Fundamenta Informaticae XXI (2006) 1001–1022
IOS Press

Computing Properties of Numerical Imperative Programs by Symbolic Computation

Jacques Carette[*] **Ryszard Janicki**[†]
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada L8S 4K1
{carette,janicki}@mcmaster.ca

Abstract. We show how properties of an interesting class of imperative programs can be calculated by means of relational modeling and symbolic computation. The ideas of [5, 26] are implemented using symbolic computations based on *Maple* [30].

1. Introduction

In the late sixties and early seventies, a technique for verifications and analysis of computer programs based on a calculus of functions and relations was proposed ([3, 4, 5, 11, 14, 18, 26, 27, 28] and others). The initial ideas were due to Robert W. Floyd [14], but *the most advanced and sophisticated version* was based on the approach proposed by Antoni Mazurkiewicz in [26] and Andrzej Blikle and Antoni Mazurkiewicz in [5] (see also [4, 18, 27, 28]).

The technique was based on calculating *Tail Function* defined in [26] as a tool to model *continuations*. If a program terminates, its meaning is defined by the value of Tail Function from its beginning. A novelty was to use the Calculus of Relations even for entirely deterministic programs. For terminating iterative programs the approach could be regarded as a predecessor (and a special case) of Kozen's Kleene Algebras with Tests [25].

Despite many theoretical and methodological advantages (it rather emphasizes *calculation* instead of *proving* as in the more popular method of Hoare [17]), the technique has never become widely accepted, probably because of the huge amount of symbolic computations that need to be performed for even

[*]Partially Supported by NSERC of Canada Grant RPG262084-03.
[†]Partially Supported by NSERC of Canada Grant.

relatively simple cases, even though in most cases these are rather easy predicate calculus computations albeit with formulas a couple of pages long.

The situation has dramatically changed today, as we have very powerful tools supporting symbolic computation such as *Maple* [30] and *Mathematica* [39], and relatively easy to use theorem provers such as *Simplify* [12]. The problem is still non-trivial, as the most general cases are undecidable, but for many practical cases an efficient solution seems to be feasible.

We were motivated by the approach of Blikle and Mazurkiewicz and we show an example of such a computation in Section 2. But instead of expressing relations as predicates, we represent them as *symbolic recurrence relations* and then use a technique called *symbolic execution*. The main idea behind symbolic execution is to use symbolic expressions as input values and to simulate the execution of the program statements on this symbolic input. Symbolic execution has wide range of potential applications, however, it has fallen out of favour for proving properties of programs. This is because naïve symbolic execution can lead to exponential blow-ups (or worse [41]). Recent work, amongst which one can find [23, 33, 36], has shown how useful this can be when used in controlled situations.

Our symbolic analysis can be seen as a kind of compiler which can translate the input programs into a symbolic expression, and then transform this expression into an output expression. From our point of view, recursion and looping are essentially equivalent, and so we will mainly restrict ourselves to loops as the source of our main difficulties. The basic technique used in such cases is to find “loop invariants” proposed by C. A. R. Hoare in 1969 [17]. Unfortunately finding them is often problematic and research on how to find them in some automatic manner, after some very interesting early work [15, 21, 22, 38] has only just restarted [23, 33].

We will show that for many frequently occurring loops, finding invariants is not necessary as the symbolic expression for the output can be generated explicitly by solving the recurrence equations generated from the loop. Even if, due to structural complexity of a loop, finding loop invariants is necessary, the technique we have proposed might often help substantially.

Since we do not represent relations as predicates but as symbolic recurrence relations over particular rings, this induces a definite restriction on the structure of the programs that can be thus handled. Despite this restriction, our method is still applicable to a large variety of numerical programs. In fact, this was a big part of the first author’s original motivation – how to deal with exactly that class of numerical programs. The intuition here came from a branch of mathematics that deals with *holonomic functions and sequences* where there exists very powerful theoretical results, as well as being the foundation for quite practical work [8, 9, 29, 34]. From that work, we knew that certain kinds of programs which correspond strongly to holonomic objects should be completely “solvable”. It was even fairly clear what those programs should look like, as packages like Maple’s `gfun` can produce code from recurrences [34].

While influenced by axiomatic semantics [17], our techniques owes much more to *denotational semantics*[1] as well as borrowing some ideas from operational semantics.

