Реализация длинной арифметики в Рефале-05

Рефал-совещание МГТУ–ИПС, 19 декабря 2025 г.

Александр Коновалов

Общие соображения

Длинная арифметика в Рефале-5

Рефал-5 — язык для символьных преобразований, по мнению докладчика, задумывался не как промышленный язык, а как язык для исследовательских проектов вроде написания суперкомпиляторов. Поэтому некоторые вещи вроде файлового ввода-вывода или арифметики довольно аскетичны.

Данные в Рефале — цепочки из символов и правильно сбалансированных круглых (структурных скобок) — объектные выражения.

Во многих ранних реализациях Рефала присутствовали символы-числа ограниченной разрядности (до 2^N), которые понимались как «макроцифры»: функции арифметики оперировали цепочками из макроцифр, понимая их как цифры по основанию 2^N.

Рефал-5 следует данной традиции (как и ряду других традиций, вроде встроенной функции чтения перфокарты `Card` ☺).

Что такое Рефал-05?

> ***Рефал-05*** *— подмножество Рефала-5 и альтернативная реализация Рефала-5, компилирующаяся в C89.*

(Цитата из `README.md`)

Репозиторий проекта:

https://github.com/Mazdaywik/Refal-05

Цели разработки: совместимость с Рефалом-5, компиляция в язык Си89 и расширяемость, минималистичность, эффективность, самоприменение, практичность, прозрачность, педантичность, переносимость.

Арифметика в Рефале-05

Первоначально в Рефале-05 не поддерживалась длинная арифметика — аргументы арифметических функций должны состоять из одной макроцифры, результат может содержать две.

Но потом докладчику пришёл в голову интересный алгоритм преобразования систем счисления и докладчик не удержался от искушения добавить новую реализацию функции `Numb` в библиотеку Рефала-05.

Ну, раз уж одна добавлена, значит, надо и остальные добавлять.

Перевод систем счисления

Основной алгоритм

Рассмотрим функцию, которая принимает два числа, представленных в виде массивов цифр S (источник) и T (назначение). Их основания систем счисления Bs и Bt и размеры Ns и Nt, соответственно.

```
Convert(in S, in Bs, in Ns, out T, in Bt, in/out Nt)
    acc ← 0
    for i ← Ns - 1 downto 0:
        acc ← acc × Bs + S[i]
    i ← 0
    while acc ≠ 0:
        assert i < Nt
        T[i] ← acc % Bt
        acc ← acc / Bt
        i ← i + 1
    Nt ← i
```

Здесь аккумулятор рассматривается как чёрный ящик с операциями умножения, сложения, остатка и деления. Примерно так можно написать функцию на Python’е, выполняющую преобразования между двумя системами счисления.

Частный случай — одно из оснований нативное (1)

На практике обычно одно из оснований нативное — поддерживаемое на уровне языка или машины. Чаще всего, это преобразование между двоичным кодом и десятичной записью.

Тогда преобразование в одну из сторон будет тривиальным — аккумулятор уже имеет требуемый вид. Получаем два алгоритма:

```
ConvertFrom(in S, in Bs, in Ns): acc
    acc ← 0
    for i ← Ns - 1 downto 0:
        acc ← acc × Bs + S[i]

ConvertTo(in S, out T, in Bt, in/out Nt)
    while S ≠ 0:
        assert i < Nt
        T[i] ← S % Bt
        acc ← S / Bt
        i ← i + 1
    Nt ← i
```

Частный случай — одно из оснований нативное (2)

Если диапазон чисел небольшой или язык поддерживает неограниченный целый тип, то мы можем пользоваться доступным числовым типом как чёрным ящиком.

Иначе нам приходится применять к аккумулятору алгоритмы длинных вычислений.

Длинный алгоритм перевода в нативную систему

```
ConvertFrom(in S, in Bs, in Ns, out T, in/out Nt)
    T[0] ← 0, k ← 0
    for i ← Ns - 1 downto 0:
        j ← 0, carry ← S[i]
        for j ← 0 to k:
            (carry, T[j]) ← muladd(T[j], Bs, carry)
        if carry ≠ 0:
            k ← k + 1
            assert k < Nt
            T[j] ← carry
    Nt ← k
```

Здесь используется операция умножения с накоплением `muladd(a, b, c)`, вычисляющая `a × b + c` в виде старшей и младшей половины результата.

