CS-Note* 44

Bill J. Ellis

June 13, 2003

Recurrence Relations and Invariant Discovery

Contents

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

1 Introduction

CS-note 34 introduced an approach for discovering invariants by returning to the program source. The approach operates by extracting patterns from the program source and then analysing these patterns to generate invariant properties.

The process of generating the patterns has been explored in CS-note 39 and refined in CS-note 43. Some relevant implementation details relating to the parsing of SPARK are given in CS-note 40. It became apparent that some of the ideas of CS-note 34 were not applicable. However, from these problems new strategies emerged, being described in CS-note 42 and in more detail in the revised version of CS-note 34, CS-note 41.

A key observation made in CS-note 41 was noting the similarity between the generalised description of patterns seen in CS-note 43 and recurrence relations[1]. This similarity can be exploited to aid in invariant discovery, especially for exception freedom proofs.

2 Recurrence relations

It is helpful to begin by summarising the different classifications of recurrence relations and their key features.

2.1 Discrete dynamical systems

Many processes are more appropriately modelled as a series of discrete changes rather than a continuous change. For example:

- **Compound interest -** Interest is added to an account in a lump sum on a particular date each year, not gradually over the whole year.
- **Program variables in loops -** Program variables change in a single step as a result of encountering an assignment statement.

These process systems are called discrete dynamical systems. They change in response to a discrete time event. Such systems can be described using recurrence relations.

2.2 Form of recurrence relations

A recurrence relation gives the terms of a sequence as a function of previous terms. To get actual values from the recurrence relation it is necessary to specify the initial values of terms. For example, the Fibonacci sequence is described by the recurrence equation (1):

$$a_{(n)} = a_{(n-1)} + a_{(n-2)} \tag{1}$$

Adding initial values (2) to the Fibonacci sequence of (1) describes the sequence (3):

$$\begin{aligned} a_{(0)} &= 0 \\ a_{(1)} &= 1 \end{aligned} \tag{2}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots \tag{3}$$

By setting different initial values different sequences can be generated. Thus while keeping the recursive part constant by tweaking only the initial values a range of sequences may be described. For example, it is noted in [2] that their are infinity many solutions to Fibonacci as each pair of initial values generates a unique sequence.

Note that recurrence relations are sometimes expressed with $a_{(n+1)}$ on the left hand, adjusting the right hand side accordingly. The slightly neater use of $a_{(n)}$ on the left hand side will be adopted here. The initial value of n is not important, so long as initial values of the recurrence relation are indexed appropriately. However, it is common to set n to 0 or 1. Although 1 seems more appropriate, 0 is more common and so will be adopted instead.

[1] It is worth noting that CS-notes are numbered as they are put on the project web page. Although they tend to observe a chronological order this is not guaranteed...

2.3 Solving recurrence relations

Recurrence relations are very expressive, allowing the straight forward presentation of complex sequences. In some cases the initial values of these recurrence relations will be known. However, it may be the case that the initial values are parametric and can take any value.

The problem with recurrence relations is they are not efficient or practical to work with. To find a value at n it is necessary to calculate previous values of n down to the initial values. Thus a common operation with recurrence relations is to find a closed formula, expressing the value of an element in the sequence at n strictly in terms of n. The solutions of recurrence relations are generally expressed independent to their initial values. Thus solving a single recurrence relation gives the solution to a potentiality infinite number of sequences. However, it is often possible to find simpler solutions for a particular sequence if initial values are assumed.

For our problem domain of invariant discovery the initial values may be known. In such cases the solved recurrence relations can be strengthened and simplified accordingly. However, their can exist cases where the initial values will only be known to be within some range. In these cases the general solution will be needed. For example, a closed form for tsfosb[2] (4) is (5):

$$a_{(n)} = a_{(n-1)} + 2 - 1 \tag{4}$$

$$a_{(n)} = a_{(0)} + n \tag{5}$$

Assuming initial values (6), a closed form for Fibonacci is (7):

$$\begin{aligned} a_{(1)} &= 0 \\ a_{(2)} &= 1 \end{aligned} \tag{6}$$

$$fib_{(n)} = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) \tag{7}$$

2.4 Classifying recurrence relations

It is helpful to classify different forms of recurrence relations.