Some preliminary results in this direction were first presented in [6]. More complete versions were presented by both authors at conferences (by the first author at a conference in honour of Sergei Abramov where the holonomic and closed-form aspects were emphasized, and by the second author at a conference in honour of Antoni Mazurkiewicz where the semantics and relations with Tail Functions were emphasized). Prototypes were implemented first by Y. Zhai [40] for Maple, and subsequently by O. Dragon

[1]The paper [26] was not initially appreciated by the denotational semantics community, but since the mid eighties it is widely credited as being one of the first papers on ‘’continuations” [35]

[13] for Fortran 77. It is planned that these prototypes be made more widely available [31].

We assume a reader has some basic knowledge of programming languages semantics, computer algebra and symbolic computation.

The next section provides the intuition (based on relations) and the initial motivation for the work. In section 3, we formally define the problem we are going to solve, while in section 4 we give precise semantics to our programming language. In section 5 we tie in the recurrences to the symbolic semantics, and then deal with issues of termination in section 6. Section 7 gives a brief overview of how to obtain closed forms, and section 8 gives a variety of examples computable with our method, as implemented. Lastly we close with some conclusion and some further work to be done on this method.

2. Intuition and Motivation

The example below (based on [4, 5, 27]) provides the initial motivation and illustrates well the main ideas. In principle we first translate a program into a *relational expression* and then we will try to obtain the program properties by analyzing this relational expression. A classical approach to the latter part involves substantial use of the *predicate calculus*.

Consider the well-known procedure factorial, written in a small subset of Maple [30]:

```
factorial:=proc(n::posint)
local i, fac;
   i:=1;
   fac:=1;
   while i < n do
    begin
      i:=i+1;
      fac:=fac*i;
    end;
return fac
end proc;
```

Since n does not change its value in the above program we may consider it as a constant, so we may assume the above program has two integer variables i and *fac*. Define $D = \mathbb{Z} \times \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers, and denote the elements of D as (i, fac). Each assignment statement can be modeled by a function $F_i : D \to D$, $i = 1, 2, 4, 5$, in the following manner:

"`i:=1`" corresponds to $F_1(i, \mathit{fac}) = (1, \mathit{fac})$,
"`fac:=1`" corresponds to $F_2(i, \mathit{fac}) = (i, 1)$,
"`i:=i+1`" corresponds to $F_4(i, \mathit{fac}) = (i + 1, \mathit{fac})$, and
"`fac:=fac*i`" maps to $F_5(i, \mathit{fac}) = (i, \mathit{fac} \cdot i)$.

The test "`i<n`" can be modeled by two partial identity functions, $I_3, \bar{I}_3 : D \to D$, where I_3 models "`i<n`", and $\bar{I}_3$ models its complement, i.e. "`i≥n`". More precisely,

"`i<n`" corresponds to $I_3(i, \mathit{fac})$, and
"`i≥n`" corresponds to $\bar{I}_3(i, \mathit{fac})$, where ($\perp$ denotes *undefined*)

$$I_3(i,\mathit{fac}) = \begin{cases} (i,\mathit{fac}) & \text{if } i < n \\ \perp & \text{otherwise} \end{cases} \qquad \bar{I}_3(i,\mathit{fac}) = \begin{cases} (i,\mathit{fac}) & \text{if } i \geq n \\ \perp & \text{otherwise} \end{cases}$$

As we had mentioned previously, for terminating programs without recursion the approach could be regarded as a predecessor (and a special case) of Kleene Algebras with Tests [25], so we can use the following scheme.

Let R, R_1, R_2 be relations (each function is a relation!) that model the program statements `S`, `S1`, `S2`, respectively. Let `T` be a test modeled by partial identities I_T and $\bar{I}_T$, and let the symbols "$\circ$" and "$*$" denote the (forward) composition of relations, and transitive and reflexive closure of relations (Kleene star), respectively.

Then :

"`S1;S2`" is modeled by $R_1 \circ R_2$,
"`if T then S1 else S2`" is modeled by $(I_T \circ R_1) \cup (\bar{I}_T \circ R_2)$, and
"`while T do S`" is modeled by $(I_T \circ R)^* \circ \bar{I}_T$.

