An Analysis of Program by Symbolic Computation

By
Yun Zhai, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M.Sc
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2006) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:
An Analysis of Program by Symbolic Computation

AUTHOR: Yun Zhai, B.Sc. (MCMASTER UNIVERSITY, CANADA)

SUPERVISORS: Dr. Jacques Carette, Dr. Ryszard Janicki

NUMBER OF PAGES: viii, 74

Abstract

We present a symbolic analysis of a class of while loop programs which can automatically derive a closed-form symbolic expression for the input-output relation embodied in that program.

We show that this is especially well-suited to analyzing programs from scientific computation, in particular programs which compute special functions (like Bessel functions) from its Taylor series expansion. Other than making heavy use of algebraic manipulations, as available in any computer algebra system, we also require the use of recurrence relations. It is from these recurrence relations that we derive most of our information.

It is important to note that we can often get interesting information about a program (like termination) without requiring closed-form solutions to the recurrences.

Acknowledgments

I would like to express my sincere appreciations to my supervisors, Dr. Jacques Carette and Dr. Ryszard Janicki, for their inspirations, invaluable and patient guidances, help, advice and comments through out the research of this thesis.

Also I want to thank all the other students I shared office with, for many fruitful discussion and for providing the environment that made this work not just possible but even enjoyable.

I am very grateful to Stephen Forrest, one of Dr. Carette's students, who helps me a lot on the code review and good performance on Maple. Thank you so much for your generous help.

I would also like to thank my parents who, for so many years, gave me the support needed to go all the way. Last not least my thanks go to my husband Lixin for his great support.

Some results of this thesis have been presented in [6]. Some parts of this thesis are the extension of appropriate parts of [6].

Contents

Abstract i

Acknowledgments ii

List of Figures vi

1 Introduction 1

1.1 Introduction 1

1.2 Applications for Symbolic Analysis 3

1.3 Outline 4

2 Intuition and Motivation 5

3 Loops and Recurrence Equations 10

3.1 Solving Recurrence Equations 10

3.2 Loop Termination 12

4 Theoretical Background 13

4.1 Denotational Semantics of Symbolic execution 14

4.1.1 Syntax of Input Domain 14

4.1.2 Grammar 15

4.1.3 Syntax of Output Domain 16
4.1.4 Semantic Functions 16
4.1.5 Semantics Equations 18
4.1.6 Code Examples 26

5 System Analysis **30**
5.1 Input Language 30
5.2 Relation Generator 31
5.3 Solving Relations: Overview 34
5.4 From Code to Recurrences: while 38
5.4.1 Generating Recurrence Relations 38
5.4.2 Generating Initial Conditions 39
5.4.3 Stopping Conditions 39
5.5 Solving with for Loops 39
5.5.1 Number of Loop Iterations 40
5.5.2 Solving Relations Involving Recursion Call 41
5.5.3 The Case of Branches in Loops 42

6 Examples of Using Symbolic Execution Tool **45**
6.1 Examples of Generating Explicit Output 45
6.1.1 Example 1: $\sum_{i=0}^{n} \frac{1}{i!}$ 45
6.1.2 Example 2: $\sum_{i=0}^{n} i!$ 47
6.1.3 Example 3: Chebyshev Polynomials 49
6.1.4 Example 3: Binomial Coefficients 50
6.1.5 Example 4: Bessel 52
6.1.6 Example 5: A Recursive Call Function 54

