW * wsize
    constant bytesT: int
    axiom BytesW: bytesT = bytes - (ptrT - ptr)

    lemma BytesW_lt_wsize: bytesT < wsize
    lemma PtrW_: ptrU < ptrB -> ptrT = ptrU

    (* количество единиц в битовом представлении байта b *)
    let function bytew (b: int): int
       requires { char_t  b}
    =  numof (BV8.nth (BV8.of_int b)) 0 8
    lemma ByteW: forall x: int. char_t x -> bytew x <= 8

    predicate i_mem (p: int) = ptr <= p < ptr + bytes
    predicate p_memw (p b: int) = b >= 0 /\ ptr <= p /\ p + b <= ptr + bytes (*
разрешить p = ptr + bytes /\ b = 0 *)
    lemma P_memw_imem: forall p b: int. p_memw p b -> b = 0 \/ (i_mem p /\ i_mem
(p+b-1))

    constant memW: array int       (* массив весов байтов из mem *)
    axiom MemW_init: forall i: int.
       i_mem i -> memW[i] = bytew mem[i]
    axiom MemW_size: length memW = length mem (* ptr + bytes *)

    let ghost function memw_i (p: int): int
       requires { i_mem p }
    =   bytew mem[p]

    lemma MemW_memw_i_eq: forall p: int. i_mem p -> memw_i p = memW[p]

    lemma Memw_i_limit1: forall p: int. i_mem p -> bytew mem[p] <= 8
    lemma Memw_i_limit2: forall p: int. i_mem p -> memw_i p <= 8
```

```
   let ghost function memw (p b: int): int
      requires { p_memw p b }
(*   =   S.sum memW.elts p (p+b) (*v1*) *)
   = if b = 0 then 0 else S.sum memW.elts p (p+b) (* v2 super-fast Z3 *)
(*   = sum memW p (p+b) (*v3*) *)

   lemma Sum_Zero: forall f: int -> int, i: int. S.sum f i i = 0
   lemma Memw_ZeroBytes: forall p: int. p_memw p 0 -> memw p 0 = 0

   lemma MemW_def2_eq: forall p b: int. p_memw p b -> memw p b = sum memW p
(p+b)
   lemma MemW_def3_eq: forall p b: int. p_memw p b -> memw p b = S.sum memw_i p
(p+b)
   lemma MemW_eq_sum_memw_i: forall p b: int. p_memw p b -> memw p b = S.sum
memw_i p (p+b)

   let rec lemma _Sum_limit (n: int) (f: int -> int) (limit a: int)
     requires { 0 <= n /\ 0 <= limit /\ 0 <= a /\ (forall i: int. a <= i < a+n -> (f i) <=
limit) }
     ensures { S.sum f a (a+n) <= n * limit }
     variant { n }
   = if n > 0 then _Sum_limit (n-1) f limit a

   let rec lemma _MemW_sum_rec (n: int) (p: int)
     requires { p_memw p n }
     ensures { memw p n <= n*8 }
     variant { n }
   = if n > 0 then _MemW_sum_rec (n-1) p

   lemma MemW_sum: forall p x: int. p_memw p x ->  memw p x <= x*8
   axiom Bytes_size_t_limit: size_t (8 * bytes) (* ??????результат memweight влезет в
size_t *)
   lemma Bytes_sum: bytes = (bytes - bytesU) + (bytesU - bytesT) + bytesT
   lemma Memw_SumParts2: forall p p1 b: int. (p <= p1 < p + b /\ p_memw p b) ->
     let b0 = p1-p in
       memw p b = (memw p b0) + (memw p1 (b-b0))
   lemma Memw_SumParts3: forall p p1 p2 b: int. (p <= p1 <= p2 < p + b /\ p_memw p
b) ->
     let b0 = p1-p in let b1 = p2-p1 in let b2 = b-b0-b1 in
        memw p b = (memw p b0) + (memw p1 b1) + (memw p2 b2)
   lemma Memw_SumParts1: forall p b1 b2: int. (p_memw p b1) /\ (p_memw (p+b1) b2)
/\ (p_memw p (b1+b2)) ->
     (memw p b1) + (memw (p+b1) b2) = memw p (b1+b2)
   lemma SumParts: memw ptr bytes =
      memw ptr (bytes - bytesU) + memw ptrU (bytesU - bytesT) + memw ptrT bytesT

   let ghost function bitw8 (p bits: int): int
      requires { size_t p  /\ 0 <= bits < 8 }
```

