A Note on Application of Program Specialization to Computer Algebra

Andrei P. Nemytykh
Program Systems Institute of RAS
Russia

5th International Conference "Computer Algebra"

Moscow, June 26-28, 2023

In this talk I am going to discuss some experiments with a general-purpose program transformation tool aiming at running time optimization.

- The experiments demonstrate that the tool is able to solve some of computer algebra tasks.
- The tasks to be solved are formulated in terms of the subject language of the transformation tool.

Automatic program specialization

Given an input program **p**, program specialization aims at running time optimization of **p** w.r.t. its syntactic structures.

- The simplest example is to generate a definition of a partial subfunction of the partial function defined by **p**.

- One may wonder in some of syntactic properties of the program **q** resulted by specialization rather than running time of **q**.

Automatic program specialization

I will:

- show a couple of corollaries of widely known mathematical constructions, which derived by a general purpose program transformation tool, a specializer;
- shortly introduce those properties of the tool, that allow it to achieve desirable results.

Automatic program specialization

The idea of using such a tool for generating mathematical formulae was originated from Alexandr Korlyukov.

The Idea

- A specialized program represents a residual code that may be of interest in itself and even is not intended to be evaluated with some input.

Let **theorem** be a program with two parameters $\mathbf{condition_1}$ and $\mathbf{condition_2}$. Then deriving a corollary $\mathbf{theorem_{condition_1}}$ from **theorem** and $\mathbf{condition_1}$ is a good example demonstrating Korlyukov's idea:

$$\mathbf{theorem_{condition_1}(condition_2) = theorem(condition_1, condition_2)}$$

- A specialized program represents a residual code that may be of interest in itself and even is not intended to be evaluated with some input.

$$\mathbf{theorem}_{\mathbf{condition_1}}(\mathbf{condition_2}) = \mathbf{theorem}(\mathbf{condition_1},\mathbf{condition_2})$$

- The corollary $\mathbf{theorem}_{\mathbf{condition_1}}$ can be more useful than the general **theorem** when we are in the scope of $\mathbf{condition_1}$.

- In general, automatic generating a good transparent statement of the corollary is nontrivial (and even undecidable) since the subject programming language of the specializer used is Turing complete.

Let an $\mathbf{N} = \overline{\mathbf{d_n d_{n-1} \ldots d_0}} \in \mathbb{N}$, where $\mathbf{d_n} > \mathbf{0}$ and $\mathbf{0} \leq \mathbf{d_i} \leq \mathbf{9}$, be given in the decimal system.

Divisibility criteria

are ways of telling whether one natural number divides another without actually carrying the division through. Divisibility criteria are constructed in terms of the digits that compose a given number. The criteria have to be simpler than the direct division of the second number by the first one.

Let $\mathbf{N}$ be the number whose divisibility by another number $\mathbf{q}$ we are going to investigate.

Let an $\mathbf{N} = \overline{\mathbf{d_n d_{n-1} \dots d_0}} \in \mathbb{N}$, where $\mathbf{d_n > 0}$ and $\mathbf{0 \le d_i \le 9}$, be given in the decimal system.

The following equality (formula) being an equality criterion of remainders of dividing two sums by $\mathbf{q} \in \mathbb{N}$ is usually called divisibility criterion for divisibility into $\mathbf{q}$. Here the function % returns the remainder of dividing the first argument by the second.

Divisibility Criterion

$$(\sum_{\mathbf{k=0}}^{\mathbf{n}} \mathbf{10^k d_k} = \sum_{\mathbf{k=0}}^{\mathbf{n}} (\mathbf{10^k}\%\mathbf{q})\mathbf{d_k}) \pmod{\mathbf{q}}$$

The right-hand side of this equality satisfies the requirement above.

Now we translate the right-hand side $(\sum_{k=0}^{n}(10^{k}\%q)d_{k})\%q$ of the equality into terms of our presentation programming language **R** that is a simplified version of Refal.

```
$ENTRY divide { q_s (ds_e) = <div 1 0 q_s (ds_e)>; }
```

```
div {
 m_s res_s q_s ( ) = res_s;      /* (∑_{k=0}^{n} 10^k d_k)%q = */
 m_s res_s q_s (ds_e d_s)
          /* ((((10 * (10^{i-1}%q))%q)d_i + ∑_{k=0}^{i-1}(10^k%q)d_k))%q */
     = <div <% <* m_s 10> q_s> <+ res_s <* d_s m_s>> q_s (ds_e)>;
}
```

$\mathcal{A}$ is a union of a set of symbols and the natural numbers.
The data set defined by: $\mathtt{d} ::= (\mathtt{d}_1) \mid \mathtt{d}_1\ \mathtt{d}_2 \mid \alpha \mid \mathtt{empty}$, where $\alpha \in \mathcal{A}$.

