Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

В.И. Шелехов

РАЗРАБОТКА ЭФФЕКТИВНЫХ ПРОГРАММ СТАНДАРТНЫХ ФУНКЦИЙ FLOOR, ISQRT И ILOG2 ПО ТЕХНОЛОГИИ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Препринт
154

Новосибирск 2010

Описывается опыт разработки по технологии предикатного программирования быстрых программ трех стандартных функций: целой части плавающего числа, целочисленного квадратного корня и целочисленного двоичного логарифма.

Siberian Branch of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

V.I. Shelekhov

DEVELOPMENT OF EFFECTIVE PROGRAMS FOR THE STANDARD FUNCTIONS FLOOR, ISQRT, AND ILOG2 UNDER PREDICATE PROGRAMMING TECHNOLOGY

Preprint
154

Novosibirsk 2010

Development of effective programs for the standard functions under predicate programming technology is described. The developed functions are integer part of float value, integer square root, and integer part of binary logarithm.

1. ВВЕДЕНИЕ

Проведен эксперимент по разработке нескольких небольших программ на C++ по технологии предикатного программирования [5] с формальной верификацией в системе автоматического доказательства PVS [1]. Среди них – быстрые программы вычисления стандартных функций**:** целой части плавающего числа, целочисленного квадратного корня и целочисленного двоичного логарифма. Опыт разработки и верификации этих трех программ описывается в данном препринте. Программы указанных стандартных функций используются в программной части известной системы спутниковой навигации «Навител Навигатор» для автомобилей; их исходные коды на C++ приведены в приложениях 2 – 4. Была поставлена задача по возможности ускорить программы данных функций и проверить их правильность посредством формальной верификации на PVS, используя ранее разработанный алгоритм генерации условий корректности предикатных программ с однозначной спецификацией [2, 3].

Последовательность работ, проводимых для каждой стандартной функции, следующая. На основе исходного кода стандартной функции (см. приложения 2 – 4) с использованием описаний алгоритмов, размещенных в Интернете, воспроизводилось математическое описание алгоритма, использованное для программирования данной стандартной функции. На базе математического описания алгоритма строилась соответствующая предикатная программа. Далее проводилась оптимизирующая трансформация [5] предикатной программы с получением программы на императивном расширении языка предикатного программирования P [4]. Наконец, программа конвертировалась с императивного расширения на язык C++. Целью описанного процесса является построение такой предикатной программы и такой последовательности трансформаций, чтобы в результате получить исходный код стандартной функции. Когда эта цель достигалась, к предикатной программе применялся алгоритм генерации условий корректности программы с построением набора теорий на языке спецификаций PVS. Далее проводился процесс доказательства на PVS сгенерированного набора лемм. Для каждой из трех стандартных функций этот процесс был длительным и трудоемким и сопровождался доказательством серии дополнительных математических свойств с построением дополнительных теорий и библиотек теорий на PVS. Отметим, что генерация условий корректности и оптимизирующие трансформации проводились вручную.

Пусть предикатная программа представлена оператором S со спецификацией в виде тройки Хоара:

pre P(x) { S } **post** Q(x, y)

Здесь x – список аргументов, y – список результатов оператора S, P(x) – предусловие, Q(x, y) – постусловие оператора. Спецификация является тотальной (реализуемой) и однозначной, если истинна формула:

$$\forall x.\ P(x) \Rightarrow \exists!\ y.\ Q(x, y).$$

Здесь x – аргументы, а y – результаты оператора. Доказательство тотальной корректности для однозначных программ с тотальной (реализуемой) спецификацией можно проводить по более простой формуле [3]:

$$P(x)\ \&\ Q(x, y) \Rightarrow LS(S)(x, y) \quad , \qquad (1)$$

где LS(S) – предикат, являющийся логическим эквивалентом оператора S в соответствии с *логической* семантикой [2]. Имеется алгоритм генерации условий корректности, эффективно декомпозирующий приведенную формулу в набор удобных для доказательства условий корректности в зависимости от структуры оператора S. Правила этого алгоритма, используемые в данном препринте, приведены в приложении 1.

Эффективная императивная программа на императивном расширении языка P [4] получается применением последовательности трансформаций предикатной программы [5]. Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование структурных объектов низкоуровневыми структурами с использованием массивов и указателей;
- развертка цикла.

В разделах 2, 3 и 4 для трех стандартных функций описывается разработка предикатных программ, генерация условий корректности и верификация на PVS. Итоги проведенного эксперимента суммируются в заключении. Правила, использованные при генерации условий корректности, описаны в приложении 1.

2. АЛГОРИТМЫ ВЫЧИСЛЕНИЯ КВАДРАТНОГО КОРНЯ

Рассмотрим алгоритмы вычисления целой части квадратного корня, т.е. функции y = isqrt(x) = floor($\sqrt{x}$), где floor – функция вычисления целой

части вещественного аргумента. Функцию isqrt можно представить следующей спецификацией:

```
isqrt(nat x : nat m) post m^2 <= x < (m + 1)^2;
```

Здесь "^" операция возведения в степень.

2.1. Простейший алгоритм

Самое простое решение задачи вычисления значения функции isqrt – посредством последовательного перебора значений аргумента. Искомое значение ближе к нулю, чем к аргументу x. поэтому лучше начать с нуля. Чтобы получить программу с хвостовой рекурсией, введем следующее обобщение задачи:

```
sq0(nat x, k : nat m) pre  k^2 <= x post m^2 <= x < (m + 1)^2;
```

Дополнительный аргумент k находится между нулем и искомым решением. Предикатная программа очевидна и представлена ниже:

```
isqrt(nat x : nat m)
{ sq0(x, 0: m) }
post m^2 <= x < (m + 1)^2;

sq0(nat x, k : nat m)
pre  k^2 <= x
{ if (x < (k + 1)^2) m = k else  sq0(x, k + 1: m) }
post m^2 <= x < (m + 1)^2;
```

2.2. Вторая версия алгоритма

Заметным недостатком алгоритма sq0 является вычисление квадрата в выражении (k + 1)^2. Между тем значение k^2 вычисляется на предыдущем шаге. С его помощью можно упростить вычисление (k + 1)^2. Рассмотрим следующую спецификацию:

```
sq1(nat x, k, n : nat m) pre  k^2 <= x & n = k^2 post m^2 <= x < (m + 1)^2;
```

Дополнительный аргумент n хранит значение k^2, вычисленное на предыдущем шаге. Реализация данной идеи воплощается в следующей программе:

```
isqrt(nat x : nat m)
{ sq1(x, 0, 0: m) }
post m^2 <= x < (m + 1)^2;
```

```
sq1(nat x, k, n : nat m)
pre  k^2 <= x & n = k^2
{     nat p = n + 2* k + 1;
      if (x < p) m = k else  sq1(x, k + 1, p: m)
}
post m^2 <= x < (m + 1)^2;
```

Построим соответствующую императивную программу для приведенной выше предикатной программы. На первом этапе трансформации предикатной программы реализуются следующие склеивания переменных.

```
sq1: m <- k; n <- p;
```

В результате склеивания получим

```
sq1(nat x, m, n : nat m)
{     n = n + 2* k + 1;
      if (x < n) m = m else  sq1(x, m + 1, n: m)
}
```

На втором этапе реализуется замена хвостовой рекурсии циклами. Результатом проведения двух этапов трансформации является следующая программа:

```
isqrt(nat x : nat m)
{ sq1(x, 0, 0: m) }

sq1(nat x, m. n : nat m)
{     for(;;) {
            n = n + 2* m + 1;
            if (x < n) break else  m = m + 1
      }
}
```

На третьем этапе проведем подстановку тела определения sq1 на место вызова. Итоговая программа приведена ниже.

```
isqrt(nat x : nat m)
{     m = 0; nat n = 0;
      for(;;) {
            n = n + 2* m + 1;
            if (x < n) break else  m = m + 1
      }
}
```

2.3. Следующее улучшение алгоритма

Вычисление 2* k + 1 реализуется быстрее, чем (k + 1)^2, В свою очередь, вычисление 2* k + 1 можно ускорить, используя простой факт, что его значение увеличивается на 2 на каждом шаге. Рассмотрим спецификацию:

```
sq2(nat x, k, n, d: nat m)
pre k^2 <= x & n = k^2 & d = 2 * k + 1
post m^2 <= x < (m + 1)^2;
```

Дополнительный параметр d обеспечивает возможность ускоренного вычисления 2 * k + 1. Соответствующая программа приведена ниже:

```
isqrt(nat x : nat m)
{ sq2(x, 0, 0, 1: m) }
post m^2 <= x < (m + 1)^2;

sq2(nat x, k, n, d: nat m)
pre k^2 <= x & n = k^2 & d = 2 * k + 1
{     nat p = n + d;
      if (x < p) m = k else  sq2(x, k + 1, p, d + 2: m)
}
post m^2 <= x < (m + 1)^2;
```

2.4. Еще одна оптимизация алгоритма

Приведем снова алгоритм sq2.

```
sq2(nat x, k, n, d: nat m)
pre k^2 <= x & n = k^2 & d = 2 * k + 1
{     nat p = n + d;
      if (x < p) m = k else  sq2(x, k + 1, p, d + 2: m)
}
```

Здесь p используется только в операции сравнения x < p, которая эквивалентна x – p < 0. Мы можем пересчитывать на каждом шаге значение x – n. В этом случае переменные p и n будут не нужны. Определим спецификацию:

```
sq3(nat x, y, k, d: nat m)
pre k^2 <= x & d = 2 * k + 1 & y = x - k^2
post m^2 <= x < (m + 1)^2;
```

Программа получается соответствующими модификациями программы sq2.

```
isqrt(nat x : nat m)
{ sq3(x, x, 0, 1: m) }
post m^2 <= x < (m + 1)^2;

sq3(nat x, y, k, d: nat m)
pre k^2 <= x & d = 2 * k + 1 & y = x - k^2
{     if (y < d) m = k else  sq3(x, y – d, k + 1, d + 2: m) }
post m^2 <= x < (m + 1)^2;
```