2.4.1 Order of recurrence relations

The order of a recurrence relation is the number of initial variables required to generate concrete values. Thus a first order recurrence relation takes the form of (8), and a second order (9).

$$a_{(n)} = f(n, a_{(n-1)}) \tag{8}$$

$$a_{(n)} = f(n, a_{(n-1)}, a_{(n-2)}) \tag{9}$$

2.4.2 Linear recurrence relations with constant coefficients (LRRCs)

This is a special category of recurrence relation of the general form (10), where:

$$a_{(n)} = c_1 * a_{(n-1)} + c_2 * a_{(n-2)} + \ldots + c_k * a_{(n-k)} + f(n) \tag{10}$$

- $n \geq k$, otherwise some of the $a_{(n-k)}$ will refer to the initial values.
- the coefficients c_i are all constant.
- Only the first powers of previous terms are used, *i.e.* $(a_{(n-i)})^1$.
- $f(k) \ldots f(k+l)$ is some sequence.

Some examples of LRRCs:

- $a_{(n)} = a_{(n-1)}$
- $a_{(n)} = 2 * a_{(n-1)} + 3 * a_{(n-2)}$
- $a_{(n)} = a_{(n-1)} + n$

[2] Two Steps Forward, One Step Back.

2.4.3 Homogeneous and non-Homogeneous LRRCs

A LRRC is homogeneous if every value in the sequence generated by $f(n)$ is zero, and non-homogeneous otherwise. The homogeneous LRRC are of the form (11). This is also known as the LRRC canonical form.

$$a_{(n)} = c_1 * a_{(n-1)} + c_2 * a_{(n-2)} + \ldots + c_k * a_{(n-k)} \tag{11}$$

Note that if $f(n)$ is non zero but constant it is always possible to reduce a non-homogeneous LRRC into a homogeneous LRRC.

2.4.4 Everything else

Very little of a general nature is known about non-linear recurrence relations. The result is that the classification of the remaining recurrence relations are reasonably referred as everything not described above.

2.5 Also known as

- **Recurrence equations -** Recurrence equations appear to be a synonym for recurrence relations. They both describe the same constructs in the same manner.
- **Difference equations -** Difference equations are concerned with the relationship between different values in sequences. They are of the form (12):

$$a_{(n)} - a_{(n-1)} = f(n, f(n-1)) \tag{12}$$

  Difference equations are slightly rewritten recurrence relations, intended for analysing the differences between elements of a sequence rather than the elements of the sequence themselves.

3 Recurrence relations and program proof

Their is a strong relationship between recurrence relations and invariant discovery for program proof. Recognising this, it is possible to formulate invariant discovery problems as solving recurrence relations. This observation allows recurrence relation solvers to be exploited for program proof.

3.1 Relationship to induction

In [3] it is noted that recursion has a lot in common with induction. However their differences are highlighted as:

- **Recursion -** Is a method of construction. (It produces functions)
- **Induction -** Is a method of proof. (It verifies properties of functions)

This remains true for our intended usage. Recursion is captured in recurrence relations describing the behaviour of a concrete program loop. These are solved to gain properties that can be added as invariants. Then, through proving the resulting VCs, their is an implicit induction proof.

3.2 How did we get here?

Initially focus was placed on on proving partial correctness. A few examples were considered, one in detail in CS-note 10. However, it became increasingly apparent that partial correctness proofs were rare in industry. Further, partial correctness proofs can be seen as a step beyond exception freedom proofs. It is reasonable to target the more retractable problems first.

The nature of exception freedom in SPARK was considered in CS-note 12 and a few examples were explored in detail in CS-note 31, CS-note 32 and CS-note 35. These examples were considered in a bottom up style, gaining insights from the source code. In CS-note 34 an alternative mixed strategy was presented, gaining insights from both bottom up analysis, looking at the code, and top down analysis, looking at the VCs.

CS-note 34 noted that tracing the changes made to variables within a loop over some finite duration procured a repeated pattern. This pattern could then be generalised to gain an invariant. The patterns looked like that in Figure 1.

$$
\begin{array}{lcll}
i_0 & = & 0 & \wedge \\
r_0 & \leq & 0 & \wedge \\
\ldots & & \ldots & \\
i_1 & = & 1 & \wedge \\
r_1 & \leq & \underbrace{0+100}_{\alpha} & \wedge \\
\ldots & & \ldots & \\
i_2 & = & 2 & \wedge \\
r_2 & \leq & \underbrace{(\underbrace{0+100}_{\alpha}) + 100}_{2*\alpha} & \wedge \\
\ldots & & \ldots & \\
i_3 & = & 3 & \wedge \\
r_3 & \leq & \underbrace{(\underbrace{(\underbrace{0+100}_{\alpha}) + 100}_{2*\alpha}) + 100}_{3*\alpha} & \wedge \\
\ldots & & \ldots &
\end{array}
$$

