Reverse engineering of holonomic functions and sequences from imperative scientific computation code

Jacques Carette*

Computing and Software, McMaster University,
Hamilton, Ontario, L8S 4K1, Canada
**E-mail: carette@mcmaster.ca*
www.cas.mcmaster.ca/~carette

Given some imperative code which either computes terms from a holonomic sequence, or an approximation to a holonomic function via truncated Taylor series, we would like to know exactly which sequence or function we are dealing with. We present a method to solve this problem. Leveraging a lot of work in finding closed-forms solutions to recurrence equations, we have a prototype which will find a closed-form (whenever possible) for such code. Many examples are provided which to show the variety of situations where our method is applicable.

Keywords: Holonomy; Reverse engineering; special functions

1. Introduction

As with a lot of scientific progress, much serendipity has gone into this present work. Recently, the author happened to get interested in reverse engineering [1]. But earlier [2], the author was well acquainted with Sergei Abramov's work in many different aspects of computer algebra. Coupled with the knowledge that there are deep techniques in computer algebra to deal with special functions, and holonomic functions in general [3, 4], it became clear that these techniques could be applied to the problem of reverse engineering scientific computation code dealing with special functions. In fact, recent work [5] indicates that these techniques may also work for some classes of programs over arrays, namely "usual" linear algebra.

Furthermore, there seemed to be quite a direct link. Software packages such as `gfun` [6] are not only quite adept at manipulating holonomic functions and recurrences, they can also produce very efficient programs to compute terms of holonomic sequences. In fact, one can do much more, as the automatically generated Encyclopedia of Special Functions of Meunier and Salvy [7] shows.

Experts already know that there is a very tight relation between holonomic functions (formal solutions of linear ordinary differential equations with polynomial coefficients) and holonomic sequences (formal solutions of linear recurrence equations with polynomial coefficients). Can we extend this relation to first, programs which compute such sequences, and thence to programs which compute approxi-

mations to holonomic functions? This current work indicates that this appears to be the case. More precisely, given code in a restricted language which comes from a holonomic source (say via `gfun`), it is possible to algorithmically invert this process and recover the initial equation. While our prototype has focused on providing closed-forms, it is just as easy to stop short of calling the solver and output the equations. Furthermore, this inversion process seems to be quite robust — minor changes in input do not affect the results. In fact, given a proper definition of a normal form for non-linear (but polynomial) first-order recurrence equations, our method could be made to produce such a normal form.

As we handle both holonomic sequences and holonomic functions, it gets tiresome (to read as well as to write) to continually specify both of these, we will use the term *holonomic object* when we want to speak about both sequences and functions indiscriminately.

At the outset, it is important to remark that the code naturally associated to holonomic objects contains only one loop, and the corresponding loop body is straight-line code. While it is certainly possible to generalize our work to more general code, it is not our intent to do this here, as our wish it to be able to reverse engineer code which came from holonomic objects.

2. Problem

Consider the following small core for a programming language SWL (for *Simple While Language*):

$E ::= \mathsf{Var}$	real or integer variable
$\mid i$	integer literal
$\mid E + E \mid E * E \mid E - E \mid E/E \mid E \mathbin{\hat{}} i$	arithmetic operations
$B ::= E = E \mid E < 0$	bool from expressions
$\mid B \text{ and } B \mid B \text{ or } B \mid \text{not } B$	boolean operations
$S ::= \mathsf{Var} := E$	variable assignment
$\mid S \ ; \ S$	sequencing
$\mid \text{while } B \text{ do } S \text{ end}$	while loop
$P ::= \text{proc}(\mathsf{Var}^*) \ \{\mathsf{localVar}^+\} \ S \ ; \ \text{return } E \text{ end}$	procedure

with the obvious operational and denotational semantics[a] [8] and we use standard regular expressions in the definition of P for brevity. SWL is not meant to be a practical programming language, but rather a core language into which one can easily translate other programs. For example, a practical programming language would allow expressions like $E_1 > E_2$ as boolean expressions; in SWL, this has to be represented as $E_2 - E_1 < 0$. What is important to note about this language is that it only has real or integer (but not boolean) variables, and more importantly,

[a] A complete denotational semantics for SWL can be found in Section 3

no conditional. Furthermore, we can only define single-level procedures, as neither the syntax for statements S nor expressions E allow procedure invocation[b], and the only control-flow mechanism is the while loop. While this may seem like an incredibly impoverished language, it is still rich enough to encode the programs corresponding to holonomic objects. In fact, it is more general than what we need, as simple holonomic objects need only a single loop. Conversely, this small programming language can be trivially mapped injectively in various programming languages, notably `C`, `Fortran`, and `Maple`.

We must make a crucial observation at this point: in conventional programming languages (whether procedural, object-oriented or functional), there is no natural programming concept which corresponds to a *derivative*. However, there is a natural programming concept which involves shifting: the while loop. More precisely, each time through the body of the loop, the *loop count* goes up by one. A loop may or may not include an explicit counter variable, but there is nevertheless an abstract "loop counter", and all the explicit state variables changed in the body of a loop "depend" on this loop counter. Our task then will be to introduce an actual loop counter variable, and then to make the dependence of each state variable on this loop counter explicit. This will be the key to turning a while loop into a recurrence.

Before we give a formal statement of the problem we are interested in, it is illustrative to consider a simpler version, which can be solved completely.

Definition 2.1. A program p is said to be *valid* if all (local) variables are initialized before they are read.

Theorem 2.1. *Let p be a program in the* while*-free fragment of SWL. Then if p denotes a valid program, $[\![p]\!]$ denotes a rational function of its inputs. If furthermore the program is division-free, then if p denotes a valid program, it in fact denotes a polynomial of its inputs.*

The above theorem is easily proved by structural induction on the syntax of programs.

Note that we are dealing with *abstract computation* [9], as we are really interested in modeling the algebraic situation over the reals. A common implementation would use floating point numbers to model the reals, which will introduce all sorts of additional complications not present in the algebraic model, and we will not concern ourselves with these issues here. We justify this by saying that it is the use of floating point which is an implementation-time approximation to the real specifications. Thus we use the usual algebraic domains (the ring of integers $\mathbb{Z}$ and the field of reals $\mathbb{R}$) as the basis for our models. However we should note that standard tricks from numerical analysis, like Kahan's summation algorithm [10, 11], are algebraically "invisible", and thus we can also deal with programs written using good numerical analysis methods.