- It is a free monoid w.r.t. the concatenation denoted by the blank.
- The second constructor is unary. It is denoted with its parenthesis only (that is without a name) and is used for constructing tree structures.

```
$ENTRY divide { q_s (ds_e) = <div 1 0 q_s (ds_e)>; }
```

```
div {
 m_s res_s q_s ( ) = res_s;      /* $(\sum_{k=0}^{n} 10^k d_k)\%q =$ */
 m_s res_s q_s (ds_e d_s)
        /* $((((10 * (10^{i-1}\%q))\%q)d_i + \sum_{k=0}^{i-1}(10^k\%q)d_k))\%q$ */
     = <div <% <* m_s 10> q_s> <+ res_s <* d_s m_s>> q_s (ds_e)>;
}
```

Programs in **R** are term rewriting systems. The semantics is based on pattern matching and call-by-value evaluation. The rewriting rules are ordered for matching from the top to the bottom.

- Every function is unary. An example of a function call:
  <**div** **1 0** $\mathtt{q_s}$ ($\mathtt{ds_e}$)>
- The function **devide** below is defined by means of one rewriting rule, while the function **div** - by means of two rules.

```
$ENTRY divide { q_s (ds_e) = <div 1 0 q_s (ds_e)>; }
```

```
div {
 m_s res_s q_s ( ) = res_s;        /* (∑_{k=0}^{n} 10^k d_k)%q = */
 m_s res_s q_s (ds_e d_s)
         /* ((((10 * (10^{i-1}%q))%q)d_i + ∑_{k=0}^{i-1}(10^k%q)d_k))%q */
     = <div <% <* m_s 10> q_s> <+ res_s <* d_s m_s>> q_s (ds_e)>;
}
```

The Presentation Programming Language **R**

Programs in **R** are term rewriting systems.

- There exist two types of variables: $\texttt{name}_e$ and $\texttt{name}_s$:
  - an e-variable can take any data as its value,
  - an s-variable ranges over $\mathcal{A}$.
- For every rewriting rule its set of variables from the left side includes its set of variables from the right side.

```
$ENTRY divide { $q_s$ ($ds_e$) = <div 1 0 $q_s$ ($ds_e$)>; }
```

```
div {
 $m_s$ $res_s$ $q_s$ ( ) = $res_s$;      /* $(\sum_{k=0}^{n} 10^k d_k)\%q =$ */
 $m_s$ $res_s$ $q_s$ ($ds_e$ $d_s$)
        /* $((((10 * (10^{i-1}\%q))\%q)d_i + \sum_{k=0}^{i-1}(10^k\%q)d_k))\%q$ */
     = <div <% <* $m_s$ 10> $q_s$> <+ $res_s$ <* $d_s$ $m_s$>> $q_s$ ($ds_e$)>;
}
```

We translate the right side $(\sum_{k=0}^{n}(10^k\%q)d_k)\%q$ of the equality into terms of the language **R** that is a simplified version of Refal.

- Given a divisor q^0 we are going to specialize the program w.r.t. the following initial configuration:

$$<\mathbf{divide}\ q^0\ (\mathtt{ds_e}\ \mathtt{d0_s})>$$

```
$ENTRY divide { q_s (ds_e) = <div 1 0 q_s (ds_e)>; }
```

```
div {
 m_s res_s q_s ( ) = res_s;      /* $(\sum_{k=0}^{n} 10^k d_k)\%q$ = */
 m_s res_s q_s (ds_e d_s)
          /* $((((10 * (10^{i-1}\%q))\%q)d_i + \sum_{k=0}^{i-1}(10^k\%q)d_k))\%q$ */
     = <div <% <* m_s 10> q_s> <+ res_s <* d_s m_s>> q_s (ds_e)>;
}
```

The initial configuration:

<divide **3** (ds$_e$ d0$_s$)>

The residual program generated by the specializer used:

```
$ENTRY divide1 { (ds_e d0_s) = <F19 (ds_e) d0_s>; }

F19 {                /* $\sum_{k=0}^{n} d_k \vdots 3$ */
 ( ) x2_s = x2_s;
 (x1_e x3_s) x2_s = <F19 (x1_e) <+ x2_s x3_s >>; }
```

The initial configuration:

<divide **18** (ds$_e$ d0$_s$)>

The residual program generated by the specializer used:

```
$ENTRY divide1 { (ds_e d0_s) = <F19 (ds_e) d0_s>; }

F19 {                /* $d_0 + 10\sum_{k=1}^{n} d_k \vdots 18$ */
 ( ) x2_s = x2_s;
 (x1_e x3_s) x2_s = <F19 (x1_e) <+ x2_s <× x3_s 10 >>>; }
```

