УДК 004.05

Верификация программы преобразования строки в целое число

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

Описывается дедуктивная верификация программы kstrtoul на языке Си из библиотеки ОС Linux. Программа kstrtoul реализует вычисление целого числа, представленного в виде последовательности литер. Чтобы упростить верификацию, применяются трансформации замены операций с указателями эквивалентными действиями без указателей. Программа преобразуется на язык предикатного программирования. Конструируется модель внутреннего состояния программы как часть спецификации программы. Дедуктивная верификация проведена в системах Why3 и Coq.

Ключевые слова: *дедуктивная верификация, трансформации программ, функциональное программирование, предикатное программирование, строковый тип.*

1. Введение

Исходной задачей является дедуктивная верификация программы kstrtoull.c на языке Си из библиотеки ядра ОС Linux. Программа kstrtoull.c реализует вычисление целого числа, представленного в строке в виде последовательности литер по правилам языка Си.

Дедуктивная верификация намного проще и быстрее для функциональных программ, чем для аналогичных императивных программ. Причина сложности императивных программ в том, что указатели, конструкции необходимые для оптимизации программ, существенно усложняют логику императивных программ. Для упрощения программ применяются трансформации, устраняющие указатели в императивной программе [6]. Операции с указателями заменяются эквивалентными действиями без указателей. Полученная программа преобразуется в эквивалентную предикатную программу. В системах автоматического доказательства Why3[18] и Coq [11] реализуется процесс дедуктивной верификации предикатной программы.

Предыдущий релиз верификации kstrtoull.c оказалася неудачным. Естественный и привычный стиль спецификации приводит к трудно доказуемым формулам корректности. Верификация в системе Why3 оказалась тяжелой. Далее, для упрощения верификации была

разработана модель внутреннего состояния исполняемой программы. Из предикатной программы в максимальной степени экстрагирована ее константная часть. Использование модели в спецификации программы существенно упростило процесс дедуктивной верификации.

Во втором разделе настоящей работы дается краткое описание языка предикатного программирования. В третьем разделе описывается трансформация исходной программы kstrtoull с получением эквивалентной предикатной программы. В четвертом разделе определяется модель программы kstrtoull и детально описывается процесс спецификации предикатной программы с использованием модели. Далее строятся формулы корректности программы относительно спецификации применением системы правил [1, 7]. Построение формул корректности документируется в Приложении 3. Совокупность формул корректности вместе с описаниями типов и переменных оформляется в виде набора теорий. В Приложении 1 представлены теории с формулами корректности. Эти теории транслируются на язык спецификаций why3 [18]. В Приложении 2 приведены теории на языке Why3 для доказательства формул корректности на момент завершения работы по верификации. Особенности процесса дедуктивной верификации предикатной программы в системах Why3[18] и Coq [11] описывается в пятом разделе. В шестом разделе представлен обзор работ. Итоги верификации подводятся в седьмом разделе.

2. Язык предикатного программирования

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [2] следующего вида:

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

Предусловие и постусловие являются формулами на языке исчисления предикатов. Они обязательны при дедуктивной верификации [3, 4, 8, 9, 15]. Мера задается только для рекурсивных программ и используется для доказательства их завершения.

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

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

Всякая переменная характеризуется *типом* – множеством допустимых значений. *Описание типа* **type** T(p) = D с возможными параметрами p связывает имя типа T с его изображением D. Типы **bool**, **int**, **real** и **char** являются *примитивными*. Значением типа **array**(T_e, T_i) является *массив* с *элементами массива* типа T_e и *индексами* конечного типа T_i.

Гиперфункция – программа с несколькими *ветвями* результатов. Гиперфункция B(x**:** y**:** z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются.

Вызов гиперфункции записывается в виде B(x**:** y #M1**:** z #M2). Здесь M1 и M2 – метки программы, содержащей вызов. Операторы перехода #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2.

Вызов гиперфункции может комбинироваться с операторами обработки ветвей:

B(x**:** y #M1**:** z #M2) **case** M1: C(y**:** u) **case** M2: D(z**:** u) .

Вызов вида B(x**:** y #M1**:** z #M2); M1**: ...** может быть представлен оператором B(x**:** y**:** z #M2).

Формально гиперфункция определяется через предикатную программу следующего вида:

B(x**:** y, z, e)
pre P(x) **post** e = E(x) & (E(x) ⇒ S(x, y)) & (¬E(x) ⇒ R(x, z))
{ ... };

Здесь x, y и z – непересекающиеся возможно пустые наборы переменных; P(x), E(x), S(x, y) и R(x, z) – логические утверждения. Предположим, что все присваивания вида e = **true** и e = **false** – последние исполняемые операторы в программе B. Программа B может быть заменена следующей программой в виде *гиперфункции*:

B(x**:** y #1**:** z #2)
pre P(x) **pre** 1**:** E(x) **post** 1: S(x, y) **post** 2: R(x, z)
{ ... };

В теле гиперфункции каждое присваивание e = **true** заменено оператором перехода #1, а e = **false** – на #2. *Метки* 1 и 2 – дополнительные параметры, определяющие два различных *выхода* гиперфункции.

Спецификация гиперфункции состоит из двух частей. Утверждение после “**pre** 1” есть предусловие первой ветви; предусловие второй ветви – отрицание предусловия первой ветви. Утверждения после “**post** 1” и “**post** 2” есть постусловия для первой и второй ветвей, соответственно.

Аппарат *гиперфункций* является более общим и гибким по сравнению с известным механизмом обработки исключений, например, в таких языках, как Java и C++. Традиционные подходы в реализации обработки аварийных ситуаций предполагают заведение дополнительных структур, усложняющих программу. Этого удается избежать при использовании гиперфункций. Использование гиперфункций делает программу короче, быстрее и проще для понимания.

3. Трансформация программы перевода строки в целое число

3.1. Описание программы

Функция kstrtoull переводит целое число, представленное строкой, в значение типа unsigned long long, то есть неотрицательное (беззнаковое) целое максимального размера. Строка в языке Си – последовательность литер, завершающаяся нулевым байтом, то есть литерой ‘\0’. Строка – первый аргумент функции kstrtoull. Второй аргумент – основание числа в строке (8,10,16) или 0; в случае нуля основание числа должно быть определено по синтаксическому представлению числа во входной строке. Третий аргумент – указатель на переменную, в которую записывается значение числа. Возвращаемое значение функции kstrtoull – *код завершения*:

- 0 – нормальное завершение с вычисленным значением по третьему аргументу;
- -EINVAL – синтаксическая ошибка в представлении числа;
- -ERANGE – значение числа не помещается в тип unsigned long long.

Функция kstrtoull вызывает функцию _kstrtoull, которая далее вызывает функции _parse_integer и _parse_integer_fixup_radix. Во всех этих функциях первый параметр s – указатель на массив байтов (типа **char**), представляющий последовательность литер входной строки. Второй параметр base – основание числа: 8, 10 или 16, либо 0.

В соответствии с правилами языка Си в функции kstrtoull используется следующее представление целого числа в виде строки литер:

\+ <основание числа> <последовательность цифр> '\n' ‘\0’

Здесь литера '\0' – нулевой байт, завершающий строку. Литера '\n' определяет переход на следующую строку. Знак числа «+», основание числа и литера '\n' могут отсутствовать в строковом представлении числа.

Ниже приведен код всех функций программы после приведения к нормальному виду условий в условных операторах и цикле **while**. Исходный код и его описание находятся на сайте: https://elixir.bootlin.com/linux/latest/source/lib/kstrtox.c.

Функция _parse_integer_fixup_radix вычисляет основание числа в случае, когда исходное значение параметра base равно нулю. В данной функции параметр base выступает как аргумент и результат. Поэтому он представлен указателем. Кроме того, для шестнадцатеричного числа в массиве s пропускается комбинация «0x». Результатом функции является продвинутый указатель по строке s, который должен указывать на первую цифру числа.

Отметим, что в соответствии с правилами языка Си шестнадцатеричное число начинается комбинацией литер «0x». Далее, восьмеричное число начинается с нулевой цифры. В остальных случаях число считается десятичным.

```
const char *_parse_integer_fixup_radix(const char *s, unsigned int *base)
{
	if (*base == 0) {
		if (s[0] == '0') {
			if (_tolower(s[1]) == 'x' && isxdigit(s[2]))
				*base = 16;
			else
				*base = 8;
		} else
			*base = 10;
	}
	if (*base == 16 && s[0] == '0' && _tolower(s[1]) == 'x')
		s += 2;
	return s;
}
```

Функция _tolower преобразует заглавные буквы к нижнему регистру, сохраняя значения остальных литер. Функция isxdigit проверяет, является ли литера шестнадцатеричной цифрой.

Далее код функции _parse_integer. Она вычисляет собственно значение целого числа по последовательности цифр в строке по указателю s. Параметр base – основание числа: 8, 10 или 16. Вычисленное значение целого числа записывается в переменной по указателю p, третьему параметру функции. Возвращаемый результат функции – число цифр,

составляющих исходное число и обработанных функцией. В случае переполнения, когда вычисляемое значение больше максимального значения ULLONG_MAX, в возвращаемый результат функции добавляется бит KSTRTOX_OVERFLOW (в 31 разряде), сигнализирующий о переполнении.

```
unsigned int _parse_integer(const char *s, unsigned int base, unsigned long long *p)
{
	unsigned long long res;
	unsigned int rv;
	res = 0;
	rv = 0;
	while (true) {
		unsigned int c = *s;
		unsigned int lc = c | 0x20; /* don't tolower() this line */
		unsigned int val;
		if ('0' <= c && c <= '9')
			val = c - '0';
		else if ('a' <= lc && lc <= 'f')
			val = lc - 'a' + 10;
		else
			break;
		if (val >= base)
			break;
		if (res & (~0ull << 60)) {
			if (res > div_u64(ULLONG_MAX - val, base))
				rv |= KSTRTOX_OVERFLOW;
		}
		res = res * base + val;
		rv++;
		s++;
	}
	*p = res;
	return rv;
}
```

Конструкция c | 0x20 эквивалентна вызову _tolower(c). Функция div_u64 реализует целочисленное деление. На каждом шаге цикла вычисления значения целого числа проводится контроль переполнения очередного значения за границы типа unsigned long long. Факт переполнения фиксируется битом KSTRTOX_OVERFLOW в результате функции. Однако само вычисление значения не прекращается, поскольку прерывание при этом не генерируется. Вычисление, конечно, будет неверным. Цикл продолжается до полного исчерпания набора цифр. Видимо, преследуется очевидная цель – отсканировать всю последовательность цифр не изменяя программу.

Ниже приведен код функции _kstrtoull, реализующей вычисление целого числа, представленного основанием числа и набором цифр. В ее теле вызываются две предыдущие функции _parse_integer_fixup_radix, вычисляющее основание, которое далее используется в вызове _parse_integer для получения значения числа. Дальнейшие действия реализуют контроль возможных ошибок в представлении числа. Вычисленное значение числа записывается в переменную по указателю res, третьему параметру функции. Возвращаемым результатом функции является код завершения: 0 – нормальное завершение без ошибок, -EINVAL – синтаксическая ошибка в представлении числа, -ERANGE – число слишком длинное и не помещается в тип unsigned long long.

```
static int _kstrtoull(const char *s, unsigned int base, unsigned long long *res)
{
	unsigned long long _res;
	unsigned int rv;
	s = _parse_integer_fixup_radix(s, &base);
	rv = _parse_integer(s, base, &_res);
	if (rv & KSTRTOX_OVERFLOW)
		return -ERANGE;
	if (rv == 0)
		return -EINVAL;
	s += rv;
	if (*s == '\n')
		s++;
	if (*s != '\0')
		return -EINVAL;
	*res = _res;
	return 0;
}
```

Далее код главной функции kstrtoull. Она вычисляет значение целого числа по его строковому представлению в параметре s. Второй параметр base – основание числа: 8, 10 или 16, либо 0, когда основание числа должно быть определено по синтаксическому представлению числа во входной строке. Третий параметр функции res – указатель на переменную, куда записывается вычисленное значение числа. Возвращаемым результатом функции является код завершения.

```
int kstrtoull(const char *s, unsigned int base, unsigned long long *res)
{
	if (s[0] == '+')
		s++;
	return _kstrtoull(s, base, res);
}
```

3.2. Устранение указателей

Для упрощения дедуктивной верификации программа kstrtoull трансформируется в эквивалентную предикатную программу. На первом этапе трансформаций устраняются указатели. Операции с указателями заменяются эквивалентными действиями без указателей. Устранение указателей существенно упрощает программу.

В программе kstrtoull используются указатели на строковые массивы и указатели на переменные. Для них применяются разные виды трансформаций. Трансформации могут быть применены при соблюдении условий, проверяемых с использованием специального потокового анализа программы.

3.2.1. Трансформация представления результатов функции

В языке Си всякая функция может иметь единственный результат, который передается оператором **return** в теле функции. Другой результат, если такой имеется, реализуется через параметр-указатель присваиванием переменной, на которую ссылается данный указатель. Одна из применяемых трансформаций – замена результатов функции *параметрами-результатами* в стиле языка предикатного программирования P [2]. Основной результат, передаваемый оператором **return**, и другие дополнительные результаты помещаются после разделителя «**:**» во второй секции параметров.

Для результата функции, передаваемого оператором **return**, вводится имя параметра. Например, для кода завершения вводится имя rep. Реализуется трансформация замены оператора **return** оператором присваивания параметру, например:

return 0 → rep = 0

Новая переменная вводится не всегда. Иногда используется локальная переменная, которая становится параметром-результатом, а оператор **return** устраняется из программы.

При наличии двух результатов функции присваивание результатам вызова функции представляется в виде оператора мульти-присваивания:

|res1, res2| = Имя функции(входные параметры)

3.2.2. Трансформации операций с указателями на переменные

Параметр-указатель, используемый для присваивания переменной, дополнительному результату функции, при трансформации заменятся параметром-результатом с прямым доступом, не через указатель. Для различных вхождений параметров-указателей в теле функции применяются следующие трансформации:

*p → p
*base → base
*res → res
&_res → _res
&base → base

Если переменная, доступная по параметру-указателю, модифицируется в теле функции, то в трансформированной программе такая переменная будет находиться одновременно в составе аргументов и результатов.