[b] we will indicate how to relax some of these restrictions later

In denotational semantics, one normally uses continuous functions on CPOs as denotations. We are instead trying to recover the algebraic meaning as implemented in a program. In other words, we are really interested in either *sequences* or *functions* over combinations of $\mathbb{Z}$ and $\mathbb{R}$. That said, we are not particularly interested in *computational* representations of these (as that is our starting point), but rather classical mathematical expressions which denote the same mathematical object. Closed forms tend to be preferred by humans, but a reasonable system of (linear) equations with initial conditions is often mathematically much more tractable. This is easiest to explain via an example, to be followed by an explicit definition.

Example 2.1 (Factorial). *Consider the following SWL procedure:*

```
proc(n) local r, i;
r := 1;
i := 1;
while i-1 <> n do
    r := r * i;
    i := i + 1;
end;
return r;
```

Let us call this procedure f. It is easy to see that

$$[\![f]\!] = \lambda n. \begin{cases} n! & n \geq 0 \\ \bot & \textit{otherwise.} \end{cases}$$

More interestingly for us, we have that $\forall t \in \mathbb{N}.[\![f]\!](t+1) = (t+1)*[\![f]\!](t)$, $[\![f]\!](0) = 1$ which, when solved in explicit terms, gets us back to $n!$. Even more interesting is that $n!$ is quite a good representation for f as over $\mathbb{Z}$ they are equivalent — if one is defined, then they are both defined and equal, and if one is undefined, they are both undefined.

The factorial procedure is special in that one can provide a closed-form for it, as well as being able to see the exact termination conditions. While we do not expect to be able to do this for all procedures, even in a fragment of SWL, we would like to extract two pieces of information from such procedures:

(1) An explicit system of recurrence equations, including initial conditions, for each while loop and,
(2) An explicit equation for the termination condition.

Referring back to ex. (2.1), we can show that the body of the while loop satisfies

$$r(t+1) = r(t) * i(t), i(t+1) = i(t) + 1, r(0) = 1, i(0) = 1$$

(where we use t as loop-counter, or "time"), and the termination condition reads

$$t_e = \min\{t \in \mathbb{N} \mid i(t) - 1 = n\},$$

were we use t_e to denote the time at which the loop ends. In this particular case, we can see how to solve the recurrence for $i(t)$ to get $i(t) = t + 1$, upon which

substitution into the equation for t_e leads us to $t_e = n$, from which we easily get the recurrence 2.1. Our task then is to make this precise.

Problem 2.1. *Given an SWL procedure p, return an explicit system of equations satisfied by all the state (local) variables of p. This system of equations should encode both correctness and termination conditions. If possible, these equations should be solved in closed-form.*

We could in fact be even more specific about what kinds of equations we will get (recurrence equations for loops and minimum equations for termination), but as we wish to later expand SWL, the above statement will be sufficient. Naturally, what we really want are closed forms; but once we have in our hands systems of recurrences, we can leverage the tremendous power of today's Computer Algebra Systems to find these closed forms. Furthermore, as this technology improves, we should be able to automatically benefit from these improvements. This is why we focus on obtaining systems of equations, and then define our solution as a two-step process of first getting recurrence equations, and then to finding potential closed-forms.

Note that our problem has a well-defined input language (SWL), a semi-formally defined intermediate language (systems of equations), and a very informal output language (closed forms). In the rest of this paper, we will endeavour to give a formal definition for the intermediate language, but leave the definition of "closed form" completely open, as we wish to be able to use whatever future technology comes along for solving our equations[c]. In this way, our solution is very modular.

Given this problem, we need to show that we have a solution which is total on code which is derived from holonomic objects. In other words, on code from holonomic objects, we need to show that our process always terminates and always gives a system of holonomic equations. Furthermore, we want our solution to be "robust"; in other words, minor variants of the input code (but with identical semantics) should give equivalent results.

Example 2.2 (Factorial revisited). *We made some claims about inverting the process of generating procedures from holonomic equations. If we give Eq. (2.1) to* `gfun[rectoproc]`*, and* `LREtools[REtoproc]`*, we obtain the procedures given in Fig. 1. For both of these programs, we obtain closed-form results equivalent to* $n!$*, namely* $\Gamma(n+1)$ *and* $n\Gamma(n)$ *respectively.*

3. Semantics

We expect the reader to be familiar with holonomic functions and sequences, but not programming language semantics, to which we will give a quick introduction. But we only introduce enough of this theory as is needed in this paper.

[c]For example, no system current uses the multiple Γ function [12] for closed-forms, although this is likely to happen in the future.

Listing 1. `gfun[rectoproc]`

```
pg1 := proc (n)
    local i1, loc0, loc1;
    loc0 := 1;
    for i1 from 0 to n-1 do
        loc1 := (i1+1)*loc0;
        loc0 := loc1
    end do;
    loc0
end proc
```

Listing 2. `LREtools[REtoproc]`

```
pg2 := proc (n::nonnegint)
    local i, u0, u1;
    u0 := 1;
    for i to n-1 do
        u1 := i*u0;
        u0 := u1
    end do;
    n*u0
end proc
```

Fig. 1. automatic factorials

There are 3 main paradigms used to specify programming language semantics: operational, denotational and axiomatic. Roughly speaking, operational semantics describe what each part of a programming languages *does*, denotational semantics describes each part of a language by a (partial) *mathematical function*, while axiomatic semantics gives *logical relations* that each part must satisfy. For small languages, it is common to give both an operational and denotational semantics, and then prove that these are equivalent. One of the cornerstones of programming language semantics is **compositionality**: the meaning of the whole is exactly the composition of the meaning of the parts.

Operational semantics is most often presented as a (conditional) reduction relation. Figure 2 presents a subset of the (standard) operational semantics for SWL. Note that it is very important to distinguish between the syntactic + of the program text from the semantic + of the underlying domain ($\mathbb{Z}$ or $\mathbb{R}$).

$$\frac{\sigma(E_1) \Rightarrow E_1' \quad \sigma(E_2) \Rightarrow E_2'}{\sigma(E_1' + E_2') \Rightarrow E_1' + E_2'} \qquad \frac{}{\sigma(v_1 + v_2) \Rightarrow v_1 + v_2}$$

$$\frac{\sigma(E) \Rightarrow \mathsf{false}}{\sigma(\mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{end}) \Rightarrow \sigma} \qquad \frac{\sigma(E) \Rightarrow \mathsf{true} \quad \sigma(S) \Rightarrow \sigma_1}{\sigma(\mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{end}) \Rightarrow \sigma_1(\mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{end}) \Rightarrow}$$

Fig. 2. Fragment of the operational semantics for SWL

In the above, σ denotes a *store*, an assignment of values to identifiers (variables), E_i is an expression and v_i is a value. A store represents the *state* associated to an imperative program. We extend the definition of this function to the whole language in two ways: applied to an expression E, we recursively evaluate to get a value; applied to a statement S, we get a new store. The main reason to present the operational semantics here is that we model most of our semantics on the denotational semantics of languages, *except* for while loops, where we model the operational semantics much more closely.