The initial configuration:

```
<divide 8 (ds_e d0_s)>
```

The residual program generated by the specializer used:

```
$ENTRY divide1 {                    /* d0 + 2d1 + 4d2 ⋮ 8 */
  (d0_s) = d0_s;
  (d1_s d0_s) = <+ d0_s <× d1_s 2>>;
  (x1_e d2_s d1_s d0_s) = <+  <+ d0_s  <× d1_s 2>>  <× d2_s 4>>;
}
```

The initial configuration:

```
<divide 10 (ds_e d0_s)>
```

The residual program generated by the specializer used:

```
$ENTRY divide1 {  (d1_s d0_s) = d0_s;  }        /* d0 ⋮ 10 */
```

Examples of divisibility criteria $\overline{\mathbf{d_n d_{n-1} \dots d_0}}$ by

6: $4 \times \sum_{k=1}^{n} d_k + d_0 \vdots 6$

- $6674 \vdots 6 \Leftrightarrow (4(6+6+7)+4) = 80 \vdots 6 \Leftrightarrow 32 \vdots 6 \Leftrightarrow 14 \vdots 6$
- $2022 \vdots 6 \Leftrightarrow (4(2+2)+2) = 18 \vdots 6 \Leftrightarrow 12 \vdots 6 \Leftrightarrow 6 \vdots 6$

37: $\sum_{k=0}^{n/3}(26 \times d_{3k+2} + 10 \times d_{3k+1} + d_{3k}) \vdots 37$

- $2023 \vdots 37 \Leftrightarrow (2+20+3) = 25 \vdots 37$
- $18446744073709551615 \vdots 37 \Leftrightarrow 1010 \vdots 37 \Leftrightarrow 11 \vdots 37$

12: $4 \times \sum_{k=2}^{n} d_k + 10 \times d_1 + d_0 \vdots 12$

999: $\sum_{k=0}^{n/3}(100 \times d_{3k+2} + 10 \times d_{3k+1} + d_{3k}) \vdots 999$

The Divisibility Criteria Specialization Tasks

Experimental average times taken by generating divisibility criteria (including input, parsing and output times) by

3	:	0m **0,045**s	**12**	:	0m **0,054**s
6	:	0m **0,046**s	**18**	:	0m **0,042**s
8	:	0m **0,058**s	**37**	:	0m **0,064**s
10	:	0m **0,039**s	**999**	:	0m **0,062**s

The computer resources:

processor name	:	AMD A6-6420K APU
cpu MHz	:	3100.000
MemAvailable	:	743376 kB
Oper. System	:	#80-20.04.1-Ubuntu x86_64 GNU/Linux

$\mathbf{F} \subsetneq \mathbf{M} \subsetneq \mathbf{K}$
Given an algebraic closed field $\mathbf{K}$ of characteristic $\mathbf{0}$ and its subfield $\mathbf{F}$ there is a classical constructive method building finite extensions of $\mathbf{F}$ inside of $\mathbf{K}$.

- The method is uniform both on $\mathbf{K}$ and its subfield set.
- Given a non-constant polynomial $\mathbf{p}(\mathbf{x}) \in \mathbf{F}[\mathbf{x}]$ which is irreducible over $\mathbf{F}$, the field of fractions of $\mathbf{F}[\mathbf{x}]/\mathbf{p}(\mathbf{x})$ is an extension of $\mathbf{F}$, which is isomorphic to the extension of $\mathbf{F}$ by a root $\mathbf{x_0}$ of $\mathbf{p}(\mathbf{x})$, $\mathbf{x_0} \in \mathbf{K}$.
    - For instance, $\mathbb{Q}(\mathbf{i}) \cong \mathbb{Q}[\mathbf{x}]/(\mathbf{x^2}+\mathbf{1})$.

Uniformness of the Construction

Theorem

Any finite extention of the field $\mathbf{F}$ may be constructed by means of a finite sequence of simple extensions of algebraic fields.

$$\mathbf{F} = \mathbf{M_0} \subsetneq \mathbf{M_1} \subsetneq \ldots \subsetneq \mathbf{M_{n-1}} \subsetneq \mathbf{M_n} = \mathbf{M}$$

The Procedure

generating a simple extension of a given field is unform over the set of subfields $\mathbf{F}$ of the field $\mathbf{K}$ of characteristic $\mathbf{0}$.

Given fields $\mathbf{F}, \mathbf{K}$ and a polynomial $\mathbf{q}(\mathbf{x}) \in \mathbf{F}[\mathbf{x}]$ irreducible over $\mathbf{F}$. The problem is to implement a small program library of arithmetic for $\mathbf{F}[\mathbf{x}]/\mathbf{q}(\mathbf{x})$, parameterized by $\mathbf{F}, \mathbf{K}$ and $\mathbf{q}(\mathbf{x})$.