Using this scheme one can easily model the above program by writing the following (symbolic) relational expression:

$$F = F_1 \circ F_2 \circ (I_3 \circ F_4 \circ F_5)^* \circ \bar{I}_3$$

There are many methods of transforming programs like `factorial` into relational expressions. In the original paper [26] a technique called "label elimination" was proposed, solving appropriate equations was proposed in [4, 5]. Both techniques can be applied to recursive programs, however for recursive programs they do not always guarantee obtaining a closed relational expression. For non-recursive programs without "`goto`" the direct translation of program statements into relational expressions seems to be the easiest method. Since the loop "`for`" can easily be simulated by "`while`" and "`if-then-else`", this method covers the loops "`for`" as well.

The techniques proposed in [4, 5, 26] can be applied to non-deterministic programs as well, however the method described in this paper and implemented in our prototypes ([40, 13]) are restricted to deterministic programs, which means that all "atomic relations" corresponding to simple assignment statements (F_1, F_2, F_4, F_5 for `factorial`) are functions, i.e. all "atomic relations" in the final relational expression are also functions.

If R_1 and R_2 are (possibly partial) functions, calculating $R = R_1 \circ R_2$ is easy: $R(x_1, ..., x_n) = R_2(R_1(x_1, ..., x_n))$. If at least one of R_1, R_2 is not a function, in general, we have to use the rule: $(x_1, ..., x_n)R_1 \circ R_2(z_1, ..., z_n) \iff \exists(y_1, ..., y_n)\ (x_1, ..., x_n)R_1(y_1, ..., y_n) \wedge (y_1, ..., y_n)R_2(z_1, ..., z_n)$. Nevertheless, it might happen that $R_1 \circ R_2$ is a function even if both R_1 and R_2 are not. In general $R_1 \cup R_2$ is not a function, even if both R_1 and R_2 are functions. Similarly, $R^* = \bigcup_0^\infty R^i$ is almost never a function, even if R is a function, since if R is a function, then $(x_1, ..., x_n)R^*(y_1, ..., y_n) \iff \exists i \geq 0.\ (y_1, ..., y_n) = R^i(x_1, ..., x_n)$, and this may happen for many, even infinite number of i's. However the following folklore result can easily be proved.

Lemma 2.1.

1. For any test T, if R_1 and R_2 are functions then $(I_T \circ R_1) \cup (\bar{I}_T \circ R_2)$ is always a *function*.

2. For any test T, if R is a function, then $(I_T \circ R)^* \circ \bar{I}_T$ is either a *function* or the empty relation.

3. For any test T, if R is a function and $(I_T \circ R)^* \circ \bar{I}_T \neq \emptyset$, then

$$((I_T \circ R)^* \circ \bar{I}_T)(x) = R^{k(x)}(x)$$

where $k(x)$ is the smallest j such that $\bar{I}_T(R^j(x_1, ..., x_n))(x) \neq \bot$. □

Lemma 2.1(3) is particularly helpful if $k(x)$ can (easily) be calculated, for instance if a closed formula for $k(x)$ can be obtained, which happens quite often (in particular for `for` loops), but not always.

Also, despite the above results, calculating functions that model even simple programs is very labour consuming and error prone. To illustrate this technique we will calculate the function $F = F_1 \circ F_2 \circ (I_3 \circ F_4 \circ F_5)^* \circ \bar{I}_3$ that models our simple program `factorial`. The most difficult part is to calculate the function $(I_3 \circ F_4 \circ F_5)^* \circ \bar{I}_3$.

Define $G = I_3 \circ F_4 \circ F_5$ and $H = G^* \circ \bar{I}_3$, so $F = F_1 \circ F_2 \circ H$. First note that $(F_1 \circ F_2)(i, \mathit{fac}) = F_2(F_1(i, \mathit{fac})) = (1, 1)$, so $F(i, \mathit{fac}) = H(F_2(F_1(i, \mathit{fac}))) = H(1, 1)$. For the function G we have:

$$G(i, \mathit{fac}) = (I_3 \circ F_4 \circ F_5)(i, \mathit{fac}) = F_5(F_4(I_3(i, \mathit{fac}))) = \begin{cases} (i+1, \mathit{fac} \cdot (i+1)) & \text{if } i < n \\ \perp & \text{if } i \geq n \end{cases}$$

Similarly :

$$G^2(i, \mathit{fac}) = G(G(i, \mathit{fac})) = \begin{cases} (i+2, \mathit{fac} \cdot (i+1) \cdot (i+2)) & \text{if } i+1 < n \\ \perp & \text{if } i+1 \geq n \end{cases}$$