$[\![\mathsf{Var}]\!]\sigma$	$=$	$\sigma(\mathsf{Var})$	$[\![i]\!]$	$=$	i
$[\![E_1 + E_2]\!]$	$=$	$[\![E_1]\!] + [\![E_2]\!]$	$[\![E_1 * E_2]\!]$	$=$	$[\![E_1]\!] * [\![E_2]\!]$
$[\![E_1 - E_2]\!]$	$=$	$[\![E_1]\!] - [\![E_2]\!]$	$[\![E_1/E_2]\!]$	$=$	$[\![E_1]\!]/[\![E_2]\!]$
$[\![E_1\hat{}\ i]\!]$	$=$	$[\![E_1]\!]^{[\![i]\!]}$	$[\![\mathsf{not}\ E_1]\!]$	$=$	$\neg[\![E_1]\!]$
$[\![E_1\ \mathsf{or}\ E_2]\!]$	$=$	$[\![E_1]\!] \wedge [\![E_2]\!]$	$[\![E_1\ \mathsf{and}\ E_2]\!]$	$=$	$[\![E_1]\!] \wedge [\![E_2]\!]$
$[\![E_1 = E_2]\!]$	$=$	$[\![E_1]\!] = [\![E_2]\!]$	$[\![E_1 < 0]\!]$	$=$	$[\![E_1]\!] < 0$
$[\![\mathsf{Var} := E]\!]\sigma$	$=$	$\sigma \oplus \{\mathsf{Var} \leftarrow [\![E]\!]\}$	$[\![S_1\ ;\ S_2]\!]\sigma$	$=$	$[\![S_2]\!]([\![S_1]\!]\sigma)$

Fig. 3. Denotational semantics for SWL

We next present the denotational semantics for SWL. Each syntactic construct in SWL denotes a mathematical function from stores to stores. Note that to make the presentation simpler, we elide the store σ from all definitions which do not explicitly use it.

Finally, we get to the thorny issue of the semantics of the while loop. This is defined as follows:

$$[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{end}]\!]\sigma = \mathsf{FIX}\ F \quad \text{where } Fg = \begin{cases} g \circ [\![S]\!] & [\![B]\!]\sigma = \mathsf{true} \\ \mathsf{id} & [\![B]\!]\sigma = \mathsf{false} \end{cases}$$

where FIX denotes the least fixed point of the operator F with respect to the information ordering on functions. Further details can be found in [13]. The semantics of a procedure can then be defined as

$$[\![\mathsf{proc}(x_1, x_2, \ldots, x_n)\ \{\mathsf{local}\ l_1, l_2, \ldots, l_m\}\ S;\ \mathsf{return}\ E\ \mathsf{end}]\!] = \lambda x_1, \ldots, x_n.[\![E]\!]([\![S]\!]\sigma_{x,l})$$

where $\sigma_{x,l}$ denotes the state where identifiers $x_1, \ldots, x_n$ and $l_1, \ldots, l_m$ are in the range. In other words, the semantics of a procedure is a function from the value of all its inputs to the value of expression E as evaluated in the environment gotten from "running" S starting from $\sigma_{x,l}$ (as expected).

What we really want to do is to:

(1) Go from denotational semantics to *symbolic semantics* [14],
(2) Replace the denotational semantics of while with a semantics closer to its operational semantics,
(3) Introduce explicit loop counters.

Luckily, these last two requirements work very well together. We will explain how this is done in the next section. We finish this section with a quick introduction to symbolic semantics.

The basic idea is very simple – so simple in fact that for most practioners of Computer Algebra, the difference with denotational semantics will be difficult to fathom. Instead of working with the *semantic* theory of state-transformers (basically partial functions), we will work one step removed, that is with a *syntactic* theory. These are commonly known as "expressions" in Computer Algebra. However the main point of expressions is that they serve two rôles: they can be syntactically

$[\![\mathsf{Var}]\!]\sigma$	$=$	$\sigma(\mathsf{Var})$	$[\![i]\!]$	$=$	i
$[\![E_1 + E_2]\!]$	$=$	$[\![E_1]\!]$ `+` $[\![E_2]\!]$	$[\![E_1 * E_2]\!]$	$=$	$[\![E_1]\!]$ `*` $[\![E_2]\!]$
$[\![E_1 - E_2]\!]$	$=$	$[\![E_1]\!]$ `-` $[\![E_2]\!]$	$[\![E_1/E_2]\!]$	$=$	$[\![E_1]\!]$ `/` $[\![E_2]\!]$
$[\![E_1$ `^` $i]\!]$	$=$	$[\![E_1]\!]$ `^` $[\![i]\!]$	$[\![$`not` $E_1]\!]$	$=$	`not` $[\![E_1]\!]$
$[\![E_1$ `or` $E_2]\!]$	$=$	$[\![E_1]\!]$ `or` $[\![E_2]\!]$	$[\![E_1$ `and` $E_2]\!]$	$=$	$[\![E_1]\!]$ `and` $[\![E_2]\!]$
$[\![E_1 = E_2]\!]$	$=$	$[\![E_1]\!]$ `=` $[\![E_2]\!]$	$[\![E_1 < 0]\!]$	$=$	$[\![E_1]\!]$ `<` 0
$[\![\mathsf{Var} := E]\!]\sigma$	$=$	$\delta(\sigma, \{\mathsf{Var} \leftarrow [\![E]\!]\sigma\})$	$[\![S_1 \ ; \ S_2]\!]\sigma$	$=$	$\delta([\![S_1]\!], [\![S_2]\!]([\![S_1]\!]\sigma))$