Formulation of the Problem

$\mathbf{F} \subsetneq \mathbf{M} \subsetneq \mathbf{K}$, the algebraic closed field $\mathbf{K}$ of characteristic $\mathbf{0}$.

- $\mathbf{F} = \mathbb{Q}$, $\mathbf{K} = \mathbb{C}$

Given a polynomial $\mathbf{q}(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}]$ irreducible over $\mathbb{Q}$.

- The problem is to implement a small program library of arithmetic for $\mathbb{Q}[\mathbf{x}]/\mathbf{q}(\mathbf{x})$, parameterized by $\mathbf{q}(\mathbf{x})$.

Specialization Task №1

$\mathbf{F} \subsetneq \mathbf{M} \subsetneq \mathbf{K}$, the algebraic closed field $\mathbf{K}$ of characteristic $\mathbf{0}$.

- $\mathbf{F} = \mathbb{Q}$, $\mathbf{K} = \mathbb{C}$, $\mathbf{q_0(x) = x^2 + 1} \in \mathbb{Q}[\mathbf{x}]$ irreducible over $\mathbb{Q}$.

These constructions are implemented as an **R**-program including two modules `FieldExt.ref` + `Q.ref`:

`ieldExt.ref`: constructions of ring $\mathbf{F}[\mathbf{x}]$ and simple extension of $\mathbf{F}$ of characteristic $\mathbf{0}$, $\underline{\text{parameterized by } \mathbf{F}}$.

`Q.ref`: arithmetic functions in $\mathbb{Q}$;

The arithmetic functions of $\mathbb{Q}$ are declared as external for the module `FieldExt.ref`.

Specialization Task №1

$\mathbf{F} \subsetneq \mathbf{M} \subsetneq \mathbf{K}$, the algebraic closed field $\mathbf{K}$ of characteristic $\mathbf{0}$.

- $\mathbf{F} = \mathbb{Q}$, $\mathbf{K} = \mathbb{C}$, $\mathbf{q_0(x) = x^2 + 1} \in \mathbb{Q}[\mathbf{x}]$ irreducible over $\mathbb{Q}$.

The constructions are implemented as an **R**-program including two modules `FieldExt.ref` + `Q.ref`:

`ieldExt.ref`: constructions of ring $\mathbf{F}[\mathbf{x}]$ and simple extension of $\mathbf{F}$ of characteristic $\mathbf{0}$, $\underline{\text{parameterized by } \mathbf{F}}$.

`Q.ref`: arithmetic functions in $\mathbb{Q}$;

The arithmetic functions of $\mathbb{Q}$ are declared as external for the module `FieldExt.ref`.

The initial configuration from the module `FieldExt.ref`:

$<\texttt{F\_Arith}\ \ \#\texttt{oper}_\texttt{s}\ \ ((\#\texttt{c}_\texttt{e})\ (\#\texttt{d}_\texttt{e}))\ \ ((\#\texttt{a}_\texttt{e})\ (\#\texttt{b}_\texttt{e}))\ \ (\ \llcorner \overline{\texttt{x}^2 + \texttt{1}} \lrcorner\ )>$

Specialization Subtask №1^{-1}

$\mathbf{F} \subsetneq \mathbf{M} \subsetneq \mathbf{K}$, the algebraic closed field $\mathbf{K}$ of characteristic $\mathbf{0}$.

- $\mathbf{F} = \mathbb{Q}$, $\mathbf{K} = \mathbb{C}$, $\mathbf{q_0(x) = x^2 + 1} \in \mathbb{Q}[\mathbf{x}]$ irreducible over $\mathbb{Q}$.

The constructions are implemented as an **R**-program including two modules `FieldExt.ref` + `Q.ref`:

`ieldExt.ref`: constructions of ring $\mathbf{F}[\mathbf{x}]$ and simple extension of $\mathbf{F}$ of characteristic $\mathbf{0}$, $\underline{\text{parameterized by } \mathbf{F}}$.

`Q.ref`: arithmetic functions in $\mathbb{Q}$;

The arithmetic functions of $\mathbb{Q}$ are declared as external for the module `FieldExt.ref`.