3.2.3. Трансформации операций с указателями на массивы

Тип указателя **char** * заменяется типом массива:

char * → **array(char, nat)**

Вместо указателя s в трансформированной программе используется массив s и переменная js – индекс в массиве s. Значение индекса js в массиве s в точности соответствует позиции указателя s в исходной программе на языке Си.

Определим трансформации для различных вхождений s в исходной программе:

*s → s[js]
s++ → js++
s += 2 → js += 2
s[0] → s[js]
s[1] → s[js+1]
_kstrtoull(s, ...) → _kstrtoull(s, js, ...)

Первый параметр всех функций – строку s будет удобнее определить глобальной переменной в трансформированной программе. С учетом этого вызов функции преобразуется к виду _kstrtoull(js, ...).

3.2.4. Программа после устранения указателей

Для упрощения верификации полезно вынести константные параметры функций в состав глобальных переменных. Определим глобальный массив s.

array(char, nat) s;

При этом переменная js остается параметром. Ее тип – size_t, соответствующий адресуемой памяти.

В функции _parse_integer_fixup_radix параметр base используется и присваивается. Поэтому он помещается также среди результатов. Вхождения *base везде заменяются на base. Оператор **return** s устраняется, а образ s, т.е. массив s и индекс js, надо было бы поместить среди результатов функции. Однако массив s – глобальный, поэтому в результаты помещается только js.

```
_parse_integer_fixup_radix(size_t js,  unsigned int base : unsigned int base, size_t js)
{
	if (base == 0) {
		if (s[js] == '0') {
			if (_tolower(s[js+1]) == 'x' && isxdigit(s[js+2]))
				base = 16;
			else
				base = 8;
		} else
			base = 10;
	}
	if (base == 16 && s[js] == '0' && _tolower(s[js+1]) == 'x')
		js += 2;
}

_parse_integer(size_t js, unsigned int base : unsigned long long p, unsigned int rv)
{
	unsigned long long res;
	res = 0;  rv = 0;
	while (true) {
		unsigned int c = s[js];
		unsigned int lc = c | 0x20; /* don't tolower() this line */
		unsigned int val;
		if ('0' <= c && c <= '9')
			val = c - '0';
		else if ('a' <= lc && lc <= 'f')
			val = lc - 'a' + 10;
		else
			break;
		if (val >= base)
			break;
		if (res & (~0ull << 60)) {
			if (res > div_u64(ULLONG_MAX - val, base))
				rv |= KSTRTOX_OVERFLOW;
		}
		res = res * base + val;
		rv++;
		js++;
	}
	p = res;
}
```

Выше представлена функция _parse_integer после трансформации. Переменная p становится результатом функции. Оператор *p = res заменяется на p = res. Оператор **return** rv устраняется. Локальная переменная rv становится параметром-результатом функции.

Ниже код функции _kstrtoull после трансформации. Переменная res становится результатом функции. Оператор *res = _res заменяется на res = _res. Вводится переменная-результат rep для кода завершения, возвращаемого оператором **return**. Оператор **return** 0 заменяется присваиванием rep = 0. Оператор **return** -ERANGE заменяется фрагментом {rep = -ERANGE; **return**}. Поскольку функции _parse_integer_fixup_radix и _parse_integer имеют два результата, присваивание результатов вызовов реализуется оператором мульти-присваивания.

```
_kstrtoull(size_t js, unsigned int base : unsigned long long res,  int rep)
{
        unsigned long long _res;
        unsigned int rv;
        |base, js| = _parse_integer_fixup_radix(js, base);
        |_res, rv | = _parse_integer(js, base);
        if (rv & KSTRTOX_OVERFLOW)
                {rep = -ERANGE; return}
        if (rv == 0)
                {rep = -EINVAL; return}
        js += rv;
        if (s[js] == '\n')
                js++;
        if (s[js] != zero)
                {rep = -EINVAL; return}
        res = _res;
        rep = 0;
}
```

Трансформация главной функции kstrtoull вводит переменные-результаты res и rep. Параметр-указатель s заменяется глобальным массивом s и локальной переменной – индексом js с начальной инициацией size_t js =0. Вхождение s[0] заменяется на s[js+0], равное s[0].

```
kstrtoull(unsigned int base : unsigned long long res, int rep)
{       size_t js =0;
        if (s[0] == '+')
                js++;
        |res, rep| = _kstrtoull(js, base);
}
```

Распознавание того, что параметр-указатель s подставляется аргументом в вызове _kstrtoull, а res подставляется результатом, реализуется потоковым анализом тела функции _kstrtoull.

3.3. Трансформации в предикатную программу

Программа после устранения указателей переводится в предикатную программу. Циклы заменяются рекурсивными программами. Некоторые фрагменты программы оформляются гиперфункциями.

Преобразуем имена типов:

```
type = nat32 = 0..2**32 – 1; // unsigned int;
type = nat64 = 0..2**64 – 1; // unsigned long long;
type = size_t = nat64;
```

Определим глобальный массив s.

```
array(char, nat) s;
```

Из программы _parse_integer вынесем фрагмент вычисления значения очередной цифры.

```
digitVal(nat32 c : nat32 val : #notDigit){
        nat32 lc = c | 0x20; /* don't tolower() this line */
        if ('0' <= c && c <= '9')
                val = c - '0';
        else if ('a' <= lc && lc <= 'f')
                val = lc - 'a' + 10;
        else
                #notDigit
        if (val >= base)
                #notDigit
}
```

Первая ветвь гиперфункции digitVal соответствует нормальному завершению программы с вычислением значения val для цифры c. Выход #notDigit по второй ветви гиперфункции соответствует случаю, когда литера c – не цифра.

В программе _parse_integer заменим цикл **while** рекурсивной программой. Ее дополнительными аргументами становятся переменные js, res и rv, модифицируемые в цикле.

```
parseInt (size_t js, nat64 res, nat32 rv, base : nat64 res, nat32 rv)
{
        digitVal(s[js] : nat32 val : return);
        if (res & (~0ull << 60)) {
                if (res > div_u64(ULLONG_MAX - val, base))
                        rv |= KSTRTOX_OVERFLOW;
        }
        parseInt (js+1, res * base + val, rv+1, base :  res, rv)
}
```

Отметим, что тип переменной js определен как size_t, соразмерно памяти, доступной по исходному указателю s.

Программа _parse_integer преобразуется к виду:

```
_parse_integer(size_t js, nat32 base : nat64 p, nat32 rv)
{
        parseInt(js, 0, 0, base: nat64 res, rv);
        p = res;
}
```

Программа _parse_integer_fixup_radix остается без изменений.

```
_parse_integer_fixup_radix(size_t js, nat32 base : nat32 base, size_t js)
{
        if (base == 0) {
                if (s[js] == '0') {
                        if (_tolower(s[js+1]) == 'x' && isxdigit(s[js+2]))
                                base = 16;
                        else
                                base = 8;
                } else
                        base = 10;
        }
        if (base == 16 && s[js] == '0' && _tolower(s[js+1]) == 'x')
                js += 2;
}
```

Программы kstrtoull и _kstrtoull заменяются гиперфункциями с тремя ветвями. Первая ветвь гиперфункции соответствует нормальному завершению с вычислением значения res для целого числа. Вторая ветвь завершается выходом #ERange в случае выхода значения константы за пределы типа nat64. Третья ветвь завершается выходом #ESint при обнаружении синтаксической ошибки в представлении числа. Устраняется переменная-результат – код завершения rep.

```
_kstrtoull(size_t js, nat32 base : nat64 res #Valid : #ERange : #ESint)
{
        nat64 _res;
        nat32 rv;
        |base, js| = _parse_integer_fixup_radix(js, base);
        |_res, rv | = _parse_integer(js, base);
        if (rv & KSTRTOX_OVERFLOW)
                #ERange
        if (rv == 0)
                #ESint
        js += rv;
        if (s[js] == '\n')
                js++;
        if (s[js] != zero)
                #ESint
        res = _res;
        #Valid
}

kstrtoull(nat32 base : nat64 res #Valid : #ERange : #ESint)
{       size_t js =0;
        if (s[js] == '+')
                js++;
        _kstrtoull(js, base: res #Valid: #ERange : #ESint);
}
```

4. Спецификация предикатной программы

Полная предикатная программа представлена выше набором функций. Спецификация программы содержит предусловие и постусловие для каждой функции.

Для упрощения верификации из предикатной программы в максимальной степени экстрагирована ее константная часть, отражающая внутренние состояния исполняемой программы. При вынесении константной части предикатная программа подверглась существенной модификации. Большая часть аргументов программ стали глобальными переменными. Экстрагированная часть определяет модель программы, см. Рис. 1. Отметим, что в предыдущий релиз дедуктивной верификации базировался на обычном стиле спецификации программы без модели. Доказательство формул корректности в системе Why3[18] оказалось громоздким и трудоемким.

Последующие изменения, реализованные при доказательстве формул корректности, в спецификации и генерации формул корректности отмечены другими цветами. Разные цвета соответствуют разным сериям изменений.

4.1. Базисная часть

Модель программы определяется набором переменных js0, ks, n, sL, resN и аксиом, определяющих их свойства. Переменная js0 определяет позицию после возможного "+" в начале строкового представления числа; ks определяет позицию первой цифры после возможного радекса, задающего основание числа (8,10,16); n фиксирует позицию после последней цифры; sL определяет позицию последней нулевой литеры '\0'; resN – значение числа.

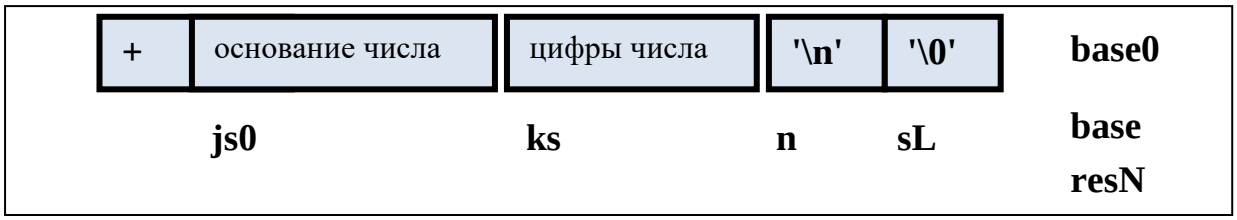

Рис. 1. Модель программы

Будем считать, что в типе **char** нет верхнего регистра для букв, что достигается применением функции _tolower. Как следствие, не рассматривается литера 'X' и опускаются вызовы функции tolower в программе. Введем константы:

```
char zero = '\0', nul = '0', nl = "\n", iks = "x", plu= "+";
```

Параметр base определяет систему счисления.

```
formula isBase(nat base) = base = 8 or base = 10 or base = 16;
type arCh = array(char, nat);
arCh s; (*массив литер s – соответствует входной строке функции kstrtoull *)
nat  sL; (* индекс последней литеры исходной строки s*)
axiom ALnat: sL >= 0;
axiom ALs: s[sL] = zero;
nat base0; (*начальное значение второго аргумента kstrtoull *)
axiom Ab0: base0 = 0 or isBase(base0);
```

Индекс после плюса:

```
nat js0 = if s[0] = plu then 1 else 0;
formula radix(nat j, k, base) =
        (if base0=0 then
           (if s[j] = '0' then (if s[j+1]='x' then base = 16  else  base =8)
            else base =10 )
         else base = base0)
        &
        (if base=16 & s[j] = '0' & s[j+1='x' then k=j+2 else k=j);
nat  ks; (* индекс начальной цифры в исходной строке s *)
nat base; (*итоговое *)
axiom Kse: radix(js0, ks, base);
```

Таким образом, base0 определяет начальное значение исходного параметра, а base – его значение после вызова _parse_integer_fixup_radix.

Цифры как подмножество типа **char** определяются отношением digit:

```
formula digitB(char c, nat base, valD);
axiom Adig:  forall char c, nat base, valD. isBase(base) ⇒
        ( 220<=c<220+base <->  digitB(c, base, valD) /\ c = valD + 220)
```

Нет необходимости определять детально, как по цифре получается ее значение. Достаточно использовать данную аксиому. Здесь предполагается, что шестнадцатеричные цифры следуют непосредственно за десятичными, что в действительности не так. Это упрощение полезно для доказательства формул корректности.

```
formula  digit(char c, nat valD) = digitB(c, base, valD);
formula isDigit(char c) = ∃nat valD. digit(c, valD);
formula isDigit16(char c) = ∃nat valD. digitB(c, 16, valD);
```

Формула digit определяет цифру в системе счисления, определяемой глобальной переменной base.

Формула endNum определяет индекс литеры после последовательности цифр.

```
formula endNum(nat n) = n >= ks & ¬isDigit(s[n]) & ∀j=ks..n-1. isDigit(s[j]);
nat  n; (* индекс за последней цифрой в строке s*)
axiom Anu: endNum(n);

formula numRes(nat k, m, nat res0, res) =
        if (k=m) res = res0
        else ∃val. digit(s[k], val) & numRes(k+1, res0 * base + val, res);
```

Формула numRes вычисляет значение числа res по оставшейся последовательности цифр в вырезке s[k .. m] в предположении, что res0 – значение числа по предыдущей последовательности цифр. Отметим, что данное рекурсивное определение не воспринимается в Why3. Вместо него пришлось использовать индуктивное определение, которое затем дважды подвергалось модификации.

```
formula number(nat res) = numRes(ks, n, 0, res);
nat resN;
axiom Ares: number(resN);
formula afterNum() = ks<n & (s[n]= zero \/ s[n]=nl /\ s[n+1]= zero);
```

Формула afterNum постулирует, что по индексу n находится либо нулевой байт, либо перевод строки с последующим нулевым байтом. Условие ks<n здесь необходимо. Оно добавлено позже при дедуктивной верификации.

Определенная выше модель программы фактически представляет специализацию входных аргументов base и js с фиксацией свойств каждой специализации, что позволяет кардинально упростить доказательство формул корректности.