Fig. 4. Symbolic semantics for SWL

manipulated, and they are denotations of mathematical functions. In other words, there is (in semantics) a large difference between the expression $x + \sin(x)$ and the mathematical function denoted by $\lambda x : \mathbb{R}.x + \sin(x)$. The first is really an abstract syntax tree (one can also think in terms of LISP s-expressions), while the second lives in the function space $\mathbb{R} \to \mathbb{R}$. Of course, we have a canonical map from the expression to its denotation, which is probably why these two concepts are so often seen as "the same". However, there is **no** reasonable converse mapping! Most functions $f \in \mathbb{R} \to \mathbb{R}$ do not have finite expressions which denote f. This structural property of possessing a finite expression is very powerful, and is part of the success of our method. In Fig. 4, we give the symbolic semantics for SWL. As for the rest of this paper we will only use *symbolic* semantics, we will re-use the $[\![\ ]\!]$ notation for this semantics. It is important to note that the types involved are quite different: the semantics of an expression is always a *syntactic expression*. There is still a store involved, but it is now a symbolic function, defined as

$$\sigma(\mathsf{Var}) = \begin{cases} v & \mathsf{Var} = v \ \in \sigma \text{ and } \sigma \text{ is a store} \\ v & \sigma = \delta(\sigma', \mathsf{Var} \leftarrow v) \\ \sigma'(\mathsf{Var}) & \sigma = \delta(\sigma', x \leftarrow v) \ \text{ and } x \neq \mathsf{Var} \\ \mathsf{Var} & \text{otherwise} \end{cases} \tag{1}$$

In other words, given the empty store $\sigma_0 = \emptyset$ (where we represent a store as a set of identifier-value pairs), we have that $[\![x + y]\!]\ \sigma_0 = x + y$, while in the store $\sigma_1 = \{(y, 3)\}$, $[\![x + y]\!]\ \sigma_1 = x + 3$. There is a simple correspondance between denotational and symbolic semantics — Theorem 4.1 in Sec. 4 makes this precise.

The semantics of statements also changes: instead of being a function from stores to stores, it now becomes a function from *store representation* to *store representation*. In Fig. 4, we use a new (syntactic) binary expression δ which represents the over-riding union $\oplus$ which was used previously in Fig. 3.

Definition 3.1. A *store representation* is a symbolic expression which can be evaluated to a unique store (of symbolic expressions).

We will use both explicit store representations ($\sigma_1 = \{(y, 3)\}$) and implicit representations ($\sigma_3 = \delta(\sigma_1, \sigma_2)$). We need to add just one more ingredient before we

can move on to recurrences for loops – names for store representations. If we were to use the symbolic semantics defined in Fig. 4 directly, even for straight-line programs we would frequently get exponential blow-up in the sizes of our expressions [15, example p. 10]. To preserve the structure of the straight-line program, as is also done by [14], each statement produces a *named* store representation, which is used in further computations. To prevent the blow-up of expressions, instead of computing $\sigma\mathsf{Var}$, we also use a symbolic representation for this step. That is we modify $[\![\mathsf{Var}]\!]\sigma$ from being $\sigma(\mathsf{Var})$ to $\epsilon(\sigma, \mathsf{Var})$ (where we pick ϵ to represent *evaluation*). For example,

```
i := 3;
r := r * i;
i := i + 1;
```

$$s_0 = \delta(\{\}, i \leftarrow \epsilon(\{\}, i))$$
$$s_1 = \delta(s_0, r \leftarrow \epsilon(s_0, r) * \epsilon(s_0, i))$$
$$s_2 = \delta(s_1, i \leftarrow \epsilon(s_1, i) + 1)$$

which is also one of the ideas in Maple's `LargeExpressions` package, which helps to produce dramatic improvements in certain large symbolic computations [16] (see also [15] for related work).

4. Recurrences

The heart of this work is to re-use a very old idea: a loop executes a certain number of times, which implicitly defines a non-negative integer "loop counter". We thus reify the number of iterations of a loop as a variable that we can manipulate. We then express the semantics of the loop body as a state transformer from the state at time t to the state at time $t+1$. A loop terminates at the first non-negative time (if it exists) that the loop condition becomes true. We will use "iteration counter" and "time" interchangeably, as we move between the traditional computer science view and the dynamical system view of code.

To do this requires not only that we have a fresh name for this new variable (easy via a standard `gensym` trick), we need to re-express the semantics of the body of a loop explicitly in terms of this new variable. Schematically, we want to perform the following transformation:

```
while Condition do
    state := F(state)
end
```

$$\Longrightarrow \qquad s_{t+1} = F(s_t)$$

where `state` should be thought of as a state vector and F as a vector-function. What we get as a result is that our loop bodies always end up translating to a first-order, generally non-linear, polynomial recurrences on the iteration counter (time). While this is certainly the correct intuition, one cannot simply add subscripts in the appropriate places in the symbolic semantics of Fig. 4 and get something semantically meaningful. If our programs were always written as explicit vector functions, this would be the case. But consider the following code fragment

```
while Condition do
    i := i + 1;
    r := r * i;
    i := i + 1;
```

$$
\begin{array}{lcl}
[\![\mathsf{Var}]\!]_w\sigma & = & \sigma(\mathsf{Var}(t)) \\
[\![\mathsf{Var} := E]\!]_w\sigma & = & \delta(\sigma, \{\mathsf{Var}(t+1) \leftarrow [\![E]\!]\sigma\}) \\
[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{end}]\!]\sigma & = & \begin{array}{l}\text{Let } B'_t = [\![B]\!]_w\sigma \\ \qquad S'_t = [\![S]\!]_w \\ \qquad e = \min(\{t \geq 0 \mid B'_t = \mathsf{true}\}) \\ \mu(S'_t, t = e, \epsilon(\vec{s}, \sigma))\end{array} \\
[\![\mathsf{proc}(\vec{x})\ \{\mathsf{local}\ \vec{l}\}\ S;\ \mathsf{return}\ E\ \mathsf{end}]\!] & = & [\![E]\!]([\![S]\!]\sigma'_{x,l})
\end{array}
$$

Fig. 5. Symbolic semantics – inside while loops, and procedures

```
    q := q + r/(i-2);
end
```

Clearly the i in the second line refers to the *new* value of i, so that as a vector function this needs to be translated to

$$\begin{pmatrix} i_{t+1} \\ r_{t+1} \\ q_{t+1} \end{pmatrix} = \begin{pmatrix} i_t + 2 \\ r_t * (i_t + 1) \\ q_t + r_t * \frac{i_t+1}{i_t} \end{pmatrix}. \tag{2}$$

But is this always possible? Yes it is, but there is a cost: one has to expand every definition of a variable into the expression using only time t values. Since at the start of a loop, all variables in the state vector will have such values, it is a matter of propagating these through. This is easy, but there is a huge potential for expression growth. Some program transformations can help mitigate this, but at the potential cost of additional state variables. In particular, conversion to Static Single Assignment (SAS) [17] improves the situation somewhat.