Построим императивную программу для данной предикатной программы. На первом этапе трансформации приведенной предикатной программы в эффективную императивную реализуется следующие склеивания переменных.

```
sq3: m <- k; x <- y;
```

В результате склеивания получим:

```
sq3(nat x, x, m, d: nat m)
{     if (x < d) m = m else sq3(x, x – d, m + 1, d + 2: m) }
```

На втором этапе реализуется замена хвостовой рекурсии циклами. Результатом проведения двух этапов трансформации является следующая программа:

```
isqrt(nat x : nat m)
{ sq3(x, x, 0, 1: m) }

sq3(nat x, x, m, d: nat m)
{
      for(;;) if (x < d) break else { x = x – d; m = m + 1; d = d + 2 }
}
```

На третьем этапе проведем подстановку тела определения sq3 на место вызова:

```
isqrt(nat x : nat m)
{     m = 0; nat d = 1;
      for(;;) if (x < d) break else { x = x – d; m = m + 1; d = d + 2 }
}
```

Наконец, реализуем вынесение оператора x = x – d перед условным оператором:

```
isqrt(nat x : nat m)
{     m = 0; nat d = 1;
      for(;;) { x = x – d; if (x < 0) break;  m = m + 1; d = d + 2  }
}
```

2.5. Нахождение квадратного корня через двоичное разложение

Последовательная оптимизация простого переборного алгоритма sq0 привела к алгоритму sq3. Все рассмотренные версии этого алгоритма имеют серьезный недостаток: число шагов алгоритма равно значению квадратного корня. Существуют алгоритмы, в которых число шагов имеет логарифмическую оценку. Один из них базируется на двоичном разложении результата, и на каждом шаге находит очередную двоичную цифру квадратного корня. Предположим, что аргумент n ограничен значением 2^{2*p}. Представим исходную задачу спецификацией:

```
isqrt(nat n, p: nat s)
pre p > 0 & n < 2^(2*p)
post s^2 <= n < (s + 1)^2;
```

Результат s будет иметь не более p двоичных цифр. Значение старшей цифры равно 0, если n < 2^(2*(p-1)) и 1 в противном случае. Допустим, мы нашли для результата s очередное приближение q в виде старших двоичных цифр с номерами от p до k. Тогда должно выполняться условие q^2 <= n < (q + 2^k)^2. Представим это спецификацией:

```
sq4(nat n, p, k, q : nat s)
pre p > 0 & n < 2^(2*p) & k <= p & q^2 <= n < (q + 2^k)^2
post s^2 <= n < (s + 1)^2;
```

Очевидно определение для isqrt.

```
isqrt(nat n, p: nat s)
pre p > 0 & n < 2^(2*p)
{ sq4(n, p, p, 0, n: s)}
post s^2 <= n < (s + 1)^2;
```

Спецификацию sq4 применим для вычисления цифры с номером k-1. Получим определение:

```
sq4(nat n, p, q, k: nat s)
pre p >= 0 & n < 2^(2*p) & k <= p & q^2 <= n < (q + 2^k)^2
{     if (k = 0) s = q
      else if ( n < (q + 2^(k-1))^2 )   sq4(n, p, k - 1, q : s)
           else  sq4(n, p, k - 1, q + 2^(k-1) : s)
}
post s^2 <= n < (s + 1)^2;
measure  k;
```

В соответствии с данным алгоритмом в зависимости от условия следующим приближением для s является q или q + 2^(k-1). Чтобы упростить вычисление условия n < (q + 2^(k-1))^2, реализуется серия оптимизаций, аналогичная переходу от sq1 к sq3. Имеет место

$$(q + 2^{(k-1)})^2 = q^2 + 2^{((k-1)*2)} + q * 2^k .$$

Тогда условие n < (q + 2^(k-1))^2 эквивалентно:

n - q^2 < 2^((k-1)*2) + q * 2^k .

Обозначим y = n - q^2. Преобразуем определение sq4 к следующему виду:

```
sq4(nat n, p, k, q, y: nat s)
pre p >= 0 & n < 2^(2*p) & k <= p & q^2 <= n < (q + 2^k)^2 & y = n - q^2
{   if (k = 0) s = q
    else  {   nat t = 2^((k-1)*2) + q * 2^k;
              if (y < t)  sq4(n, p, k - 1, q , y: s)
              else  sq4(n, p, k - 1, q or 2^(k-1), y – t : s)
    }
}
post s^2 <= n < (s + 1)^2;
measure  k;
```

Кроме того, мы заменили q + 2^(k-1) эквивалентным q **or** 2^(k-1), вычисляемым быстрее, где **or** – побитовая операция над натуральными.

Построим императивную программу для данной предикатной программы. На первом этапе трансформации приведенной предикатной программы в эффективную императивную реализуется следующие склеивания переменных:

sq4: n <- y; s <- q; k <- p .

В результате склеивания получим:

```
isqrt(nat n, p: nat s)
{ sq4(n, p, p, 0, n: s)}

sq4(nat n, k, k, s, n: nat s)
{   if (k = 0) s = s
    else  {   nat t = 2^((k-1)*2) + s * 2^k;
              if (n < t)  sq4(n, k, k - 1, s , n: s)
              else  sq4(n, k, k - 1, s or 2^(k-1), n – t : s)
    }
}
```

На втором этапе реализуется замена хвостовой рекурсии циклами. Результатом проведения двух этапов трансформации является следующая программа:

```
isqrt(nat n, p: nat s)
{ sq4(n, p, p, 0, n: s) }
```

```
sq4(nat n, k, k, s, n: nat s)
{  for (;;) {
        if (k = 0) break;
        nat t = 2^((k-1)*2) + s * 2^k;
        if (n < t)      k = k - 1
        else { n = n – t; s = s or 2^(k-1); k = k – 1 }
   }
}
```

На третьем этапе проведем подстановку тела определения sq4 на место вызова и упрощения. Получим программу:

```
nat n, p;
nat  s = 0;
for (nat k = p; k = 0; k = k - 1) {
        nat t = 2^((k-1)*2) + s * 2^k;
        if (n >=t) { n = n – t; s = s or 2^(k-1); }
}
```

Обозначим:

```
sq(k) = { t = 2^((k-1)*2) + s * 2^k; if (n >=t) { n = n – t; s = s or 2^(k-1); } }
```

Проведем развертку цикла. Получим

```
nat n, p;
nat  s = 0;
sq(p); sq(p – 1); …; sq(2); sq(1)
```

Проведем подстановку тела для sq(p) и sq(1). Получим итоговую программу.

```
sq(k) = { t = 2^((k-1)*2) + s * 2^k; if (n >=t) { n = n – t; s = s or 2^(k-1); } }
nat n, p;
nat  s = 0;
if (n >= 2^((p-1)*2)) { n = n – 2^((p-1)*2); s = 2^(p-1); }
sq(p – 1); …; sq(2);
if (n >= s * 2)  s = s or 1
```

По сравнению с программой, представленной на сайте

http://www.finesse.demon.co.uk/steven/sqrt.html,

параметр k смещен на единицу. Однако поскольку тело sq(k) подставляется открыто на место вызовов sq(p – 1), ..., sq(2), то обе программы идентичны.

На примере данной задачи видно, что проблема автоматизированной трансформации предикатной программы в эффективную императивную программу далеко не проста.

2.6. Верификация программы sq4

Объектом верификации является программа sq4:

```
isqrt(nat n, p: nat s)
pre p > 0 & n < 2^(2*p)
{ sq4(n, p, p, 0, n: s)}
post s^2 <= n < (s + 1)^2;

sq4(nat n, p, k, q, y: nat s)
pre p >= 0 & n < 2^(2*p) & k <= p & q^2 <= n < (q + 2^k)^2 & y = n - q^2
{     if (k = 0) s = q
      else        {  nat t = 2^((k-1)*2) + q * 2^k;
                  if (y < t)   sq4(n, p, k - 1, q , y: s)
                  else  sq4(n, p, k - 1, q or 2^(k-1), y – t : s)
      }
}
post s^2 <= n < (s + 1)^2;
measure  k;
```

Предложение **measure** k задает функцию меры m(k) = k. Мера задается на части аргументов определения предиката. Должно выполняться требование: мера для аргументов рекурсивного вызова должна быть строго меньше меры для аргументов определения. Мера применяется для одного из обобщений метода доказательства по индукции. Мера используется для генерации условий корректности для рекурсивных вызовов; см. правило FB3 в приложении 1.

Представим начальную часть теории для определения предиката isqrt.

```
isqrt : THEORY
BEGIN
      IMPORTING sq4
      n, p, s: VAR nat
      P_isqrt(n, p): bool = n < 2^(2*p)
      Q_isqrt(n, s): bool = s^2 <= n & n < (s + 1)^2
      satisfy_spec: LEMMA
            P_isqrt(n, p) IMPLIES EXISTS s: Q_isqrt(n, s)
```

Здесь представлены функции для предусловия и постусловия и лемма, определяющая тотальность (реализуемость) спецификации. Язык спецификаций является достаточно богатым и позволяет легко представить (сконвертировать) любые типы и выражения языка P.