6.2 Examples of Generating Implicit Output 56
6.2.1 Example 1: GCD 56
6.2.2 Example 2: LCM 58

7 Related Work **60**

8 Contributions and Future Work **62**

A Specification by Maude **64**

List of Figures

5.1 Symbolic execution system 32
5.2 Symbolic execution system in detail 37

List of Tables

2.1 Relations between statements and modelled relations 7

3.1 Input program corresponding to recurrence equations 11

4.1 Boolean symbols and their opposite values 21

5.1 Rules for program transformation 31

5.2 Translation of `factorial` into the set of appropriate relations 33

5.3 Mapping relations between the generated relations and the semantic functions 33

5.4 Inert functions and their meanings 35

5.5 Transformation and rules 35

5.6 Recurrence equations and initial conditions for `factorial`. 38

6.1 Recursive and initial functions for computing $\sum_{i=0}^{n} \frac{1}{i!}$ 47

6.2 Recursive and initial functions for computing $\sum_{i=0}^{n} i!$ 48

6.3 Another example to compute $\sum_{i=0}^{n} i!$ 49

6.4 Recursive and initial functions for `chebyshev` 50

6.5 Recursive and initial functions for `Binomial Coefficients` 51

6.6 Recursive and initial functions for `Bessel` 53

6.7 Recursive and initial functions for `chebyshev1` 55

6.8 Translation of program into the input of generating invariants [29] for GCD 57

6.9 Translation of program into the input of generating invariants [29] for LCM 59

Chapter 1

Introduction

This chapter provides a brief introduction to the background, purpose and outline of this thesis.

1.1 Introduction

In late sixties and early seventies, a technique for verifications and analysis of computer programs based on a calculus of relations was proposed ([2, 4, 26, 16] and others). Despite many theoretical and methodological advantages (it rather emphasises *calculation* instead of *proving*), the technique has never become widely accepted because of the huge amount of symbolic computations that need to be performed for even relatively simple cases.

The situation has dramatically changed today, as we have very powerful tools supporting symbolic computation such as *Maple* [27] and *Mathematica* [34]. The problem is still non-trivial, as the most general cases can be proved undecidable, but for many less general cases an efficient solution seems to be feasible.

In this thesis we show how to build a Maple [27] based tool [28] that can either automatically compute a closed form for simple programs with loops, or considerably simplify that task by computing *polynomial* invariants of such programs. Simple cases of recursion can also be treated. For straight-line programs, this reduces to 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 inputs. The formal specification of our system was done using *Maude* [25].

Symbolic execution has wide range of potential applications, however, it has been rarely used for proving properties of programs ([29] is one of few exceptions). This is because, in general, naive symbolic execution can lead to exponential blow-ups.

Our symbolic analysis can be seen as a kind of compiler which can translate the input programs into a symbolic expression, and then can 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 [17]. The basic technique used in such cases is to find "loop invariants" proposed by C. A. R. Hoare in 1969 [14]. Unfortunately finding them is often problematic and research on how to find them in some automatic manner has only just begun [29].

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.

1.2 Applications for Symbolic Analysis

Symbolic execution was first introduced by J. King in [19]. It is a technique that executing a program supplying symbolic input value and returning a symbolic output value. Since then, a number of researchers have developed it further for several different purposes.

Symbolic analysis can be applied to a variety of problems. One of operational area of symbolic analysis is centered around compilers. Classical techniques usually fail to provide accurate data dependence information to support parallelization techniques such as array privatization [33] and communication optimization [12]. Therefore, sophisticated symbolic analysis that can cope with program unknown is needed to overcome these complier deficiencies.

Another application for symbolic analysis are program development tools which assist the programmer and system designer. Instead of testing and debugging, symbolic analysis techniques provide analysis information for statically detecting program anomalies.

Other applications can be found in safety-critical real-time computer systems which can be characterized by the presence of timing constraints. For example, symbolic analysis has been employed to deduce time functions [3] for real-time programs.

Symbolic analysis techniques can be equally important for program testing [19], program verification [7], Software specialization [10], software reuse, pattern matching and concept comprehension [24], and other areas where complex programs with unknowns must be examined and accurate program analysis is required [13].

1.3 Outline

Chapter 2 introduces the intuition and motivation of our thesis.

Chapter 3 discusses the solving recurrences function and loop termination.

Chapter 4 introduces theoretical background and discuss denotational semantics.

Chapter 5 analyzes our symbolic system in detail.

Chapter 6 gives some examples of using our symbolic computation system.

Chapter 7 summarizes the contribution and future work of this thesis.

Chapter 2

Intuition and Motivation

The example below (from [4]) provides a 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. The full theory of those expressions can be found in [21].