Another question is, why is the resulting system non-linear if the original system is holonomic? Essentially because the most natural way of coding some of these programs is to use implicit formulas for what are simply polynomials, because they can be computed more easily that way. So instead of having an $n+2$ in a coefficient, there will be an i_n with $i_{n+1} = i_n + 1, i_0 = 2$. Similarly, even though $\sum_{i=0}^{n} \frac{x^i}{i!}$ can be represented using a single recurrence, a program representing this will typically have 4 — one for each of i, $i!$, x^i and the sum itself.

The first 3 equations in Fig. 5 gives the semantics for a while loop, where only those aspects which differ from those in Fig. 4 are given, and where t is a *fresh* variable. We use the symbol $[\![_]\!]_w$ for clarity. In Fig. 5, we use $\vec{s}$ to denote those state variables which are assigned to during the execution of the loop body. Via μ, we pack up the results, where S'_t encodes the system of recurrences, $t = e$ gives the termination condition and $\epsilon(\vec{s}, \sigma)$ records the initial conditions of the relevant state variables. Implicit in this notation is that a μ encodes a change to all the state variables in $\vec{s}$, as a simultaneous assignment. In other words, μ represents running the "whole loop" and records the final state upon eventual termination. Note that e may or may not be given in closed form. Just as important, while $B'(t)$ will depend

on some of the variables in $\vec{s}$, it is frequently the case that it does not depend on all of them. Therefore only part of the solution of the system of recurrences S' is needed for determining termination. The last equation in Fig. 5 gives the semantics for whole procedures, where we use $\vec{x}$ to abbreviate $@_1, \ldots, @_n$ and similarly for $\vec{l}$. $\sigma'_{x,l}$ here denotes an environment where explicitly the identifiers $\vec{x}$ are *free*, in other words, *undefined*. Thus we will see (a subset of) those variables appear free in the result. The evaluation function σ needs to be modified a bit further from its original definition in Eq. (1) to

$$\sigma(\mathsf{Var}) = \begin{cases} v & \mathsf{Var} = v \ \in \sigma \text{ and } \sigma \text{ is a store} \\ v & \sigma = \delta(\sigma', \mathsf{Var} \leftarrow v) \\ \sigma'(\mathsf{Var}) & \sigma = \delta(\sigma', x \leftarrow v) \ \text{ and } x \neq \mathsf{Var} \\ ((S'_t)^{(e)}\sigma)(\mathsf{Var}) & \sigma = \mu(S'_t, t = e, \epsilon(\vec{s}, \sigma)) \\ \mathsf{Var} & \text{otherwise} \end{cases} \tag{3}$$

where $F^{(i)}$ denotes the i-times iterate of F. The μ case can be "optimized" if $\mathsf{Var} \notin \vec{s}$ since S'_t is the identity in that case.

While SWL does not allow nested loops, one can easily see what would be need to be done to allow this. Changing the syntax is trivial; more complex would be the changes to the symbolic semantics. The semantics of an inner-most loop would remain unchanged. But what is $[\![\mu]\!]_w$? When this can be given in closed-form, the interpretation is straightforward. But in other cases, it is quite unclear when it is even useful to trying to write out a mixed system encoding the nesting. This is the main reason we have chosen to leave SWL with only single loops at this point.

A further point to notice is that a lot of programs, especially those that come from holonomic functions, have a particular triangular structure visible in the recurrences. This structure is clearly visible in Eq. (2). Given such a triangular structure, it is much easier to get a closed-form, as each recurrence can be solved independently in order of dataflow dependence. We will comment further on this in Section 7.

We are now in a position to relate traditional denotational semantics to symbolic semantics. Since we use the same notation for both kinds of semantics, in the theorem below we use a superscript D for denotational and superscript S for symbolic. Below $\sqsubseteq$ is the natural information ordering on partial functions, where $g \sqsubseteq f$ means that f is defined everywhere g is (and equal to g there), and may be defined on a bigger set than g.

Theorem 4.1. *Let p be a valid SWL program, and V the sequence of input variables of p. Then we have that $[\![p]\!]^D \sqsubseteq \lambda V.[\![p]\!]^S$.*

Sketch. At the level of procedures, it is clear that one needs to take the free variables of $[\![p]\!]^S$ and abstract them out, thus the λV. Then proof proceeds by structural induction on the program syntax. The main lemmas required relate the action of $\oplus$ on stores and δ on store representations, and the behaviour of the least-fixed-point operator FIX versus the encoding of μ. The reason we only get $\sqsubseteq$ is

related to the fact that symbolic equations preserve definedness, but may remove some singularities (like that present in x/x). □

5. Termination

As mentionned above, rewriting the semantics as a system of recurrences can make it quite clear when termination depends on only a subset of the quantities computed. For example, this is always the case for `for` loops encoded using a `while`, as well as when an explicit but variable number of terms of a sequence are needed. When termination depends on a more complex criteria, for example that a term has gotten smaller than a certain ϵ, finding a closed-form for the termination condition becomes much harder, and is naturally undecidable in the general case (even in this restricted language).

It is worthwhile to examine a little closer what $\min(\{t \geq 0 \mid B'_t = \mathsf{true}\})$ really means. It says that the *semantics* of a loop is related to the *first time* $t \geq 0$ that the boolean condition B becomes `true`. Using this *stopping time* in the solution to the recurrence S'_t (along with the initial conditions) gives the full semantics. This should be intuitively clear – we claim that this is in fact much more intuitive than either the denotational semantics given as an operator fixed-point in a CPO [13], or when using relational semantics [18], as the Kleene closure of the relational version of S'. The correspondance in fact is much closer to that of the operational semantics (see Fig. 2).

Another aspect to consider is what happens when B'_t is particularly simple. For example, assume that B'_t is $u(t) < 0$ for some integer-valued state variable u. While solving $B'_t = \mathsf{true}$ may look like a satisfaction (SAT) problem, this is not a very fruitful way to approach the problem. What is much more important is the behaviour of $u(t)$ as a function of t. In particular, if we could determine that for $t \geq 0$, $u(t)$ is monotonically decreasing and $u(0) > 0$, then we can immediately deduce termination (although not in general in closed form). On the other hand, if for $t \geq 0$, $u(0) > 0$ but $u(t)$ is non-decreasing, then we have an immediate proof of non-termination. In this case, it is the **analytic** properties of $u(t)$ which are of interest, not its discrete properties. This should be a rich source of (semi-)decision procedures for loop termination, and we hope to get back to this problem in the future. Let us record the discussion above more formally.