В соответствии с формулой (1) целью является доказательство истинности правила:

$$P\_isqrt(n, p) \;\&\; Q\_isqrt(n, s) \vdash LS(sq4(n, p, p, 0, n: s))$$

В соответствии с правилом FB1 генерируется лемма:

```
FB1: LEMMA
   P_isqrt(n, p) & Q_isqrt(n, s) IMPLIES  P_sq4(n, p, p, 0, n) & Q_sq4(n, s)
```

Правило FB1 и все другие используемые далее правила описаны в приложении 1. Приведенная лемма завершает теорию isqrt. Представим начальную часть теории sq4.

```
sq4 : THEORY
BEGIN
   n, p, s, q, k, y: VAR nat
   P_sq4(n, p, k, q, y): bool =
         p >= 0 & n < 2^(2*p) & k <= p & q^2 <= n  &
               n < (q + 2^k)^2 & y = n - q^2
   Q_sq4(n, s): bool = s^2 <= n & n < (s + 1)^2
   satisfy_spec: LEMMA
      P_sq4(n, p, k, q, y) IMPLIES EXISTS s: Q_sq4(n, s)
   m(k): nat = k
```

Целью является доказательство истинности правила:

```
FR1: Induct(k) & P_sq4(n, p, k, q, y) & Q_sq4(n, s) ├  LS(<тело sq4>).
```

В соответствии с правилом FC1 для первой альтернативы условного оператора генерируется лемма:

```
FC1: LEMMA
            P_sq4(n, p, k, q, y) & Q_sq4(n, s) & k = 0 IMPLIES s = q.
```

В соответствии с правилом FC2 для второй альтернативы условного оператора генерируется цель:

```
FC2: Induct(k) & P_sq4(n, p, k, q, y) & Q_sq4(n, s) & NOT k = 0 ├
      LS(<вторая альтернатива условного> ).
```

По правилу FS5 для оператора суперпозиции генерируется лемма:

```
FS5: LEMMA
      P_sq4(n, p, k, q, y) & Q_sq4(n, s) & NOT k = 0 IMPLIES k – 1 >= 0.
```

По правилу FS6 для оператора суперпозиции генерируется цель:

```
FS6: Induct(k) & P_sq4(n, p, k, q, y) & Q_sq4(n, s) & NOT k = 0 &
                                          t = 2^((k-1)*2) + q * 2^k ├
  LS(if (y < t) sq4(n, p, k - 1, q , y: s) else  sq4(n, p, k - 1, q + 2^(k-1), y – t : s)).
```

В соответствии с правилом FC1 для первой альтернативы условного оператора генерируется цель:

```
FC1: Induct(k) & P_sq4(n, p, k, q, y) & Q_sq4(n, s) & NOT k = 0 &
      t = 2^((k-1)*2) + q * 2^k &  y < t ├ LS(sq4(n, p, k - 1, q , y: s)).
```

По правилу FB3 для рекурсивного вызова sq4(n, p, k - 1, q , y: s) генерируется лемма:

```
FB3: LEMMA
    P_sq4(n, p, k, q, y) & Q_sq4(n, s) & NOT k = 0 &
        t = 2^((k-1)*2) + q * 2^k & y < t
            IMPLIES  m(k - 1) < m(k) & P_sq4(n, p, k - 1, q, y) & Q_sq4(n, s).
```

В соответствии с правилом FC2 для второй альтернативы условного оператора генерируется цель:

```
FC2: Induct(k) & P_sq4(n, p, k, q, y) & Q_sq4(n, s) & NOT k = 0 &
        t = 2^((k-1)*2) + q * 2^k & & NOT y < t ├
            LS(sq4(n, p, k - 1, q + 2^(k-1), y – t : s)).
```

По правилу FB3 для рекурсивного вызова sq4(n, p, k - 1, q + 2^(k-1), y – t : s) генерируется лемма:

```
FB3.1: LEMMA
    P_sq4(n, p, k, q, y) & Q_sq4(n, s) & NOT k = 0 & t = 2^((k-1)*2) + q * 2^k &
        NOT y < t IMPLIES
            m(k - 1) < m(k) &
              P_sq4(n, p, k - 1, bv2nat(nat2bv(q) OR nat2bv(2^(k - 1))), y – t) &
                Q_sq4(n, s) .
```

На этом завершается генерация формул корректности теории sq4.

Для доказательства леммы FF3.1 сначала требуется доказать эквивалентность выражений q **or** 2^(k-1) и q + 2^(k-1), для чего необходимо будет доказать лемму:

```
or_eq_plus: LEMMA
    k > 0 & k < p & mod(q, 2^k) = 0 IMPLIES
                bv2nat(nat2bv(q) OR nat2bv(2^(k - 1))) = q + 2^(k - 1) .
```

Условие mod(q, 2^k) = 0 абсолютно необходимо для доказательства леммы, поэтому требуется включить его в состав предусловия P_sq4. Как следствие, будет необходимо доказывать истинность этого условия для второго вызова sq4, что реализуется с использованием дополнительной леммы:

```
mod0_2k: LEMMA
        k > 0 & mod(q, 2^k) = 0 IMPLIES  mod(2^(k - 1) + q, 2^(k - 1)) = 0.
```

Доказательство леммы or_eq_plus оказалось на порядок сложнее доказательства всех остальных лемм. Ее доказательство использовало следующие вспомогательные леммы, включенные в теорию sq4:

```
fill_eq_bv0: LEMMA FORALL (k:posnat): fill[k](FALSE) = nat2bv[k](0)
or_symms: LEMMA FORALL (k:posnat):
                                FORALL (A, B: bvec[k]): (A OR B)  = (B OR A)
mod0_2k_1: LEMMA  k > 0 & mod(q, 2^k) = 0 IMPLIES  mod(q, 2 ^ (k - 1)) = 0
mult2_ge0: LEMMA   d > 0 & q = d * x IMPLIES x >= 0
nat_div_ex: LEMMA   d > 0 & mod(q, d) = 0 IMPLIES EXISTS x: q = d * x
```

```
le_exp2p_k: LEMMA  k < p & q < exp2(p) &  q = x * exp2(k)  IMPLIES
                              x < exp2(p - k)
le_exp2p: LEMMA  p >= 0 & n < 2^(2*p) & q^2 <= n  IMPLIES q < exp2(p + 1)

x: VAR nat
bv2n_or_d: LEMMA
  k > 0 & d > 0  & x < exp2(d) &
  (nat2bv[d](div(x * exp2(k), exp2(k))) OR nat2bv[d](div(exp2((k - 1)), exp2(k)))) =
                                          nat2bv[d](x)
              IMPLIES
  bv2nat[d]((nat2bv[d](div(x * exp2(k), exp2(k))) OR
                    nat2bv[d](div(exp2((k - 1)), exp2(k))))  =   x

or_eq_bvd: LEMMA
    k > 0 & d > 0 & div(exp2(k - 1), exp2(k)) = 0 & div(x * exp2(k) , exp2(k)) = x &
          div(x * exp2(k), exp2(k)) < exp2(d)
                IMPLIES
    (nat2bv[d](div(x * exp2(k), exp2(k))) OR nat2bv[d](div(exp2((k - 1)), exp2(k))))
          = ( nat2bv[d](x) OR nat2bv[d](0) ) .
```

При доказательстве or_eq_plus использовались также леммы из девяти разных библиотек, поставляемых с PVS. Последние две леммы появились при попытке обойти ошибку системы PVS.

Собственная библиотека fl.pvs, включающая пять лемм, использовалась при доказательстве сгенерированных лемм.

Вот итоговая выдача о результатах доказательства основных теорий:

```
Proof summary for theory sq4
  P_sq4_TCC1.....proved - complete   [shostak]( 0.15 s)
  P_sq4_TCC2.....proved - complete   [shostak]( 0.16 s)
  Q_sq4_TCC1.....proved - complete   [shostak]( 0.11 s)
  Q_sq4_TCC2.....proved - complete   [shostak]( 0.11 s)
  satisfy_spec.....proved - complete   [shostak]( 5.54 s)
  FC1.....proved - complete   [shostak]( 4.01 s)
  FS5.....proved - complete   [shostak]( 4.56 s)
  mod0_2k_1_TCC1.....proved - complete   [shostak]( 1.04 s)
  mod0_2k_1..... .proved - complete   [shostak]( 2.33 s)
  FF3_TCC1..... ...proved - complete   [shostak]( 3.27 s)
  FF3..... ...proved - complete   [shostak]( 7.34 s)
  fill_eq_bv0_TCC1..... ..proved - complete   [shostak]( 0.20 s)
  fill_eq_bv0.....proved - complete   [shostak]( 0.43 s)
  or_symms.....proved - complete   [shostak]( 0.35 s)
```

or_eq_ex.....proved - complete [shostak](0.15 s)
bv2n_or_d_TCC1.....proved - complete [shostak](1.39 s)
bv2n_or_d_TCC2.....proved - complete [shostak](0.40 s)
bv2n_or_d_TCC3.....proved - complete [shostak](7.67 s)
bv2n_or_d.....proved - complete [shostak](0.25 s)
or_eq_bvd_TCC1.....proved - complete [shostak](0.30 s)
or_eq_bvd_TCC2.....proved - complete [shostak](2.23 s)
or_eq_bvd_TCC3.....proved - complete [shostak](1.53 s)
or_eq_bvd_TCC4.....proved - complete [shostak](2.16 s)
or_eq_bvd.....proved - complete [shostak](0.22 s)
mult2_ge0.....proved - complete [shostak](1.12 s)
nat_div_ex_TCC1.....proved - complete [shostak](0.21 s)
nat_div_ex.....proved - complete [shostak](1.94 s)
le_exp2p_k_TCC1.....proved - complete [shostak](0.77 s)
le_exp2p_k.....proved - complete [shostak](1.08 s)
lt_trans.....proved - complete [shostak](0.32 s)
le_exp2p_TCC1.....proved - complete [shostak](0.15 s)
le_exp2p.....proved - complete [shostak](2.02 s)
or_eq_plus_TCC1.....proved - complete [shostak](5.14 s)
or_eq_plus.....unfinished [shostak](29.92 s)
mod0_2k_TCC1.....proved - complete [shostak](1.17 s)
mod0_2k.....proved - complete [shostak](3.58 s)
FF3_1_TCC1.....proved - complete [shostak](3.28 s)
FF3_1_TCC2.....proved - complete [shostak](6.98 s)
FF3_1_TCC3.....proved - complete [shostak](1.88 s)
FF3_1_TCC4.....proved - complete [shostak](4.57 s)
FF3_1.....proved - incomplete [shostak](15.20 s)
Theory totals: 41 formulas, 41 attempted, 40 succeeded (125.23 s)

Proof summary for theory fl
uniq_flp.....proved - complete [shostak](0.64 s)
floor_int.....proved - complete [shostak](0.12 s)
sq_2p_TCC1.....proved - complete [shostak](0.10 s)
sq_2p.....proved - complete [shostak](0.10 s)
lt_m.....proved - complete [shostak](0.64 s)
lt_not.....proved - complete [shostak](0.29 s)
Theory totals: 6 formulas, 6 attempted, 6 succeeded (1.89 s)

```
Proof summary for theory isqrt
 Q_isqrt_TCC1.....proved - complete   [shostak](0.11 s)
 Q_isqrt_TCC2.....proved - complete   [shostak](0.11 s)
 satisfy_spec.....proved - complete   [shostak](3.09 s)
 FB1.....proved - complete   [shostak](2.91 s)
Theory totals: 4 formulas, 4 attempted, 4 succeeded (6.22 s)

Grand Totals: 51 proofs, 51 attempted, 50 succeeded (133.34 s)
```

В таблице леммы с именем ***_TCC генерируются системой PVS для доказательства условий совместимости типов выражений, используемых в основных леммах.