Длинный алгоритм перевода из нативной системы

```
ConvertTo(in S, in Ns, out T, in Bt, in/out Nt)
    C ← создать копию массива S, Nc ← Ns
    i ← 0
    while Nc > 0:
        carry ← 0
        for j ← Nc - 1 downto 0:
            (C[j], carry) ← divmod(carry, C[j], Bt)
        if C[Nc - 1] = 0:
            Nc ← Nc - 1
        assert i < Nt
        T[i] ← carry
        i ← i + 1
    Nt ← i
```

Здесь используется двухсловная операция деления с остатком `divmod(high, low, den)`, возвращающая пару из частного и остатка. Предполагается, что `high` $<$ `den`.

Сложность алгоритмов перевода (1)

Можно заметить, что сложность этих алгоритмов квадратичная: они оба требуют $O(\texttt{Ns} \cdot \texttt{Nt})$ операций, причём `Nt` понимается как окончательное значение. В частности, `muladd` и `divmod` будут выполняться точно $\texttt{Ns} \times \texttt{Nt}$ раз.

Примечание для математиков. *Длина числа в любой системе пропорциональна логарифму значения, поэтому оценку сложности можно описать как квадрат логарифма значения преобразуемого числа. Но нам сейчас интересна оценка сложности именно в том виде, как она записана.*

Сложность алгоритмов перевода (2)

Из полученной формулы следует, что чем больше основания систем счисления, тем меньше требуется операций.

Также заметим, что операции `muladd` и `divmod` работают с двойными словами, т.е. работать с длинным числом как массивом целых чисел максимального размера будет довольно затруднительно. Удобнее будет работать с массивом целых меньшего размера (`short` или `int`, если `sizeof(int) < sizeof(long)`).

Но работа с целыми меньшего размера число операций увеличит.

Частные случаи линейного преобразования (1)

Очевидно, что если одно из оснований систем счисления (исходное или целевое) будет степенью второго, то для преобразования достаточно сгруппировать или разгруппировать цифры в количестве, равном показателю степени.

Известный программистам пример: преобразование восьмеричных и шестнадцатеричных чисел в двоичную систему и обратно.

Ещё пример. Если нам дан массив из чисел по основанию `100`, а нам нужно перейти в систему счисления по `1000000`, то достаточно сгруппировать тройки и подсчитать `T[k] ← S[3k+2]×10000 + S[3k+1]×100 + S[3k]`.

Частные случаи линейного преобразования (2)

Также этот частный случай интересен тем, что допусает параллельное преобразование: группы можно вычислять полностью независимо.

Также, если величина группы меньше машинного слова, то работать с двойными словами не требуется.

Промежуточные следствия (1)

Используя правило группировки, можно заметно упростить или повысить эффективность преобразований систем счисления.

Действительно, работа с двойными словами в высокоуровневых языках зачастую недоступна (хотя процессор может поддерживать, например, команда умножения на Intel сохраняет двойной результат в паре `rdx`, `rax`). Но если квадрат второго основания влезает в слово, то слова можно рассматривать как пары полуслов. Старшую и младшую половины, очевидно, можно выделять битовыми операциями.

Промежуточные следствия (2)

С другой стороны, если основание другой системы счисления 10, то преобразование можно выполнять в два этапа с промежуточной системой счисления 10^n.

В частности, при преобразовании между 32-разрядными словами и десятичной системой можно использовать промежуточную систему $10\,000 < 2^{16}$, которая влезает в половину слова. Для 64-разрядных слов, соответственно, используем 10^8.

В Рефале-05 именно так реализована функция `Symb`.

Ещё частный случай преобразования (1)

Между общим случаем, когда основания $\mathtt{Bs}$ и $\mathtt{Bt}$ никак не связаны, и частным, когда $\mathtt{Bs} = \mathtt{Bt}^n$ или $\mathtt{Bt} = \mathtt{Bs}^n$, есть промежуточный, где одно из оснований пропорционально другому.

Он квадратичный, как и общий случай, но при этом допусаает ограниченную форму распараллеливания и не требует операций с двойными словами.