```
 =   numof (BV8.nth (BV8.of_int mem[p])) 0 bits

let ghost function bitw (p bits: int): int
   requires { p_memw p (div bits 8) /\ 0 <= bits }
=   if bits >= 8 then
       let bt = div bits 8 in memw p bt + bitw8 (p + bt) (mod bits 8)
   else bitw8 p bits

type memL_t = array int
constant memL: memL_t
axiom MemL_length: length memL = longW
axiom MemL_elem_limit: forall k: int. 0 <= k < longW -> ulong_t memL[k]

let ghost function valL (k: int): int
   requires {  0 <= k < longW }
=   let j = k*wsize in    (*  mem[j .. j+7] *)
   ((((((mem[j]*256 + mem[j+1]) * 256 + mem[j+2]) * 256 + mem[j+3]) * 256 +
            mem[j+4]) * 256+ mem[j+5]) * 256 + mem[j+6]) * 256 + mem[j+7]


axiom MemL: forall k: int. if 0 <= k < longW then memL[k] = valL k else memL[k] = 0
lemma MemL_limit: forall k: int. 0 <= memL[k] <= ulong_max

(* количество единиц в битовом представлении слова  w *)
let function wordw (w: int): int
   requires { ulong_t  w}
=   numof (BV64.nth (BV64.of_int w)) 0 64

axiom BV64_int_conv: forall w: int. ulong_t w -> BV64.to_int (BV64.of_int w) = w
axiom BV64_of_int_Zero: BV64.of_int 0 = BV64.zeros
lemma Wordw_Limit: forall w: int. ulong_t w -> wordw w <= 64

let ghost function memlw (i: int): int
   (*requires { 0 <= i < longW }   ?ломается ли корректность?*)
=   if 0 <= i < longW then wordw memL[i] else 0

let ghost function diapw (a b: int): int
   requires { 0 <= a <= b <= longW /\ a < longW }
=   S.sum memlw a b

lemma Diapw_Zero: forall a: int. 0 <= a < longW -> diapw a a = 0

let function wordwB (w bits: int): int
   requires { ulong_t  w /\ 0 <= bits <= 64}
=   if (bits = 0 || w = 0) then 0
   else numof (BV64.nth (BV64.of_int w)) 0 bits

lemma WordwB_Zero: forall b: int. ulong_t b -> wordwB b 0 = 0
```

```
   lemma WordwB_Limit: forall w bits: int. ulong_t w /\ 0 <= bits <= 64 -> wordwB w
bits <= bits

   let ghost function bitmw (nbits: int): int
      requires { 0 <= nbits <= longW * 64 }
   =   let wholewords = div nbits 64 in let tailbits = mod nbits 64 in
      (S.sum memlw 0 wholewords) + if tailbits = 0 then 0 else (wordwB
(memL[wholewords]) tailbits)

   let rec lemma _Bitmw_whole_limit (n: int)
     requires { 0 <= n <= longW }
     ensures { (S.sum memlw 0 n) <= n*64 }
     variant { n }
   = if n > 0 then _Bitmw_whole_limit (n-1)

   lemma Bitmw_Limit: forall nbits: int. 0 <= nbits <= longW * 64 -> bitmw nbits <=
nbits

   (* simpler implementation *)
   function bitmap_last_word_mask   (nbits: int): BV64.t =
      if nbits = 0 then BV64.ones
      else BV64.lsr BV64.ones (bits_per_long - nbits)

   (* = (~0UL >> (-(nbits) & (bits_per_long - 1)))*)
   function long_mask_nbits(v nbits: int): int =
      (BV64.to_int
         (BV64.bw_and (BV64.of_int v) (bitmap_last_word_mask nbits)))

end (*MemweightBase*)


theory Memweight
   use MemweightBase
   use int.Int, int.EuclideanDivision

   predicate qmemweight(ret: int) = ret = (memw ptr bytes) && ret <= size_max &&
size_t ret

   predicate pbits1 (bytes0 p ret0: int) =
       size_t bytes0 /\ size_t p /\ size_t ret0 /\
       ptr <= p <= ptrU /\ p+bytes0 = ptr+bytes /\ ret0 = memw ptr (p-ptr)
   predicate qbits1 (bytes0 p bytes1 p1 ret1: int) =
       (ret1 = memw ptr (p1-ptr) /\ p+bytes0 = p1+bytes1 /\ bytes1 = bytesU /\ p1 =
ptrU)
         && (size_t bytes1 /\ size_t p1 /\ size_t ret1)