Из всех перечисленных лемм недоказанной осталась лемма or_eq_plus. Ее доказательство завершилось секвентом:

```
or_eq_plus.1.1.1.1.1.1.1.1.1.1.1.1.1.1 :

[-1]  bv2nat[p - k]
  ((nat2bv[p - k](div(x * exp2(k), exp2(k))) OR nat2bv[p - k]
                                                 (div(exp2((k - 1)), exp2(k)))))
            = x
[-2]  k > 0
[-3]  k < p
[-4]  q < exp2(p)
|-------
[1]   bv2nat[p - k]
     (( nat2bv[p - k](div(x * exp2(k), exp2(k))) OR
        nat2bv[p - k](div(exp2((k - 1)), exp2(k)))))
              = x

Rule?
```

Здесь формулы [–1] и [1] тождественны, а значит, лемма доказана, хотя PVS не способна распознать такой простой факт, что свидетельствует о явной ошибке PVS.

3. АЛГОРИТМ ВЫЧИСЛЕНИЯ ЦЕЛОЙ ЧАСТИ ЧИСЛА

Хорошо известна функция y = floor(x), вычисляющая наименьшее целое, не превосходящее вещественного аргумента x. Определим предикат:

```
formula flp(real x, int  i) = i <= x & x < i + 1;
```

Спецификация функции floor представляется определением:

```
floor(real x: int y) post  flp(x, y);
```

3.1. Постановка задачи

Нам требуется построить быстрый алгоритм вычисления floor для вещественных чисел в плавающем формате, определяемом в языке C++ типами **float**, **double** и **quad**. В алгоритме плавающие числа представлены в бинарном формате, определяемом стандартом IEEE 754. Использовалась последняя версия IEEE 754-2008 этого стандарта. Плавающее число представляется тройкой (S, E, T), где $S = \{0, 1\}$ – знак числа, E – сдвинутая экспонента, а T – мантисса в нормализованном представлении без старшего разряда (равного 1). В соответствии со стандартом, значение плавающего числа определяется по формуле:

$$y = (-1)^S * 2^{E-bias} * (1 + 2^{1-p} * T).$$

Здесь bias – сдвиг экспоненты, p – число битов в представлении мантиссы, $1 \le E \le 2^w - 2$, w – число битов в представлении экспоненты. Значение $E = 0$ предназначено для кодирования нулей и ненормализованных малых значений; случай $E = 2^w - 1$ соответствует бесконечности (положительной или отрицательной в зависимости от знака S) и числу NaN (not a number). Для формата 32 (**float**) значения bias = 127, w = 8 и p = 24. Для формата 64 (**double**) значения bias = 1023, w = 11 и p = 53.

Значение функции floor для положительной и отрицательной бесконечностей и числа NaN не определено. Для $E = 0$ значением функции floor является 0 при $S = 0$ и -1 при $S = 1$. Особым является случай отрицательного нуля. Если трактовать его как сверхмалое отрицательное значение, не представимое в формате ненормализованных малых значений, то значением floor должно быть –1. Однако это никак не следует из стандарта: при отрицательном переполнении (underflow) отрицательного результата нет явного требования представлять этот результат отрицательным нулем. Впрочем, в стандарте декларирован режим округления roundTowardNegative, при котором возможно получение отрицательных нулей в операциях типа $x - x$. В этом случае решение floor(-0) = –1 может показаться сомнительным.

3.2. Спецификация

В нашем алгоритме ограничимся случаем нормализованного числа при $0 < E < 2^w - 1$. Однако мы не выставляем этого ограничения в предусловии. Алгоритм определен также для $E = 0$ и $E = 2^w - 1$, однако в соответствии с бинарным представлением плавающих чисел не может быть там применен.

Определим константы и типы:

```
const nat bias;
const subtype (nat i: i > 0) p;
const subtype (nat i: i >= 2) w;
const nat m = p + w; % число битов в представлении числа y.
```

Здесь m определяет разрядность числа, т.е. число битов в представлении плавающего числа. Значения bias, p и w представлены неинтерпретированными константами. Однако ограничения p > 0 и w >= 2 по существу используются при доказательстве корректности программы.

```
type bit = {nat a | a = 0 or a = 1};
type nate = {nat e |  e < bias + m - 2};
type natp = {nat i | i < 2 ^ (p - 1)}
type intm = {int x | - 2^(m-1) < x & x < 2^(m-1)};
```

Дадим определение спецификации:

```
formula val(bit S, nate E, natp T: real) =
                                (-1)^S * 2^(E - bias) * (1 + 2^(1 - p) * T);
floor(bit S, nate E, natp T: intm y) post  flp(val(S, E , T), y);
```

Функция val определяет вещественное значение плавающего числа в соответствии со стандартом. В спецификации предиката floor предусловие отсутствует, поскольку все ограничения на входные значения представлены описаниями типов bit, nate и natp. Тип natp определяет очевидное ограничение для мантиссы T, поскольку в соответствии со стандартом для T используется p – 1 битов. Тип intm определяет естественное требование: результат функции floor должен помещаться в m битов. Тип nate вводит ограничение для значений экспоненты E. Далее будет автоматически доказано, что для указанных параметров S, E и T значение результата y будет находиться в пределах типа intm.

3.3. Предикатная программа

Преобразуем формулу для функции val следующим образом:
val(S, E, T) = (-1)^S * 2^(E – bias – p + 1) * (2^(p-1) + T) = (-1)^S * 2^d * z ,
где d = E – bias – p + 1, z = 2^(p-1) + T. При этом z >= 2^(p-1) и z < 2^p.

Определим функцию:

```
formula val1(bit S, nat d, z: real) = (-1)^S * 2^d * z;
```

Тогда вычисление функции floor можно свести к вычислению функции floor1 со спецификацией:

```
floor1(bit S, int d, nat z: intm y)
pre z >= 2^(p-1) & z < 2^p & d < w post  flp(val1(S, d , z), y);
```

Таким образом получаем следующее определение предиката floor:

```
floor(bit S, nat E, T: intm y) pre T < 2^(p-1) & E – bias < m – 2                (1)
{ floor1(S, E – bias – p + 1, 2^(p-1) + T: y) } post  flp(val(S, E , T), y);
```

Далее, требуемое значение функции floor получается соответствующим сдвигом натурального z. Полный алгоритм представляется следующим определением:

```
floor1(bit S, int d, nat z: intm y) pre z >= 2^(p-1) & z < 2^p & d < w          (2)
{ if (S = 0) { if (d >= 0) y = 2^d * z else y = div(z, 2^(-d)) }
   else   if (d >= 0) y =  - 2^d * z
             else if (mod(z, 2^(-d)) = 0) y = - div(z, 2^(-d))
                  else y = - div(z, 2^(-d)) – 1
} post  flp(val1(S, d , z), y);
```

Для отрицательных чисел алгоритм сложнее – фактически реализуется функция ceiling, реализующая приведение к целому в другую сторону.

Следует отметить следующие особенности алгоритма. Результат вычисления y = 2^d * z (и y = - 2^d * z) может оказаться неточным, если неточным было исходное представление числа в плавающем формате. Причем погрешность может оказаться большой – до 2^d. Из-за неточности исходного представления возможна погрешность результата y от -1 до +1 при вычислении div(z, 2^(-d)), а также ошибочность вычисления условия mod(z, 2^(-d)) = 0.

3.4. Оптимизация предикатной программы

Реализуется замена конструкций программы на более эффективные эквивалентные конструкции с использованием битовых операций. В определении (1) реализуется замена 2^(p-1) + T на 2^(p-1) **or** T:

```
floor(bit S, nat E, T: intm y) pre T < 2^(p-1) & E – bias < m – 2                (3)
{ floor1(S, E – bias – p + 1, 2^(p-1) or T: y) } post  flp(val(S, E , T), y);
```

Реализуется замена конструкций определения (2) на более эффективные эквивалентные конструкции с использованием битовых операций. Применяются следующие замены:

- z * 2^d на z << d
- div(z, 2^(-d)) на z >> (-d)
- mod(z, 2^(-d)) = 0 на z & (2^(-d) – 1) = 0.

Значение 2^(-d) – 1 состоит из (-d) единичных битов. Такое значение можно сгенерировать посредством сдвига: (2^p – 1) >> (p + d).