Далее будем считать, что $\mathtt{Bs} = \beta \cdot \mathtt{Bt}$. Обратный случай можно получить, проведя рассуждения в обратном порядке.

Ещё частный случай преобразования (2)

Рассмотрим число в системе по основанию $\mathtt{Bs}$:

$$C = \sum_{i=0}^{n-1} c_i \cdot \mathtt{Bs}^i$$

Обозначим $q_i = \lfloor c_i / \mathtt{Bt} \rfloor$ и $r_i = c_i \bmod \mathtt{Bt}$ — частное и остаток цифры от деления на $\mathtt{Bt}$. Тогда

$$C = \sum_{i=0}^{n-1} c_i \cdot \mathtt{Bs}^i = \sum_{i=0}^{n-1} (q_i \cdot \mathtt{Bt} + r_i) \cdot \mathtt{Bs}^i$$

Ещё частный случай преобразования (2)

Рассмотрим два соседних слагаемых:

$$\ldots + c_{i+1} \cdot \mathtt{Bs}^{i+1} + c_i \cdot \mathtt{Bs}^i + \ldots =$$

$$= \ldots (q_{i+1} \cdot \mathtt{Bt} + r_{i+1}) \cdot \mathtt{Bs}^{i+1} + (q_i \cdot \mathtt{Bt} + r_i) \cdot \mathtt{Bs}^i + \cdots =$$

$$= \ldots + q_{i+1} \cdot \mathtt{Bt} \cdot \mathtt{Bs}^{i+1} + r_{i+1} \cdot \mathtt{Bs}^{i+1} + q_i \cdot \mathtt{Bt} \cdot \mathtt{Bs}^i + r_i \cdot \mathtt{Bs}^i + \ldots =$$

$$= \ldots + r_{i+1} \cdot \beta^{i+1} \mathtt{Bt}^{i+1} + q_i \cdot \mathtt{Bt} \cdot \beta^i \mathtt{Bt}^i =$$

$$= \ldots + r_{i+1}\beta \cdot \mathtt{Bs}^i \cdot \mathtt{Bt} + q_i \cdot \mathtt{Bs}^i \cdot \mathtt{Bt} + \ldots =$$

$$= \ldots + (r_{i+1}\beta + q_i) \cdot \beta^i \mathtt{Bt}^i \cdot \mathtt{Bt} + \ldots =$$

$$= \ldots + (r_{i+1}\beta + q_i) \cdot \mathtt{Bs}^i \cdot \mathtt{Bt} + \ldots =$$

$$= \ldots + q_{i+1} \cdot \mathtt{Bt} \cdot \mathtt{Bs}^{i+1} + c'_i \cdot \mathtt{Bs}^i \cdot \mathtt{Bt} + r_i \cdot \mathtt{Bs}^i + \ldots$$

где $c'_i = r_{i+1}\beta + q_i$.

Ещё частный случай преобразования (3)

Тогда:

$$C = \sum_{i=0}^{n-1} c_i \cdot \mathtt{Bs}^i = \sum_{i=0}^{n-1} (q_i \cdot \mathtt{Bt} + r_i) \cdot \mathtt{Bs}^i =$$

$$= \mathtt{Bt} \sum_{i=0}^{n-1} (r_{i+1} \cdot \beta + q_i) \cdot \mathtt{Bs}^i = \mathtt{Bt} \sum_{i=0}^{n-1} c'_i + r_0 = C' \mathtt{Bt} + r_0$$

$$q_i = \lfloor c_i / \mathtt{Bt} \rfloor, \quad r_i = c_i \ \mathbf{mod}\ \mathtt{Bt}, \quad c'_i = r_{i+1}\beta + q_i, \quad \mathtt{Bs} = \beta \mathtt{Bt}$$

Здесь предполагается, что $r_n = 0$.