4.2. Спецификация программы kstrtoull

Программа kstrtoull модифицирована в соответствии с моделью. Аргументы перенесены в глобальные переменные. Аргументы исчезли также и в вызове _kstrtoull. Вхождения переменной js заменены на js0.

```
kstrtoull( : nat64 res #Valid : #ERange : #ESint)
pre Valid: pValid() pre ERange: pERange() pre ESint: pESint()
post Valid: qValid(res)
{       size_t js;
        if (s[0] == '+') js0 = 1 else js0 =0;
        _kstrtoull(js, base : res #Valid : #ERange : #ESint);
}
```

При отсутствии аргументов нет общего предусловия. Но есть предусловия по ветвям гиперфункции. Предусловие третье ветви ESint далее определяется как дополнение предусловий первых двух ветвей в рамках общего предусловия.

```
nat max64 = 2**64 – 1; // = ULLONG_MAX
formula pValid() = isDigit(s[ks]) & afterNum() & resN <= max64;
formula pERange() = isDigit(s[ks]) & resN > max64;
lemma Lpre12:  ¬ (pValid() & pERange())
formula pESint() = not pValid() /\ not pERange();
lemma Lpre3: not (pValid() /\ pESint());
formula qValid(nat64 res) =  isDigit(s[ks]) & res = resN
formula qkstrtoull(nat64  res, nat e) =
    (e = 0 -> pValid() /\ qValid(res)) /\
    (e = 1 -> pERange()) /\
    (e = 2 -> pESint());
```

При дедуктивной верификации гиперфункция kstrtoull преобразуется в функцию с дополнительным результатом e. Постусловие qkstrtoull этой функции составляется указанным образом из постусловий и предусловий ветвей.

4.3. Спецификация программы _kstrtoull

```
formula q_kstr(nat64 res, nat c) =
    (c = 0 -> pValid() /\ qValid(res)) /\
    (c = 1 -> pERange()) /\
    (c = 2 -> pESint())
```

Формула q_kstr определяет общее постусловие для гиперфункции _kstrtoull. Предусловия и постусловия ветвей наследуются от главной программы-гиперфункции.

```
_kstrtoull(  : nat64 res #Valid : #ERange : #ESint)
pre Valid: pValid() pre ERange: pERange() pre ESint: pESint()
post Valid: qValid(res)
{
        _parse_integer_fixup_radix( : base, ks);
        _parse_integer(  : nat64 _res, nat32 rv, bool of);
        if (of) #ERange
        else if (rv == 0) #ESint
        else { size_t js2 = ks + rv;
                if (s[js2] == '\n') js3 = js2+1 else js3 = js2;
                if (s[js3] != zero) #ESint
                else {res = _res; #Valid }
        }
}
```

Переменная of истинна в случае переполнения при вычислении значения числа.

4.4. Спецификация программы _parse_integer_fixup_radix

```
_parse_integer_fixup_radix( : nat base, size_t js)
post radix(js0, js, base)
{
        if (base0 == 0) {
                if (s[js0] == '0') {
                        if (s[js0+1]) == 'x' (*& isxdigit(s[js0+2])*)) base = 16
                        else base = 8;
                } else  base = 10;
        };
        if (base == 16 & s[js0] == '0' & s[js0+1] == 'x') js= js0 + 2 else js = js0;
}
```

Вызов isxdigit(s[js0+2]) является избыточным и он удален. Если не шестнадцатеричная цифра, то base = 8 и далее диагностируется синтаксическая ошибка. Функция _parse_integer_fixup_radix находится в интерфейсе пользователя библиотекой ОС Linux. Однако подобная особенность нигде не оговаривается. Поэтому вызов sxdigit(s[js0+2]) не является необходимым и для независимого вызова _parse_integer_fixup_radix.

```
formula isxdigit(char c: bool) = isDigit(c, 16);
```

4.5. Спецификация программы _parse_integer

В данной программе три ошибки. Хотя внесены они вполне сознательно с расчетом: крайне маловероятно, что в реальных приложениях кто-то от них не пострадает. Тип переменная rv выше определен как nat32. При достаточно длинной последовательности цифр значение rv выйдет за границу типа nat32. Оператор rv |= KSTRTOX_OVERFLOW исходной программы также вставлен с расчетом недостижимости 32-го бита значением rv.

При дедуктивной верификации эти ошибки неизбежно фиксируются. Чтобы эти ошибки не мешали проведению верификации, вместо бита KSTRTOX_OVERFLOW введена переменная of, регистрирующая переполнение итогового значения числа. Тип переменной rv определен как **nat**. Переполнение при вычислении целого числа хотя и фиксируется, однако далее вычисление, приводящее к переполнению, все же зачем-то производится. Видимо, в предположении, что оно не приведет к аварийному завершению исполнения. Возможно для того, чтобы отсканировать число до конца. Однако это нигде не оговорено. Чтобы купировать данную ошибку при верификации, для переменной res вместо nat64 используется тип **nat**.

```
formula qParse(nat64 p, nat rv, bool of) =
                rv = n – ks & of = (p>max64) & p = resN;

_parse_integer(  : nat64 p, nat rv, bool of)
post qParse(p, rv, of)
{       parseInt(ks, 0, 0, false: nat64 res, rv, of);
        p = res;
}
```

Проведена замена res & (~0ull << 60) на res >= 2**60. При строгом подходе эквивалентность такой замены необходимо формально доказывать. Занятие это утомительное и неблагодарное. Вызов digitVal заменен на композицию **if** (isDigit(s[js], 16)) digit(s[js], 16, val).

```
formula pParseInt(size_t js, nat64 res, nat rv, bool of0) =
            ks<=js<=n & (∀nat j=ks..js-1. isDigit(s[j])) &
                        numRes(ks, js, 0, res) & rv = js-ks & of0 =(res>max64);
formula qParseInt(size_t js, nat64 res, res', nat rv, rv', of) =
    numRes(js, n, res, res') & rv' = rv + n – js & of = res'>max64;

parseInt (size_t js, nat64 res, nat rv, bool of0: nat res', nat rv', bool of)
pre pParseInt(js, res, rv, of0) post qParseInt(js, res, res', rv, rv', of) measure sL – js;
{       if (isDigit(s[js])) {
                digit(s[js], val);
                nat res1 = res*base + val;
                if (res >= 2**60 & res > (max64 – val) / base) of= true else of = of0;
                parseInt (js+1, res*base + val, rv+1, of0 :  res', rv', of);
        } else { res' = res || rv' = rv || of = of0 }
}
```

5. Процесс дедуктивной верификации

По завершению построения спецификации была проведена генерация формул корректности применением системы правил [7]. Построение формул корректности подробно документировано в Приложении 3. Формулы корректности вместе со спецификацией

собраны в теории, представленные в Приложении 1. Теории были переведены на язык спецификаций системы Why3 [18].

Верификация в системе Why3 оказалась тяжелой. Естественный и привычный стиль спецификации иногда приводит к громоздким доказательствам формул корректности, когда доказательство любого простого утверждения требует длительной работы. Чтобы упростить верификацию, из предикатной программы в максимальной степени экстрагирована ее константная часть в виде модели, описанной в разд. 4.1, Рис. 1. Фактически проведена специализация по аргументам base и js, что позволило упростить программу и спецификацию.

В процессе верификации последовательно исправлялись ошибки в спецификации. Трижды модифицировалась программа parseInt. Применялись разные способы обхода ошибок, указанных в разд. 4.5. Конечные теории по завершению процесса верификации представлены в Приложении 2.

Две леммы NuEq и NuNext, требующие доказательства по индукции, доказаны с помощью аппарата лемма-функций [17]. Для этого строится вспомогательная предикатная программа, спецификация которой совпадает с исходной леммой. Покажем эту ставшую популярной технику на примере леммы NuEq.

```
lemma NuEq :   forall k:int, m:int, res0:int, res:int, res1:int.
          numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
```

Строится следующая предикатная программа.

```
NuEq(int k, m, res, res0, res1)
   post numRes(k, m, res0, res) & numRes(k, m, res0, res1) ⇒ res1 = res;
   measure m
{   if (k = m) {  }  else  { NuEq(k, m-1, res, res0, res1)} }
```

Программа не обязана быть правильной. Поэтому первая ветвь условного оператора пустая. Постусловие здесь в точности формула леммы NuEq. По данной программе генерируется следующая формула корректности:

RB5: k <> m & qNuEq(k, m-1, res, res0, res1) ⇒ qNuEq(k, m, res, res0, res1)

Используя формулу **RB5**, SMT-решатели Z3 и CVC4, входящие в составе системы Why3 [18], уже способны автоматически доказать лемму NuEq.

Соответствующие программы и генерация формул корректности для лемма-функций приведены в конце Приложения 3. Были трудности с построением правильных программ для лемма-функций. При доказательстве последней леммы NuNext обнаружилась ошибка в

индуктивном определении предиката numRes: индуктивно порождаемое множество оказалось пустым.

Дополнительных лемм относительно немного: 4 леммы при доказательстве формул корректности программы parseInt и 15 лемм как расширение исходной модели. В системе Coq проведено только три коротких доказательства.

6. Обзор работ

Трансформации, устраняющие указатели – новое направление исследований. Наиболее близкими здесь являются работы по анализу видов структур данных (shape analysis) [10, 12, 13] и обратной трансляции [14,16] на язык более высокого уровня. Причем в этих работах не ставится задача устранения указателей.

Наборы трансформаций для устранения указателей, применяемых для разных программ на языке Си, отличаются. И пока рано говорить о создании универсальной системы трансформаций. Трансформации для программы конкатенации строк [6] оперируют с несколькими указателями на одном массиве, чего нет в настоящей работе. Трансформации устранения указателей для программы пирамидальной сортировки [5] более просты: там не требуется введения переменных-индексов. В настоящей работе используется больше видов трансформаций. Новыми являются трансформации для параметров-указателей и вхождений указателей в качестве фактических параметров вызовов функций.

Параметр-указатель оказывается необходимым для второго результата функции, поскольку в языке Си всякая функция может иметь единственный результат, передаваемый оператором **return** в теле функции. В языке предикатного программирования P [2] и языке функционального программирования WhyML [18] допускается несколько результатов. Кроме того, в языке P допускается несколько выходов в гиперфункциях, причем каждый выход может иметь несколько результатов. Схожие по выразительности возможности в языке WhyML: исключения с несколькими параметрами.

Вычисление значения целого числа по его строковому представлению представляется в виде стандартной библиотечной функции практически во всех популярных языках программирования. Тем не менее, формальной верификации этой функции, по-видимому, никогда не проводилось.

7. Итоги верификации

Казалось бы, программа kstrtoull проста, и в ней не должно быть ошибок. Однако иногда ошибки вносятся сознательно в целях оптимизации программы по времени исполнению или по размеру объектного кода. Здесь расчет на то, что в реальных приложениях такие ошибки никогда не проявятся. Разумеется, такая практика провоцирует дурной стиль программирования.

Программа _parse_integer содержит следующие ошибки:

1. Тип переменная rv определен как **unsigned int**. При достаточно длинной последовательности нулевых цифр значение rv выйдет за границу типа. Для устранения ошибки необходим контроль выхода за границы типа с выдачей дополнительного флага ошибки. Более правильно было бы определить тип rv как **size_t**, соразмерно типу указателя переменной s.

2. При достаточно длинной последовательности цифр значение переменной rv будет содержать единицу в 32-м бите, что будет диагностировано как переполнение. Поэтому ошибочно использовать оператор rv |= KSTRTOX_OVERFLOW для диагностики переполнения.

3. Контроль переполнения при вычислении целого числа реализуется. Однако вычисление, приводящее к переполнению, все же зачем-то производится. Что, безусловно, ошибочно.

Данные ошибки хорошо известны. Они не исправляются по той причине, что на реальных приложениях эти ошибки не проявились.

4. В верифицируемой программе _parse_integer_fixup_radix удалена проверка условия isxdigit(s[2]). В результате такого изменения, если после «0x» далее следует не шестнадцатеричная цифра, то программа выдаст base=16, хотя ранее выдавала base=8. Для ошибочного числа это не принципиально. Поэтому следует рекомендовать устранить проверку isxdigit(s[2]) в библиотеке стандартных программ ОС Linux.

В остальном, программа kstrtoull в точности соответствует спецификации и не содержит других ошибок.

Список литературы

1. Доказательство правил корректности операторов предикатной программы. [Электронный ресурс]. URL:http://www.iis.nsk.su/persons/vshel/files/rules.zip . (дата обращения 12.08.2020)
2. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Pascal. Версия 0.14. Новосибирск, 2018. 28с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf. (дата обращения 12.12.2020)
3. Чушкин М.С. Система дедуктивной верификации предикатных программ. *Программная инженерия*. 2016. № 5. С. 202-210. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/paper.pdf. (дата обращения 12.12.2020)
4. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования. *Программная инженерия*, 2011, № 2. С. 14-21.
5. Шелехов В.И. Верификация предикатной программы пирамидальной сортировки с применением обратной трансформации. — ИСИ СО РАН, Новосибирск, 2020. 36с. [Электронный ресурс]. URL: https://persons.iis.nsk.su/files/persons/pages/sort9.pdf (дата обращения 12.12.2020)
6. Шелехов В.И. Дедуктивная верификация программы конкатенации строк с применением обратной трансформации // *Знания-Онтологии-Теории (ЗОНТ-19*). — Новосибирск, 2019. — 19с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/logcflc1.pdf. (дата обращения 12.08.2020)
7. Шелехов В.И. Правила доказательства корректности предикатных программ. — Новосибирск, ИСИ СО РАН, 2019. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/prrules.pdf (дата обращения 12.12.2020)
8. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
9. Шелехов В.И., Чушкин М.С. Верификация программы быстрой сортировки с двумя опорными элементами. *Научный сервис в сети Интернет*. М.: ИПМ им. М.В.Келдыша, 2018. 26с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf. (дата обращения 12.08.2020)
10. Boockmann J.H., Lüttgen G., Mühlberg J.T. Generating Inductive Shape Predicates for Runtime Checking and Formal Verification // Leveraging Applications of Formal Methods, Verification and Validation. Verification. 2018. LNCS 11245. P. 64-74.
11. The Coq Proof Assistant. [Электронный ресурс]. URL: https://coq.inria.fr/. (дата обращения 12.08.2020)
12. Haller I., Slowinska A., Bos H. Scalable data structure detection and classification for C/C++ binaries // Empirical Software Engineering. 2016, v. 21, Issue 3. P. 778–810.
13. Jung C., Clark N. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage // 42nd Int. Symposium on Microarchitecture (MICRO 42), NY, 2009. P. 56-66.
14. Novosoft C2J: a C to Java translator. http://www.novosoft-us.com/ solutions/product c2j.shtml, 2001.
15. Shelekhov V. I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, No. 7, P. 421–427.
16. Trudel M., Furia C. A., Nordio M., Meyer B., Oriol M. C to O-O Translation: Beyond the Easy Stuff // 19th Working Conference on Reverse Engineering, 2012. P. 19-28.
17. Volkov, G., Mandrykin, M., Efremov, D.: Lemma functions for Frama-C: C programs as proofs. In: Proceedings of the 2018 Ivannikov ISPRAS Open Conference (ISPRAS-2018). pp. 31–38.
18. Why 3. Where Programs Meet Provers. [Электронный ресурс]. URL:http://why3.lri.fr. (дата обращения 12.08.2020)