Consider the well-known procedure factorial, written in a small subset of Maple [27]:

```
procedure factorial(n)
   i:=1;
   fac:=1;
   while i < n do
   begin
      i:=i+1;
      fac:=fac*i;
   end;
   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 $D = \mathbb{Z} \times \mathbb{Z}$ and denote the elements of D as (i, fac). Each instruction can be modeled by a function $F_i : D \to D$, $i = 1, 2, 4, 5$, in the following manner:

`"i:=1"` corresponds to $F_1(i, fac) = (1, fac)$,

`"fac:=1"` corresponds to $F_2(i, fac) = (i, 1)$,

`"i:=i+1"` corresponds to $F_4(i, fac) = (i + 1, fac)$,

and `"fac:=fac*i"` to $F_5(i, fac) = (i, fac \cdot i)$.

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<1"` corresponds to $I_3(i, fac)$, and

`"i≥1"` corresponds to $\bar{I}_3(i, fac)$, where

$$I_3(i, fac) = \begin{cases} (\text{i}, fac) & \text{if } i < n \\ \perp & \text{if } i \geq n \end{cases}$$

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

It is a well known fact that non-recursive programs can be modeled adequately with Kleene Algebras of Relations with Tests (see [21]) by using 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 modelled by 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 table 2.1 shows the relations between statements and modelled relations.

Statement	Modeled By
`S1;S2`	$R_1 \circ R_2$
`if T then S1 else S2`	$(I_T \circ R_1) \cup (\bar{I}_T \circ R_2)$
`while T do S`	$(I_T \circ R)^* \bar{I}_T$

Table 2.1: Relations between statements and modelled relations

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

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

Calculating "$\circ$" is easy in this case, but calculating "*" is not. Let $G = I_3 \circ F_4 \circ F_5$. Then we have:

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

Similarly :

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

Hence :

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

Since G^* is *not* a function, we need to express G^k in terms of relation calculus:

$(i, fac)G^k(i', fac') \iff i' = i + k \wedge fac' = fac * (i+1) * \ldots * (i+k) \wedge i + k - 1 < n.$

We have $G^* = \bigcup_{i=0}^{\infty} G^i$, hence:

$(i, fac)G^*(i', fac')$

$\iff \exists k \geq 0, (i, fac)G^k(i', fac')$

$\iff \exists k, 0 \leq k < n - i + 1 \ \wedge \ i' = i + k \ \wedge fac' = fac * (i+1) * \ldots * (i+k).$

We may now make some simplification:

$(F_1 \circ F_2)(i, fac) = F_2(F_1(i, fac)) = (1, 1).$

This means:

$(i, fac)F_1 \circ F_2 \circ G^*(i', fac')$

$\iff (1, 1)G^*(i', fac')$

$\iff \exists k, 0 \leq k < n \wedge i' = k + 1 \wedge fac' = (k+1)!$

Let us calculate : $(1, 1)G^* \circ \bar{I}_3(i', fac')$.

The partial function $\bar{I}_3$ in the relational representation looks as follows:

$(i, fac)\bar{I}_3(i', fac') \iff i \geq n \wedge i = i' \wedge fac = fac'.$

From the definition of " $\circ$ " we have :

$(1, 1)G^* \circ \bar{I}_3(i', fac') \iff \exists i, fac, (1, 1)G^*(i, fac)$ and

$(i, fac)\bar{I}_3(i', fac')$

$\iff (\exists k, 0 \leq k < n \wedge i = k + 1 \wedge fac = (k+1)!) \wedge (i \geq n \wedge i = i' \wedge fac = fac').$

Note that: $i = k + 1 \wedge i \geq n \Rightarrow k + 1 \geq n \iff k \geq n - 1$, and

$0 \leq k < n \wedge k \geq n - 1 \Rightarrow k = n - 1 \iff n = k + 1.$

So now, we do not have a general $\exists k$, but a very specific k=n-1, which means $G^* \circ \bar{I}_3$ is a function again, and the statement:

$$(\exists k, 0 \leq k < n \wedge i = k + 1 \wedge fac = (k+1)!) \wedge (i \geq n \wedge i = i' \wedge fac = fac')$$

is reduced to:

$$i' = k + 1 = n - 1 + 1 = n \wedge fac' = n!$$

In this way we have proved that $F(i, fac) = (n, n!)$, so the program is correct. To make this technique feasible for bigger, more realistic programs, we need a tool that would be able to do all those symbolic calculations. Our tool [28] will take the text of the program `factorial` as an input and will return the text "$n!$" as an output. In the following chapters we will show how it can be done with some help from *Maple* [27]. Our system [28] can also deal with some kind of limited recursion as well.

Chapter 3

Loops and Recurrence Equations

A formula that expresses the meaning of a loop can be explicitly derived (in some cases) by solving appropriate recurrence equations.

3.1 Solving Recurrence Equations

Consider our program `factorial` presented in chapter 2. Every time when the loop is executed, the value of i is incremented by one and the value of *fac* is incremented i times. We may express this change in a form of recurrences. For this example, the recurrence relation is following $i(t+1) = i(t) + 1, fac(t+1) = fac(t) \cdot i(t+1)$, where $i(t+1)$ and $fac(t+1)$, for $t \geq 0$ are the values of variable i and *fac* at the end of iteration $t+1$. In this sense, t represents *time*, which is the important new concept in this representation. Because *time* is explicitly used in the recurrence, this allows many techniques from symbolic computation to apply. Before entering the loop, the value of i is 1 and the value of *fac* is 1, so we initialize the recurrence by $i(0) = 1$ and $fac(0) = 1$. Table 3.1 describes the meaning of our loop.