Hence :

$$G^j(i, \mathit{fac}) = \begin{cases} (i+j, \mathit{fac} \cdot (i+1) \cdot (i+2) \cdot \ldots \cdot (i+j)) & \text{if } i+j-1 < n \\ \perp & \text{if } i+j-1 \geq n \end{cases}$$

Notice that this last step requires a small amount of human ingenuity to "see" the pattern (although we will show how this can be automated in some cases). From Lemma 2.1(3) it follows $H(i, \mathit{fac}) = G^k(i, \mathit{fac})$ where $k = k(i, \mathit{fac})$ is the smallest j such that $\bar{I}_3(G^j(i, \mathit{fac})) \neq \perp$. In this case we can easily show that there is only one such j and that $k(i, \mathit{fac}) = n - i$. First note that $\bar{I}_3(G^j(i, \mathit{fac})) \neq \perp$ implies $G^j(i, \mathit{fac}) \neq \perp$, i.e. $G^j(i, \mathit{fac}) = (i+j, \mathit{fac}')$ and $i + j - 1 < n$. Furthermore $\bar{I}_3(i+j, \mathit{fac}') \neq \perp$ implies $i + j \geq n$. From $i + j - 1 < n$ and $i + j \geq n$ we immediately get $i + j = n$, or $j = n - i$. Hence $k(i, \mathit{fac}) = n - i$, i.e.

$$H(i, \mathit{fac}) = G^{n-i}(i, \mathit{fac}) = (n, \mathit{fac} \cdot (i+1) \cdot (i+2) \cdot \ldots \cdot n).$$

This means $F(i, \mathit{fac}) = H(1, 1) = (n, n!)$, so the program is correct (in the sense of partial correctness with respect to the specification that $\forall n \in \mathbb{N}.\texttt{factorial}(n) = n!$), although this last step still requires a non-trivial proof. To make this technique feasible for bigger, more realistic programs, we need a tool that would be able to do all those symbolic calculations. The reasoning presented above rely heavily on Lemma 2.1(3) and is rather typical for human beings. Many steps and observations are not easy to mechanize, which suggest that perhaps we should be looking for a different approach.

Our prototypes [40, 13] will take the text of the program `factorial` as an input (in Maple and Fortran 77 respectively) and will return the text "$n!$" as the main output. In the next sections we will show how it can be done with some help from *Maple* [30]. *The main idea is not to represent relations as predicates but to instead translate them into symbolic recurrence relations.* Note the Maple prototype [40] can deal with limited recursion as well, while the Fortran 77 prototype [13] can deal with vector and matrix algebra.

3. Formal description of the problem

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

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

with the obvious operational and denotational semantics[2] 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, no conditional[3]. Furthermore, we can only define single-level procedures, as neither the syntax for statements S nor expressions E allow procedure invocation[4] , 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, if our target is just to deal with holonomic objects, this language is is more general than needed, as simple holonomic objects require only a single loop. Conversely, this small programming language can be trivially mapped injectively in various programming languages, notably `C`, `Fortran`, and `Maple`.

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 3.1.
A program p is said to be *valid* if all (local) variables are initialized before they are read. □

[2]A complete denotational semantics for SWL can be found in Section 4
[3]Although one can still encode a condition via two while loops, and a boolean via integers
[4]we can easily allow calls to procedures which return polynomial functions of their parameters

Suppose that we denote by $[\![p]\!]$ the mathematical function that a program p computes (i.e. its denotational semantics, given for SWL in section 4). It is already easy to see the following:

Proposition 3.1. 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 proposition is easily proved by structural induction on the syntax of programs.

Note that we are dealing with *abstract computation* [37], 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 [16, 19], are algebraically "invisible", and thus we can also deal with programs written using good numerical analysis methods.

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 in classical mathematical expressions which denote the same mathematical object. Certain 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 3.1. (Factorial again)
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;
end proc
```

Let us call this procedure f. As we have seen before,

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

where we λ notation for function from the λ-calculus ([35] for its use in denotational semantics). More interestingly for us, we have that

$$\forall t \in \mathbb{N}. [\![f]\!](t+1) = (t+1) \cdot [\![f]\!](t),\ [\![f]\!](0) = 1 \tag{1}$$

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. (3.1), we can show that the body of the while loop satisfies

$$r(t+1) = r(t) \cdot i(t), i(t+1) = i(t) + 1, r(0) = 1, i(0) = 1$$

(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 (1). Our task then is to make this precise.

Problem 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. We often want 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 [5]. 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. In this way, our solution is very modular.

Example 3.2. (Factorial revisited)
We made some claims about inverting the process of generating procedures from holonomic equations. If we give eq. 1 to the Maple routines [6] `gfun[rectoproc]` (which given a linear recurrence equation with polynomial coefficients will return a procedure for computing the n^{th} term), and `LREtools[REtoproc]`

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
```