Приложение 1

Теории для формул корректности

Определим набор теорий с формулами корректности для всех программ, а также набор формул, вызываемых в формулах корректности, в частности, в предусловиях и постусловиях.

```
theory Base {
char zero = '\0', nul = '0', nl = "\n", iks = "x", plu= "+";
formula isBase(nat base) = base = 8 or base = 10 or base = 16;
type arCh = array(char, nat);
arCh s; (*первый аргумент kstrtoull *)
nat  sL; (* индекс последней литеры исходной строки s*)
axiom ALnat: sL >= 0;
axiom ALs: s[sL] = zero;
nat base0; (*второй аргумент kstrtoull *)
axiom Ab0: base0 = 0 or isBase(base0);
nat js0 = if s[0] = plu then 1 else 0;
formula radix(nat j, k, base) =
        (if base0=0 then
           (if s[j] = '0' then (if s[j+1]='x' then base = 16  else  base =8)
            else base =10 )
         else base = base0)
        &
        (if base=16 & s[j] = '0' & s[j+1='x' then k=j+2 else k=j);
nat  ks; (* индекс начальной цифры в исходной строке s*)
nat base; (*итоговое *)
axiom Kse: radix(js0, ks, base);
formula digitB(char c, nat base, valD);
axiom Adig:  forall char c, nat base, valD. isBase(base) ⇒
         ( 220<=c<220+base <->  digitB(c, base, valD) /\ c = valD + 220)
formula  digit(char c, nat valD) = digitB(c, base, valD);
formula isDigit(char c) = ∃nat valD. digit(c, valD);
formula isDigit16(char c) = ∃nat valD. digitB(c, 16, valD);
formula endNum(nat n) = n >= ks & ¬isDigit(s[n]) & ∀j=ks..n-1. isDigit(s[j]);
nat  n; (* индекс за последней цифрой в строке s*)
axiom Anu: endNum(n);
formula numRes(nat k, m, nat res0, res) =
        if (k= m) res = res0
        else ∃val. digit(s[k], val) & numRes(k+1, res0 * base + val, res);
formula number(nat res) = numRes(k, n, 0, res);
nat resN;
axiom Ares: number(resN);
formula afterNum() = s[n]= zero \/ s[n]=nl /\ s[n+1]= zero;
} Base;
```

Формулы корректности программы **kstrtoull**

```
theory Kstrtoull {
import Base;
```

```
formula pValid() = isDigit(s[ks]) & afterNum() & resN <= max64;
formula pERange() = isDigit(s[ks]) & resN > max64;
lemma Lpre12:  ¬ (pValid() & pERange())
formula pESint() = not pValid() /\ not pERange();
lemma Lpre3: not (pValid() /\ pESint());
formula qValid(nat64 res) =  isDigit(s[ks]) & res = resN
formula qkstrtoull(nat64 res, nat e) =
       (e = 0 ⇒ pValid() & qValid(res)) &
       (e = 1 ⇒ pERange()) &
       (e = 2 ⇒ pESint())
formula S(size_t js) = if s[0] = '+' then js =1 else js =0;
formula q_kstr(nat64 res, nat c) =
    (c = 0 -> pValid() /\ qValid(res)) /\
    (c = 1 -> pERange()) /\
    (c = 2 -> pESint())
HSB2: S(js) ⇒ p_kstr(js);
COR1: S(js0) & qValid( res) & pValid() & e=0 ⇒  qkstrtoull( res, e);
COR2: S(js0) & pERange() & e=1 ⇒ qkstrtoull( res, e);
COR3: S(js0) & pESint() & e=2 ⇒ qkstrtoull(res, e);
}Kstrtoull;
```

Формулы корректности программы _kstrtoull

```
theory Kstrtoull_ {
import Base, Kstrtoull;
type size_t = nat;
formula qParse(nat64 p, nat32 rv, bool of) =
                rv = n – ks & of = (p>max64) & p = resN;
QSB1: p_kstr(js) & radix(js, js1, base1) ⇒ pParse(js1, base1)
formula p3(nat64 _res, nat32 rv, bool of) = radix(js0, ks, base) & qParse( _res, rv, of);

COR4: p3( _res, rv, of) & of & c=1 ⇒ q_kstr(res, c);
COR5: p3( _res, rv, of) & ¬of & rv = 0 & c=2 ⇒ q_kstr(res, c);
formula p4(size_t js2, nat64 _res, nat32 rv, bool of) =
    p3( _res, rv, of) & ¬of & ¬rv = 0 & js2 = ks + rv;
formula S3(size_t js2, js3) = if (s[js2] = '\n') js3 = js2+1 else js3 = js2;

COR6: p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] != zero & c=2 ⇒ q_kstr(res, c);
COR7: p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] = zero & res = _res & c=0  ⇒ q_kstr(res, c);
}Kstrtoull_;
```

Формулы корректности программы _parse_integer_fixup_radix

```
theory Parse_integer_fixup_radix {
import Base, Kstrtoull;
formula S4( nat base) =
    if base0 = 0 then  ( if s[js0] = '0' then
                                ( if s[js0+1]) = 'x' then base = 16 else base = 8)
                       else  base = 10 )
    else base = base0;
```

QS1: ∃ base. S4(base);
formula C2(**nat** base) = base = 16 & s[js0] = '0' & s[js0+1] = 'x';
COR8: S4(base) & C2(base) & js= js0 + 2 ⇒ radix(js0, js, base);
COR9: S4(base) & ¬C2(base) & js = js0 ⇒ radix(js0, js, base);
} Parse_integer_fixup_radix;

Формулы корректности программы _parse_integer

theory Parse_integer {
import Base, Kstrtoull, Kstrtoul_;
formula pParseInt(size_t js, nat64 res, nat32 rv, **bool** of0) =
ks<=js<=n & (∀**nat** j=ks..js-1. isDigit(s[j])) &
numRes(ks, js, 0, res) & rv = js-ks & of0 =(res>max64);
formula qParseInt(size_t js, nat64 res, res', nat32 rv, rv', of) =
numRes(js, n, res, res') & rv' = rv + n – js & of = res'>max64;
QSB2: pParseInt(ks, 0, 0, **false**);
formula qParse(nat64 p, nat32 rv, **bool** of) =
rv = n – ks & of = (p>max64) & p = resN;
COR10: qParseInt(ks, 0, res, 0, rv, of) & p = res ⇒ qParse(p, rv, of);
formula m(size_t js) = sL – js;
RP1: pParseInt(js, res, rv, of0) & ¬isDigit(s[js]) & res' = res & rv' = rv & of = of0 ⇒
qParseInt(js, res, res', rv, rv', of);
formula E6(**nat** res, val, **bool** of0, of1) =
if (res >= 2**60 & res > (max64 – val) / base) **then** of1= **true** **else** of1 = of0
~~**RP3**: pParseInt(js, res, rv) & isDigit(s[js]) & digit(s[js], val) & res1 = res*base + val &~~
~~E6(res,val) & res' = res1 & rv' = rv & of= **true** & jl=js ⇒~~
~~qParseInt(js , jl, res, res', rv, rv', of);~~
formula p6(size_t js, **nat** res, val, nat32 rv, **bool** of0, of1) =
pParseInt(js, res, rv, of0) & isDigit(s[js]) & digit(s[js], val) & E6(res, val, of0, of1);
formula pB(size_t js, **nat** rv) = rv+1 < 2**32;
RB1: p6(js, res, val, rv, of0, of1) ⇒ pParseInt(js+1, res*base + val, rv+1, of1) & m(js+1)<m(js);
RB2: p6(js, res, val, rv, of0, of1) & qParseInt(js+1, res*base + val, res', rv1+1, rv', of) ⇒
qParseInt(js, res, res', rv, rv', of)
}Parse_integer;