   goal QSB1: pbits1 bytes ptr 0
```

```
   predicate pbits2 (b p ret0: int) =
      size_t b /\ size_t p /\ size_t ret0 /\ p_memw p b /\ ret0 = memw ptr (p-ptr)
   predicate qbits2 (p b ret0 ret: int) =  ret = ret0 + (memw p b) && size_t ret

   predicate pbitmap_weight (nbits: int) =
      uint_t nbits && div nbits bits_per_long <= longW

   predicate qbitmap_weight (nbits ret: int) =
      ret = bitmw nbits && int_t ret  (*bitw ptrT bits*)

   goal QSB0: forall bytes1 p1 ret1 longs: int.
         qbits1 bytes ptr bytes1 p1 ret1 /\ longs = div bytes1 wsize  /\ not (longs = 0)
          ->   pbitmap_weight (longs * bits_per_long)

   lemma RB1a : forall longs: int.    longs = div bytesU wsize ->
      (memw ptr (ptrU - ptr) + bitmw (longs * bits_per_long)) = memw ptr ((ptrU +
(longs * wsize)) - ptr)
   lemma RB2a : forall longs: int.    longs = div bytesU wsize ->
       ((memw ptr (ptrU - ptr) + bitmw (longs * bits_per_long))
             + memw (ptrU + (longs * wsize)) (bytesU - (longs * wsize)))
                = memw ptr bytes

   goal RB1: forall bytes1 p1 ret1 bytes2 p2 ret2 longs tt: int.
      size_t bytes1 /\ size_t p1 /\ size_t ret1 /\ size_t bytes2 /\ size_t p2 /\ size_t ret2 /\
size_t longs /\ size_t tt /\
      qbits1 bytes ptr bytes1 p1 ret1 /\
      longs = div bytes1 wsize /\
      longs <> 0 /\
      qbitmap_weight (longs * bits_per_long) tt /\ (*tt = bitw ptrT (longs * bits_per_long)
*)
      ret2 = ret1 + tt /\
      bytes2 = bytes1 - longs * wsize /\
      p2 = p1 + longs * wsize
      -> pbits2 bytes2 p2 ret2

   goal RB2: forall bytes1 p1 ret1 bytes2 p2 ret2 ret longs tt: int.
      size_t bytes1 /\ size_t p1 /\ size_t ret1 /\ size_t bytes2 /\ size_t p2 /\ size_t ret2 /\
size_t ret /\ size_t longs /\
      qbits1 bytes ptr bytes1 p1 ret1 /\
      longs = div bytes1 wsize /\
      longs <> 0 /\
      qbitmap_weight (longs * bits_per_long) tt /\
      ret2 = ret1 + tt /\
      bytes2 = bytes1 - longs * wsize /\
      p2 = p1 + longs * wsize /\
      qbits2 p2 bytes2 ret2 ret
      -> qmemweight ret
```

```
   goal RB3: forall bytes1 p1 ret1 bytes2 p2 ret2 longs: int.
      size_t bytes1 /\ size_t p1 /\ size_t ret1 /\ size_t bytes2 /\ size_t p2 /\ size_t ret2 /\
size_t longs /\
      qbits1 bytes ptr bytes1 p1 ret1 /\
      longs = div bytes1 wsize /\
      longs = 0 /\
      ret2 = ret1 /\
      bytes2 = bytes1 /\
      p2 = p1
      -> pbits2 bytes2 p2 ret2
   goal RB4: forall bytes1 p1 ret1 bytes2 p2 ret2 ret longs: int.
      size_t bytes1 /\ size_t p1 /\ size_t ret1 /\ size_t bytes2 /\ size_t p2 /\ size_t ret2 /\
size_t ret /\ size_t longs /\
      qbits1 bytes ptr bytes1 p1 ret1 /\
      longs = div bytes1 wsize /\
      longs = 0 /\
      ret2 = ret1 /\
      bytes2 = bytes1 /\
      p2 = p1 /\
      qbits2 p2 bytes2 ret2 ret
      -> qmemweight ret
end (*Memweight*)

theory Bits1
   use MemweightBase, Memweight
   use int.Int, int.EuclideanDivision
   use array.Array