Proposition 5.1. *Consider the semantics for a while loop given in Fig. 5. Then we have that:*

(1) If B'_t is of the form $E_t < 0$ for some integer-valued expression E, $E_0 > 0$ and E_t is a monotonically decreasing function of t, then there exists a least t such that $E_t < 0$.

(2) If B'_t is of the form $E_t < 0$ for some real-valued continuous expression E, $E_0 > 0$, $\lim_{t\to\infty} E_t < 0$ and E_t is a monotonically decreasing function of t, then there exists a least t such that $E_t < 0$.

Proof. The first item is because the positive integers are a discrete total order with no infinite descending chains. The second is a rephrasing of the intermediate value theorem. □

Of course, there are dual versions of the above statements with the inequalities reversed and with increasing functions. It is also easy to formalize the negative statements. While these conditions might seem quite special, they are nevertheless very useful to deal with practical problems.

Interestingly, we can obtain most of our results without worrying about termination at all. In other words, our method (like that of [19]) cleanly splits the process into one of dealing with invariants *first*, and then dealing with termination. In other words, we are claiming that the symbolic semantics given in the previous section are *modular*. In fact, 3 different aspects are completely separated: the recurrence relations from the body of the loop, the termination condition, and the initial conditions.

In our examples, termination will turn out to be either very easy or essentially impossible. More precisely, we'll either have a termination condition which is both explicit and monotonic (over the positive integers), or a completely implicit equation which is either not monotonic or whose monotonicity is a difficult theorem.

6. Closed-forms

Obtaining closed forms is in many ways the easiest part! This is because of the tremendous pre-existing technology already in place for dealing with recurrences, and especially those with polynomial coefficients. Here we crucially rely on much of the work of Sergei Abramov (and many of his co-authors), as implemented in Maple [2]. Many of the recurrences we will encounter can be reduced to hypergeometric summations, where Sergei Abramov has also contributed greatly [20–23] to cite but just a few.

As mentionned before, the basic algorithm involves sorting the recurrence equations in data-dependence order[d], solving each such system, substituting in the remaining equations, and iterating. Explicit initial conditions are always given, as this can sometimes significantly improve the solving process. Solving for the termination condition relies solely on Maple's `solve` command being able to invert the termination condition.

7. Examples

While in Sec. 2, we define an abstract programming language SWL, and proceed to give it proper semantics, in this example section we will give examples coming from a concrete programming language (Maple), in a subset which is trivially translatable

[d]something which we thought would be done by Maple's own `rsolve`, but apparently this is not the case.

into SWL. We have a prototype implementation for Maple (first described in [24]), as well as a Fortran 77 implementation (described in [5]).

Translating from Maple syntax or Fortran 77 syntax to SWL, and then to symbolic semantics, is a tedious engineering problem. Thankfully in Maple the *inert form* returned by `ToInert` makes this quite straightforward; a lot more technology is required for Fortran 77, but again is straightforward via leveraging standard compiler tools (such as ANTLR [25]).

Listing 3. Factorial

```
ff:=proc(n)
    local j, fac;
    j:=1;
    fac:=1;
    while j<>n do
        j:=j+1;
        fac:=fac*j;
    end do;
   fac;
end proc:
```

Listing 4. Factorial variant

```
gg:=proc(a, b, n)
    local j, fac;
    j:=a;
    fac:=b;
    while j<>n do
        j:=j+1;
        fac:=fac*j;
    end do;
   fac;
end proc:
```

Example 7.1 (Factorial). *We get back to our factorial example (Listing 3). While this example is extremely simple, it is outside the reach of most other methods since the loop invariants are not polynomial. On this code, our method returns the explicit formula* $n!$.

Example 7.2 (Shifted factorial). *Since the method relies on recurrences, it is quite robust against minor code variations, unlike other methods. In particular, the code in Listing 4 gives* $\frac{\Gamma(n+1)b}{\Gamma(a+1)}$*, as expected. But one can make further modifications, say changing j:=j+1 to j:=j+c with* `c` *a new input, the result is now computed as*

$$c^{\left(\frac{-a+n}{c}\right)}\Gamma\left(\frac{n+c}{c}\right)b\Gamma\left(\frac{a+c}{c}\right)^{-1}$$

The previous examples also exhibit an additional benefit of this method for debugging code. If one is expecting a particular function (say factorial) and the result computed is clearly different, one might be able to see from the returned formula what the actual problem is. In listing 4 for example, only with $b/\Gamma(a+1) = 1$ will the result computed actually be factorial.

Other discrete functions can be dealt with too, and these functions can involve either discrete or continuous parameters.

Example 7.3 (Generalized Binomial). *Listing 5 shows a routine that purports*

Listing 5. Generalized binomial

```
bin := proc(u, k)
local res, i;
    res := 1;
    for i from 1 to k do
        res := res * (u-i+1)/i;
    end do;
    res;
end proc:
```

Listing 6. Chebyshev polynomial

```
chebyshev:=proc(n)
local i, u0, u1, v;
    u0 := 1;
    u1 := x;
    for i from 2 to n do
        v := u1;
        u1 := -u0+2*x*u1;
        u0 := v;
    end do;
    u1;
end proc:
```

to compute $\binom{u}{k}$. *Our program says that this computes*

$$\frac{(-1)^k \Gamma(-u+k)}{\Gamma(-u)\Gamma(k+1)}. \tag{4}$$

Lemma 7.1 shows that these denote the same function.

Lemma 7.1. *Equation (4)* $= \binom{u}{k}$ *for all positive integers* k *and all* $u \in \mathbb{R} \setminus \mathbb{N}$. *Furthermore, for all* $n \in \mathbb{N}$ *and* $k \in \mathbb{N}$,

$$\lim_{x \to n} \frac{(-1)^k \Gamma(-x+k)}{\Gamma(-x)\Gamma(k+1)} = \binom{n}{k}.$$

Proof. If we denote by E_n the shift operator for the variable n and I the identity operator, then we have that both functions satisfy the recurrences $E_k + \frac{k-u}{k+1} I = 0$ and $E_u + \frac{u+1}{k-u-1} I = 0$, and are equal at $k = 1$ for all $u \in \mathbb{R} \setminus \mathbb{N}$, which proves the first assertion. The second part is proved by analytic continuation. □

Equation (7.3) illustrates an interesting phenomenon which is fairly pervasive in computer algebra: how does one interpret the closed-form results from various algorithms? Sergei's ISSAC 2006 paper [26] provides a very nicer answer to this problem for the case of summative of P-recursive sequences.