Приложение 2

Теории для формул корректности на языке why3

Теории, приведенные в Приложении 2, преобразованы на язык спецификаций системы верификации Why3. Здесь приведена финальная версия после проведнния всех доказательств.

```
theory Base
  use export int.Int
  use export map.Map

  type nat32 = int
  type nat64 = int
  type size_t = int
```

```
type char = int
constant zerO: char = 0
constant nul: char = 220
constant nl: char = 1
constant iks: char = 2
constant plu: char = 3

type arCh = int -> char
constant s: arCh                (*first parameter of kstrtoull*)
constant sL: int
axiom Alnat: sL >= 0
axiom ALs: s[sL] = zerO

predicate isBase(base: int) = base = 8 \/ base = 10 \/ base = 16
constant base0: int           (*second kstrtoull  parameter *)
axiom Ab0: base0 = 0 \/ isBase(base0)
constant js0: int = if s[0] = plu then 1 else 0
lemma Ljs0: js0 <= sL

predicate radix(j k base: int) =
  (if base0=0 then
      (if s[j] = zerO then (if s[j+1]=iks then base = 16  else  base = 8)
       else base =10 )
   else base = base0)
  /\
  (if base=16 /\ s[j] = nul /\ s[j+1] = iks then k=j+2 else k=j)
lemma RaEq: forall k k1 base1 base2: int.
    radix js0 k base1 /\ radix js0 k1 base2 -> k = k1 /\ base1 = base2
lemma RaEx: exists k, base1: int. radix js0 k base1
constant ks: int
constant base: int
axiom Kse: radix js0 ks base
lemma Lba: isBase base
lemma Lks: ks <= sL
lemma Lks0: js0 <= ks

predicate digitB(c: char) (base valD: int)
axiom Adig:  forall c: char, base valD: int. isBase base ->
         ( 220<=c<220+base <->  digitB c base valD /\ c = valD + 220)
predicate  digit(c: char) (valD: int) = digitB c base valD
lemma LA0:   forall valD: int. not (digit zerO valD)
lemma LAnul: digit nul 0
lemma LAv:   forall c: char, base valD: int. digitB c base valD -> 0<=valD<base
predicate isDigit(c: char) = exists valD: int. digit c valD
predicate isDigit16(c: char) = exists valD: int. digitB c 16 valD

predicate endNum(n: int) =
   n >= ks /\ not (isDigit s[n]) /\ (forall j: int. ks<=j<n -> isDigit s[j])
```

```
  lemma LnuEq: forall n1 n2: int. endNum n1 /\ endNum n2 -> n1 = n2
  lemma LnuEx: exists n: int. endNum n
  constant  n: int
  axiom Anu: endNum n
  lemma LnuLs: n <= sL
  lemma LnuKs: ks <= n

  inductive numRes int int int int =
      | Last: forall k m res0: int. k = m -> numRes k m res0 res0
      | Next: forall k m res0 res valD: int.
              k <= m /\ digit s[m] valD /\ numRes k m res0 res ->
                              numRes k (m+1) res0 (res * base + valD)
(*    | Next: forall k m res0 res valD: int.
              k < m /\ digit s[k] valD /\ numRes k m res0 res ->
                                numRes (k+1) m (res0 * base + valD) res *)
  lemma NuEq: forall k m res0 res res1: int.
                    numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
  lemma NuNext: forall res res0 k m valD: int. k < m /\ digit s[k] valD /\
                    (forall j: int. k<=j<m -> isDigit s[j]) /\
                     numRes (k+1) m (res0 * base + valD) res -> numRes k m res0 res
  predicate number(res: int) = numRes ks n 0 res
(*  lemma NuEx: exists res: int. number res *)
  constant resN: int
  axiom Ares: number resN
  predicate afterNum() = ks<n /\ (s[n]= zerO \/ s[n]=nl /\ s[n+1]= zerO)
end (*Base*)

theory Kstrtoull
  use Base
  use export int.EuclideanDivision
  use export bv.Pow2int

  constant max64: int = pow2 64 - 1         (*= ULLONG_MAX*)
  predicate pValid() = afterNum() /\ resN <= max64
  predicate pERange() = resN > max64
  lemma Lpre12: not (pValid() /\ pERange())
(*  predicate pESint() =  not (isDigit s[ks]) \/ not afterNum()*)
  predicate pESint() =  not pValid() /\ not pERange()
  lemma Lpre3: not (pValid() /\ pESint())
  predicate qValid(res: int) = isDigit s[ks] /\ res = resN
  predicate qkstrtoull( res e: int) =
     (e = 0 -> pValid() /\ qValid res) /\
     (e = 1 -> pERange()) /\
     (e = 2 -> pESint())
  predicate sS( js: int) = if s[0]=plu then js = 1 else js = 0
  goal COR1: forall res e: int. sS(js0) /\ qValid(res) /\ pValid() /\ e=0 -> qkstrtoull res e
  goal COR2: forall res e: int. sS(js0) /\ pERange() /\ e=1 -> qkstrtoull res e
  goal COR3: forall res e: int. sS(js0) /\ pESint() /\ e=2 -> qkstrtoull res e
```

```
end (*Kstrtoull*)

theory Kstrtoull_
  use Base
  use Kstrtoull
(*  predicate p_kstr(js: int) = js=js0*)
  predicate q_kstr(res c: int) =
     (c = 0 -> pValid() /\ qValid res) /\
     (c = 1 -> pERange()) /\
     (c = 2 -> pESint())
  predicate qParse(p: int)(rv: nat32)(of: bool) =
     rv = n - ks /\ of = (p>max64) /\ p = resN
  predicate p3(_res: int)(rv: nat32)(of: bool) = radix js0 ks base /\ qParse _res rv of
  goal COR4: forall res _res c: int, rv: nat32, of: bool.
     p3 _res rv of /\ of /\ c=1 -> q_kstr res c
  goal COR5: forall res _res c: int, rv: nat32, of: bool.
     p3 _res rv of /\ not of /\ rv = 0 /\ c=2 -> q_kstr res c
  predicate p4(js2 _res: int)(rv: nat32)(of: bool) =
     p3 _res rv of /\ not of /\ rv <> 0 /\ js2 = ks + rv
  predicate s3(js2 js3: int) = if s[js2] = nl then js3 = js2+1 else js3 = js2
  goal COR6: forall res _res c js2 js3: int, rv: nat32, of: bool.
     p4  js2 _res rv of /\ s3 js2 js3 /\ s[js3] <> zerO /\ c=2 -> q_kstr res c
  goal COR7: forall res _res c js2 js3: int, rv: nat32, of: bool.
     p4  js2 _res rv of /\ s3 js2 js3 /\ s[js3] = zerO /\ res = _res /\ c=0  -> q_kstr res c
end (*Kstrtoull_*)

theory Parse_integer_fixup_radix
  use Base
  use Kstrtoull
  use Kstrtoull_

  predicate s4() =
     if base0 = 0 then  ( if s[js0] = nul then
                                  ( if s[js0+1] = iks then base = 16 else base = 8 )
                          else  base = 10 )
     else base = base0
(*  goal QS1: forall js: int. p_kstr js -> s4() *)
  predicate c2() = base = 16 /\ s[js0] = nul /\ s[js0+1] = iks
  goal COR8: forall js: int.
     s4() /\ c2() /\ js= js0 + 2 -> radix js0 js base
  goal COR9: forall js: int.
     s4() /\ not c2() /\ js= js0 -> radix js0 js base
end  (*Parse_integer_fixup_radix*)

theory Parse_integer
  use Base
  use Kstrtoull
  use Kstrtoull_
```

```
  predicate pParseInt(js res rv: int)(of0: bool) =
    ks<=js<=n /\ (forall j: int. ks<=j<js -> isDigit s[j]) /\
    numRes ks js 0 res /\ rv = js-ks /\ of0 = (res>max64)
  goal QSB2: pParseInt ks 0 0 false
  predicate qParseInt(js res res9: int)(rv rv9: nat32)(of: bool) =
     numRes js n res res9  /\ rv9 = rv + n - js /\ of = (res9 > max64)
  function m(js: int): int = sL - js
  goal COR10: forall p res: int, rv: nat32, of: bool.
     qParseInt ks 0 res 0 rv of /\ p = res -> qParse p rv of
  lemma JsEqN: forall js:int. js>=ks /\ (forall j:int. ks<=j<js -> isDigit s[j]) /\
                  not (isDigit s[js])  -> js = n
  goal RP1: forall js res res9: int, rv rv9: nat32, of of0: bool.
     pParseInt js res rv of0 /\ not (isDigit s[js]) /\ res9 = res /\ rv9 = rv /\ of=of0 ->
          qParseInt js res res9 rv rv9 of
  predicate e6(res valD: int)(of0 of1: bool)  =
     if res >= pow2 60 /\ res > div (max64 - valD) base then of1 = true else of1 = of0

  lemma RnumR: forall js:int, valD:int, res:int.
     js <= n /\ (forall j:int. ks <= j < js -> isDigit s[j]) /\
     numRes ks js 0 res /\ digit s[js] valD  ->
                  numRes ks (js + 1) 0 ((res * base) + valD)
  lemma Rgr60: forall valD:int, res:int.
              0<=valD<base /\ max64 < ((res * base) + valD) -> res >= pow2 60
  predicate p6(js res valD rv: int)(of0 of1: bool) = pParseInt js res rv of0 /\
      isDigit s[js] /\ digit s[js] valD /\  e6 res valD of0 of1
  goal RB1: forall js valD res: int, rv: nat32, of0 of1: bool. p6 js res valD rv of0 of1 ->
                       pParseInt (js+1) (res*base+valD) (rv+1) of1 /\ m(js+1) < m(js)
  lemma Rlsn: forall js: int. js<=n /\ isDigit(s[js]) -> js<n
  goal RB2: forall js valD res res9: int, rv rv9: nat32, of of0 of1: bool.
     p6 js res valD rv of0 of1 /\ qParseInt (js+1) (res*base+valD) res9 (rv+1) rv9 of ->
          qParseInt js res res9 rv rv9 of
end (*Parse_integer*)

theory LemmaFunctions
  use Base
  use Kstrtoull
(*correctness formulas of lemma function for the lemma:
  lemma NuNext: forall res:int, res0:int, k:int, m:int, valD:int.
   k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res ->
                     numRes k m res0 res
*)
  predicate pNun(k m valD res0 res: int) =
      k < m /\ digit s[k] valD /\ (forall j: int. k <=j< m -> isDigit s[j])
  predicate qNun(k m valD res0 res: int) =
           numRes (k + 1) m ((res0 * base) + valD) res -> numRes k m res0 res
  goal COR11: forall k m valD res0 res: int.
              pNun k m valD res0 res  /\ m = k+1 -> qNun k m valD res0 res
  goal QSB3:  forall k m valD res0 res res1: int.
```

```
                pNun k m valD res0 res  /\ m <> k+1 ->
                    pNun k (m-1) valD res0 res1 /\ isDigit s[m-1]
  goal COR12: forall k m valD va res0 res res1: int.
          pNun k m valD res0 res /\ m <> (k+1) /\ qNun k (m-1) valD res0 res1 /\
               digit s[m-1] va /\ res = ((res1*base)+va) -> qNun k m valD res0 res

(* correctness formulas of lemma function for the lemma:
   lemma NuEq: forall k m res0 res res1: int.
                   numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
*)
  predicate qNuEq(k m res res0 res1: int) =
        numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
  goal COR13: forall k m res0 res res1: int. k = m -> qNuEq k m res res0 res1
  goal RB5: forall k m res0 res res1: int.
             k <> m /\ qNuEq k (m-1) res res0 res1 -> qNuEq k m res res0 res1

end (*LemmaFunctions*)
```

Приложение 3

Генерация формул корректности

Приведем набор определений, необходимых для верификации программы kstrtoull и используемых в предусловиях и постусловиях.

```
char zero = '\0', nul = '0', nl = "\n", iks = "x", plu= "+";
formula isBase(nat base) = base = 8 or base = 10 or base = 16;
type arCh = array(char, nat);
arCh s; (*первый аргумент kstrtoull *)
nat  sL; (* индекс последней литеры исходной строки s*)
axiom ALnat: sL >= 0;
axiom ALs: s[sL] = zero;
nat base0; (*второй аргумент kstrtoull *)
axiom Ab0: base0 = 0 or isBase(base0);
nat js0 = if s[0] = plu then 1 else 0;
formula radix(nat j, k, base) =
        (if base0=0 then
           (if s[j] = '0' then (if s[j+1]='x' then base = 16  else  base =8)
            else base =10 )
         else base = base0)
        &
        (if base=16 & s[j] = '0' & s[j+1='x' then k=j+2 else k=j);
nat  ks; (* индекс начальной цифры в исходной строке s*)
nat base; (*итоговое *)
axiom Kse: radix(js0, ks, base);
formula digitB(char c, nat base, valD);
axiom Adig:  forall char c, nat base, valD. isBase(base) ⇒
         ( 220<=c<220+base <->  digitB(c, base, valD) /\ c = valD + 220)
formula  digit(char c, nat valD) = digitB(c, base, valD);
formula isDigit(char c) = ∃nat valD. digit(c, valD);
formula isDigit16(char c) = ∃nat valD. digitB(c, 16, valD);
formula endNum(nat n) = n >= ks & ¬isDigit(s[n]) & ∀j=ks..n-1. isDigit(s[j]);
nat  n; (* индекс за последней цифрой в строке s*)
axiom Anu: endNum(n);
formula numRes(nat k, m, nat res0, res) =
        if (k= m) res = res0
        else ∃val. digit(s[k], val) & numRes(k+1, res0 * base + val, res);
formula number(nat res) = numRes(k, 0, res);
nat resN;
axiom Ares: number(resN);
formula afterNum() = s[n]= zero \/ s[n]=nl /\ s[n+1]= zero;
```

Формулы корректности программы **kstrtoull**

```
nat max64 = 2**64 – 1; // = ULLONG_MAX
formula pValid() = isDigit(s[ks]) & afterNum() & resN <= max64;
formula pERange() = isDigit(s[ks]) & resN > max64;
lemma Lpre12:  ¬ (pValid() & pERange())
formula pESint() = not pValid() /\ not pERange();
lemma Lpre3: not (pValid() /\ pESint());
formula qValid(nat64 res) =  isDigit(s[ks]) & res = resN

kstrtoull( : nat64 res #Valid : #ERange : #ESint)
pre Valid: pValid() pre ERange: pERange() pre ESint: pESint()
post Valid: qValid(res)
{       if (s[0] == '+') js0 = 1 else js0 =0;
        _kstrtoull( : res #Valid : #ERange : #ESint);
}
```

Построим правила корректности для гиперфункции kstrtoull.

Для гиперфункции вводим параметр-результат e – номер ветви гиперфункции со значениями 0, 1 и 2. Постусловие для гиперфункции:

```
formula qkstrtoull(nat64 res, nat e) =
      (e = 0 ⇒ pValid() & qValid(res)) &
      (e = 1 ⇒ pERange()) &
      (e = 2 ⇒ pESint())
```

Применим правило **QS**.

$$\mathbf{QS}:\ \frac{\begin{array}{l}P(x) \Rightarrow \exists z.\ B(x\mathbf{:}\ z);\\ \forall z.\ C(x, z\mathbf{:}\ y)\ \mathbf{corr}\ [P(x)\ \&\ B(x\mathbf{:}\ z),\ Q(x, y)];\end{array}}{B(x\mathbf{:}\ z);\ C(x, z\mathbf{:}\ y)\ \mathbf{corr}\ [P(x),\ Q(x,y)]}$$

Конкретизация правила:

$$\mathbf{QS}:\ \frac{\begin{array}{l}\exists\ js.\ \mathbf{if}\ (s[0] = '+')\ js0 = 1\ \mathbf{else}\ js0 =0;\\ \_kstrtoull(\ \mathbf{:}\ res\ \#Valid\ \mathbf{:}\ \#ERange\ \mathbf{:}\ \#ESint)\ \mathbf{corr}\\ [S(js0),\ qkstrtoull(res, e)];\end{array}}{\begin{array}{l}\mathbf{if}\ (s[0] = '+')\ js0 = 1\ \mathbf{else}\ js0 =0;\ \_kstrtoull(\ \mathbf{:}\ res\ \#Valid\ \mathbf{:}\ \#ERange\ \mathbf{:}\ \#ESint)\ \mathbf{corr}\\ [\mathbf{true},\ qkstrtoull(\ res,\ e)]\end{array}}$$

```
formula S(size_t js) = (s[0] = '+' ⇒ js =1) & (¬s[0] = '+' ⇒ js =0);
```

Вторая посылка правила определяет новую цель:

```
_kstrtoull(  : res #Valid : #ERange : #ESint) corr [S(js0), qkstrtoull( res, e)]
```

Применим правило **HSB** [27, Раздел 7].

HSB:
$$\frac{\begin{array}{l}B(x\textbf{:}\ y, z, e)\ \textbf{corr}^*\ [P(x), Q(x, y, z, e)];\qquad P1(x) \Rightarrow P^*(x);\\ C(y\textbf{:}\ u)\ \textbf{corr}\ [P1(x)\ \&\ S(x, y)\ \&\ E(x), Q1(x,v,u)];\\ D(z\textbf{:}\ u)\ \textbf{corr}\ [P1(x)\ \&\ R(x, z)\ \&\ \neg E(x), Q1(x,v,u)]\end{array}}{B(x\textbf{:}\ y\ \#M1\textbf{:}\ z\ \#M2)\ \textbf{case}\ M1:\ C(y\textbf{:}\ u)\ \textbf{case}\ M2:\ D(z\textbf{:}\ u)\ \ \textbf{corr}\ [P1(x), Q1(x,v,u)]}$$

Здесь B – гиперфункция вида

B(x**:** y #1**:** z #2) **pre** P(x) **pre** 1**:** E(x) **post** 1: S(x, y) **post** 2: R(x, z){ … };

Вызов гиперфункции преобразуется к виду:

_kstrtoull(**:** res #M1 **:** #M2 **:** #M3) **case** M1:{e=0} **case** M2: {e=1} **case** M3:{e=2}

Определим предусловие и постусловие для _kstrtoull.

```
formula p_kstr(size_t js) = true;
formula q_kstr(nat64 res, nat c) =
    (c = 0 -> pValid() /\ qValid(res)) /\
    (c = 1 -> pERange()) /\
    (c = 2 -> pESint())
```

Ниже конкретизация правила **HSB**.

HSB:
$$\frac{\begin{array}{l}\text{\_kstrtoull(\textbf{:} res \#M1 \textbf{:} \#M2 \textbf{:} \#M3)}\ \textbf{corr}^*\ [p\_kstr(js), q\_kstr(res, c)];\\ S(js0) \Rightarrow p\_kstr(js, base);\\ e=0\ \textbf{corr}\ [S(js0)\ \&\ qValid(\ res)\ \&\ pValid(),\ \ qkstrtoull(\ res, e)];\\ e=1\ \textbf{corr}\ [S(js0)\ \&\ pERange(), qkstrtoull(\ res, e)]\\ e=2\ \textbf{corr}\ [S(js0)\ \&\ pESint(), qkstrtoull(\ res, e)]\end{array}}{\begin{array}{l}\text{\_kstrtoull(\textbf{:} res \#M1 \textbf{:} \#M2 \textbf{:} \#M3)}\\ \textbf{case}\ M1:\{e=0\}\ \textbf{case}\ M2:\ \{e=1\}\ \textbf{case}\ M3:\{e=2\}\ \textbf{corr}\\ [S(js0), qkstrtoull(\ res, e)]\end{array}}$$

Вторая посылка правила определяет формулу корректности:

~~**HSB2:** S(js) ⇒ p_kstr(js);~~

Для посылок 3-5 применяется правило **COR**.