Figure 1: Patterns generated by loop assignments

CS-note 39 considered the generation of these patterns up to a finite depth. The result was a symbolic trace similar in style to that above. However, it was observed that the finite depth being considered was unnecessary as the patterns were constant between iterations. The result is that describing the patterns before and after each iteration is sufficient to describe the pattern on every iteration. This refinement was described in CS-note 43, which adopted the form of patterns as shown in Figure 2.

$$
\begin{array}{l|l}
\textbf{Assign} & [] \rightarrow i_{(now,1)} = i_{(pred(now),top)} + 1 \\
\textbf{Assign} & [(1)] \rightarrow result_{(now,1)} = result_{(pred(now),top)} + (element(data\_set, i_{(pred(now),top)}))
\end{array}
$$

Figure 2: Patterns as symbolic before and after

CS-note 41 proposed a means for working with these generalised patterns to generate invariants. The main step underlying this approach was the rewrite rule (13).

$$X \underbrace{Y}_{N} \Rightarrow X * Y^N \tag{13}$$

Finally, as a late addition to CS-note 41, the similarity between these and recurrence relations was spotted. From this observation the solving techniques of CS-note 34 could be integrated within the algorithm of CS-note 43 by exploiting recurrence relation solvers, as considered in this note.

4 Solving recurrence relations

Many techniques exist for the solving of recurrence relations. These techniques generally focus entirely on LRRCs or very special cases of recurrence relations.

4.1 Our kind of problems

The recurrence relations generated from regular data structure manipulation in program code are reasonably likely to be LRRC, as programs often transform variables in a linear form. Even where the transformations do not produce LRRC recurrence relations they may still be solved by automatic tools. For example (14) will be produced from a subprogram performing `A:=A*A` within a loop. This is not LRRC but can be solved automatically.

$$a_{(n)} = a_{(n-1)} * a_{(n-1)} \tag{14}$$

Their is scope for trying to simplify hard recurrence relations by weakening their accuracy (Using the technique described in §4.2.2). This simplification may exploit:

- Implicit knowledge of mathematical functions.
- The explicit postconditions of SPARK subprograms.
- Program context information, such as `if` statements.
- The properties generated on a previous execution, especially where their are multiple loops.

In some cases, solving recurrence relations will reveal useful information that leads to good invariants being discovered for exception freedom proofs.

4.2 Only work with integers?

Recurrence relations describe discrete changes, indexed by the variable n. This variable n must be an integer. However, the recurrence relation itself may be any mathematical formula.

4.2.1 Integer

Integers are straight forward to process within recurrence relations.

4.2.2 Floating point

Any proofs in the presence of floating point numbers is difficult due to the inherent lack of precision in real numbers. However, this need not exclude proofs of exception freedom in this domain.

If real numbers are being used it is reasonable to assume that they are not being used as if as integers. Thus the presence of real numbers indicates that fractions or irrational numbers are expected. However, where proving floating point numbers within SPARK, the floating point arithmetic is prefect. All decimal numbers are converted into rational numbers. All operators that may lead to irrational numbers can not be reduced.

It may be possible to use simplifications to approximate functions that return irrational numbers. The rational numbers can be used directly. For example the code `R:=R+(1/2)*cos(X)` produces recurrence relation (15) which can be reduced to the two extreme recurrence relations (16) and (17) which can then be solved as regular recurrence relations.

$$R_{(n)} = R_{(n-1)} + (1/2) * cos(X_n) \tag{15}$$

$$R_{(n)} = R_{(n-1)} + (1/2) * -1 \tag{16}$$

$$R_{(n)} = R_{(n-1)} + (1/2) * 1 \tag{17}$$

4.2.3 Enumerated types

All enumerated types have a definite numeric value. SPARK forbids comparing enumerated types from different enumerated collections. However, if the code has successfully passed the EXAMINER then this has already been checked and the enumerated values can be safely processed using their numeric value. Once processing has completed it is necessary to convert back from the numeric values to the corresponding enumerated value.

4.2.4 Boolean

The Boolean type is essentially a special enumerated type, containing the values `True` and `False`. However, unlike other enumerated types, the Boolean types are not sorted. The EXAMINER will check to ensure that Booleans are only compared using equality, allowing Booleans to be safely processed with some suitable numeric values. Once processing has completed it is necessary to convert back from these numeric values to the corresponding Boolean value.