Figure 1. automatic factorials

(which does the same but via a different algorithm), we obtain the procedures given in fig. 1. For both of these programs, we obtain closed-form $\Gamma(n+1)$ and $n\Gamma(n)$ respectively. As is well-known, Γ is the unique convex continuous function which interpolates $n!$, and it satisfies the functional equation $\Gamma(k+1) = k\Gamma(k)$. From these, we see that these results are equivalent to $n!$ on the positive integers, as required.

4. Semantics

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 $\mathbb{R}$).

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

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

Figure 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

[5] However, it seems that what "closed forms" means depends on the user. When giving talks about this work, some people loved to see the exact special functions coming out, while others wanted to see unevaluated sums. Our prototype now returns both.

[6] `gfun` is a Maple package revolving around generating functions, while `LREtools` is a package for dealing with linear recurrence equations.

languages, *except* for while loops, where we model the operational semantics much more closely. It is important to note that this is quite close in spirit to what is done in [26].

We next present the denotational semantics for SWL. Each syntactic construct in SWL denotes a mathematical function from stores to stores. The semantics of the constructs without while loops and procedures is given in fig. 3, where $\oplus$ represents the overriding union operator. Note that to make the presentation simpler, we elide the store σ from all definitions which do not explicitly use it.

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

Figure 3. Denotational semantics for SWL

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 [35]. The semantics of a procedure can then be defined as

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

where $\sigma_{x,l}$ denotes the state where identifiers $x_1, \dots, x_n$ and $l_1, \dots, l_m$ 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* [36],
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. In the case of constructs without while loops and procedures, this is given in fig. 4, where δ represents the overriding union operator over symbolic values.

The basic idea is very simple – so simple in fact that for most practitioners of Computer Algebra, the difference with denotational semantics is 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

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

Figure 4. Symbolic semantics for SWL

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 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 $\mathbb{R} \rightarrow \mathbb{R}$. 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 $f \in \mathbb{R} \rightarrow \mathbb{R}$ 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 $\llbracket\ \rrbracket$ notation for this semantics. It is important to note that the types involved are quite different: the symbolic semantics of an expression is always a *syntactic expression*. There is still a store involved, but now it is a function σ with symbolic values $\sigma(\mathsf{Var})$ such that

$$\sigma(\mathsf{Var}) = \begin{cases} v & (\mathsf{Var}, v) \in \sigma \\ v & \sigma = \delta(\sigma', \mathsf{Var} \leftarrow v) \\ \sigma'(\mathsf{Var}) & \sigma = \delta(\sigma', x \leftarrow v) \text{ and } x \neq \mathsf{Var} \\ \mathsf{Var} & \text{otherwise} \end{cases} \tag{2}$$

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 $\llbracket x+y \rrbracket\ \sigma_0 = x+y$, while in the store $\sigma_1 = \{(y,3)\}$, $\llbracket x+y \rrbracket\ \sigma_1 = x+3$. There is a simple correspondence between denotational and symbolic semantics — Theorem 5.1 in section 5 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*.