Ещё частный случай преобразования (4)

Так мы получаем следующий алгоритм:

```
ConvertTo(in S, in Ns, out T, in Bt, in/out Nt)
    C ← создать копию массива S, Nc ← Ns
    i ← 0
    while Nc > 0:
        assert i < Nt
        (q, T[i]) ← divmod(C[0], Bt)
        for j ← 0 to Nc - 1:
            if j < Nc - 1:
                (q', r) ← C[j + 1] % Bt
            else:
                r ← 0
            C[j] ← muladd(r, (Bs/Bt), q)
            q ← q'
        i ← i + 1
    Nt ← i
```

Заметим, что этот алгоритм одновременно использует и операцию деления с остатком, и операцию умножения с накоплением. Но обе эти операции уже не требуют двойных слов!

Ещё частный случай преобразования (5)

Этот алгоритм можно распараллелить: элементы векторов q_i, r_i и c_i могут вычисляться параллельно:

$$q_i = \lfloor c_i / \mathtt{Bt} \rfloor$$

$$r_i = c_i \ \textbf{mod} \ \mathtt{Bt}$$

$$c'_i = r_{i+1}\beta + q_i$$

Но в Рефале-05 это никак не используется.

Ещё частный случай преобразования (6)

Разновидность этого алгоритма используется в цифровой схемотехнике для преобразования между двоичной и двоично-десятичной формами, используется тот факт, что $10 = 5 \cdot 2, c'_i = (c_{i+1} \textbf{ mod } 2) \cdot 5 + \lfloor c_i/2 \rfloor$. Параллелизм в цифровой схеме используется естественно: к каждой тетраде можно независимо прибавлять пятёрку в зависимости от чётности числа в старшей тетраде.

Об известности этого алгоритма в компьютерном программировании докладчик ничего не знает, в книгах вроде «Искусства программирования» Кнута или «Алгоритмических трюков для программистов» Уоррена об этом ничего не говорится.

Использование этого алгоритма в Рефале-05 (1)

Описанный алгоритм используется в реализации функции `Numb` Рефала-05: используется тот факт, что $10^n = 5^n \cdot 2^n$.

Сначала аргумент функции (десятичная запись числа при помощи литер) преобразуется в систему по основанию $10^{N/4}$, где N — размер слова (32 или 64, можно выбирать при компиляции) — используем правило группировки.

Затем выполняются проходы, каждый из которых вытягивает из числа $N/4$ битов — каждые четыре прохода формируют очередную макроцифру. Вместо деления с остатком на $2^{N/4}$ используются двоичные сдвиги.

Использование этого алгоритма в Рефале-05 (2)

Преимущества алгоритма:

- он максимально заполняет значением машинное слово, а чем больше основание системы счисления, тем меньше требуется итераций — формула $O(\texttt{Ns} \cdot \texttt{Nt})$,
- алгоритм не требует двухсловных операций, все расчёты помещаются в машинное слово.

Для 64-разрядной арифметики можно было бы воспользоваться тем, что $10^{19} < 2^{64}$, но это сильно усложнило бы алгоритм, т.к. 19 бит не кратно размеру слова.

Вопросы реализации длинной арифметики

Общие принципы, положенные в основу

- Следует избегать неоправданных отказов: если можно (и не сложно) написать алгоритм, не требующий дополнительной памяти, то так и следует написать — тогда функция всегда будет завершаться успешно. Альтернативный вариант создаёт *возможность* того, что функция может завершаться из-за нехватки памяти.
- Код должен быть максимально чистым и прозрачным, на сколько возможно.
- Код должен быть переносимым.
- Код должен быть эффективным — не следует делать плохой алгоритм, если не намного сложнее можно реализовать хороший.

Следствия из общих принципов

- Функции арифметики не выделяют временных массивов, т.к. это создаёт возможность отказа, если условный `malloc()` вернёт нулевой указатель.
- Арифметические функции используют простые школьные алгоритмы «в столбик», более сложные алгоритмы вроде Карацубы не используются.
- Большинство функций арифметики повторно используют звенья из аргумента для построения результата.
- Цифрами для арифметических операций являются макроцифры, а не их половины, поэтому пришлось написать функции деления с остатком и двойного умножения, оперирующими парами слов.
- Рассматривался вариант сделать деление «побитовым» — каждая итерация давала бы бит частного, делимое сдвигалось на разряд, но он был бы крайне неэффективен.