В соответствии со стандартом языка C++ результат операции сдвига не определен, если величина сдвига превышает длину сдвигаемого значения. Сдвиг вправо ограничен в предусловии значением w. Поэтому при замене

div(z, 2^(-d)) на z >> (-d) следует отдельно рассмотреть случай –d > p, при котором div(z, 2^(-d)) = 0. Проведя указанные замены получим следующую программу:

```
floor1(bit S, int d, nat z: intm y) pre z >= 2^(p-1) & z < 2^p & d < w
      (4)
{ if (S = 0) {if (d >= 0) y = z << d else if (–d > p) y = 0 else y = z >> (-d) }
   else  if (d >= 0) y =  - (z << d)
         else if (–d > p) y = - 1
              else if (z & ((2^p – 1) >> (p + d)) = 0) y = - (z >> (-d))
                   else y = - (z >> (-d)) – 1
} post  flp(val1(S, d , z), y);
```

Отметим, что полученная программа по-прежнему остается предикатной, и она могла бы быть объектом доказательства корректности вместо исходной предикатной программы (1) и (2).

3.5. Полная программа

В полной программе необходимо учесть случай E = 0 для нулей и ненормализованных малых значений. Для $E = 2^w – 1$ (случай бесконечностей и NaN) результат функции floor не определен.

```
  if (E = 0) { if (S= 0) y = 0 else y = -1 }
  else  floor(S, E, T: y)
```

Напомним, что решение floor(-0) = -1 не является очевидным. Подстановка определений предикатов floor и floor1 на место вызовов дает следующую программу:

```
 if (E = 0) { if (S= 0) y = 0 else y = -1 }                                  (5)
 else {  int d = E – bias – p + 1; nat z = 2^(p-1) or T;
         if (S = 0) { if (d >= 0) y = z << d
                      else if (–d > p) y = 0 else y = z >> (-d)
         } else if (d >= 0) y =  - (z << d)
                else if (–d > p) y = - 1
                     else if (z & ((2^p – 1) >> (p + d)) = 0) y = - (z >> (-d))
                          else y = - (z >> (-d)) – 1
 }
```

3.6. Верификация программы

Рассмотрим генерацию формул корректности программы (1, 2) и процесс автоматического доказательства истинности сгенерированных формул корректности. Представим теорию main, содержащую образ глобальных описаний программы.

```
main: THEORY
BEGIN
        IMPORTING fl
        bias: nat  %constants
        p: posnat
        w: {i: nat | i >= 2}
        m: nat = p + w
        nbit: TYPE = {n: nat | n = 0 OR n = 1}
        intm: TYPE = {x: int | - 2 ^ (m - 1) < x & x < 2 ^ (m - 1)}
        natp: TYPE = {i: nat | i < 2 ^ (p - 1)}
        nate: TYPE = {e: nat | e < bias + m - 2}
        T: VAR natp
        y: VAR intm
        S: VAR nbit
        E: VAR nate
        d, z: VAR nat
        val(S, E, T) = (-1)^S * 2^(E - bias) * (1 + 2^(1 - p) * T)
        val1(S, d, z) = (-1)^S * 2^d * z
END main
```

Представим начальную часть теории floor_val для определения предиката floor.

floor(bit S, **nat** E, T**:** intm y) **pre** T < 2^(p-1) & E – bias < m – 2 (1)
{ floor1(S, E – bias – p + 1, 2^(p-1) + T: y) } **post** flp(val(S, E , T), y);

```
floor_val: THEORY
BEGIN
        IMPORTING main, floor1
        T: VAR natp
        y: VAR intm
        S: VAR nbit
        E: VAR nate
        P_floor(E, T): bool = T < 2^(p-1) & E – bias < m – 2
        Q_floor(S, E, T, y): bool = flp(val(S, E , T), y)
        satisfy_spec: LEMMA P_floor(E, T) IMPLIES EXISTS y: Q_floor(S, E, T, y)
```

Целью является доказательство истинности правила:

P_floor(E, T) & Q_floor(S, E, T, y) ├ LS(floor1(S, E – bias – p + 1, 2^(p-1) + T))

В соответствии с правилом FB1 генерируется лемма:

```
FB1: LEMMA
                P_floor(E, T) & Q_floor(S, E, T, y) IMPLIES
                P_floor1(E – bias – p + 1, 2^(p-1) + T) &
                Q_floor1(S, E – bias – p + 1, 2^(p-1) + T, y) .
```

Представим начальную часть теории floor1.

```
floor1: THEORY
BEGIN
        IMPORTING main
        y: VAR intm
        E, T, d, z: VAR nat
        S: VAR nbit
        P_floor1(d, z): bool = z >= 2^(p-1) & z < 2^p & d < w
        Q_floor1(S, d, z, y): bool = flp(val1(S, d , z), y)
        satisfy_spec: LEMMA P_floor1(d, z) IMPLIES EXISTS r: Q_floor1(S, d, z, y)
```

Целью является доказательство истинности правила:

FB1: P_floor1(d, z) & Q_floor1(S, d, z, y) ├ LS(<тело floor1>).

В соответствии с правилом FC1 для первой альтернативы условного оператора генерируется цель:

FC1: P_floor1(d, z) & Q_floor1(S, d, z, y) & S = 0 ├ LS(<первый условный >).

В соответствии с правилом FC1 для первой альтернативы первого условного оператора генерируется лемма:

```
FC1: LEMMA
      P_floor1(d, z) & Q_floor1(S, d, z, y) & S = 0 & & d > 0 IMPLIES y = 2^d * z .
```

В соответствии с правилом FC2 для второй альтернативы первого условного оператора генерируется лемма:

```
FC2: LEMMA
      P_floor1(d, z) & Q_floor1(S, d, z, y) & S = 0 & & NOT d > 0 IMPLIES
                                                        y = div(z, 2^(-d)) .
```

В соответствии с правилом FC2 для второй альтернативы условного оператора генерируется цель:

FC2: P_floor1(d, z) & Q_floor1(S, d, z, y) & NOT S = 0 ├ LS(<второй условный >).

В соответствии с правилом FC1 для первой альтернативы второго условного оператора генерируется лемма:

```
FC11: LEMMA
  P_floor1(d, z) & Q_floor1(S, d, z, y) & NOT S = 0 & d > 0 IMPLIES y =  - 2^d * z.
```

В соответствии с правилом FC2 для второй альтернативы второго условного оператора генерируется цель:

```
FC2: P_floor1(d, z) & Q_floor1(S, d, z, y) & NOT S = 0 & NOT d > 0  ├
                                                    LS(<третий условный >).
```

В соответствии с правилом FC1 для первой альтернативы третьего условного оператора генерируется лемма:

```
FC12: LEMMA
      P_floor1(d, z) & Q_floor1(S, d, z, y) & NOT S = 0 & NOT d > 0 &
            mod(z, 2^(-d)) = 0 IMPLIES
                                  y = - div(z, 2^(-d)) .
```

В соответствии с правилом FC2 для второй альтернативы третьего условного оператора генерируется лемма:

```
FC21: LEMMA
    P_floor1(d, z) & Q_floor1(S, d, z, y) & NOT S = 0 & NOT d > 0 &
          NOT mod(z, 2^(-d)) = 0 IMPLIES
                                        y = - div(z, 2^(-d)) – 1.
```

Автоматическое доказательство сгенерированных лемм оказалось относительно несложным. Доказательство последней леммы потребовало заметных усилий, примерно равных доказательству всех предыдущих лемм. Однако весьма серьезную трудность вызвало доказательство принадлежности переменной y типу intm, т.е. того, что результат функции floor помещается в m битов. Потребовалось доказать 12 промежуточных лемм. В итоге теория main приняла следующий вид:

```
main: THEORY
BEGIN
  IMPORTING fl
  bias: nat  %constants
  p: posnat
  w: {i: nat | i >= 2}
  m: nat = p + w
  nbit: TYPE = {n: nat | n = 0 OR n = 1}
  intm: TYPE = {x: int | - 2 ^ (m - 1) < x & x < 2 ^ (m - 1)}
  natp: TYPE = {i: nat | i < 2 ^ (p - 1)}
  nate: TYPE = {e: nat | e < bias + m - 2}
  T: VAR natp
  y: VAR intm
  S: VAR nbit
  E: VAR nate
  me(E, T): real = 2 ^ (E - bias) * (1 + 2 ^ (1 - p) * T)

  me_eq:  LEMMA   me(E, T) = 2 ^ (E - bias - p + 1) * (2 ^ (p - 1) + T)
  le_pe:  LEMMA  E < bias - 2 + m IMPLIES  1 + E - bias < m - 1
  lepm:   LEMMA  2 ^ p < 2 ^ (m - 1) - 1
  gr2:    LEMMA  1 + 2 ^ (1 - p) * T < 2
  gr2p:   LEMMA  2 ^ (p - 1) + T < 2 ^ p
  me_gt0: LEMMA  me(E, T) > 0
  me_m:   LEMMA  me(E, T) < 2 ^ (m - 1) - 1

  val(S, E, T): real = (-1)^S * me(E, T)
  val_m:  LEMMA
            val(S, E, T) < 2 ^ (m - 1) - 1 & - 2 ^ (m - 1) + 1 < val(S, E, T)
```

```
  fval_m: LEMMA
        floor(val(S, E, T)) < 2 ^ (m - 1) & - 2 ^ (m - 1) < floor(val(S, E, T))

  flval(S, E, T): intm = floor(val(S, E, T))
  intd: TYPE = {j: int | j < w}
  natz: TYPE = {y: nat | y >= 2 ^ (p - 1) & y < 2 ^ p}
  z: VAR natz
  d: VAR intd
  me1(d, z): real = 2 ^ d * z
  me1_gt0: LEMMA  me1(d, z) > 0
  me1_m:   LEMMA  me1(d, z) < 2 ^ (m - 1) - 1
  val1(S, d, z): real = (-1)^S * me1(d, z)
  va1l_m:  LEMMA
        val1(S, d, z) < 2 ^ (m - 1) - 1 & - 2 ^ (m - 1) + 1 < val1(S, d, z)
  fval1_m: LEMMA
        floor(val1(S, d, z)) < 2 ^ (m - 1) & - 2 ^ (m - 1) < floor(val1(S, d, z))
  flval1(S, d, z): intm =  floor(val1(S, d, z))
END main
```

3.7. Сравнение с версией программы floor в файле fast_floor.cpp

В программе, находящейся в файле fast_floor.cpp (приложение 3) обнаружено две ошибки:

1. При E – bias >= m – 2 (или d >= w) сдвигаемое влево значение выходит за пределы m – 1 битов. В частности, при d = w старшая единица попадает в позицию знака в представлении целого числа, в результате чего результат floor становится отрицательным.