The initial configuration from the module `FieldExt.ref`:

$<$F_Arith Inv $((\#c_e)\,(\#d_e))\ ((\#a_e)\,(\#b_e))\ (\,\overline{\llcorner x^2 + 1 \lrcorner}\,)>$

Specialization Subtask №1^{-1}

The initial configuration from the module `FieldExt.ref`:

$$\texttt{<F\_Arith Inv ((\#b}_e\texttt{) (\#a}_e\texttt{)) (}\overline{\llcorner x^2+1 \lrcorner}\texttt{)>}$$

The specialization result by the supercompiler SCP4

$$\texttt{\$ENTRY formulai \{}$$
$$(\mathtt{b}_e)\ \mathtt{a}_e = \ulcorner\!\!\llcorner(-\mathtt{b}_e/(\mathtt{b}_e{}^2+\mathtt{a}_e{}^2))x + (\mathtt{a}_e/(\mathtt{b}_e{}^2+\mathtt{a}_e{}^2))\urcorner\!\!\lrcorner\ ;$$
$$\texttt{\}}$$

- Rough outline - encoding; $\mathtt{a}_e, \mathtt{b}_e \in \mathbb{Q}$.
- First approximation.

Specialization Subtask №1^{-1}

The arithmetic functions of $\mathbb{Q}$ are declared as external for the module `FieldExt.ref`. Their definitions and properties are unavailable for SCP4.

The initial configuration from the module `FieldExt.ref`:

```
$EXTERN Q_Div, Q_Mul, Q_Sub, Q_Add;
        <F_Arith Inv ((#b_e) (#a_e)) ( ⌞x^2 + 1⌟ )>
```

The specialization result by the supercompiler SCP4

```
$ENTRY formulai {
(b_e) a_e = ⌜(−1/b_e)/(1−a_e(0 − (a_e/b_e))/b_e)x +
          (0 − (0 + 1 × ((0 − (a_e/b_e))/b_e)))/(1−a_e(0 − (a_e/b_e))/b_e)⌝ ;
}
```

- Encoding; $a_e, b_e \in \mathbb{Q}$.
- `Q_Div` $= /$, `Q_Mul` $= \times$, `Q_Sub` $= -$, `Q_Add` $= +$;
- Approximation.

Specialization Subtask №1^{-1}

The arithmetic functions of $\mathbb{Q}$ are declared as external for the module `FieldExt.ref`. `Q_Div` $= /$, `Q_Mul` $= \times$, `Q_Sub` $= -$, `Q_Add` $= +$.
Their definitions and properties are unavailable for SCP4.

The specialization result by the supercompiler SCP4

```
$ENTRY formulai {
(bₑ) aₑ = ⌜(−1/bₑ)/(1−aₑ(0−(aₑ/bₑ))/bₑ)x +
          (0−(0 + 1×((0−(aₑ/bₑ))/bₑ)))/(1−aₑ(0−(aₑ/bₑ))/bₑ)⌟ ;
}
```

Even the simplest field properties are unavailable for SCP4.

- Encoding; $a_e, b_e \in \mathbb{Q}$.
- Approximation.

Specialization Subtask №1^{-1}

The initial configuration $\mathtt{Arith}_{\mathtt{Inv}}(\mathtt{b_e},\mathtt{a_e})$

$EXTERN Q_Div, Q_Mul, Q_Sub, Q_Add;

$\quad$ <F_Arith Inv $((\#\mathtt{b_e})\ (\#\mathtt{a_e}))\ (\ \llcorner \overline{x^2+1} \lrcorner\ )$>

The specialization result by the supercompiler SCP4

$ENTRY formulai {

$(\mathtt{b_e})\ \mathtt{a_e} = \ulcorner(-1/\mathtt{b_e})/(1-\mathtt{a_e}(0-(\mathtt{a_e}/\mathtt{b_e}))/\mathtt{b_e})x +$

$\quad (0-(0+1\times((0-(\mathtt{a_e}/\mathtt{b_e}))/\mathtt{b_e})))/(1-\mathtt{a_e}(0-(\mathtt{a_e}/\mathtt{b_e}))/\mathtt{b_e})\urcorner$;

}

This residual program reflects that SCP4 recognized uniformness of the theoretical construction implemented. Let $\mathtt{T}_{\mathbb{Q}}(\mathtt{b_e},\mathtt{a_e})$ stand for the evaluation time of the right hand side above, then

$\exists\, \mathbf{C} \in \mathbb{R}$ s.t.$\forall\, \mathtt{b_e},\mathtt{a_e} \in \mathbb{Q},\ \mathtt{b_e} \neq \mathbf{0}$:

$$\mathtt{Time}_{\mathtt{spec}}(\ulcorner \mathtt{Arith}_{\mathtt{Inv}}(\mathtt{b_e},\mathtt{a_e})\urcorner) \leq \mathbf{C} \times \mathtt{Time}_{\mathtt{run}}([\![\mathtt{Arith}_{\mathtt{Inv}}(\mathtt{b_e},\mathtt{a_e})]\!]) - \mathtt{T}_{\mathbb{Q}}(\mathtt{b_e},\mathtt{a_e})$$

Specialization Task №2

$\mathbf{F} \subsetneqq \mathbf{M} \subsetneqq \mathbf{K}$, the algebraic closed field $\mathbf{K}$ of characteristic $\mathbf{0}$.