Переиспользование памяти (1)

Большинство операций не требуют выделения памяти:

- Операция сложения переиспользует звенья более длинного аргумента.
- Операция вычитания переиспользуеют звенья аргумента, большего по модулю,
- Функции деления `Div`, `Mod` и `Divmod` вызывают функцию, которая вычисляет частное и остаток. Она строит звенья делителя и частного из звеньев делимого и служебных звеньев (звено с круглой скобкой, именем функции, скобкой вызова).

Операция сложения вызывается из `Add` для аргументов одинаковых знаков и из `Sub` для аргументов разных знаков, операция вычитания — наоборот.

Переиспользование памяти (2)

- Функция `Numb` преобразования строки в число переинициализирует звенья с десятичными цифрами в звенья с макроцифрами (их, очевидно, нужно в несколько раз меньше).
- Результат функции сравнения `Compare` — одна литера `'-'`, `'0'` или `'+'` (знак разности), выделять память ей не нужно.

Переиспользование памяти — исключения

- Функция умножения `Mul` вынуждена выделять память — выделяет столько звеньев, сколько имеется в более коротком аргументе.
- Функция преобразования числа в строку `Symb` тоже вынуждена выделять память: выделяет память под промежуточную цепочку по основанию 10^N и под цифры результата, это неизбежно. Ради упрощения реализации звенья из поля зрения не переиспользуются.

Настраиваемый размер макроцифр

Компилятор Рефала-05 компилирует в Си, размер макроцифры задаётся макросом препроцессора, передаваемым компилятору Си. Однако, чтобы Рефал-05 поддерживал 64-разрядные макроцифры в исходниках (не сообщал о синтаксической ошибке на них), он сам должен быть перекомпилирован с соответствующей опцией (он самоприменимый).

Макросы:

- `R05_NUMBER_INT` — `unsigned int`,
- `R05_NUMBER_LONG` — `unsigned long`,
- `R05_NUMBER_LONGLONG` — `unsigned long long` (Си99+),
- `R05_NUMBER_UINT32_T` — `uint32_t` (Си99+),
- `R05_NUMBER_UINT64_T` — `uint64_t` (Си99+),
- `R05_NUMBER_CUSTOM=…` — пользовательский тип (например, можно указать `size_t`).

Подробнее — в документации.

Особенности операции умножения

Операция умножения использует школьный алгоритм «в столбик».

Если сомножители состоят из M и N цифр, то результат потребует $M + N - 1 \ldots M + N$ цифр.

У операции умножения рассматриваются три частных случая:

- однозначного на однозначное,
- однозначного на многозначное — оптимизация, дополнительная память не выделяется,
- многозначного на многозначное — выделяется память размером с более короткий аргумент + одно слово, по мере умножения цифры второго сомножителя используются повторно.

Используется вспомогательная операция умножения с накоплением, дающая двойное слово.

Операция умножения, дающая двойной результат

```
struct mul_res {
  r05_number high, low;
};

static struct mul_res mul(r05_number x, r05_number y, r05_number acc) {
  enum { HALF = R05_NUMBER_BITS / 2 };
  const r05_number HALF_MASK = ((r05_number) 1 << HALF) - 1;

  r05_number x_high = x >> HALF;
  r05_number x_low = x & HALF_MASK;
  r05_number y_high = y >> HALF;
  r05_number y_low = y & HALF_MASK;

  r05_number hh = x_high * y_high;
  r05_number hl = x_high * y_low;
  r05_number lh = x_low * y_high;
  r05_number ll = x_low * y_low + (acc & HALF_MASK);
  r05_number mid =
    (ll >> HALF) + (hl & HALF_MASK) + (lh & HALF_MASK) + (acc >> HALF);

  struct mul_res result;
  result.high = hh + (hl >> HALF) + (lh >> HALF) + (mid >> HALF);
  result.low = (ll & HALF_MASK) | (mid << HALF);
  return result;
}
```

Особенности операции деления (1)

Операция умножения тоже школьный алгоритм «в столбик».

Если длина делимого M, делителя N, $M > N$, длина частного будет не более $M - N \ldots M - N + 1$, ллина остатка не более длины делителя.

Идея алгоритма (почерпнул из книги Уоррена «Алгоритмические трюки для программистов», Уоррен ссылается на Кнута): делимое и делитель нужно сдвинуть влево так, чтобы старший бит первой цифры делителя был единичным. В этом случае потребуется не более одной коррекции делимого. В конце выполняется обратный сдвиг остатка.