В реализации следовало бы запретить использовать функцию floor при d > 0, т.е. при E – bias – p + 1 > 0 из-за экспоненциально растущей погрешности.

2. Ненормализованное сверхмалое число обрабатывается так же, как нормализованное (с добавлением 1 в разряд p). Тем не менее это не меняет значения функции floor из-за малости числа. Однако при вычислении других функций результат был бы ошибочным.

Программа в файле fast_floor.cpp с точностью до обозначений и за исключением незначительных деталей подобна конечной программе (5) без первого оператора **if** (E = 0) { **if** (S=0) y = 0 **else** y = -1 }, который можно было бы опустить. Операторы **int** d = E – bias – p + 1; **nat** z = 2^(p-1) **or** T, присутствующие в (5), разнесены в fast_floor.cpp по разным ветвям, за счет чего программа в fast_floor.cpp, возможно, работает чуть быстрее.

4.АЛГОРИТМЫ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО ДВОИЧНОГО ЛОГАРИФМА

Рассмотрим алгоритмы вычисления целой части двоичного логарифма, т.е. функции y = ilog2(x) = floor($\log_2$(x)), где floor – функция вычисления целой части вещественного аргумента. Функцию ilog2 можно представить следующей спецификацией:

```
log2(real x: real y) pre x >0 post y = ln(x) / ln(2);
ilog2(nat a : nat r) pre a >= 1 post r = floor(log2(a));
```

4.1. Вычисление логарифма подсчетом числа сдвигов

Известно, что целочисленный двоичный логарифм – это номер старшей единицы в двоичном разложении числа a. Простейший способ вычисления этого номера заключается в определении числа сдвигов (на один бит) вправо числа a до полного его обнуления. Результат сдвига вправо числа a на один бит есть div(a, 2). Используя приведенные соображения нетрудно построить следующий алгоритм:

```
ilog2(nat a : nat r) pre a >= 1
{  if (a = 1) r = 0 else r = ilog2(div(a, 2)) +1
} post r = floor(log2(a));
```

Однако далеко не просто доказать корректность данного алгоритма.

Сначала построим программу с хвостовой рекурсией, используя следующее обобщение задачи:

```
ilg(nat a, m : nat r) pre a >= 1 post r = m + floor(log2(a));
```

Здесь параметр m используется в качестве накопителя числа сдвигов.

Программа представлена ниже в виде двух определений предикатов:

```
ilog2(nat a : nat r) pre a >= 1
{  ilg(a, 0: r)
} post r = floor(log2(a));

ilg(nat a, m : nat r) pre a >= 1
{  if (a = 1) r = m else ilg(div(a, 2), m +1: r)
} post r = m + floor(log2(a));
measure e(a) = a - 1
```

Построим соответствующую императивную программу для приведенной выше предикатной программы. На первом этапе трансформации предикатной программы реализуются следующие склеивания переменных.

```
ilg: r <- m;
```

В результате склеивания получим:

```
ilog2(nat a : nat r)
{  ilg(a, 0: r) }

ilg(nat a, r : nat r)
{  if (a = 1) r = r else ilg(div(a, 2), r +1: r) }
```

Далее реализуется замена хвостовой рекурсии циклами и подстановка определения ilg на место вызова. Результатом трансформации является следующая программа:

```
ilog2(nat a : nat r)
{  for (r = 0; a != 1; r = r +1) a = div(a, 2); }
```

Представим теорию для определения предиката ilog2.

```
ilog2: THEORY
BEGIN
        IMPORTING ilg, log
        a, r: VAR nat
        P_ilog2(a): bool = a >= 1
        Q_ilog2(a, r): bool = r = floor(log(2, a))
        satisfy_spec: LEMMA P_ilog2(a) IMPLIES EXISTS r: Q_ilog2(a, r)
```

Целью является доказательство истинности правила:

P_ilog2(a) & Q_ilog2(a, r) ├ LS(ilg(a, 0: r)) .

В соответствии с правилом FB1 генерируется лемма:

```
FB1: LEMMA
            P_ilog2(a) & Q_ilog2(a, r) IMPLIES  P_ilg(a) & Q_ilg(a, 0, r) .
```

Эта лемма завершает теорию ilog2. Представим начальную часть теории ilg.

```
ilg : THEORY
BEGIN
        a, m, r: VAR nat
        P_ilg(a): bool = a >= 1
        Q_ilg(a, m, r): bool = r = m + floor(log(2, a))
        satisfy_spec: LEMMA P_ilg(a) IMPLIES EXISTS r: Q_ilg(a, m, r)
        e(a): nat = a - 1
```

Целью является доказательство истинности правила:

```
FR1: Induct(a) & P_ilg(a) & Q_ilg(a, m, r) ├
          LS(if (a = 1) r = m else ilg(div(a, 2), m +1: r)).
```

В соответствии с правилом FC1 для первой альтернативы условного оператора генерируется лемма:

```
FC1: LEMMA
              P_ilg(a) & Q_ilg(a, m, r) & a = 1 IMPLIES r = m.
```

В соответствии с правилом FC2 для второй альтернативы условного оператора генерируется цель:

```
FC2: Induct(a) & P_ilg(a) & Q_ilg(a, m, r)& NOT a = 1 ├
        LS(ilg(div(a, 2), m +1: r)).
```

Для рекурсивного вызова ilg по правилу FB3 генерируется лемма:

```
FB3: LEMMA P_ilg(a) & Q_ilg(a, m, r) & NOT a = 1 IMPLIES
    e(div(a, 2)) < e(a) & P_ilg(div(a, 2)) & Q_ilg(div(a, 2), m +1, r) .
```

4.2. Алгоритм вычисления логарифма сдвигами по степеням двойки

Для ускорения вычисления целочисленного двоичного логарифма вместо сдвига по одному биту можно использовать более длинные сдвиги. Допустим, требуется вычислить ilog2(a) для a < 2^(2^p) и a >= 2^(2^(p-1)), где p > 1. Тогда в соответствии со свойствами логарифма

```
log2(a) = log2(a / 2^(2^(p-1))) + log2(2^(2^(p-1))) =
        = log2(a / 2^(2^(p-1))) + 2^(p-1).
```

Поскольку a / 2^(2^(p-1)) < 2^(2^(p-1)), то приведенное тождество определяет схему вычисления двоичного логарифма. Можно показать, что при вычислении ilog2 вещественное деление можно заменить целочисленным. Это дает возможность представить алгоритм в виде следующего определения предиката:

```
ilogB(nat a, p : nat r) pre a >= 1 & a < 2^(2^p)
{     if (p = 0) r = 0
      else if (a >= 2^(2^(p-1))) r = ilogB(div(a, 2^(2^(p-1))), p - 1) + 2^(p-1)
           else ilogB(a, p - 1: r)
} post r = floor(log2(a));
```

Рекурсия в данном определении не является хвостовой. Введем дополнительный параметр – накопитель m. Предикатная программа с хвостовой рекурсией представлена следующими двумя определениями:

```
ilogB(nat a, p : nat r) pre a >= 1 & a < 2^(2^p)
{     lgB(a, p, 0: r) } post r = floor(log2(a));

lgB(nat a, p, m : nat r) pre a >= 1 & a < 2^(2^p)
{     if (p = 0) r = m
      else if (a >= 2^(2^(p-1))) lgB(div(a, 2^(2^(p-1))), p - 1, m + 2^(p-1): r)
           else lgB(a, p - 1, m: r)
} post r = floor(log2(a)) + m;
```

Построим соответствующую императивную программу для приведенной выше предикатной программы. На первом этапе трансформации предикатной программы реализуются следующие склеивания переменных.

```
lgB: r <- m;
```

В результате склеивания получим

```
ilogB(nat a, p : nat r)
{     lgB(a, p, 0: r) }

lgB(nat a, p, r : nat r)
{     if p = 0 then r = r
      else if (a >= 2^(2^(p-1))) lgB(div(a, 2^(2^(p-1))), p - 1, r + 2^(p-1): r)
           else lgB(a, p - 1, r: r)
}
```

Далее реализуется замена хвостовой рекурсии циклами и подстановка определения lgB на место вызова. Результатом трансформации является следующая программа:

```
ilogB(nat a, p : nat r)
{     r = 0;
      for (; p != 0; p = p - 1)
            if (a >= 2^(2^(p-1))) {a = div(a, 2^(2^(p-1))); r = r + 2^(p-1)}
}
```

Для p = 5, что соответствует 32-битным натуральным числам, проведем развертку цикла. Получим программу:

```
r = 0;
if (a >= 2^16) {a = div(a, 2^16); r = r + 16}
if (a >= 2^8) {a = div(a, 2^8); r = r + 8}
if (a >= 2^4) {a = div(a,2^4); r = r + 4}
if (a >= 2^2) {a = div(a, 2^2); r = r + 2}
if (a >= 2^1) {a = div(a, 2^1); r = r + 1}
```

Наконец, используя операции сдвига, получим окончательную программу:

```
r = 0;
if (a >= 1<<16) {a = a >>16; r = r + 16}
if (a >= 1<< 8) {a = a >>8); r = r + 8}
if (a >= 1<< 4) {a = a >>4; r = r + 4}
if (a >= 1<< 2) {a = a >>2; r = r + 2}
if (a >= 1<< 1) { r = r + 1}
```

Вернемся к предикатной программе и рассмотрим ее верификацию относительно спецификаций.

```
ilogB(nat a, p : nat r) pre a >= 1 & a < 2^(2^p)
{     lgB(a, p, 0: r) } post r = floor(log(2, a));
```

```
lgB(nat a, p, m : nat r) pre a >= 1 & a < 2^(2^p)
{     if p = 0 then r = m
      else if a >= 2^(2^(p-1)) then lgB(div(a, 2^(2^(p-1))), p - 1, m + 2^(p-1): r)
           else lgB(a, p - 1, m: r)
} post r = floor(log(2, a)) + m;
measure e(p) = p
```

Представим теорию для определения предиката ilogB.

```
ilogB: THEORY
BEGIN
        IMPORTING lgb, ilog2
        a: VAR posnat
        r, p: VAR nat
        P_ilogB(a, p): bool = a < 2^(2^p)
        Q_ilogB(a, r): bool = r = ilog2(a))
        satisfy_spec: LEMMA P_ilogB(a, p) IMPLIES EXISTS r: Q_ilogB(a, r)
```

Целью является доказательство истинности правила:

P_ilogB(a, p) & Q_ilogB(a, r) ├ LS(lgB(a, p, 0: r)).