COR:
$$\frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)\end{array}}{H(x\textbf{:}\ y)\ \textbf{corr}\ \ [P(x), Q(x, y)]}$$

Конкретизация правила для трех посылок:

COR:
$$\frac{\begin{array}{l}S(js0)\ \&\ qValid(\ res)\ \&\ pValid()\ \&\ e=0 \Rightarrow\ \ qkstrtoull(\ res, e);\\ S(js0)\ \&\ qValid(\ res)\ \&\ pValid() \Rightarrow \exists\ e.\ e=0\end{array}}{e=0\ \textbf{corr}\ [S(js0)\ \&\ qValid(\ res)\ \&\ pValid(), qkstrtoull(\ res, e)];}$$

COR:
$$\frac{\begin{array}{l}S(js0)\ \&\ pERange()\ \&\ e=1 \Rightarrow\ \ qkstrtoull(\ res, e);\\ S(js0)\ \&\ pERange() \Rightarrow \exists\ e.\ e=1\end{array}}{e=1\ \textbf{corr}\ [S(js0)\ \&\ pERange(), qkstrtoull(\ res, e)]}$$

COR: $\dfrac{\begin{array}{l}S(js0)\ \&\ pESint()\ \&\ e{=}2 \Rightarrow qkstrtoull(res, e);\\ S(js0)\ \&\ pESint() \Rightarrow \exists\ e.\ e{=}2\end{array}}{e{=}2\ \textbf{corr}\ [S(js0)\ \&\ pESint(), qkstrtoull(res, e)];}$

Первые посылки в каждой их трех конкретизаций дают формулы корректности:

COR1: S(js0) & qValid(res) & pValid() & e=0 ⇒ qkstrtoull(res, e);
COR2: S(js0) & pERange() & e=1 ⇒ qkstrtoull(res, e);
COR3: S(js0) & pESint() & e=2 ⇒ qkstrtoull(res, e);

Формулы корректности программы _kstrtoull

```
_kstrtoull(  : nat64 res #Valid : #ERange : #ESint)
pre Valid: pValid() pre ERange: pERange() pre ESint: pESint()
post Valid: qValid(res)
{
        _parse_integer_fixup_radix(js0: base, ks);
        _parse_integer(ks, base1: nat64 _res, nat32 rv, bool of);
        if (of) #ERange
        else if (rv == 0) #ESint
        else { size_t js2 = ks + rv;
                if (s[js2] == '\n') js3 = js2+1 else js3 = js2;
                if (s[js3] != zero) #ESint
                else {res = _res; #Valid }
        }
}

 predicate p_kstr(js: int) = js=js0
formula q_kstr(nat64 res, nat c) =
    (c = 0 -> pValid() /\ qValid(res)) /\
    (c = 1 -> pERange()) /\
    (c = 2 -> pESint())
nat sizemax;
type size_t = subtype(nat x: x<= sizemax);
```

Для тела программы _kstrtoull применяется правило **QSB**.

QSB: $\dfrac{\begin{array}{l}B(x: z)\ \textbf{corr}^*\ [P_B(x), Q_B(x, z)];\ P(x) \Rightarrow P^*_B(x);\\ \forall z.\ C(x, z: y)\ \textbf{corr}\ [P(x)\ \&\ Q_B(x, z), Q(x, y)];\end{array}}{B(x: z);\ C(x, z: y)\ \textbf{corr}\ [P(x), Q(x,y)]}$

Конкретизация правила **QSB**.

_parse_integer_fixup_radix(**:** base, ks) **corr***
[**true**, radix(js0, ks, base)];
p_kstr(js, base) ⇒ p_kstr(js, base);
QSB: Z **corr** [radix(js0, ks, base), q_kstr(res, c)];
_parse_integer_fixup_radix(**:** base, ks); Z **corr**
[**true**, q_kstr(res, c)]

Здесь и далее Z обозначает оставшуюся часть программы. Третья посылка определяет новую цель.

_parse_integer(**:** nat64 _res, nat32 rv, **bool** of); Z **corr**
[radix(js0, ks, base), q_kstr(res, c)];

Применяется правило **QSB**.

$$\textbf{QSB:}\ \frac{\begin{array}{l}B(x{:}\ z)\ \textbf{corr}^*\ [P_B(x), Q_B(x, z)];\ P(x) \Rightarrow P^*_B(x);\\ \forall z.\ C(x, z{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ Q_B(x, z), Q(x, y)];\end{array}}{B(x{:}\ z);\ C(x, z{:}\ y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

formula qParse(nat64 p, nat32 rv, **bool** of) =
rv = n – ks & of = (p>max64) & (p>max64 ⇒ p = resN);
_parse_integer(~~size_t js, base~~ **:** nat64 p, nat32 rv, **bool** of)
~~**pre** isBase(base) & js=ks~~ **post** qParse(p, rv, of)

Конкретизация правила **QSB**.

_parse_integer(**:** _res, rv, of) **corr*** [**true**, qParse(_res, rv, of)];
radix(js0, ks, base) ⇒ **true**;
QSB: Z **corr** [radix(js0, ks, base) & qParse(_res, rv, of), q_kstr(res, c)];
_parse_integer(**:** _res, rv, of); Z **corr** [radix(js0, ks, base), q_kstr(res, c)]

Вторая посылка определяет формулу корректности:

~~**QSB1**: p_kstr(js) & radix(js, js1, base1) ⇒ pParse(js1, base1)~~

Третья посылка определяет новую цель.

formula p3(nat64 _res, nat32 rv, **bool** of) = radix(js0, ks, base) & qParse(_res, rv, of);
if (of) #ERange **else** Z **corr** [p3(_res, rv, of), q_kstr(res, c)];

Применяется правило **QC**.

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x), Q(x, y)];\\ C(x{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x), Q(x, y)]\end{array}}{\{\textbf{if}\ (E(x))\ B(x{:}\ y)\ \textbf{else}\ C(x{:}\ y)\}\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Конкретизация правила **QC**:

$$\textbf{QC:}\ \frac{\begin{array}{l}c{=}1\ \textbf{corr}\ [p3(\ \_res, rv, of)\ \&\ of, q\_kstr(res, c)];\\ Z\ \ \textbf{corr}\ [p3(\ \_res, rv, of)\ \&\ \neg\ of, q\_kstr(res, c)]\end{array}}{\textbf{if}\ (of)\ c{=}1\ \textbf{else}\ Z\ \textbf{corr}\ [p3(\ \_res, rv, of), q\_kstr(res, c)]}$$

К первой посылке применим правило **COR**.

$$\textbf{COR:}\ \frac{\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x,y);\quad \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)}{H(x\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{p3(\ \_res, rv, of)\ \&\ of\ \&\ c{=}1 \Rightarrow q\_kstr(res, c);\quad p3(\ \_res, rv, of)\ \&\ of \Rightarrow \exists\ c.\ c{=}1}{c{=}1\ \textbf{corr}\ [p3(\ \_res, rv, of)\ \&\ of, q\_kstr(res, c)]}$$

Первая посылка определяет формулу корректности.

COR4: p3(_res, rv, of) & of & c=1 ⇒ q_kstr(res, c);

Вторая посылка правила **QC** определяет новую цель.

if (rv = 0) #ESint **else** Z **corr** [p3(_res, rv, of) & ¬ of, q_kstr(res, c)]

Применяется правило **QC**.

$$\textbf{QC:}\ \frac{B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x), Q(x,y)];\quad C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x), Q(x,y)]}{\{\textbf{if}\ (E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила **QC**:

$$\textbf{QC:}\ \frac{c{=}2\ \textbf{corr}\ [p3(\ \_res, rv, of)\ \&\ \neg of\ \&\ rv = 0, q\_kstr(res, c)];\quad Z\ \ \textbf{corr}\ [p3(\ \_res, rv, of)\ \&\ \neg of\ \&\ \neg rv = 0, q\_kstr(res, c)]}{\textbf{if}\ (rv = 0)\ c{=}2\ \textbf{else}\ Z\ \textbf{corr}\ [p3(\ \_res, rv, of)\ \&\ \neg\ of, q\_kstr(res, c)]}$$

К первой посылке применим правило **COR**.

$$\textbf{COR:}\ \frac{\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x,y);\quad \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)}{H(x\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{p3(\ \_res, rv, of)\ \&\ \neg of\ \&\ rv = 0\ \&\ c{=}2 \Rightarrow q\_kstr(res, c);\quad p3(\ \_res, rv, of)\ \&\ \neg of\ \&\ rv = 0 \Rightarrow \exists\ c.\ c{=}2}{c{=}2\ \textbf{corr}\ [p3(\ \_res, rv, of)\ \&\ \neg of\ \&\ rv = 0, q\_kstr(res, c)];}$$

Первая посылка определяет формулу корректности.

COR5: p3(_res, rv, of) & ¬of & rv = 0 & c=2 ⇒ q_kstr(res, c);

Вторая посылка правила **QC** определяет новую цель.

size_t js2 = ks + rv; Z **corr** [p3(_res, rv, of) & ¬of & ¬rv = 0, q_kstr(res, c)]

Применяется правило **QS**.

$$\textbf{QS:}\ \frac{P(x) \Rightarrow \exists z.\ B(x\textbf{:}\ z);\quad \forall z.\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ B(x\textbf{:}\ z), Q(x,y)];}{B(x\textbf{:}\ z);\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

QS: $\dfrac{\begin{array}{l} p3(\ \_res, rv, of) \ \&\ \neg of \ \&\ \neg rv = 0 \Rightarrow \exists js2.\ js2 = ks + rv; \\ Z\ \ \textbf{corr}\ [p3(\ \_res, rv, of) \ \&\ \neg of \ \&\ \neg rv = 0 \ \&\ js2 = ks + rv,\ q\_kstr(res, c)]; \end{array}}{js2 = ks + rv;\ Z\ \ \textbf{corr}\ [p3(\ \_res, rv, of) \ \&\ \neg of \ \&\ \neg rv = 0,\ q\_kstr(res, c)]}$

Вторая посылка определяет новую цель.

formula p4(size_t js2, nat64 _res, nat32 rv, **bool** of) =
p3(_res, rv, of) & ¬of & ¬rv = 0 & js2 = ks + rv;
if (s[js2] == '\n') js3 = js2+1 **else** js3 = js2; Z **corr** [p4(js2, _res, rv, of), q_kstr(res, c)];
formula S3(size_t js2, js3) = **if** (s[js2] = '\n') js3 = js2+1 **else** js3 = js2;

Применяется правило **QS**.

$$\textbf{QS:}\ \frac{\begin{array}{l} P(x) \Rightarrow \exists z.\ B(x\textbf{:}\ z); \\ \forall z.\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x) \ \&\ B(x\textbf{:}\ z),\ Q(x, y)]; \end{array}}{B(x\textbf{:}\ z);\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,y)]}$$

Конкретизация правила:

QS: $\dfrac{\begin{array}{l} p4(js2,\ \_res, rv, of) \Rightarrow \exists js3.\ \textbf{if}\ (s[js2] = '\backslash n')\ js3 = js2+1\ \textbf{else}\ js3 = js2; \\ Z\ \textbf{corr}\ [p4(js2, \_res, rv, of) \ \&\ S3(js2, js3),\ q\_kstr(res, c)]; \end{array}}{\textbf{if}\ (s[js2] == '\backslash n')\ js3 = js2+1\ \textbf{else}\ js3 = js2;\ Z\ \textbf{corr}\ [p4(js2, \_res, rv, of),\ q\_kstr(res, c)]}$

Вторая посылка определяет новую цель.

if (s[js3] != zero) #ESint **else** {res = _res; #Valid } **corr**
[p4(js2, _res, rv, of) & S3(js2, js3), q_kstr(res, c)];

Применяется правило **QC**.

$$\textbf{QC:}\ \frac{\begin{array}{l} B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x) \ \&\ E(x),\ Q(x, y)]; \\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x) \ \&\ \neg E(x),\ Q(x, y)] \end{array}}{\{\textbf{if}\ (E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x),\ Q(x, y)]}$$

Конкретизация правила **QC**:

QC: $\dfrac{\begin{array}{l} c=2\ \textbf{corr}\ [p4(js2, \_res, rv, of) \ \&\ S3(js2, js3) \ \&\ s[js3] != zero,\ q\_kstr(res, c)]; \\ res = \_res;\ \#Valid\ \ \textbf{corr}\ [p4(js2, \_res, rv, of) \ \&\ S3(js2, js3) \ \&\ s[js3] = zero,\ q\_kstr(res, c)] \end{array}}{\begin{array}{l} \textbf{if}\ (s[js3] != zero)\ \#ESint\ \textbf{else}\ \{res = \_res;\ \#Valid\ \}\ \textbf{corr} \\ [p4(js2, \_res, rv, of) \ \&\ S3(js2, js3),\ q\_kstr(res, c)] \end{array}}$

К первой посылке применим правило **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l} \forall x,y.\ P(x) \ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x, y); \\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y) \end{array}}{H(x\textbf{:}\ y)\ \textbf{corr}\ \ [P(x),\ Q(x, y)]}$$

Конкретизация правила:

COR: $\dfrac{\begin{array}{l} p4(js2, \_res, rv, of) \ \&\ S3(js2, js3) \ \&\ s[js3] != zero \ \&\ c=2 \Rightarrow q\_kstr(res, c); \\ p4(js2, \_res, rv, of) \ \&\ S3(js2, js3) \ \&\ s[js3] != zero \Rightarrow \exists\ c.\ c=2 \end{array}}{c=2\ \textbf{corr}\ [p4(js2, \_res, rv, of) \ \&\ S3(js2, js3) \ \&\ s[js3] != zero,\ q\_kstr(res, c)];}$

Первая посылка определяет формулу корректности.