Особенности операции деления (2)

Рассматриваются следующие частные случаи:

- деление однозначного на однозначное,
- деление многозначного на однозначное — это не оптимизация, просто требуется отдельный алгоритм,
- деление многозначного на многозначное, $M < N$ — остаток равен делимому, частное нулевое,
- деление многозначного на многозначное, $M \geq N$ — алгоритм в столбик, если после вычитания получися заём, прибавляем делитель, пока не получится перенос (должно быть не более одной итерации, но, как и Уоррен, использую `while`).

Операция деления с остатком двойного слова реализована через операцию деления с остатком полуторного слова.

Операция деления двойного слова

```
struct div_res {
  r05_number quot, rem;
};

static struct div_res div_double_word(
  r05_number num_high, r05_number num_low, r05_number denom
) {
  enum { HALF = R05_NUMBER_BITS / 2 };
  const r05_number HALF_MASK = ((r05_number) 1 << HALF) - 1;
  struct div_res res, res_high;

  assert(num_high < denom);
  assert(1 == (denom >> (R05_NUMBER_BITS - 1)));

  res_high = div_sesquialter_words(
    num_high >> HALF, (num_high << HALF) | (num_low >> HALF), denom
  );
  res = div_sesquialter_words(
    res_high.rem >> HALF,
    (res_high.rem << HALF) | (num_low & HALF_MASK),
    denom
  );
  res.quot |= res_high.quot << HALF;
  return res;
}
```

Операция деления полуторного слова (1)

```
static struct div_res div_sesquialter_words(
  r05_number num_high_half, r05_number num_low, r05_number denom
) {
  enum { HALF = R05_NUMBER_BITS / 2 };
  const r05_number HALF_MASK = ((r05_number) 1 << HALF) - 1;

  r05_number num_high = (num_high_half << HALF) | (num_low >> HALF);
  r05_number denom_high = denom >> HALF;
  r05_number denom_low = denom & HALF_MASK;
  r05_number guess = num_high / denom_high;
  r05_number part_low = denom_low * guess;
  r05_number part_high = denom_high * guess;
  r05_number part_hl = part_high << HALF;
  r05_number part_hh = part_high >> HALF;
  struct div_res res;

  part_low += part_hl;
  if (part_low < part_hl) {
    part_hh += 1;
  }

  . . .
```

Операция деления полуторного слова (2)

```
    . . .

    while (
      part_hh > num_high_half
      || (part_hh == num_high_half && part_low > num_low)
    ) {
      guess -= 1;
      if (part_low < denom) {
        part_hh -= 1;
      }
      part_low -= denom;
    }

    res.quot = guess;
    res.rem = num_low - part_low;

    return res;
}
```

Оптимизации и замеры производительности

Бенчмарк

Исходники бенчмарка и версия библиотеки встроенных функций с оптимизациями находится здесь:

https://gitflic.ru/project/mazdaywik/slides-refal-05-arithm

Тестированию подвергались:

- актуальная версия Рефала-05, https://github.com/Mazdaywik/Refal-05, фиксация `cb71f84fb72587b81280abaa779be1f7f486a810`,
- ветка `v3.4-staged` Рефала-5λ, https://github.com/bmstu-iu9/refal-5-lambda, фиксация `871712ce78ec8989709e9d1568f7fbbe987dd383`,
- Рефал-5 версии PZ, http://www.botik.ru/pub/local/scp/refal5/bin/ref5_041029_bin_winNT.zip, и версии П$\underline{\check{з}}$, http://www.botik.ru/pub/local/scp/refal5/ref5_win10_x86_64_exe_252309.zip.

Тестовая машина

Замеры выполнялись на машине ASUS Vivobook Go 15 E1504F с процессором AMD Ryzen 3 7320U, базовая скорость 2,40 ГГц (но при вычислениях повышается до ≈3,5 ГГц), кэш 256 Кбайт/2,0 Мбайт/4,0 Мбайт, 8 Гбайт ОЗУ.

Операционная система Windows 10 Pro, версия 22H2, внутренняя версия 10.0.19045.6456.