В соответствии с правилом FB1 генерируется лемма:

```
FB1: LEMMA
        P_ilogB(a, p) & Q_ilogB(a, r) IMPLIES  P_lgB(a, p) & Q_lgB(a, p, 0, r).
```

Эта лемма завершает теорию ilogB. Представим начальную часть теории lgB.

```
lgB : THEORY
BEGIN
        p, m, r: VAR nat
        a: VAR posnat
        P_lgB(a, p): bool = a < 2^(2^p)
        Q_lgB(a, m, r): bool = r = m + ilog2(a)
        satisfy_spec: LEMMA P_lgB(a, p) IMPLIES EXISTS r: Q_lgB(a, m, r)
        e(p): nat = p
```

Целью является доказательство истинности правила

FR1: Induct(e) & P_lgB(a, p) & Q_lgB(a, m, r) ├ LS(< тело lgB >).

В соответствии с правилом FC1 для первой альтернативы условного оператора генерируется лемма:

```
FC1: LEMMA
        P_lgB(a, p) & Q_lgB(a, m, r) & p = 0 IMPLIES r = m.
```

В соответствии с правилом FC2 для второй альтернативы условного оператора генерируется цель:

```
FC2: Induct(e) & P_lgB(a, p) & Q_lgB(a, m, r) & NOT p = 0 ├
        LS(<внутренний условный оператор>).
```

В соответствии с правилом FC1 для первой альтернативы условного оператора генерируется цель:

```
FC1: Induct(e) & P_lgB(a, p) & Q_lgB(a, m, r) & NOT p = 0 & a >= 2^(2^(p-1))
        ├ LS(lgB(div(a, 2^(2^(p-1))), p - 1, m + 2^(p-1): r)) .
```

Для первого рекурсивного вызова lgB по правилу FB3 генерируется лемма:

```
FB3: LEMMA P_lgB(a, p) & Q_lgB(a, m, r) & NOT p = 0 & a >= 2^(2^(p-1))
   IMPLIES e(p - 1) < e(p) & P_lgB(div(a, 2^(2^(p-1))), p - 1) &
                            Q_lgB(div(a, 2^(2^(p-1))), m + 2^(p-1), r) .
```

В соответствии с правилом FC2 для второй альтернативы условного оператора генерируется цель:

```
FC2: Induct(e) & P_lgB(a, p) & Q_lgB(a, m, r) &
                NOT p = 0 & NOT a >= 2^(2^(p-1)) ├ LS(lgB(a, p - 1, m: r)).
```

Для второго рекурсивного вызова lgB по правилу FB3 генерируется лемма:

```
FB3_1: LEMMA P_lgB(a, p) & Q_lgB(a, m, r) & NOT p = 0 & NOT a >= 2^(2^(p-1))
          IMPLIES e(p - 1) < e(p) & P_lgB(a, p - 1) & Q_lgB(a, m, r) .
```

Автоматическая верификация приведенных условий самих корректности не потребовала много усилий. Однако для их доказательства потребовалось построить теории log, log2, ilog2 и доказать 32 леммы в них. Вот эти теории:

```
log: THEORY
BEGIN
  p: VAR {y: nat | y > 1}
  IMPORTING lnexp@ln_exp
  x, y: VAR posreal
  n: VAR posnat
  i: VAR int
  log(p, x): real = ln(x) / ln(p)
  log_1:    LEMMA log(p, 1) = 0
  log_p:    LEMMA log(p, p) = 1
  log_mult: LEMMA log(p, x * y) = log(p, x) + log(p, y)
  log_div:  LEMMA log(p, x / y) = log(p, x) - log(p, y)
  log_expt: LEMMA log(p, x^i)  = i * log(p, x)
  log_eq_0: LEMMA log(p, x) = 0 IMPLIES x = 1
  log_ge_0: LEMMA x >= 1 IMPLIES log(p, x) >= 0
  log_gt_0: LEMMA x > 1 IMPLIES log(p, x) > 0
  log_strict_incr: LEMMA FORALL x, y : x < y IMPLIES log(p, x) < log(p, y)
  log_incr: LEMMA FORALL  x, y : x <= y IMPLIES log(p, x) <= log(p, y)
  z, u: VAR real
  exp(p, z): real = exp(z * ln(p))
```

```
  log_exp:  LEMMA log(p,exp(p, z)) = z
  exp_log:  LEMMA exp(p,log(p, x)) = x
  exp_strict_incr: LEMMA FORALL z, u : z < u IMPLIES exp(p, z) < exp(p, u)
  exp_incr: LEMMA FORALL z, u : z <= u IMPLIES exp(p, z) <= exp(p, u)
END log

log2: THEORY
BEGIN
  IMPORTING log
  x, y: VAR posreal
  i, n: VAR posnat
  log2(x): real = log(2, x)
  log2_1:    LEMMA log2(1) = 0
  log2_2:    LEMMA log2(2) = 1
  log2_mult: LEMMA log2(x * y) = log2(x) + log2(y)
  log2_div:  LEMMA log2(x / y) = log2(x) - log2(y)
  log2_expt: LEMMA log2(x^i)  = i * log2(x)
  log2_exp2: LEMMA log2(2^n) = n
  log2_div_eq: LEMMA log2(x) = log2(x / 2^n) + n
  log2_eq_0: LEMMA log2(x) = 0 IMPLIES x = 1
  log2_ge_0: LEMMA x >= 1 IMPLIES log2(x) >= 0
  log2_gt_0: LEMMA x > 1 IMPLIES log2(x) > 0
  log2_lt1:  LEMMA x < 2 IMPLIES log2(x) < 1
  log2_strict_incr: LEMMA FORALL x, y : x < y IMPLIES log2(x) < log2(y)
  log2_incr: LEMMA FORALL  x, y : x <= y IMPLIES log2(x) <= log2(y)
  z, u: VAR real
  k: VAR nat
  ex2(z): real = exp(2, z)
  log2_ex2:    LEMMA log2(ex2(z)) = z
  ex2_log2:    LEMMA ex2(log2(x)) = x
  ex2_exp2:    LEMMA ex2(k) = exp2(k)
  ex2_strict_incr: LEMMA FORALL z, u : z < u IMPLIES ex2(z) < ex2(u)
  ex2_incr: LEMMA FORALL z, u : z <= u IMPLIES ex2(z) <= ex2(u)
END log2

ilog2: THEORY
BEGIN
  IMPORTING log2, ints@div_nat, fl
  a, m: VAR posnat
  x: VAR posreal
  y: VAR { z: real | z >= 1}
  ilog2(y): nat = floor(log2(y))
```

```
  ilog2_1:     LEMMA ilog2(1) = 0
  ilog2_floor: LEMMA ilog2(y) = ilog2(floor(y))
  ilog2_ge0:   LEMMA ilog2(a) >= 0
  ilog2_next:  LEMMA x < 1 IMPLIES  ilog2(a) =  ilog2(a + x)
  ilog2_div:   LEMMA a >= 2^m IMPLIES  ilog2(div(a, 2^m)) =  ilog2(a / 2^m)
  ilog2_divm:  LEMMA a >= 2^m IMPLIES  ilog2(div(a, 2^m)) = ilog2(a) - m
  ilog2_div2:  LEMMA a >= 2 IMPLIES  ilog2(div(a, 2)) = ilog2(a) - 1
END ilog2
```

Ключевой и наиболее сложной для доказательства здесь была лемма ilog2_div.

4.3. Итоги

Программа, находящаяся в файле fast_log2.cpp (приложение 4), соответствует программе ilog2, представленной в разд. 4.1. Имеется различие: fast_log2(0) = 0, тогда как ilog2 не определена для 0. На других сайтах встречались программы, для которых ilog2(0) = 0 под режимом, включаемом по желанию пользователя.

Императивная программа, построенная для ilogB в разд. 4.2, должна быть значительно быстрее, чем fast_log2.cpp. На разных сайтах встречались другие алгоритмы для ilog2. Имеется алгоритм, представленный на сайте

`http://www.aggregate.org/MAGIC/`

Это более быстрый алгоритм, использующий подпрограмму подсчета числа единиц. Оба эти алгоритма не используют условных операторов и в сумме должны быть быстрее ilogB.

5. ЗАКЛЮЧЕНИЕ

Проведенный эксперимент подтверждает универсальность технологии предикатного программирования, позволяющей воспроизвести любую императивную программу в классе задач дискретной и вычислительной математики. Полученные программы для функций isqrt и floor на императивном расширении языка P близки к соответствующему исходному коду на C++; отличия имеются, но они не являются принципиальными. Полученная программа для функции ilog2 существенно быстрее исходной программы на C++.

Алгоритм генерации условий корректности не претерпел существенных изменений в ходе данного эксперимента. Проведены лишь уточнения для

генерации на PVS битовых операций над натуральными. Можно констатировать, что задача генерации условий корректности решена успешно. Генерируемые условия корректности декомпозированы достаточно хорошо. Узким местом является верификация условий корректности на PVS. Доказательство на PVS оказалось нетривиальным и трудоемким процессом. Трудности возникали не столько при доказательстве самих условий корректности, сколько при доказательстве используемых математических свойств.