   goal QC3: forall bytes0 p ret0 bytes1 p1 ret: int.
      size_t bytes0 /\ size_t p /\ size_t ret0 /\ size_t bytes1 /\ size_t p1 /\ size_t ret /\
      pbits1 bytes0 p ret0 /\
      ( bytes0 = 0 \/ (mod p wsize) = 0 ) /\
      bytes1 = bytes0 /\ p1 = p /\ ret = ret0
      -> qbits1 bytes0 p bytes1 p1 ret

   predicate qhweight8 (v w: int) = w = bytew v && uint_t w

   goal RB7: forall bytes0 p ret0 tt: int.
      size_t bytes0 /\ size_t p /\ size_t ret0 /\ uint_t tt /\
      pbits1 bytes0 p ret0 /\
      bytes0 > 0 /\
      mod p wsize <> 0 /\
      qhweight8 mem[p] tt
      -> pbits1 (bytes0-1) (p+1) (ret0+tt)

   goal RB8: forall bytes0 p ret0 bytes1 p1 ret tt: int.
      size_t bytes0 /\ size_t p /\ uint_t ret0 /\ size_t bytes1 /\ size_t p1 /\
      size_t ret /\ uint_t tt /\
```

```
        pbits1 (bytes0-1) p ret0 /\
        bytes0 > 0 /\
        mod p wsize <> 0 /\
        qhweight8 mem[p] tt /\
        qbits1 (bytes0-1) (p+1) bytes1 p1 ret
        -> qbits1 bytes0 p bytes1 p1 ret
end (*bits1*)


theory Bits2
    use MemweightBase, Memweight, Bits1
    use int.Int
    use array.Array

    goal QC4: forall b p ret0 ret: int.
        size_t b /\ size_t p /\ size_t ret0 /\ size_t ret /\
        pbits2 b p ret0 /\
        b = 0 /\ ret = ret0
        -> qbits2 p b ret0 ret
    goal RB9: forall b p ret0 tt: int.
        size_t b /\ size_t p /\ size_t ret0 /\  size_t tt /\
        pbits2 b p ret0 /\
        b > 0 /\
        qhweight8 mem[p] tt
        -> pbits2 (b-1) (p+1) (ret0+tt)
    goal RB10: forall b p ret0 ret tt: int.
        size_t b /\ size_t p /\ size_t ret0 /\ size_t ret /\ uint_t tt /\
        pbits2 b p ret0 /\
        b > 0 /\
        qhweight8 mem[p] tt /\
        qbits2 (p+1) (b-1) (ret0 + tt) ret
        -> qbits2 p b ret0 ret
end (*bits2*)


theory WeightL
    use MemweightBase
    use int.Int, int.NumOf, int.EuclideanDivision
    use array.Array

    predicate pweightL (w0 k lim: int) = int_t w0 /\ uint_t k /\ k <= lim /\ lim < longW /\
w0 = diapw 0 k (*memw ptrU k*wsize*)
    predicate qweightL (w0 lim k1 w: int) = int_t w0 /\ uint_t lim /\ uint_t k1 /\ int_t w
            -> w = diapw 0 lim /\ k1 = lim (*w = w0 + (memw ptrT (lim * wsize)) /\
k1=lim*)
    predicate qhweight_long (w ret: int) = ulong_t w ->
            ret = numof (BV64.nth (BV64.of_int w)) 0 (wsize * 8) && ulong_t ret
```

```
   (* additional restrictions *)
   lemma Bitmw_Eq_Diapw_longs: forall n: int. 0 <= n < longW ->
      bitmw (n * bits_per_long) = diapw 0 n
   lemma Bitmw_Eq_Diapw_bits: forall bits longs nbits: int.
      0 <= bits < longW * bits_per_long /\ nbits = (mod bits bits_per_long) /\ longs =
(div bits bits_per_long)
     ->
      longs <= longW /\
      bitmw bits = (diapw 0 longs) + if nbits = 0 then 0 else (wordwB memL[longs] nbits)

   let rec lemma _DiapW_Bitmw_eq (b: int)
    requires { 0 <= b < longW }
    ensures { diapw 0 b = bitmw (b * bits_per_long) }
    variant { b }
   = if (0 < b) then _DiapW_Bitmw_eq (b-1)

   axiom Diapw_Limit_Int: forall k: int. 0 <= k < longW -> int_t (diapw 0 k)
   lemma Diapw_Step: forall n: int. 0 <= n < longW-1 -> diapw 0 n + wordwB memL[n]
(wsize * 8) = diapw 0 (n+1)
   lemma Wordw_eq_wordwB: forall w: int. ulong_t w -> wordw w = wordwB w (wsize *
8)
   lemma Qhweight_long_eq_wordw: forall w: int. ulong_t w -> qhweight_long w (wordw
w)
   (* end additional restrictions *)