- $\mathbf{F} = \mathbb{Q}$, $\mathbf{K} = \mathbb{C}$, $\mathbf{q_0}(\mathbf{x}) = \mathbf{x^2} - \mathbf{2} \in \mathbb{Q}[\mathbf{x}]$ irreducible over $\mathbb{Q}$.

The constructions are implemented as an **R**-program including two modules `FieldExt.ref` + `Q.ref`. The arithmetic functions of $\mathbb{Q}$ are declared as external for the module `FieldExt.ref`.

Specialization Subtask №2×

The initial configuration from the module `FieldExt.ref`:

$$\texttt{<F\_Arith Mul}\ ((\#c_e)\ (\#d_e))\ ((\#a_e)\ (\#b_e))\ (\ \llcorner\overline{x^2-2}\lrcorner\ )\texttt{>}$$

The specialization result by the supercompiler SCP4

$$\begin{array}{l}\texttt{\$ENTRY formula2m}\ \{\\ \quad (c_e)\ (d_e)\ \ (a_e)\ b_e\ =\ \ulcorner\!\!\llcorner (d_e a_e + c_e b_e)x + (d_e b_e + 2c_e a_e)\urcorner\!\!\lrcorner\ \ ;\\ \}\end{array}$$

- Rough outline - encoding; $a_e, b_e, c_e, d_e \in \mathbb{Q}$.
- First approximation.

Specialization Subtask №2×

The initial configuration from the module `FieldExt.ref`:

$<\texttt{F\_Arith Mul}\ ((\#c_e)\ (\#d_e))\ ((\#a_e)\ (\#b_e))\ (\ \overline{\llcorner x^2 - 2 \lrcorner}\ )>$

The specialization result by the supercompiler SCP4

$\texttt{\$ENTRY formula2m}\ \{$
$(c_e)(d_e)(a_e)\ b_e = \ulcorner\llcorner((d_e a_e + c_e b_e) - 0 \times (c_e a_e))x + (d_e b_e - (-2((c_e a_e)/1)))\urcorner\lrcorner;$
$\}$

- Encoding; $a_e, b_e, c_e, d_e \in \mathbb{Q}$.
- `Q_Div` $= /$, `Q_Mul` $= \times$, `Q_Sub` $= -$, `Q_Add` $= +$;
- Approximation.

Arithmetic in Finite Field Extensions Specialization Tasks

The size of the source module `FieldExt.ref` is about 70 lines including comments.

Experimental average times taken by generating arithmetic operations (including input, parsing and output times)

Inv in $\mathbb{Q}[\mathbf{x}]/(\mathbf{x}^2+\mathbf{1})$	:	**0**m **0**,**122**s
$\times$ in $\mathbb{Q}[\mathbf{x}]/(\mathbf{x}^2-\mathbf{2})$	:	**0**m **0**,**114**s

The computer resources:

processor name	:	AMD A6-6420K APU
cpu MHz	:	3100.000
MemAvailable	:	743376 kB
Oper. System	:	#80-20.04.1-Ubuntu x86_64 GNU/Linux

How it Does Work

- The residual programs above were produced by the supercompiler SCP4,
    - a specializer based on a specialization method known as Turchin's supercompilation.
- The language **R** specified above is a functional programming language,
    - for the sake of simplicity only **R**-programs defining partial predicates rather than arbitrary partial functions will be considered below.

How it Does Work

- Any **R**-program **P** can be seen as an evaluation tree,
    - as a rule infinite,
    - i.e. any recursion is unfolded.
    - The edges are labeled with **P**-patterns.
    - Parameterized states of **P** label the tree nodes.
      They are predicates.
- The tree root is labeled with the initial parameterized state.

SCP vs. TM

Turing machine

- A few simple instructions moving a pointer along a tape and writing/reading the tape cells.
  - The programmer juggles with the instructions during generating source programs.
  - In general, even very simple properties of program behavior are undecidable.

SCP vs. TM

The supercompiler

- A number of simple instructions moving a pointer along an evaluation tree, analyzing and transforming the tree; the instruction language is Turing complete.
    - The supercompiler juggles with the instructions while generating residual programs, trying to solve undecidable problems. It is forced to approximate the problems to be solved.
    - In general, even for very simple source programs the supercompiler behavior is undecidable and very complicated.

Roughly speaking,

the main specialization aim - the efficiency of the target program contradicts the efficiency of the supercompiler.

Generating Hypotheses and Theorem Proving

- Given a program and its initial parameterized state
    - SCP4 explores the corresponding evaluation tree, starting from the tree root.
    - It considers the predicates labeling some of the nodes as hypotheses to be proven.
- Given such a node
    - SCP4 tries to prove the corresponding hypothesis by induction on the number of **R**-machine steps along every path originating in this node.
- There may be a number of hypotheses labeling different nodes to be simultaneously proven.