Example 7.4 (Chebyshev polynomial). *Listing 6 shows a routine to compute the Chebyshev polynomials of the first-kind. It is worthwhile noting that these polynomials satisfy a second-order linear recurrence, while single loops naturally encode first-order recurrences. Thus the standard trick used to change an* n^{th} *order linear recurrence into a system of* n *first-order (linear) recurrences is used. Our method returns*

$$\frac{1}{2}\left(\left(x+\sqrt{x^2-1}\right)^{-n} + \left(x-\sqrt{x^2-1}\right)^{-n}\right)$$

which is indeed the closed-form for the Chebyshev polynomials.

The next example is quite peculiar as the code is derived from the computation of a *constant* which comes from the evaluation of a holonomic function (in this case the simplest such function, the exponential) at a point. Normally constants are outside the realm of holonomic techniques, since a variable is needed with respect to which one can either take a difference or a differential. However, since e^1 cannot be computed exactly, one has to find an approximation. The simplest such approximation is to truncate the power series for e^x after a certain number of terms. Holonomic techniques now apply directly, because the computation in the inner-loop depend on the loop counter t, inducing a recurrence.

Listing 7. e^1

```
exp1 := proc(n)
local res, j, i;
    res := 1;
    j := 1;
    for i from 1 to n do
        j := j/i;
        res := res + j;
    end do;
    res;
end proc:
```

Listing 8. BesselJ

```
Bessel := proc(z, nu, m::posint)
    local res, i, t;
    (res,t) := (0,1);
    for i from 0 to m-1 do
        res := res + t;
        t := -t*z^2/(4*(i+nu+1)*(i+1));
    end do;
    res;
end proc;
```

Example 7.5 (exp(1)). *While our intuition is that Listing 7 shows a program that computes e^1, it in fact computes an approximation. More precisely, it computes* exactly

$$\frac{e^1\Gamma(n+1,1)}{\Gamma(n+1)}$$

(where both the Γ and the 2-argument incomplete Γ function appear). As expected, the error term $\Gamma(n+1,1)/\Gamma(n+1)$ converges to 1 very quickly with rising n.

Example 7.6 (BesselJ). *Our system reports that this computes* exactly

$$\frac{\Gamma(\nu+1)}{z^\nu}\left[J_\nu(z)\,2^\nu - \frac{\left(z^{2m+\nu} - s^{(+)}_{2m+1+\nu,\nu}(z)\right)\left(\frac{-1}{4}\right)^m}{\Gamma(m+1+\nu)\,\Gamma(m+1)}\right]$$

where J_ν is the Bessel function of the first kind, while $s^{(+)}$ is known as Lommel's s function.

What is interesting about the previous example is not what it computes exactly, but that we can recognize (and with a bit more work, compute) that this is a *Taylor approximation* for the non-singular part of Bessel's function at the origin.

It is worthwhile noting that the examples above are generally distinct from the ones of related work, like that of Rodríguez-Carbonell and Kapur [19] and of Kovács

and Jebelean [27, 28]. Below, we give the examples from [27] and [19] which can be translated in to SWL.

Listing 9. Integer Division

```
div := proc(x, y)
    local lquo, lrem;
    lquo := 0;
    lrem := x;
    while (y<=lrem) do
        lrem := lrem - y;
        lquo := lquo + 1;
    end do;
    (lrem, lquo);
end;
```

Listing 10. Fibonacci

```
fibo := proc(n)
    local F, H, i;
    i := n;
    F := 1;
    H := 1;
    while i>1 do
        H := H + F;
        F := H - F;
        i := i - 1;
    end do
    F;
end;
```

Example 7.7 (Integer division). *This code needs a minor extension to SWL — returning multiple values. Given the code in Listing 9, we return the explicit form*

$$(\mathsf{lrem} = x - y\left\lfloor\frac{x}{y}\right\rfloor,\ \mathsf{lquo} = \left\lfloor\frac{x}{y}\right\rfloor).$$

The floor *function appears courtesy of solving for the stopping condition, as the recurrences are trivial with solutions* $\mathsf{lquo}_t = t$ *and* $\mathsf{lrem}_t = x - yt$.

Example 7.8 (Fibonacci). *Given the code (Listing 10) to compute the Fibonacci numbers, our system returns a fairly complex expression in term of* $1/\phi$ *where* $\phi = \frac{(1+\sqrt{5})}{2}$ *is the golden ratio instead of the expected* $\frac{\phi^n+\hat{\phi}^n}{\sqrt{5}}$. *One can nevertheless verify that the answer is correct (and derived automatically).*

In direct contrast to ex. (4), the previous example is an instance (along with ex. (5)) where the tools' answer must be recognized as equivalent to the "correct" answer, rather than an instance of the tool detecting a problem.

```
Sqrt := proc(n::posint) local a, s, t;
    a := 0; s := 1; t := 1;
    while s<=n do
        a := a + 1;
        s := s + t + 2;
        t := t + 2;
    end;
    a;
end proc;
```

Example 7.9 (Square root). *The code in Listing 11 computes an approximation to the square root of a positive integer n. In fact, it computes exactly $\lfloor\sqrt{n}\rfloor$, where again the floor function appears courtesy of solving the stopping condition.*

8. Approximation

Consider the results that our method gives on the following examples [e]:

	exact	approximates
ex. (7)	$\frac{e\Gamma(n+1,1)}{\Gamma(n+1)}$	e
$\sum_{i=0}^{n} i!$	${}_2F_0(1,1 \mid 1) - (n+1)!{}_2F_0(1,n+2 \mid 1)$	∞ or ${}_2F_0(1,1 \mid 1)$
ex. (8)	big mess	Bessel J
ex. (6)	Chebyshev T	∞

We get a very interesting situation when we ignore the stopping condition for a particular piece of code. We get in a sense an "abstract" version of the meaning of a particular piece of code, independent of its termination condition. In some cases, the result is not very meaningful and informative, while in others this "abstract" meaning is in fact exactly the code's intent! This dichotomy reveals itself to be deeper still: if it makes sense to interpret a certain piece of code as an *approximation*, then we are usually most interested in the limit as the iteration count goes to infinity, while for discrete computations, it really is the value at the stopping time which is of interest. It certainly would be interesting to see if it would be possible to re-interpret this in the setting of stream computations [29].