COR6: p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] != zero & c=2 ⇒ q_kstr(res, c);

Вторая посылка правила **QC** определяет новую цель.

res = _res; #Valid **corr** [p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] = zero, q_kstr(res, c)];

Применяется правило **QS**.

$$\textbf{QS:}\ \frac{\begin{array}{l}P(x) \Rightarrow \exists z.\ B(x\textbf{:}\ z);\\ \forall z.\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x) \ \&\ B(x\textbf{:}\ z),\ Q(x, y)];\end{array}}{\begin{array}{l}B(x\textbf{:}\ z);\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,y)]\end{array}}$$

Конкретизация правила:

$$\textbf{QS:}\ \frac{\begin{array}{l}p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero \Rightarrow \exists res.\ res = \_res;\\ c=0\ \textbf{corr}\ [p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero\ \&\ res = \_res,\ q\_kstr(res, c)];\end{array}}{\begin{array}{l}res = \_res;\ \#Valid\ \ \textbf{corr}\ [p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero,\ q\_kstr(res, c)]\end{array}}$$

Для второй посылки применяется **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)\end{array}}{\begin{array}{l}H(x\textbf{:}\ y)\ \textbf{corr}\ \ [P(x),\ Q(x, y)]\end{array}}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{\begin{array}{l}p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero\ \&\ res = \_res\ \&\ c=0\ \Rightarrow q\_kstr(res, c);\\ p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero\ \&\ res = \_res \Rightarrow \exists\ c.\ c=0\end{array}}{\begin{array}{l}c=0\ \textbf{corr}\ [p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero\ \&\ res = \_res,\ q\_kstr(res, c)]\end{array}}$$

Первая посылка определяет формулу корректности.

COR7: p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] = zero & res = _res & c=0 ⇒ q_kstr(res, c);

Формулы корректности программы _parse_integer_fixup_radix

```
_parse_integer_fixup_radix( : nat base, size_t js)
pre js=js0 post radix(js0, js, base)
{
        if (base0 == 0) {
                if (s[js0] == '0') {
                        if (s[js0+1]) == 'x' (*& isxdigit(s[js+2])*)) base = 16
                        else base = 8;
                } else  base = 10;
        };
        if (base == 16 & s[js0] == '0' & s[js0+1] == 'x') js= js0 + 2 else js = js0;
}

formula S4( nat base) =
    if base0 = 0 then  ( if s[js0] = '0' then
                                    ( if s[js0+1]) = 'x' then base = 16 else base = 8)
                               else  base = 10 )
    else base = base0;
```

Для тела программы _parse_integer_fixup_radix применяется правило **QS**.

$$\textbf{QS:}\ \frac{\begin{array}{l}P(x) \Rightarrow \exists z.\ B(x\textbf{:}\ z);\\ \forall z.\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ B(x\textbf{:}\ z), Q(x, y)];\end{array}}{\begin{array}{l}B(x\textbf{:}\ z);\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x,y)]\end{array}}$$

Конкретизация правила:

$$\textbf{QS:}\ \frac{\begin{array}{l}\exists\ base.\ S4(base);\\ Z\ \textbf{corr}\ [S4(base), radix(js0, js, base)];\end{array}}{\begin{array}{l}\textbf{if}\ (base0 == 0)\ldots;\ Z\ \textbf{corr}\ [\textbf{true}, radix(js0, js, base)]\end{array}}$$

Первая посылка определяет формулу корректности:

QS1: ∃ base. S4(base);

Для второй посылки применяется **QC**.

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x), Q(x, y)];\\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x), Q(x, y)]\end{array}}{\begin{array}{l}\{\textbf{if}\ (E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x), Q(x, y)]\end{array}}$$

formula C2(**nat** base) = base = 16 & s[js0] = '0' & s[js0+1] = 'x';

Конкретизация правила **QC**:

$$\textbf{QC:}\ \frac{\begin{array}{l}js= js0 + 2\ \textbf{corr}\ [S4(base)\ \&\ C2(base), radix(js0, js, base)];\\ js = js0\ \ \textbf{corr}\ [S4(base)\ \&\ \neg C2(base), radix(js0, js, base)];\end{array}}{\begin{array}{l}\textbf{if}\ (base == 16\ \&\ s[js0] == '0'\ \&\ s[js0+1] == 'x')\ js= js0 + 2\ \textbf{else}\ js = js0\ \textbf{corr}\\ [S4(base), radix(js0, js, base)]\end{array}}$$

К первой и второй посылке применим правило **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)\end{array}}{\begin{array}{l}H(x\textbf{:}\ y)\ \textbf{corr}\ \ [P(x), Q(x, y)]\end{array}}$$

Конкретизации правила **COR**:

$$\textbf{COR:}\ \frac{\begin{array}{l}S4(base)\ \&\ C2(base)\ \&\ js= js0 + 2 \Rightarrow radix(js0, js, base);\\ S4(base)\ \&\ C2(base) \Rightarrow \exists\ js'.\ js= js0 + 2\end{array}}{\begin{array}{l}js= js0 + 2\ \textbf{corr}\ [S4(base)\ \&\ C2(base), radix(js0, js, base)];\end{array}}$$

$$\textbf{COR:}\ \frac{\begin{array}{l}S4(base)\ \&\ \neg C2(base)\ \&\ js = js0 \Rightarrow radix(js0, js, base);\\ S4(base)\ \&\ \neg C2(base) \Rightarrow \exists\ js.\ js = js0\end{array}}{\begin{array}{l}js = js0\ \ \textbf{corr}\ [S4(base)\ \&\ \neg C2(base), radix(js0, js, base)];\end{array}}$$

Первые посылки в каждой из конкретизаций дают формулы корректности:

COR8: S4(base) & C2(base) & js= js0 + 2 ⇒ radix(js0, js, base);
COR9: S4(base) & ¬C2(base) & js = js0 ⇒ radix(js0, js, base);

Формулы корректности программы _parse_integer

```
formula qParse(nat64 p, nat32 rv, bool of) =
                rv = n – ks & of = (p>max64) & p = resN;
_parse_integer(  : nat64 p, nat32 rv, bool of)
pre isBase(base) & js=ks post qParse(p, rv, of)
{       parseInt(ks, 0, 0, false: nat64 res, rv, of);
        p = res;
}
```

Для тела программы _parse_integer применяется правило **QSB**.

$$\textbf{QSB:}\ \frac{\begin{array}{l}B(x: z)\ \textbf{corr}^*\ [P_B(x), Q_B(x, z)];\ P(x) \Rightarrow P^*_B(x);\\ \forall z.\ C(x, z: y)\ \textbf{corr}\ [P(x)\ \&\ Q_B(x, z), Q(x, y)];\end{array}}{B(x: z);\ C(x, z: y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила **QSB**.

$$\textbf{QSB:}\ \frac{\begin{array}{l}\text{parseInt(ks, 0, 0, false: nat64 res, rv, of) } \textbf{corr}^*\\ \text{[pParseInt(ks, 0, 0, false), qParseInt(ks, 0, res, 0, rv, of)];}\\ \text{pParseInt(ks, 0, 0, false);}\\ \text{p = res } \textbf{corr} \text{ [qParseInt(ks, 0, res, 0, rv, of), qParse(p, rv, of)];}\end{array}}{\text{parseInt(ks, 0, 0, false: nat64 res, rv, of); p = res } \textbf{corr} \text{ [true, qParse(p, rv, of)]}}$$

Вторая посылка определяет формулу корректности:

QSB2: pParseInt(ks, 0, 0, false);

Для третьей посылки применяется правило **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)\end{array}}{H(x\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Конкретизация правила **COR**:

$$\textbf{COR:}\ \frac{\begin{array}{l}\text{qParseInt(ks, 0, res, 0, rv, of) \& p = res} \Rightarrow \text{qParse(p, rv, of)}\\ \text{qParseInt(ks, 0, res, 0, rv, of)} \Rightarrow \exists\ \text{p. p = res}\end{array}}{\text{p = res } \textbf{corr} \text{ [qParseInt(ks, 0, res, 0, rv, of), qParse(p, rv, of)]}}$$

Первая посылка дает формулу корректности:

COR10: qParseInt(ks, 0, res, 0, rv, of) & p = res ⇒ qParse(p, rv, of);

Построим формулы корректности для программы parseInt.

```
formula m(size_t js) = sL – js;
formula pParseInt(size_t js, nat64 res, nat32 rv, bool of0) =
            ks<=js<=n & (∀nat j=ks..js-1. isDigit(s[j])) &
                        numRes(ks, js, 0, res) & rv = js-ks & of0 =(res>max64);
formula qParseInt(size_t js, nat64 res, res', nat32 rv, rv', of) =
    numRes(js, n, res, res') & rv' = rv + n – js & of = res'>max64;

parseInt (size_t js, nat64 res, nat32 rv, bool of0: nat res', nat32 rv', bool of)
```

```
pre pParseInt(js, res, rv, of0) post qParseInt(js, res, res', rv, rv', of) measure sL – js;
{       if (isDigit(s[js])) {
                digit(s[js], val);
                nat res1 = res*base + val;
                if (res >= 2**60 & res > (max64 – val) / base) of1= true else of1 = of0;
                parseInt (js+1, res*base + val, rv+1, of1 :  res', rv', of);
        } else { res' = res || rv' = rv || of = of0 }
}
```

Для тела программы parseInt применяется правило **QC**.

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x),\ Q(x,\ y)];\\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x),\ Q(x,\ y)]\end{array}}{\{\textbf{if}\ (E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Конкретизация правила **QC**:

$$\textbf{QC:}\ \frac{\begin{array}{l}Z1\ \textbf{corr}\ \ [pParseInt(js,\ res,\ rv,\ of0)\ \&\ isDigit(s[js]),\ \ qParseInt(js,\ res,\ res',\ rv,\ rv',\ of)];\\ Z2\ \ \textbf{corr}\ \ [pParseInt(js,\ res,\ rv,\ of0)\ \&\ \neg isDigit(s[js]),\ \ qParseInt(js,\ res,\ res',\ rv,\ rv',\ of)];\end{array}}{\textbf{if}\ (isDigit(s[js]))\ \ Z1\ \textbf{else}\ \ Z2\ \textbf{corr}\ \ [pParseInt(js,\ res,\ rv,\ of0),\ qParseInt(js,\ res,\ res',\ rv,\ rv',\ of)]}$$

Посылки правила **QC** определяют две новые цели.

```
digit(s[js], val);Z1 corr
[pParseInt(js, res, rv, of0) & isDigit(s[js]),  qParseInt(js, res, res', rv, rv', of)]
res' = res || rv' = rv || of = of0  corr
        [pParseInt(js, res, rv, of0) & ¬isDigit(s[js]),  qParseInt(js, res, res', rv, rv', of)]
```

Для второй цели применим правило **RP**.

$$\textbf{RP:}\ \frac{\begin{array}{l}B(x\textbf{:}\ y)\ \textbf{corr*}\ [P_B(x),\ Q_B(x,\ y)];\\ C(x\textbf{:}\ z)\ \textbf{corr*}\ [P_C(x),\ Q_C(x,\ z)];\\ \forall y,\ z\ (P(x)\ \&\ Q_B(x,\ y)\ \&\ Q_C(x,\ z) \Rightarrow Q(x,\ y,\ z));\\ P(x) \Rightarrow P^*_B(x)\ \&\ P^*_C(x);\end{array}}{\{B(x\textbf{:}\ y)\ ||\ C(x\textbf{:}\ z)\}\ \textbf{corr}\ [P(x),\ Q(x,\ y,\ z)]}$$

Конкретизация правила **RP**:

$$\textbf{RP:}\ \frac{\begin{array}{l}res' = res\ \textbf{corr*}\ [\textbf{true},\ res' = res];\\ rv' = rv\ \textbf{corr*}\ [\textbf{true},\ rv' = rv];\\ of = \textbf{false}\ \textbf{corr*}\ [\textbf{true},\ of = of0];\\ pParseInt(js,\ res,\ rv,\ of0)\ \&\ \neg isDigit(s[js])\ \&\ res' = res\ \&\ rv' = rv\ \&\ of = of0 \Rightarrow\\ \qquad qParseInt(js,\ res,\ res',\ rv,\ rv',\ of);\\ pParseInt(js,\ res,\ rv,\ of0)\ \&\ \neg isDigit(s[js]) \Rightarrow true\ \&\ true\ \&\ true;\end{array}}{\begin{array}{l}\{\ res' = res\ ||\ rv' = rv\ ||\ of = of0\ \}\ \textbf{corr}\\ pParseInt(js,\ res,\ rv,\ of0)\ \&\ \neg isDigit(s[js]),\ \ qParseInt(js,\ res,\ res',\ rv,\ rv',\ of)]\end{array}}$$

Четвертая посылка определяет формулу корректности:

```
RP1: pParseInt(js, res, rv, of0) & ¬isDigit(s[js]) & res' = res & rv' = rv & of = of0 ⇒
                qParseInt(js, res, res', rv, rv', of);
```

Для первой цели правила **QC**: применим правило **QS**.

$$\textbf{QS:}\ \frac{\begin{array}{l}P(x) \Rightarrow \exists z.\ B(x\textbf{:}\ z);\\ \forall z.\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ B(x\textbf{:}\ z),\ Q(x, y)];\end{array}}{\begin{array}{l}B(x\textbf{:}\ z);\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,y)]\end{array}}$$

Конкретизация правила:

$$\textbf{QS:}\ \frac{\begin{array}{l}pParseInt(js, res, rv, of0)\ \&\ isDigit(s[js]) \Rightarrow \exists\ val.\ digit(s[js], val);\\ Z1\ \textbf{corr}\ [pParseInt(js, res, rv, of0)\ \&\ isDigit(s[js])\ \&\ digit(s[js], val),\\ \qquad qParseInt(js, res, res', rv, rv', of)];\end{array}}{\begin{array}{l}digit(s[js], val);Z1\ \textbf{corr}\\ [pParseInt(js, res, rv, of0)\ \&\ isDigit(s[js]),\ \ qParseInt(js, res, res', rv, rv', of)]\end{array}}$$

Вторая посылка определяет новую цель:

E6(res, val, of0, of1); parseInt (js+1, res*base + val, rv+1, of1 **:** res', rv', of) **corr**
[pParseInt(js, res, rv, of0) & isDigit(s[js], 16) & digit(s[js], 16, val),
qParseInt(js, res, res', rv, rv', of)];

formula E6(**nat** res, val, **bool** of0, of1) =
if (res >= 2**60 & res > (max64 – val) / base) **then** of1= **true** **else** of1 = of0

Снова применяется правило **QS**.

$$\textbf{QS:}\ \frac{\begin{array}{l}P(x) \Rightarrow \exists z.\ B(x\textbf{:}\ z);\\ \forall z.\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ B(x\textbf{:}\ z),\ Q(x, y)];\end{array}}{\begin{array}{l}B(x\textbf{:}\ z);\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,y)]\end{array}}$$