Тестовое окружение

Использовались компиляторы:

- Borland C++ Compiler 5.5.1, выпущен в 2000 году,
- GCC 15.2.0, сборки (`i686-mcf-dwarf-rev0, Built by MinGW-Builds project`) и (`x86_64-mcf-seh-rev0, Built by MinGW-Builds project`).
- Microsoft Visual C++ 2022 версии 19.44.35220.

Перечисленные компиляторы вызывались с максимальной оптимизацией: «`-O2`» для BCC, «`-O3`» для GCC и «`/O2`» для MSVC.

Для Рефала-5 (PZ и Пʒ) использовались уже готовые скомпилированные модули.

Компилятор Рефала-5λ был собран с флагами `set RLMAKE_FLAGS=-X-OiADPRS` и компилятором BCC без оптимизации.

Оптимизации

Сначала докладчик думал, что узким местом алгоритмов будут мультипликативные операции: двухсловные функции умножения с накоплением и деления с остатком.

Было сделано несколько оптимизаций этих операций:

- использование более широкого типа, доступного в компиляторе (например, макроцифра — `uint32_t`, «двухсловный» тип — `uint64_t`),
- использование вещественной арифметики для 32-разрядных макроцифр и полуторасловного деления,
- в задумке были даже ассемблерные вставки, но докладчик не успех их к докладу.

Но замер показал, что узким местом является не арифметика, а списковое представление.

Особенности бенчмарка (1)

Входными данными для бенчмарка являются пары аргументов, определяемых множеством

$$\{(41 \cdot 3^i, 71 \cdot 3^j), (71 \cdot 3^j, 41 \cdot 3^i) | i \in 0 \ldots 399, j \in 0 \ldots 399\}$$

В машинном представлении цепочки из не более чем 20 32-разрядных или 10 64-разрядных макроцифр, длины цепочек распределены равномерно.

Для получения тестовых данных для функции `Symb` пары чисел (цепочек макроцифр конкатенируются) в цепочки длиной от 2 до 40 или 20 макроцифр (в зависимости от размера макроцифры). Тестовые данные для `Numb` формируются путём преобразования исходных пар чисел в строки при помощи `Symb` и их конкатенации.

Особенности бенчмарка (2)

При этом, сначала генерируются входные данные в виде огромного списка пар чисел, затем для каждого элемента этого списка вызывается тестируемая функция. Сгенерированные звенья списка для аргументов располагаются, как правило, по соседству, благодаря чему обращение к ним выполняется эффективно.

Но если в рамках одного запуска программы-бенчмарка запустить тестирование некоторой функции несколько раз, то список свободных звеньев основательно перемешается и из-за промахов кэша время вычислений существенно замедлится.

Как это выглядит — на следующем слайде.

Особенности бенчмарка (3) — промахи кэша

```
X:\>benchmark.exe aaa
PrepareData 0.297
Add         2.203
Add         2.344
Add         2.391
Add         2.750
Add         2.828
Add         2.843
Add         2.813
```

Опция `aaa` предписывает несколько раз вызвать прогон для `Add`. Прогон выше был сделан для компилятора BCC. Замедление выполнения заметно.

Результаты замеров производительности (1)

К сожалению, докладчик не успел к докладу сделать чистовые замеры производительности: когда на компьютере выполнялся только бенчмарк без каких-либо других ресурсозатратных программ.

Поэтому последующие замеры будут проводиться на основе немного «грязных» данных, полученных во время отладки бенчмарка, а значит, данные могут быть не очень точны.

Впоследствии будет произведён аккуратный замер и его результаты будут добавлены в презентацию в репозитории

https://gitflic.ru/project/mazdaywik/slides-refal-05-arithm

Результаты замеров производительности (2) — функция Add

Компилятор	Тип макроцифры	Время, с
BCC	`unsigned int`	0,875
GCC 32	`uint32_t`	0,983
GCC 68	`uint64_t`	0,894
MSVC 32	`uint32_t`	1,218
MSVC 64	`uint64_t`	0,826
Рефал-5λ	`unsigned int`	17,969
Рефал-5 PZ	32-разрядное	1,287
Рефал-5 ПƷ	64-разрядное	1,279