However this highlights a particular problem which we do not know, at present, how to solve: certain codes are clearly intended to be finite approximations to infinite computations, while others are complete computations in and of themselves. Certainly there is no syntactic difference between these two cases. Might there be a semantic difference? There cannot be a definitive answer to this problem, as approximating programs might be of independent interest. Also, for those who have not yet seen this phenomenon, it is interesting to see that holonomic technology often performs resummation of divergent series without any need to specifically ask for this. Some may consider this a problem, as it is in fact very difficult to prevent this from happening.

9. Conclusion

We have added one more piece to the holonomy dictionary: programs. Whether one presents a holonomic function as a differential equation, a recurrence for its coefficients, as a program to compute approximations to the Taylor series or as a program to compute the coefficients, it appears that we now have the technology in place to translate between these representations. It is important to note that in some cases (approximation), we do not care as much about the *values* that these

[e] We do not show the code for the second example, but it is a trivial modification of that of ex. (7)

programs output, rather it is the program text itself which embodies the information we need. In other words, it is the explicit intentional content of the program rather than its extentional (observable) behaviour which is the key to our results.

This work definitely opens some questions, even for code as simple as what is allowed by SWL. Given a general SWL program, what kind of function does it represent? Can one classify exactly those SWL programs that are obtained from straightforward encodings of holonomic objects? In most of the examples we have looked at, the systems of recurrences exhibit a very clear block structure. In fact, we are lead to conjecture that holonomic objects *must* exhibit a natural block structure. Furthermore, there also seems to be some natural relations between the sizes of these blocks and the order of the underlying linear recurrence. This seems worth further investigation.

Of course, none of the more spectacular results (ex. (8) comes to mind) would have been possible without Sergei Abramov's decisive contributions to the theoretical and practical advances in summation and recurrences equations.

Acknowledgements

Work supported in part by NSERC Discovery Grant RPG262084-03.

The author wishes to thank Yun Zhai who built a first Maple prototype, described in [24], and Olivier Dragon who extended this work to Fortran 77 [5]. Ryszard Janicki provided funding for Yun Zhai and the encouragement necessary to get this work completed.

References

[1] J. Carette and P. K. Chowdhury, Symbolic interpretation of legacy assembly language., in *WCRE*, eds. A. D. Lucia and S. E. Sim (IEEE Computer Society, November 2005).
[2] S. A. Abramov, J. J. Carette, K. O. Geddes and H. Q. Le, *Journal of Symbolic Computation* **38**, 1303(October 2004).
[3] F. Chyzak and B. Salvy, *Journal of Symbolic Computation* **26**, 187(August 1998).
[4] F. Chyzak, Gröbner bases, symbolic summation and symbolic integration, in *Gröbner Bases and Applications*, eds. B. Buchberger and F. Winkler, London Mathematical Society Lecture Notes Series, Vol. 251 (Cambridge University Press, 1998) p. 3260. Proceedings of the Conference 33 Years of Grbner Bases, ISBN 0-521-63298-6.
[5] O. Dragon, Reverse engineering of scientific computation fortran code, Master's thesis, Dept. of Computing and Software, McMaster University(July 2006).
[6] B. Salvy and P. Zimmermann, *ACM Transactions on Mathematical Software* **20**, 163 (1994).
[7] L. Meunier and B. Salvy, ESF: An automatically generated encyclopedia of special functions, in *Symbolic and Algebraic Computation*, ed. J. R. Sendra (ACM Press, 2003). Proceedings of ISSAC'03, Philadelphia, August 2003.
[8] A. Tucker and R. Noonan, *Programming Languages: Principles and Paradigms* (McGraw Hill, 2002).
[9] J. V. Tucker and J. I. Zucker, *ACM Trans. Comput. Logic* **5**, 611 (2004).
[10] D. Goldberg, *ACM Comput. Surv.* **23**, 5 (1991).

[11] W. Kahan, *Commun. ACM* **8**, p. 40 (1965).
[12] V. Adamchik, *The Ramanujan Journal* **9**, 271(June 2005).
[13] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* (MIT Press, 1981).
[14] T. Fahringer and B. Scholz, *Advanced Symbolic Analysis for Compilers: New Techniques and Algorithms for Symbolic Program Analysis and Optimization*, Lecture Notes in Computer Science, Vol. 2628 (Springer-Verlag, 2003).
[15] E. Kaltofen, *Journal of the ACM* **35**, 231 (1988).
[16] W. Zhou, J. Carette, D. J. Jeffrey and M. B. Monagan, Hierarchical representations with signatures for large expression management, in *Proceedings of Artificial Intelligence and Symbolic Computation*, 2006.
[17] A. W. Appel, *Modern Compiler Implementation: In ML* (Cambridge University Press, New York, NY, USA, 1998).
[18] D. Kozen, *Transactions on Programming Languages and Systems* **19**, 427(May 1997).
[19] E. Rodríguez-Carbonell and D. Kapur, Program verification using automatic generation of invariants, in *Proceedings of ICTAC 2004*, , Lecture Notes in Computer Science Vol. 3407 (Springer-Verlag, 2005).
[20] S. A. Abramov and M. Petkovšek, Gosper's algorithm, accurate summation, and the discrete Newton-Leibniz formula., in *ISSAC*, 2005.
[21] S. A. Abramov and M. Petkovsek, *J. Symb. Comput.* **33**, 521 (2002).
[22] S. A. Abramov and M. Bronstein, On solutions of linear functional systems., in *ISSAC*, 2001.
[23] S. A. Abramov, M. Petkovsek and A. A. Ryabenko, *Discrete Mathematics* **210**, 3 (2000).
[24] J. Carette, R. Janicki and Y. Zhai, Program verification by calculating relations, in *Proc. 15th IASTEAD Conf. on Applied Simulation and Modelling*, (Acta Press, Rhodos, 2006).
[25] T. J. Parr, *ANTLR, ANother Tool for Language Recognition*).
[26] S. A. Abramov, On the summation of p-recursive sequences, in *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, (ACM Press, New York, NY, USA, 2006).
[27] L. Kovacs and T. Jebelean, Automated Generation of Loop Invariants by Recurrence Solving in Theorema, in *Proc. of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04)*, eds. D. Petcu, V. Negru, D. Zaharie and T. Jebelean (Mirton Publisher, Timisoara, Romania, 26-30 September 2004).
[28] L. Kovács and T. Jebelean, Finding Polynomial Invariants for Imperative Loops in the Theorema System, in *Proceedings of Verify'06 Workshop, IJCAR'06, The 2006 Federated Logic Conference*, eds. S. Autexier and H. MantelAugust 15-16 2006.
[29] J. V. Tucker and J. I. Zucker, Theory of computation over stream algebras, and its applications., in *MFCS*, 1992.