Generating Hypotheses and Theorem Proving

The main problems:

- How does SCP4 determine the hypotheses-nodes to be proven?
- How does it decide that a current hypothesis is too strong (or too weak) to be automatically proven?
    - How should (the assumption of) the statement be weakened?

SCP4 approximates the corresponding solutions. It is based on variants of Higman–Kruskal relation, being well quasi-orders on the program states along the evaluation tree paths.

It is possible to manually create annotations in the input program to be specialized, which may provide some support for SCP4.

How it Does Work

SCP4 approximates the corresponding solutions

- How should (the assumption of) the statement be weakened?

Weakening of the assumptions of the statements / Example

In order to prove the exact uniform lower bound of the worst case complexity of evaluating polynomials $\mathbf{a_n x^n + \ldots + a_1 x + a_0}$, it is usefull to extend the polynomial set to $\mathbf{a_n x^n + \ldots + a_1 x + a_0 (mod\ x^{n+1})}$ (S. A. Abramov, Lectures on Complexity of Computations, p. 230.
After E. M. Reingold and A. I. Stocks.)

- How does it decide that a current hypothesis is too strong (or too weak) to be automatically proven?

SCP4 approximates the corresponding solutions.

It is possible to manually create annotations in the input program to be specialized, which may provide some support for SCP4.

The initial configuration $\mathbf{Divide}(\mathbf{q^0})$ to be specialized:

$<\mathbf{divide}\ \mathbf{q^0}\ (\mathtt{ds_e}\ \mathtt{d0_s})>$

```
$ENTRY divide { q_s (ds_e) = <div 1 0 q_s (ds_e)>; }
```

```
div {
 m_s res_s q_s ( ) = res_s;      /* $(\sum_{k=0}^{n} 10^k d_k)\%q =$ */
 m_s res_s q_s (ds_e d_s)
        /* $((((10 * (10^{i-1}\%q))\%q)d_i + \sum_{k=0}^{i-1}(10^k\%q)d_k))\%q$ */
     = <div <% <* m_s 10> q_s> <+ res_s <* d_s m_s>> q_s (ds_e)>;
}
```

How it Does Work

The initial configuration $\texttt{Divide}(\texttt{q}^0)$ to be specialized:

$<$**divide** $\mathbf{q^0}$ $(\texttt{ds}_\texttt{e}\ \texttt{d0}_\texttt{s})>$

```
$ENTRY divide { $q_s$ ($ds_e$) = <div 1 0 $q_s$ ($ds_e$)>; }
```

```
div {
 $m_s$ $res_s$ $q_s$ ( ) = $res_s$;     /* $(\sum_{k=0}^{n} 10^k d_k)\%q =$ */
 $m_s$ $res_s$ $q_s$ ($ds_e$ $d_s$)
        /* $((((10*(10^{i-1}\%q))\%q)d_i + \sum_{k=0}^{i-1}(10^k\%q)d_k))\%q$ */
     = <div <% <* $m_s$ 10> $q_s$> <+ $res_s$ <* $d_s$ $m_s$>> $q_s$ ($ds_e$)>;
}
```

$\exists\, \mathbf{C} \in \mathbb{R}$ s.t. $\forall\, \mathbf{q} \in \mathbb{N}$, $\mathbf{q} \neq \mathbf{0}$: $\mathtt{Time}_{\mathtt{spec}}(\ulcorner \mathtt{Divide}(\mathtt{q}) \urcorner) \leq \mathbf{C} \times \mathtt{q}^2$

Conclusion

Given a specializer Spec of programs written in a programming language $\mathcal{L}$. It is possible to formulate specialization tasks in such a way that Spec will attempt:

- to compile programs written in a language $\mathcal{M}$ into the language $\mathcal{L}$;
    - Quality of such an indirect compilation is another question.
- to verify safety properties of nondeterministic communication protocols;
- to describe root's sets of some of classes of word equations.

The last item maybe has some relationship to the field of computer algebra.

Thanks you for your attention!

Specialization Subtask №1^{-1}

The arithmetic functions of $\mathbb{Q}$ are declared as external for the module `FieldExt.ref`. Their definitions and properties are unavailable for SCP4.

The initial configuration from the module `FieldExt.ref`:

```
$EXTERN Q_Div, Q_Mul, Q_Sub, Q_Add;
        <F_Arith Inv ((#b_e) (#a_e)) ( ⌞x^2 + 1⌟ )>
```

The specialization result by the supercompiler SCP4

```
$ENTRY formulai {
(b_e) a_e = ⌜(−1/b_e)/(1−a_e(0 − (a_e/b_e))/b_e)x +
        (0 − (0 + 1 × ((0 − (a_e/b_e))/b_e)))/(1−a_e(0 − (a_e/b_e))/b_e)⌝ ;
}
```

- Encoding; $a_e, b_e \in \mathbb{Q}$.
- `Q_Div` $= /$, `Q_Mul` $= \times$, `Q_Sub` $= -$, `Q_Add` $= +$;
- Approximation.