Результаты замеров производительности (3) — функция `Mul`

Компилятор	Тип макроцифры	Время, с
BCC	`unsigned int`	2,672
GCC 32	`uint32_t`	1,472
GCC 68	`uint64_t`	1,003
MSVC 32	`uint32_t`	1,839
MSVC 64	`uint64_t`	0,94
Рефал-5λ	`unsigned int`	218,953
Рефал-5 PZ	32-разрядное	4,423
Рефал-5 ПЗ	64-разрядное	4,413

Результаты замеров производительности (4) — функция `Divmod`

Компилятор	Тип макроцифры	Время, с
BCC	`unsigned int`	1,609
GCC 32	`uint32_t`	1,199
GCC 68	`uint64_t`	1,011
MSVC 32	`uint32_t`	1,493
MSVC 64	`uint64_t`	0,941
Рефал-5λ	`unsigned int`	102,375
Рефал-5 PZ	32-разрядное	4823,013
Рефал-5 П$\breve{3}$	64-разрядное	нет замера

Это был отладочный прогон, замера не сохранилось.

Результаты замеров производительности (4) — функция `Numb`

Компилятор	Тип макроцифры	Время, с
BCC	`unsigned int`	1,047
GCC 32	`uint32_t`	0,856
GCC 68	`uint64_t`	0,738
MSVC 32	`uint32_t`	1,066
MSVC 64	`uint64_t`	0,716
Рефал-5λ	`unsigned int`	749,125
Рефал-5 PZ	32-разрядное	24,746
Рефал-5 ПЗ	64-разрядное	нет замера

Результаты замеров производительности (4) — функция `Symb`

Компилятор	Тип макроцифры	Время, с
BCC	`unsigned int`	23,0
GCC 32	`uint32_t`	6,041
GCC 68	`uint64_t`	2,929
MSVC 32	`uint32_t`	7,052
MSVC 64	`uint64_t`	3,155
Рефал-5λ	`unsigned int`	176,922
Рефал-5 PZ	32-разрядное	119,748
Рефал-5 ПЗ	64-разрядное	130,569

Что не влезло в доклад

Были сделаны замеры производительности для 32-битных макроцифр на 64-битных платформах и наоборот. Компиляторы GCC и MSVC позволяют использовать тип `uint64_t` при компиляции для 32-битной платформы, понятно, что этот тип аппаратно не поддерживается и эмулируется.

Для мультипликативных операций (`Mul`, `Divmod`) были замеры производительности оптимизации числа вдвое большей разрядности (`uint64_t` для `uint32_t`, `unsigned __int128` для `uint64_t`). Результаты на разных компиляторах разные.

Для деления была проверена оптимизация полуторасловного деления. Результаты плохие для всех компиляторов — медленнее, чем исходный вариант для целых чисел

Оптимизация для широкого типа

```
static struct div_res div_double_word(
  r05_number num_high, r05_number num_low, r05_number denom
) {
  struct div_res res;
  r05_twice_number numerator = num_high;
  numerator <<= R05_NUMBER_BITS;
  numerator |= num_low;
  res.quot = (r05_number) (numerator / denom);
  res.rem = (r05_number) (numerator % denom);
  return res;
}
```

Для умножения аналогично.

«Оптимизация» половинного деления через double

```
static struct div_res div_sesquialter_words(
  r05_number num_high_half, r05_number num_low, r05_number de
) {
  const double POWER32 = 4294967296.0;
  double numerator = num_high_half * POWER32 + num_low;
  r05_number quot = (r05_number) (numerator / denom);
  double rem = numerator - (double) quot * denom;
  struct div_res res;

  assert(0 ⩽ rem && rem < denom);
  res.quot = quot;
  res.rem = (r05_number) rem;
  return res;
}
```

Выводы

- Большой бенчмарк мерять и отлаживать долго, можно не успеть к докладу ☺.
- В Рефале-5λ длинная арифметика написана на смеси Рефала и C++, бóльшая часть на Рефале. Это медленно.
- В Рефале-5 деление о-о-о-чень медленное.
- Алгоритм, использованный в `Numb` интересный и работает очень неплохо, имеет смысл попробовать переписать `Symb` в той же манере.