Definition 4.1. 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 [20, example p. 10]. To preserve the structure of the straight-line program, as is also done by [36], 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;
```

$$s_0 = \delta(\{\}, i \leftarrow \epsilon(\{\}, 3))$$
$$s_1 = \delta(s_0, r \leftarrow \epsilon(s_0, r) \cdot \epsilon(s_0, i))$$
$$s_2 = \delta(s_1, i \leftarrow \epsilon(s_1, i) + 1)$$

which is also one of the ideas in Maple's `LargeExpressions` package, which helps to produce dramatic improvements in certain large symbolic computations [41] (see also [20] for related work).

5. 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 reflect 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. *Hopefully the reader will be struck by the resemblance between this last equation and the* Tail Function *of Mazurkiewicz [26].*

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;
    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

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

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

translated to

$$\begin{pmatrix} i_{t+1} \\ r_{t+1} \\ q_{t+1} \end{pmatrix} = \begin{pmatrix} i_t + 2 \\ r_t \cdot (i_t + 1) \\ q_t + r_t \cdot \frac{i_t+1}{i_t} \end{pmatrix}. \tag{3}$$

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) [2] improves the situation somewhat.

Another question is, why is the resulting system non-linear if the original system is holonomic[7]? 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 resulting system is non-linear, but has a very natural hierarchical structure in which it every "higher" variable depends linearly on "lower" ones.

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, where the subscript w is meant to indicate that this is the *inner* semantics of a loop. In fig. 5, we use $\vec{s}$ to denote those state variables which are assigned to during the execution of the loop body. S'_t is the semantics of a sequence of assignments; but if we unravel what that means, it really encodes a set of $\mathsf{Var}(t + 1) \leftarrow [\![E]\!]\sigma$ which can readily be seen to be recurrence equations. μ is a symbolic constructor that we use to pack up the results. The first argument encodes the 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. Going back

[7]Holonomic systems are, by definition, linear

to ex. (3.1), we would get that the semantics of the while loop are

$$\begin{aligned}&\mu(\delta(\delta(\{\}, r(t+1) \leftarrow r(t) * i(t)), i(t+1) \leftarrow i(t)+1),\\ &\quad t = \min\{t \in \mathbb{N} \mid i(t) - 1 = n\},\\ &\quad \epsilon(\{r, i\}, \{r(0) = 1, i(0) = 1\}))\end{aligned}$$

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 $x_1, \ldots, x_n$ 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. Equation 2 can be completed as follows:

$$\sigma(\mathsf{Var}) = \begin{cases} v & (\mathsf{Var}, v) \ \in \sigma \\ v & \sigma = \delta(\sigma', \mathsf{Var} \leftarrow v) \\ \sigma'(\mathsf{Var}) & \sigma = \delta(\sigma', x \leftarrow v) \ \text{ and } x \neq \mathsf{Var} \\ ((S'_t)^{(e)}\sigma)(\mathsf{Var}) & \sigma = \mu(S'_t, t = e, \epsilon(\vec{s}, \sigma)) \\ \mathsf{Var} & \text{otherwise} \end{cases} \tag{4}$$

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 try 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 numerical programs, have a particular triangular structure visible in the recurrences. This structure is clearly visible in eq. 3. Given such a triangular structure, it is much easier to get a closed-form, as each recurrence can be solved independently in order of data flow dependence. We will comment further on this in Section 8.

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

Proof:
[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). $\square$

6. Termination

As mentioned 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 [35], or when using relational semantics [25], as the Kleene closure of the relational version of S'. The correspondence 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 6.1. 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. $\square$

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 [33]) 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.

7. 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. See [1] for a survey of some of these technologies.

As mentioned before, the basic algorithm involves sorting the recurrence equations in data dependence order , 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.

We do not always compute full closed-forms, as sometimes a sum or a product better expresses the intent of some code, while the occurrence of a strange special function coming from mathematical physics may confuse more than it enlightens.

8. Examples

While in section 3, 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 easily translatable into SWL. We have a prototype implementation of what we describe here for Maple (first described in [6]), as well as a Fortran 77 implementation (described in [13]). This latter implementation is why we have chosen an abstract intermediate language (SWL) instead of simply using a subset of Maple, even though our concrete examples in this paper use Maple.

Translating from Maple syntax or Fortran 77 syntax to SWL is a tedious engineering problem. The further translation to symbolic semantics is straightforward. Thankfully in Maple the *inert form* returned by `ToInert` makes the initial translation from Maple programs to SWL quite simple. This inert form [30] is a fully faithful representation of the abstract syntax tree of all Maple objects, including procedures, as a simple Maple data-structure. A lot more technology is required for Fortran 77, but again is straightforward via leveraging standard compiler tools (such as ANTLR [32]).

Example 8.1. (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 $n!$.

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:
```

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:
```

Example 8.2. (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 8.3. (Generalized Binomial)
Listing 5 shows a routine that purports to compute $\binom{u}{k}$. Our program says that this computes

$$\frac{(-1)^k\,\Gamma(-u+k)}{\Gamma(-u)\,\Gamma(k+1)}. \qquad (5)$$

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;
```

Lemma 8.1 shows that these denote the same function.

Lemma 8.1. Equation 5 $= \binom{u}{k}$ holds 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. □

Example 8.4. (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.