Конкретизация правила:

$$\textbf{QS:}\ \frac{\begin{array}{l}pParseInt(js, res, rv, of0)\ \&\ isDigit(s[js], 16)\ \&\ digit(s[js], 16, val) \Rightarrow \exists\ of1.\ E6(res, val, of0, of1)\\ parseInt\ (js+1, res*base + val, rv+1, of1\ \textbf{:}\ res', rv', of)\ \textbf{corr}\\ [pParseInt(js, res, rv, of0)\ \&\ isDigit(s[js], 16)\ \&\ digit(s[js], 16, val)\ \&\ E6(res, val, of0, of1),\\ \qquad qParseInt(js, res, res', rv, rv', of)];\end{array}}{\begin{array}{l}E6(res, val, of0, of1);\ \ parseInt\ (js+1, res*base + val, rv+1, of1\ \textbf{:}\ res', rv', of)\ \textbf{corr}\\ [pParseInt(js, res, rv, of0)\ \&\ isDigit(s[js], 16)\ \&\ digit(s[js], 16, val),\\ \qquad qParseInt(js, res, res', rv, rv', of)];\end{array}}$$

Вторая посылка определяет новую цель:

parseInt (js+1, res*base + val, rv+1, of1 **:** res', rv', of) **corr**
[pParseInt(js, res, rv, of0) & isDigit(s[js], 16) & digit(s[js], 16, val) & E6(res, val, of0, of1),
qParseInt(js, res, res', rv, rv', of)];

применим правило **RB**:

$$\textbf{RB:}\ \frac{\begin{array}{l}\forall z\ C(x, z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x, z), Q_C(x, y)];\\ SV(P_B, B)(x);\\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x, B(x));\\ \forall y\ (\ P(x)\ \&\ Q_C(B(x), y) \Rightarrow Q(x, y)\ );\end{array}}{\begin{array}{l}C(x, B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]\end{array}}$$

formula p6(size_t js, **nat** res, val, nat32 rv, **bool** of0, of1) =
pParseInt(js, res, rv, of0) & isDigit(s[js]) & digit(s[js], val) & E6(res, val, of0, of1);

formula pB(size_t js, **nat** rv) = rv+1 < 2**32;

Конкретизация правила:

RB:
parseInt(js, res, rv, of0**:** res′, rv′, of) **corr***
[pParseInt(js, res, rv, of0), qParseInt(js , res, res′, rv, rv′, of)];
p6(js, res, val, rv, of0, of1) ⇒ pB(rv) & pParseInt(js+1, res*base + val, rv+1, of1) & m(js+1)<m(js);
p6(js, res, val, rv, of0, of1) & qParseInt(js+1, res*base + val, res′, rv1+1, rv′, of) ⇒
qParseInt(js , res, res′, rv, rv′, of);
---
parseInt (js+1, res*base + val, rv+1, of1 **:** res′, rv′, of) **corr**
[p6(js, res, val, rv, of0, of1), qParseInt(js , res, res′, rv, rv′, of)]

Здесь B(js, res, rv) = (js+1, res * base + val, rv1+1). Причем $P_B(x)$ = ~~js+1 < 2**32 &~~ = rv+1 < 2**32. Вторая и третья посылки определяют следующие формулы корректности:

RB1: p6(js, res, val, rv, of0, of1) ⇒ pParseInt(js+1, res*base + val, rv+1, of1) & m(js+1)<m(js);
RB2: p6(js, res, val, rv, of0, of1) & qParseInt(js+1, res*base + val, res′, rv1+1, rv′, of) ⇒
qParseInt(js, res, res′, rv, rv′, of)

Генерация формул корректности для лемм-предикатов

B.1. Лемма **NuNext**.

При доказательстве возникла следующая лемма:

```
lemma NuNext: forall res:int, res0:int, k:int, m:int, valD:int.
  k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res ->
                  numRes k m res0 res
```

Построим следующую соответствующую программу в качестве леммы-предиката:

```
NuNext(int k, m, valD, res0, res)
      pre k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res
      post numRes k m res0 res
      measure m
{   if (m = k+1) true
    else  {{ NuNext(k, m-1, valD, res0, res1) || digit(s[m-1], va)}; res= res1*base+va}
}

formula pNun(k, m, valD, res0, res) =
          k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res
formula qNun(k, m, valD, res0, res) = numRes k m res0 res
```

formula hm(**nat** m) = m;

Применим правило **QC**.

QC:
B(x**:** y) **corr** [P(x) & E(x), Q(x, y)];
C(x**:** z) **corr** [P(x) & ¬E(x), Q(x, y)]

{**if (**E(x)) B(x**:** y) **else** C(x**:** y)} **corr** [P(x), Q(x, y)]

Конкретизация правила **QC**:

true corr [pNun(k, m, valD, res0, res) & m = k+1, qNun(k, m, valD, res0, res)];
Z; res= res1*base+va **corr**
QC: [pNun(k, m, valD, res0, res) & m <> k+1, qNun(k, m, valD, res0, res)]
if (m = k+1) **true else** Z **corr** [pNun(k, m, valD, res0, res), qNun(k, m, valD, res0, res)]

Для первой посылки применим правило **COR**.

$$\textbf{COR:}\ \frac{\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x,y);\quad \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)}{H(x\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{pNun(k,m,valD,res0,res)\ \&\ m = k+1\ \&\ \textbf{true} \Rightarrow qNun(k,m,valD,res0,res);\quad pNun(k,m,valD,res0,res)\ \&\ m = k+1 \Rightarrow \exists b.\ \textbf{true}}{\textbf{true corr}\ [pNun(k,m,valD,res0,res)\ \&\ m = k+1, qNun(k,m,valD,res0,res)]}$$

Первая посылка определяет формулу корректности:

COR11: pNun(k, m, valD, res0, res) & m = k+1 ⇒ qNun(k, m, valD, res0, res);

Вторая посылка правила **QC** определяет цель:

Z; res= res1*base+va **corr**
[pNun(k, m, valD, res0, res) & m <> k+1, qNun(k, m, valD, res0, res)]

Применяется правило **QSB**.

$$\textbf{QSB:}\ \frac{B(x\textbf{:}\ z)\ \textbf{corr}^{*}\ [P_B(x), Q_B(x,z)];\ P(x) \Rightarrow P^{*}_B(x);\quad \forall z.\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ Q_B(x,z), Q(x,y)];}{B(x\textbf{:}\ z);\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

Z **corr** [pNun(k, m-1, valD, res0, res1) & isDigit(s[m-1]),
qNun(k, m-1, valD, res0, res1) & digit(s[m-1], va)]
pNun(k, m, valD, res0, res) & m <> k+1 -> pNun(k, m-1, valD, res0, res1) & isDigit(s[m-1])
res= res1*base+va **corr**
QSB: [pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) & digit(s[m-1], va),
qNun(k, m, valD, res0, res)];
Z; res= res1*base+va **corr**
[pNun(k, m, valD, res0, res) & m <> k+1, qNun(k, m, valD, res0, res)]

Z **corr** [pNun(k, m-1, valD, res0, res1) & isDigit(s[m-1]),
qNun(k, m-1, valD, res0, res1) & digit(s[m-1], va)]

Вторая посылка определяет формулу корректности:

QSB3: pNun(k, m, valD, res0, res) & m <> k+1 ->
pNun(k, m-1, valD, res0, res1) & isDigit(s[m-1]);

Третьяя посылка определяет новую цель. Применяется правило **COR**.

$$\textbf{COR:}\ \frac{\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x,y);\quad \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)}{H(x\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{\begin{array}{l}pNun(k, m, valD, res0, res)\ \&\ m <> k+1\ \&\ qNun(k, m-1, valD, res0, res1)\ \&\ digit(s[m-1], va)\ \&\\ res= res1*base+va \rightarrow qNun(k, m, valD, res0, res)\end{array}}{\begin{array}{l}pNun(k, m, valD, res0, res)\ \&\ m <> k+1\ \&\ qNun(k, m-1, valD, res0, res1)\ \&\ digit(s[m-1], va)\\ res= res1*base+va\ \textbf{corr}\\ [pNun(k, m, valD, res0, res)\ \&\ m <> k+1\ \&\ qNun(k, m-1, valD, res0, res1)\ \&\ digit(s[m-1], va),\\ qNun(k, m, valD, res0, res)]\end{array}}$$

Первая посылка определяет формулу корректности:

COR12: pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) &
digit(s[m-1], va) & res= res1*base+va -> qNun(k, m, valD, res0, res)

В итоге получим теорию:

```
formula pNun(k, m, valD, res0, res) =
        k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res
formula qNun(k, m, valD, res0, res) = numRes k m res0 res
COR11: pNun(k, m, valD, res0, res) & m = k+1 ⇒ qNun(k, m, valD, res0, res);
QSB3: pNun(k, m, valD, res0, res) & m <> k+1 ->
                pNun(k, m-1, valD, res0, res1) & isDigit(s[m-1]);
COR12: pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) &
                digit(s[m-1], va) & res= res1*base+va -> qNun(k, m, valD, res0, res)

predicate pNun(k m valD res0 res: int) =
        k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res
predicate qNun(k m valD res0 res) = numRes k m res0 res
goal COR11: pNun k m valD res0 res  /\ m = k+1 -> qNun k m valD res0 res
goal QSB3: pNun k m valD res0 res  /\ m <> k+1 ->
                pNun k (m-1) valD res0 res1 /\ isDigit s[m-1]
goal COR12: pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) /\
                digit s[m-1] va /\ res = ((res1*base)+va) -> qNun k m valD res0 res
```

B.2. Лемма **NuEq**.

При доказательстве возникла следующая лемма:

```
lemma NuEq :   forall k:int, m:int, res0:int, res:int, res1:int.
               numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
```

Построим следующую соответствующую программу в качестве леммы-предиката:

```
NuEq(int k, m, res, res0, res1)
      post numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res;
      measure m
{   if (k = m) true  else  { NuEq(k, m-1, res, res0, res1)} }
```

formula pNuEq(**int** k, m, res, res0, res1) = **true**;
formula qNuEq(**int** k, m, res, res0, res1) =
numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res;

Введем формулу для меры.

formula hm(**nat** m**: nat**) = m;

Применим правило **QC**.

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x),\ Q(x,\ y)];\\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x),\ Q(x,\ y)]\end{array}}{\{\textbf{if (}E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Конкретизация правила **QC**:

$$\textbf{QC:}\ \frac{\begin{array}{l}\textbf{true corr}\ [pNuEq(k,\ m,\ res,\ res0,\ res1)\ \&\ k = m,\ qNuEq(k,\ m,\ res,\ res0,\ res1)];\\ NuEq(k,\ m\text{-}1,\ res,\ res0,\ res1)\ \textbf{corr}\\ [pNuEq(k,\ m,\ res,\ res0,\ res1)\ \&\ k <> m,\ qNuEq(k,\ m,\ res,\ res0,\ res1)]\end{array}}{\begin{array}{l}numRes\ k\ m\ res0\ res\ /\backslash\ numRes\ k\ m\ res0\ res1 \Rightarrow res1 = res\ \textbf{corr}\\ [pNuEq(k,\ m,\ res,\ res0,\ res1),\ qNuEq(k,\ m,\ res,\ res0,\ res1)]\end{array}}$$

Для первой посылки применим правило **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x,\ y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)\end{array}}{H(x\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{\begin{array}{l}pNuEq(k,\ m,\ res,\ res0,\ res1)\ \&\ k = m\ \&\ \textbf{true} \Rightarrow qNuEq(k,\ m,\ res,\ res0,\ res1);\\ pRnumR(js,\ valD,\ res)\ \&\ ks = js \Rightarrow \exists b.\ \textbf{true}\end{array}}{\textbf{true corr}\ [pNuEq(k,\ m,\ res,\ res0,\ res1)\ \&\ k = m,\ qNuEq(k,\ m,\ res,\ res0,\ res1)]}$$

Первая посылка определяет формулу корректности:

COR13: pNuEq(k, m, res, res0, res1) & k = m ⇒ qNuEq(k, m, res, res0, res1);

Вторая посылка правила **QC** определяет цель:

NuEq(k, m-1, res, res0, res1) **corr** [k <> m, qNuEq(k, m, res, res0, res1)]

Здесь учтено применение правила **QS**.

Применим правило **RB**.

$$\textbf{RB:}\ \frac{\begin{array}{l}\forall z\ C(x,\ z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x,\ z),\ Q_C(x,\ y)];\\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x,\ B(x));\\ \forall y\ (\ P(x)\ \&\ Q_C(B(x),\ y) \Rightarrow Q(x,\ y)\ );\end{array}}{C(x,\ B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Конкретизации правила:

$$\textbf{RB:}\ \frac{\begin{array}{l}RnumR(\textbf{int}\ js,\ valD,\ res)\ \textbf{corr}^*\ [pRnumR(js\text{-}1,\ va,\ res),\ qRnumR(js\text{-}1,\ va,\ res)];\\ k <> m \Rightarrow m\text{-}1<m;\\ k <> m\ \&\ qNuEq(k,\ m\text{-}1,\ res,\ res0,\ res1) \Rightarrow qNuEq(k,\ m,\ res,\ res0,\ res1)\end{array}}{NuEq(k,\ m\text{-}1,\ res,\ res0,\ res1)\ \textbf{corr}\ [k <> m,\ qNuEq(k,\ m,\ res,\ res0,\ res1)]}$$

Посылки правила **RB** определяют формулы корректности:

RB5: k <> m & qNuEq(k, m-1, res, res0, res1) ⇒ qNuEq(k, m, res, res0, res1)

В итоге получим теорию:

```
formula qNuEq(int k, m, res, res0, res1) =
        numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res;
  COR13: k = m ⇒ qNuEq(k, m, res, res0, res1);
  RB5: k <> m & qNuEq(k, m-1, res, res0, res1) ⇒ qNuEq(k, m, res, res0, res1)

predicate qNuEq(k m res res0 res1: int) =
                numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
goal COR13: k = m -> qNuEq k m res res0 res1
goal RB5: k <> m /\ qNuEq k (m-1) res res0 res1 -> qNuEq k m res res0 res1
```