Specialization Subtask №1^{-1}

The residual program:

```
$EXTERN Q_Div, Q_Mul, Q_Sub, Q_Add;

* p(x) = ((b) (a)) = b*x+a – многочлен; требуется вычислить 1/p(x)
$ENTRY formulai { ($b_e$) $a_e$ = <C1  ($b_e$) ($a_e$)  <Q_Div (1) $b_e$>>; }

C1  { (e.1) (e.2) e.x1 =                    /* 0-(a/b) = -a/b */
          <C2 (e.1) (e.2) (e.x1) <Q_Sub (0) <Q_Mul (e.2) e.x1>>>; }
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
* e.x4 = (0-(a/b))/b = -a/$b^2$;      e.x6 = 1 - a(0-(a/b))/b = 1 - (-$a^2$/$b^2$),
* e.y3 = (-1/b)/(1 - a(0-(a/b))/b) = -b/($b^2$ + $a^2$)
C6  { (e.x4) (e.x6) e.y3 =      /* e.y4 = 0-(0+1*((0-(a/b))/b)) = a/$b^2$ */
      <C7 (e.x6) (e.y3) <Q_Sub (0) <Q_Add (0) <Q_Mul (1) e.x4>>>>;
}
C7  {            /* (0-(0+1*((0-(a/b))/b)))/(1 - a(0-(a/b))/b)
                   = (a/$b^2$)/(1 + $a^2$/$b^2$) = a/($b^2$ + $a^2$) */
(e.x6) (e.y3) e.y4 ,  <Q_Div (e.y4) e.x6>:  e.y7 = (e.y3) (e.y7); }
* 1/p(x) = (-1/b)/(1-a(0-(a/b))/b)_ _(0-(0+1*((0-(a/b))/b)))/(1-a(0-(a/b))/b)
```

Specialization Subtask №2×

The initial configuration from the module `FieldExt.ref`:

$<\texttt{F\_Arith Mul}\ ((\#c_e)\ (\#d_e))\ ((\#a_e)\ (\#b_e))\ (\ \overline{\llcorner x^2 - 2 \lrcorner}\ )>$

The specialization result by the supercompiler SCP4

$\texttt{\$ENTRY formula2m}\ \{$

$(c_e)(d_e)(a_e)\ b_e = \ulcorner((d_e a_e + c_e b_e) - 0 \times (c_e a_e))x + (d_e b_e - (-2((c_e a_e)/1)))\urcorner;$

$\}$

- Encoding; $a_e, b_e, c_e, d_e \in \mathbb{Q}$.
- $\texttt{Q\_Div} = /,\ \texttt{Q\_Mul} = \times,\ \texttt{Q\_Sub} = -,\ \texttt{Q\_Add} = +;$
- Approximation.

Specialization Subtask №2×

The residual program:

```
$EXTERN Q_Div, Q_Mul, Q_Sub, Q_Add;
/* p(x) = ((c) (d)) = c*x+d, q(x) = ((a) (b)) = a*x+b – многочлены;
   требуется вычислить p(x)q(x) */
$ENTRY formula2m1 {                                    /* e.y3 = c*a */
 (c_e) (d_e) (a_e) b_e = <C1 (c_e) (d_e) (a_e) (b_e) <Q_Mul (c_e) a_e>>; }

C1  { (e.2) (e.3) (e.4) (e.5) e.y3 =                /* e.y6 = d*a */
           <C2 (e.2) (e.3) (e.5) (e.y3) <Q_Mul (e.3) e.4>>; }
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
* e.y9 =(c*a)/1 = c*a;   e.y8 = d*b, e.y7 = d*a + c*b
C5  { (e.y7) (e.y8) e.y9 =        /* e.z6 = (d*a + c*b) - 0*(c*a) */
        <C6 (e.y8) (e.y9) <Q_Sub (e.y7) <Q_Mul (0) e.y9>>>; }
C6  { (e.y8) (e.y9) e.z6   /* (d*b) -(-2*((c*a)/1)) = d*b + 2*c*a */
       , <Q_Sub (e.y8) <Q_Mul ('-'2) e.y9>>: e.z7 = (e.z6) (e.z7); }
/* p(x)q(x) = _(((d*a + c*b)-0*(c*a))_ _(d*b-(-2*((c*a)/1))))_
           = ((d*a + c*b) (d*b + 2*c*a)) */
```