Example 8.5. (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*

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

(where both the Γ and the 2-argument incomplete Γ function appear). As expected, the error term $\Gamma\left(n+1,1\right)/\Gamma\left(n+1\right)$ converges to 1 very quickly with rising n. More intuitively, our program also reports that this computes

$$1+\sum_{t=1}^{n}\frac{1}{\Gamma(t+1)}$$

Example 8.6. (BesselJ)
Our system reports that this computes *exactly*

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

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

$$\sum_{t=1}^{m}\frac{\left(-1\right)^{t-1}\left(z^2\right)^{t-1}\Gamma(\nu+1)}{4^{t-1}\Gamma\left(t+\nu\right)\Gamma(t)}$$

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 [33] and of Kovács and Jebelean [23, 24]. Below, we give the examples from [23] and [33] which can be translated in to SWL.

Example 8.7. (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 8.8. (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 equivalent to the usual closed-form for the Fibonacci numbers (and derived automatically).

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;
```

Listing 11. Square Root

```
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;
```

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.

Example 8.9. (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.

9. Conclusions and Future Work

We have described a symbolic execution system that can be used to analyze properties of programs. It is especially well-suited to numerical programs which compute so-called *Special Functions*. The most important tool is the transformation of loops into explicit systems of recurrence equations over *time*.

For future work, we would like to loosen the restriction for `while` loop we have now. The most promising line of investigation is to see if we can include branches in loops, but where the branch condition depends on a *monotonic* function of time.

We are also experimenting with the "pure" Blikle and Mazurkiewicz approach [4, 5, 26] using a theorem prover Simplify [12] and the results look very promising.

References

[1] S. A. Abramov, J. J. Carette, K. O. Geddes, and H. Q. Le, Telescoping in the Context of Symbolic Summation in Maple, *Journal of Symbolic Computation* 38 (4), 2004, pp. 1303-1326.

[2] A. W. Appel, *Modern Compiler Implementation: In ML*, Cambridge University Press, New York, NY, USA, 1998.

[3] H. Bekić, Definable operations in general algebras and the theory of automata and flowcharts, Unpublished Manuscript, IBM Laboratory, Vienna 1969.

[4] A. Blikle, An analysis of programs by algebraic means, In A. Mazurkiewicz, Z Pawlak (eds), *Mathematical Foundation of Computer Science*, Banach Center Publications, Vol. 2, pp. 167–213, Polish Scientific Publishers, Warsaw 1977.

[5] A. Blikle, A. Mazurkiewicz, An algebraic approach to the theory of programs, algorithms, languages and recursiveness, *Proc. of 1st MFCS (Mathematical Foundations of Computer Science*), Jabłonna, Poland 1972.

[6] J. Carette, R. Janicki, Y. Zhai, Program Verification by Calculating Relations, *Proc. of 15th IASTED ASM'06 (Applied Simulation and Modeling)*, Rhodos, Greece 2006, pp.150-156, Acta Press.

[7] T. E. Cheatham, J. A. Townley, Symbolic Evaluation of Programs: A look at Loop Analysis, *Proc. of ACM Symposium on Symbolic and Algebraic Computation*, 1976, pp. 90-96.

[8] F. Chyzak, B .Salvy, Non-commutative Elimination in Ore Algebras Proves Multivariate Holonomic Identities, *Journal of Symbolic Computation* 26 (2), 1998, pp. 187-227.

[9] F. Chyzak, B .Salvy, Gröbner Bases, Symbolic Summation and Symbolic Integration, in B. Buchberger, F. Winkler (eds.), *Gröbner Bases and Applications*, London Mathematical Society Lecture Notes Series, Vol. 251, Cambridge University Press 1998, pp. 32-60.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Program. Lang. Syst.*, 13, 4 (1991), 451-490.

[11] J. W. DeBakker, D. Scott, A Theory of Programs, Unpublished Manuscript, IBM Laboratory, Vienna 1969.