   goal QC5: forall k w0 lim k1 w: int.
      uint_t k /\ int_t w0 /\ uint_t lim /\ uint_t k1 /\ int_t w /\
      pweightL w0 k lim /\
      k >= lim /\
      w = w0 /\ k1 = k
      -> qweightL w0 lim k1 w
   goal RB11: forall k w0 lim tt: int.
      uint_t k /\ int_t w0 /\ w0 = diapw 0 k /\ uint_t lim /\ int_t tt /\
      pweightL w0 k lim /\
      k < lim /\
      qhweight_long memL[k] tt
      -> pweightL (w0+tt) (k+1) lim
   goal RB12: forall k w0 lim k1 w tt: int.
      uint_t k /\ int_t w0 /\ uint_t lim /\ uint_t k1 /\ int_t w /\ int_t tt /\
      pweightL w0 k lim /\
      k < lim /\
      qhweight_long memL[k] tt /\
      qweightL (w0+tt) lim (k+1) w
      -> qweightL w0 lim k w
end  (*weightL*)
```

```
theory A__bitmap_weight
  use MemweightBase, WeightL
  use int.Int, int.EuclideanDivision, int.NumOf
  use array.Array

  predicate p__bitmap_weight (bits: int) =
    uint_t bits && div bits bits_per_long <= longW

  predicate q__bitmap_weight (bits ret: int) =
    ret = bitmw bits && int_t ret (*bitw ptrT bits*)

  goal QSB2: forall bits lim: int.
    uint_t bits /\ uint_t lim /\
    lim = div bits bits_per_long /\ lim < longW /\ longW > 0
    -> pweightL 0 0 lim
  goal QC2: forall bits lim w k ret: int.
    uint_t bits /\ uint_t lim /\ int_t w /\ uint_t k /\ int_t ret /\
    lim = div bits bits_per_long /\ lim < longW /\
    qweightL 0 lim k w /\
    mod bits bits_per_long = 0 /\
    ret = w
    -> q__bitmap_weight bits ret

  axiom WordwB_eq_mask: forall w b: int. ulong_t w /\ 0<=b<=bits_per_long ->
    wordw (long_mask_nbits w b) = wordwB w b
  lemma Qhweight_long_eq_wordwB: forall w tt: int. ulong_t w /\ qhweight_long w tt ->
tt = wordw w
  lemma Qhweight_long_eq_wordwB2: forall w tt: int. ulong_t w -> numof (BV64.nth
(BV64.of_int w)) 0 (wsize * 8) = wordw w
  lemma Long_mask_nbits_Limit: forall w, nbits: int. ulong_t w /\ 0 <= nbits <=
bits_per_long -> long_mask_nbits w nbits <= w

  goal COR1: forall bits lim k w tt ret nbits: int.
    uint_t bits /\ uint_t lim /\ uint_t k /\ int_t w /\ ulong_t tt /\ uint_t ret /\
    lim = div bits bits_per_long /\ lim < longW /\
    qweightL 0 lim k w /\
    nbits = mod bits bits_per_long /\ nbits <> 0 /\
    qhweight_long (long_mask_nbits memL[k] nbits) tt /\          (* memL[k] &
bitmap_last_word_mask(bits) *)
    ret = w + tt
    -> q__bitmap_weight bits ret
end (*A__bitmap_weight*)

theory Bitmap_weight
  use MemweightBase, Memweight, WeightL, A__bitmap_weight
  use int.Int, int.EuclideanDivision
  use int.NumOf
  use array.Array
```

```
   predicate lessLong (nbits: int) = 0 < nbits <= bits_per_long

   goal RB5: forall nbits ret: int.
      uint_t nbits /\ int_t ret /\
      div nbits bits_per_long <= longW /\
      lessLong nbits /\
      qhweight_long (long_mask_nbits memL[0] nbits) ret (* memL[0] &
bitmap_last_word_mask(nbits) *)
      -> qbitmap_weight nbits ret
   goal RB6: forall nbits ret: int.
      uint_t nbits /\ int_t ret /\
      not lessLong nbits /\
      q__bitmap_weight nbits ret
      -> qbitmap_weight nbits ret
end (*bitmap_weight*)
```