4.3 The Parma University's Recurrence Relation Solver (PURRS)

The Parma University's Recurrence Relation Solver (PURRS) [1] is a new tool[3] aimed specifically at automatically finding solutions to recurrence relations. PURRS only deals with integers. It claims to support:

- **Homogeneous and non-homogeneous** LRRCs
  These are normal LRRCs.

- **Homogeneous and non-homogeneous** LRRCs **of order 1 with a parametric coefficient**
  These are LRRCs of the form (18):

$$a_{(n)} = c_1 * a_{(n-1)} + f(n) \tag{18}$$

  Where $a_{(n-1)}$ can be parametric or constant. A parametric value is an unknown constant.

- **Linear Recurrences of Order 1 with Variable Coefficients**
  These are recurrences of the form (19):

$$a_{(n)} = b(n) * a_{(n-1)} + f(n) \tag{19}$$

  Where $b(n)$ is a function of n. These are considered more difficult problems and may not automate as favourably as LRRCs.

- **Generalised (Divide-Et-Impera) Recurrences**
  These are recurrences of the form (20):

$$a_{(n)} = b * a_{(n/c)} + f(n) \tag{20}$$

  Where $b \geq 0$, $c \geq 1$, $f(n)$ returns only positive integers and the value of n/c is taken as its integer part. These are considered very hard problems, with exact solutions not existing in many cases. PURRS may give up on an exact solution and instead return the upper and lower bounds of the solution.

4.3.1 Some examples

- **A counter**

```
Code: loop I:=I+1; end loop;
Gives recurrence relation: x(n)=x(n-1)+1

PURRS returns:
----------------------------------------
Exact solution for: x(n) = 1+x(-1+n)

x(n) = x(0)+n
----------------------------------------
```

- **Addition in loop**

```
Code: loop R:=R+100; end loop;
Gives recurrence relation: x(n)=x(n-1)+100

PURRS returns:
```

[3]The project went public in Jul 24, 2002.

```
----------------------------------------
Exact solution for x(n) = 100+x(-1+n)

x(n) = x(0)+100*n
----------------------------------------
```

- **Fibonacci**

```
Recurrence relation: x(n)=x(n-1)+x(n-2)

PURRS returns:
----------------------------------------
Exact solution for x(n) = x(-2+n)+x(-1+n)

x(n) = (-1/5*(1/2-1/2*sqrt(5))^(1+n)*sqrt(5)+1/5*(1/2+1/2*sqrt(5))^(1+n)*
       sqrt(5)+1/5*(1/2-1/2*sqrt(5))^n*sqrt(5)-1/5*(1/2+1/2*sqrt(5))^n*
       sqrt(5))*x(0)+(-1/5*(1/2-1/2*sqrt(5))^n*sqrt(5)+1/5*(1/2+1/2*
       sqrt(5))^n*sqrt(5))*x(1)
----------------------------------------
```

On first glance it looks as if PURRS has got tied up during simplification. However, this is the solution for all initial values. If starting from $x_{(0)} = 0$ and $x_{(1)} = 1$ then this trivially reduces to (21).

```
x(n) = (-1/5*(1/2-1/2*sqrt(5))^(1+n)*sqrt(5)+1/5*(1/2+1/2*sqrt(5))^(1+n)*
       sqrt(5)+1/5*(1/2-1/2*sqrt(5))^n*sqrt(5)-1/5*(1/2+1/2*sqrt(5))^n*
       sqrt(5))*0+(-1/5*(1/2-1/2*sqrt(5))^n*sqrt(5)+1/5*(1/2+1/2*
       sqrt(5))^n*sqrt(5))*1

x(n) = -1/5*(1/2-1/2*sqrt(5))^n*sqrt(5)+1/5*(1/2+1/2*sqrt(5))^n*sqrt(5)
```

$$a_{(n)} = -\frac{1}{5} * \left(\frac{1}{2} - \frac{1}{2 * \sqrt{5}}\right)^{n * \sqrt{5}} + \frac{1}{5} * \left(\frac{1}{2} + \frac{1}{2 * \sqrt{5}}\right)^{n * \sqrt{5}} \tag{21}$$

This is fairly close to the polished closed solution. It may even be the case that PURRS or another system could successfully simplify the specific case solution into the polished form shown in (7).

References

[1] Purrs: The parma university's recurrence relation solver. http://www.cs.unipr.it/purrs/.

[2] Zeph Grunschlag. Recurrence relations (lecture slides). http://www.cs.columbia.edu/ zeph/3203/l21/l21.ppt.

[3] Harold Simmons. Forms of recursion and induction (lecture notes). http://www.cs.man.ac.uk/ hsimmons/NOTES/FULL/FormsRec.ps.gz.