[12] D. Detlefs, G. Nelson, J. B. Saxe, Simplify: A Theorem Prover for Program Checking, *Journal of ACM*, 53,3 (2005), 365-373.

[13] O. Dragon, Reverse Engineering of Scientific Computation FORTRAN Code, Master Thesis, Dept. of Computing and Software, McMaster University, Hamilton, Canada 2006.

[14] R. W. Floyd, Assigning Meaning to Programs, *Proc. of 19th Symposium on Applied Mathematics*, 1967, pp. 19-32.

[15] S. M. German and B. Wegbreit, A Synthesizer of Inductive Assertions, *IEEE Trans. Software Eng.*, vol. 1 (1), 1975, pp. 68-75.

[16] D. Goldberg, What every computer scientist should know about floating-point arithmetic, *ACM Comput. Surv.* 23 (1), 1991, pp. 5-48.

[17] C. A. R. Hoare, An Axiomatic Basis of Computer Programming, *Comm. of ACM* 12 (1969), 576-580.

[18] R. Janicki, Analysis of Coroutines by Means of Vectors of Coroutines, *Fundamenta Informaticae*, 2, 2 (1979), 289-316.

[19] W. Kahan, Pracniques: further remarks on reducing truncation errors, *Commun. of ACM*, 8 (1), 1965.

[20] E. Kaltofen, Greatest Common Divisors of Polynomials Given by Straight-Line Programs, *Journal of the ACM*, 35 (1), 1988, pp. 231-264.

[21] M. Karr, Affine Relationships Among Variables of a Program, *Acta Informatica*, 6 (1976), 133-151.

[22] S. Katz and Z. Manna, A Closer Look at Termination, *Acta Informatica*, 5 (1975), 333-352.

[23] L. I. Kovács, T. Jebelean, Automated Generation of Loop Invariants by Recurrence Solving in *Theorema*, *Proc. of SNASC'04* (Symbolic and Numeric Algorithms for Scientific Computing), 2004.

[24] L. I. Kovács, T. Jebelean, Finding Polynomial Invariants for Imperative Loops in the Theorema System, *Proc. of Verify'06 Workshop*, IJCAR'06, The 2006 Federated Logic Conference, pp. 52-67.

[25] D. Kozen, Kleene Algebras with tests, *Transactions on Programming Languages and Systems* 3, 19 (1997), 427-443.

[26] A. Mazurkiewicz, Proving Algorithms by Tail Function, *Information and Control*, 18 (1971) 793-798.

[27] A. Mazurkiewicz, Iteratively Computable Relations, *Bull. Acad. Polon. Sci., Ser. Sci. Math. Astronom. Phys.*, 20 (1972), 793-798.

[28] A. Mazurkiewicz, Recursive Algorithms and Formal Languages, *Bull. Acad. Polon. Sci., Ser. Sci. Math. Astronom. Phys.*, 20 (1972), 799-803.

[29] L. Meunier, B. Salvy, ESF: An Automatically Generated Encyclopedia of Special Functions, *Proc. of ISSAC'03*, Philadelphia 2003. ACM Press. pp. 199-205.

[30] M. B. Monagan and K. O. Geddes and K. M. Heal and G. Labahn and S. M. Vorkoetter, *Maple Programming Guide*, Springer Verlag, 1998.

[31] Reverse Engineering at McMaster, `http://www.cas.mcmaster.ca/~carette/ReverseEngineering`

[32] T. J. Parr, *ANTLR, ANother Tool for Language Recognition*, `http://www.antlr.org/`

[33] E. Rodríguez-Carbonell, D. Kapur, Program Verification Using Automatic Generation of Invariants, Proc. of ICTAC'04, *Lecture Notes in Computer Science* 3407, Springer 2005, pp. 325-340.

[34] B. Salvy, P. Zimmermann, Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable, *ACM Transactions on Mathematical Software*, 20 (2), 1994, pp. 164-177.

[35] D. Schmidt, *Denotational Semantics*, Allyn and Bacon, 1986.

[36] B. Scholz, T. Fahringer, *Advanced Symbolic Analysis for Compilers*. Springer-Berlin, 2003.

[37] J. V. Tucker, J. I. Zucker,Abstract versus concrete computation on metric partial algebras, *ACM Trans. Comput. Logic* 5 (4), 2004, 611-668.

[38] B. Wegbreit, The Synthesis of Loop Predicates, *Commun. of ACM* 17, 2 (1974), 102-112.

[39] S. Wolfram, *The Mathematica Book*, Cambridge University Press, 1999.

[40] Y. Zhai, An Analysis of Programs by Symbolic Computations, Master Thesis, Dept. of Computing and Software, McMaster University, Hamilton, Canada 2006.

[41] W. Zhou, J. Carette, D. J. Jeffrey and M. B. Monagan, Hierarchical representations with signatures for large expression managemen, Proc. of Artificial Intelligence and Symbolic Computation, *Lecture Notes in Computer Science* 4120, Springer 2006, pp. 254-268.