В процессе верификации обнаружено 5 ошибок. Из них четыре – чисто технические ошибки в предикатных программах или при генерации вручную условий корректности. Принципиальной является ошибка, обнаруженная для функции floor: результат функции floor не помещается в отведенные для него m битов для достаточно большого плавающего числа такого, что E – bias >= m – 2 (или d >= w). Ситуация вроде бы очевидная, однако на практике многие ее не учитывают. Ошибка обнаружена в ходе доказательства условия корректности, сгенерированного PVS для функции flval1: требуется доказать принадлежность результата flval1 типу intm. Обнаруживается, что из-за ошибки доказать такое невозможно. На примере данной ошибки становится очевидным, что формальная верификация является наиболее мощным инструментом в обеспечении надежности программ.

В заключение приведем информацию о времени выполнения различных этапов работ по трем функциям:

Для функции isqrt:

- поиск по Интернету, разработка алгоритма по технологии предикатного программирования – 4 дня;
- верификация – 9 дней, из них 5 дней на доказательно леммы or_eq_plus;
- написание отчета – 3 дня.

Для функции floor:

- поиск по Интернету, изучение стандартов и разработка алгоритма – 5 дней;
- верификация – 8 дней, из них полтора дня на доказательно собственно условий корректности;
- написание отчета - 4 дня.

Для функций ilog2 и ilogB:

- - разработка алгоритмов ilog2 и ilogB: 2 дня;
- - верификация: 6 дней;
- - написание отчета: 2 дня.

Относительно большое время, затраченное на верификацию, отчасти объясняется тем, что автор еще не освоил систему PVS в достаточной степени.

СПИСОК ЛИТЕРАТУРЫ

1. PVS Specification and Verification System. . – SRI International. – http://pvs.csl.sri.com/doc/
2. Шелехов В.И. Исчисление вычислимых предикатов. – Новосибирск, 2007. – 24с. – (Препр. / ИСИ СО РАН; N 143).
3. Шелехов В.И. Модель корректности программ на языке исчисления вычислимых предикатов. – Новосибирск, 2007. – 50с. – (Препр. / ИСИ СО РАН; N 145).
4. Шелехов В.И. Язык предикатного программирования Р. – Новосибирск, 2002. – 40с. – (Препр. / ИСИ СО РАН; N 101).
5. Шелехов В.И. Введение в предикатное программирование. – Новосибирск, 2002. – 82с. – (Препр. / ИСИ СО РАН; N 100).

В данном приложении приведены правила, используемые при генерации условий корректности.

Теорема 2.1 тождества спецификации и программы. Рассмотрим программу со спецификацией в виде тройке Хоара:

$$\{P(x)\}\ S(x{:}\ y)\ \{Q(x, y)\} \tag{2.15}$$

Здесь x и y – разные наборы переменных, причем x может быть пустым. Допустим, оператор S является однозначным в области истинности предусловия P(x), а спецификация оператора S является реализуемой. Предположим, LS(S(x: y)) выводима из спецификации, т.е.:

$$P(x)\ \&\ Q(x, y) \Rightarrow LS(S(x{:}\ y)) \tag{2.16}$$

Тогда программа (2.15) является корректной.

Пусть имеется нерекурсивный вызов предиката A(x: y) со спецификацией:

$$\{P(x)\}\ A(x{:}\ y)\ \{Q(x, y)\} \tag{3}$$

Для нерекурсивного вызова предиката A(x: y) определим правило:

Правило FB1. $\quad R(x, y) \vdash P(x)\ \&\ Q(x, y)$

Лемма 15. Допустим, нерекурсивный вызов предиката A(x: y) является корректным, а его спецификация (3) – однозначна. Если истинно правило FB1, то истинна следующая формула:

$$R(x, y) \Rightarrow LS(A(x{:}\ y))$$

Для условного оператора **if** (C) A(x: y) **else** B(x: y) определим правила:

Правило FC1. $\quad R(x, y)\ \&\ C \vdash LS(A(x{:}\ y))$
Правило FC2. $\quad R(x, y)\ \&\ \neg C \vdash LS(B(x{:}\ y))$

Лемма 9. Если истинны правила FC1 и FC2, то истинна следующая формула:

$$R(x, y) \Rightarrow LS(\textbf{if}\ (C)\ A(x{:}\ y)\ \textbf{else}\ B(x{:}\ y))$$

Допустим A(x: z) – нерекурсивный вызов предиката со спецификацией:

$$\{P(x)\}\ A(x{:}\ z)\ \{Q(x, z)\} \tag{11}$$

Для оператора суперпозиции A(x: z); B(x, z: y) определим специализацию правил FS1 и FS2:

Правило FS5. R(x, y) ├ P(x)

Правило FS6. R(x, y) & Q(x, z) ├ LS(B(x, z: y))

Лемма 27. Допустим, нерекурсивный вызов предиката A(x: z) является корректным относительно спецификации (11). Если правила FS5 и FS6 истинны, то истинна следующая формула:

$$R(x, y) \Rightarrow LS(A(x: z); B(x, z: y))$$

Допустим, рекурсивное кольцо состоит из единственного определения предиката:

$$A(t, x: y) \equiv P(t, x) \{ K(t, x: y) \} Q(t, x, y) \qquad (5.20)$$

Формула корректности определения (5.20) для серии F принимает вид:

$$V(t) \equiv P(t, x) \& Q(t, x, y) \Rightarrow LS(K(t, x: y)) \qquad (2)$$

Правила корректности FR для (5.20) принимает вид:

Правило FR1. Induct(t) & P(t, x) & Q(t, x, y) ├ LS(K(t, x: y)).

Лемма 14. Допустим, спецификация (предусловие P(t, x) и постусловие Q(t, x, y)) является реализуемой, а оператор K(t, x: y) – однозначный. Если правило FR1 истинно, то определение (5.20) является корректным.

Пусть имеется рекурсивный вызов предиката A(u, x1: y1) в составе оператора K(t, x: y) в определении (5.20) предиката A. Для рекурсивного вызова A(u, x1: y1) определим правило:

Правило FB3. R(u, t, x1, y1) ├ Less(u, t) & P(u, x1) & Q(u, x1, y1)

Здесь предикат Less(u, t) обозначает отношение $u \sqsubset t$ или $m(u) < m(t)$, находящееся в составе предиката Induct(t).

Лемма 17. Если истинно правило FB3, то истинна следующая формула:

$$Induct(t) \& R(u, t, x1, y1) \Rightarrow LS(A(u, x1: y1))$$

Код функции isqrt из файла fast_sqrt.cpp

```
static const uint32_t small_limit = 17;
static const uint32_t small_sqrts[small_limit] =
//  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
{0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4};
#define sqrtBit(k) \
    t = s + (1UL << (k - 1)); t <<= k + 1; if (_n >= t) {_n -= t; s |= 1UL << k;}

uint32_t os::isqrt(uint32_t _n) {
    if (_n < small_limit)
        return small_sqrts[_n];
    uint32_t s = 0UL;
    if (_n >= 1UL<<30) {
        _n -= 1UL<<30;
        s = 1UL<<15;
    }
    uint32_t t;
    sqrtBit(14);
    sqrtBit(13);
    sqrtBit(12);
    sqrtBit(11);
    sqrtBit(10);
    sqrtBit(9);
    sqrtBit(8);
    sqrtBit(7);
    sqrtBit(6);
    sqrtBit(5);
    sqrtBit(4);
    sqrtBit(3);
    sqrtBit(2);
    sqrtBit(1);
    if (_n > s<<1)
        s |= 1UL;

    return s;
}
```

Код функции floor из файла fast_floor.cpp

```
int32_t os::fast_floor(float _f) {
    uint32_t dw = reinterpret_cast<uint32_t &> (_f);
    if (int32_t(dw) < 0) {
            //
            // For negative values.
            //
            dw &= 0x7FFFFFFF;
            if (dw == 0)
                    return 0;

            const int32_t sh = 23 + 127 - (dw >> 23);
            if (sh >= 24)
                    return -1;
            else if (sh < 0) {
                    // NOTE: precision is lost.
                    return -int32_t((0x00800000 | (dw & 0x007FFFFF)) << (-sh));
            } else {
                    if (dw & (0x007FFFFF >> (23 - sh)))
                            // NOTE: the number has fractional part.
                            return -int32_t((0x00800000 | (dw & 0x007FFFFF)) >> sh) - 1;
                    else
                            // NOTE: the number is whole.
                            return -int32_t((0x00800000 | (dw & 0x007FFFFF)) >> sh);
            }
    } else {
            //
            // For positive values.
            //
            if (dw == 0)
                    return 0;
            const int32_t sh = 23 + 127 - (dw >> 23);
            if (sh >= 24)
                    return 0;
            else if (sh < 0)
                    // NOTE: the precision is lost.
```

```
            return (0x00800000 | (dw & 0x007FFFFF)) << (-sh);
        else
            return (0x00800000 | (dw & 0x007FFFFF)) >> sh;
    }
}
```

Приложение 4

Код функции ilog2 из файла fast_log2.cpp

```
uint32_t os::fast_log2(uint32_t _dw) {
    uint32_t log2 = 0;
    while (_dw != 0) {
        ++log2;
        _dw >>= 1;
    }
    return log2;
}
```

В.И. Шелехов

РАЗРАБОТКА ЭФФЕКТИВНЫХ ПРОГРАММ СТАНДАРТНЫХ ФУНКЦИЙ FLOOR, ISQRT И ILOG2 ПО ТЕХНОЛОГИИ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Препринт
154

Рукопись поступила в редакцию 25.04.10
Редактор Т. М. Бульонкова
Рецензент И.С. Ануреев

Подписано в печать 10.06.10
Формат бумаги 60 × 84 1/16 Объем 2.45 уч.-изд.л., 2.7 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42