Input program	Generated into Recurrence Equations
`factorial:=proc(n)`	
`local i, fac;`	
`i:=1;`	*i(0):=1*
`fac:=1;`	*fac(0):=1*
`while` $i < n$ `do`	
`i:=i+1;`	*i(t+1):= i(t) + 1*
`fac:=fac * i;`	*fac(t+1):= fac(t)*i(t+1)*
`end do;`	
`end proc:`	

Table 3.1: Input program corresponding to recurrence equations

These recursive functions can not be solved by Maple [27] directly since they are *non-linear* recurrences. However, we can clearly see that $i(t+1)$ is independent of $fac(t)$, but $fac(t+1)$ is not independent of $i(t+1)$, i.e. we may not be able to solve the system directly, but we might be able to solve it incrementally. From a simple data-flow analysis, this order can be determined. If we solve for $i(t)$ first (including initial conditions), we get $i(t) = t + 1$. Replacing $i(t)$ by $t + 1$, in the equation for fac, we get $fac(t+1) = fac(t) \cdot (t+2)$, which is also solvable. Putting the solution together, we get

$$i(t) = t + 1 \tag{3.1}$$

$$fac(t) = (t+1)! \tag{3.2}$$

Note that the above solution is still in terms of *time*; however, regardless of whether the loop terminates or not, we have found the *core meaning* of the loop!

This technique can be applied to any loop if the set of appropriate recurrence equations that the loop defines is essentially triangular, with polynomial solutions for the non-linearities (or it can be transformed into such a system).

3.2 Loop Termination

To determine the value of recurrence variables after the loop, we need the recurrence condition which *symbolically* determines the number of iterations for the recurrence. In our example, the recurrence variables are i and *fac*, and the recurrence condition is given by $i(t) < n$, which, together with equation (3.1) gives us the following formula for the loop termination (see [30]):

$$\min_{t\geq 0}\{t \mid i(t) \geq n\} = \min_{t\geq 0}\{t \mid t+1 \geq n\} = \begin{cases} n-1 & \text{if } n \geq 1 \\ 0 & \text{if } n \leq 0 \end{cases}$$

In order to compute the value of *fac* at the loop exit, we have to substitute $n-1$ for t in (3.2). So, we get "$n!$".

Note that it is only necessary to obtain a closed-form solution [23] to the recurrences involving for those variables which actually *occur* in the stopping condition to determine loop termination. This can frequently be much simpler, as is the case for all *for* loops!

Chapter 4

Theoretical Background

There are two main aspects of computer language, its syntax and its semantics. The syntax is concerned with the grammatical structure of the program while the semantics gives the meaning of grammatically correct programs. There are a number of different ways of describing the semantics of a programming or specification language, the most common ones are followings:

`Operational Semantics`: The meaning of a construct of the language is specified by the computation it induces when it is executed on a machine. In particular, it is of interest how the effect of a computation is produced.

`Denotational Semantics`: The meaning of a construct of the language is described by translating it into a different structure and modeling the effect of statements of the language there [31].

`Axiomatic Semantics` : The meaning of a language is described by axioms that can be used to prove theorems about specification terms in the language [31].

In our system, we are mainly concerned about the denotational semantics. Denotational semantics is a methodology to define the precise meaning of a computer

language. In denotational semantics, a computer language is given by a valuation function that map programs into mathematical objects considered as their denotation, i.e. meaning. Thus the valuation function of a computer program reveals the meaning of computer programs while neglecting its syntactic structure.

4.1 Denotational Semantics of Symbolic execution

4.1.1 Syntax of Input Domain

We illustrate the syntax of Input domain in this section. `Type` denotes data type that we use in the language. `ArithmeticExpr` denotes the abstract Arithmetic expression that we use in the language. `int` and `constant` are `Int` type and `symbol` is `Symbol` type. `var` is a `Variable`. `ArithmeticOp` denotes the Arithmetic operation that we use in the language. `var(ArithmeticExpr)` denotes recursive variable in the loop. We describe them by the followings:

```
Type           : Int | Bool | Symbol
Variable       : Identifier : Type
ArithmeticOp   : + | - | * | /
ArithmeticExpr : int | symbol | constant | var(ArithmeticExpr)
                 | ArithmeticExpr ArithmeticOp ArithmeticExpr
```

`BooleanOp`, `RelationOp` and `UnaryOp` denote the boolean operation, relational boolean operation and unary boolean operation that we use in the language, respectively. `Expr` denotes the boolean expression or arithmetic expression that we use in the language. We describe them by the followings:

```
BooleanOp   : && | ||
RelationOp1 : < | <= | > | >=
RelationOp2 : == | ! =
RelationOp  : RelationOp1 | RelationOp2
UnaryOp     : ¬
BooleanExpr : Bool
            | BooleanExpr BooleanOp BooleanExpr
            | BooleanExpr RelationOp2 BooleanExpr
            | ArithmeticExpr RelationOp ArithmeticExpr
            | UnaryOp BoolExpr
Expr        : BooleanExpr | ArithmeticExpr
```

`BinaryOp` denotes the binary operation that we use in the language. `Op` denotes the operation including binary operation and unary operation that we use in the language. We describe them by the followings:

```
BinaryOp : BooleanOp | RelationOp | ArithmeticOp
Op       : BinaryOp | UnaryOp
```

4.1.2 Grammar

We will illustrate the abstract interpretation using the language described by the following simple grammar:

```
Statement : var := expr
          | if booleanExpr then Statement end if
          | if booleanExpr then Statement else Statement end if
          | while booleanExpr do Statement end do
          | for var from int to int by int do Statement end do
          | Statement ; Statement
```

In the language, we have `assignment` statement, `if condition` statement, `while loop` and `for loop` statement, and statement sequences. var, expr and booleanExpr are the type of `Expr`, `Expr` and `BooleanExpr` individually.

4.1.3 Syntax of Output Domain

We define the syntax of the output domain in the followings:

```
OutputExpr : Expr | δ(StateRep, (Identifier, Expr))
           | γ(BooleanExpr, StateRep, StateRep)
           | μ(BooleanExpr, {Identifier}, StateRep)
           | η([Variable, Int, Int, Int], {Identifier}, StateRep)
StateRep   : δ | γ | μ | η | ⊥
```

Where `OutputExpr` represents the domain of Output Expression. State is a map from an identifier to an output expression. `StateRep` is the representation of the state, i.e. for a StateRep, one can frequently reconstruct a state.

4.1.4 Semantic Functions

The detailed semantics requires that we show how the expressions and statement are modelled in terms of those basic semantic domains. We need to define the following

functions:

δ : StateRep $\rightarrow$ (Identifier, Expr) $\rightarrow$ StateRep

γ : BooleanExpr $\rightarrow$ StateRep $\rightarrow$ StateRep $\rightarrow$ StateRep

μ : BooleanExpr $\rightarrow$ {Identifier} $\rightarrow$ StateRep $\rightarrow$ StateRep

η : [Variable, Int, Int, Int] $\rightarrow$ {Identifier} $\rightarrow$ StateRep $\rightarrow$ StateRep

$[\![\ ]\!]$: Statement $\rightarrow$ (StateRep $\rightarrow$ StateRep)

$[\![\![\ ]\!]\!]$: Statement $\rightarrow$ (StateRep $\rightarrow$ StateRep)

evalExpr : Expr $\rightarrow$ StateRep $\rightarrow$ OutputExpr

getRecVar : Statement $\rightarrow$ {Identifier}

Reduce : StateRep $\rightarrow$ OutputExpr

The constructor δ, γ, μ and η describe the updating StateRep to a new StateRep. The function $[\![\ ]\!]$ and $[\![\![\ ]\!]\!]$ describe the mapping from one StateRep to a new **StateRep** under the state commands. The function evalExpr evaluates an expression with the StateRep to the outputExpr. Function Reduce can evaluate δ, γ, μ and η when possible.

Let id and expr be the type of Identifier and Expression individually, then (id, expr) can be denoted by id=expr in δ.

4.1.5 Semantics Equations

(AS) $[\![\texttt{var:=expr}]\!] = \lambda\ \texttt{s} \in \texttt{StateRep}\ .\ \delta(\texttt{s}, \texttt{var=evalExpr(expr, s)})$

(SQ) $[\![\texttt{stmt}_1; \texttt{stmt}_2]\!] = \lambda\ \texttt{s} \in \texttt{StateRep}.\ [\![\texttt{stmt}_2]\!]([\![\texttt{stmt}_1]\!](\texttt{s}))$

(I1) $[\![\texttt{if cond then stmt end if}]\!] = \lambda\ \texttt{s} \in \texttt{StateRep}.\ \gamma(\texttt{cond}, [\![\texttt{stmt}]\!](\texttt{s}), \texttt{s})$

(I2) $[\![\texttt{If cond then stmt}_1 \texttt{ else stmt}_2 \texttt{ end if}]\!] =$
$\lambda\ \texttt{s} \in \texttt{StateRep}.\ \gamma\ (\texttt{cond}, [\![\texttt{stmt}_1]\!](\texttt{s}), [\![\texttt{stmt}_2]\!](\texttt{s}))$

(WH) $[\![\texttt{while cond do stmt end do}]\!] =$
$\lambda\ \texttt{s} \in \texttt{StateRep}.\ \mu(\texttt{cond, getRecVar(stmt)}, [\![\texttt{stmt}]\!](\texttt{s}))$

(WA) $[\![\texttt{var:=expr}]\!](\texttt{s}) = \delta(\texttt{s, var(t+1)=evalRecExpr(expr, s)})$

(WI1) $[\![\texttt{if cond then stmt end if}]\!](\texttt{s}) = \gamma(\texttt{cond}, [\![\texttt{stmt}]\!](\texttt{s}), \texttt{s})$

(WI2) $[\![\texttt{if cond then stmt}_1 \texttt{ else stmt}_2 \texttt{ end if}]\!](\texttt{s}) =$
$\gamma(\texttt{cond}, [\![\texttt{stmt}_1]\!](\texttt{s}), [\![\texttt{stmt}_2]\!](\texttt{s}))$

(WS) $[\![\texttt{stmt}_1 ; \texttt{stmt}_2]\!](\texttt{s}) = [\![\texttt{stmt}_2]\!]([\![\texttt{stmt}_1]\!](\texttt{s}))$

(FR) $[\![\texttt{for i from s to e by step do stmt end do}]\!] =$
$\lambda\ \texttt{s} \in \texttt{StateRep}.\ \eta([\texttt{i, s, e, step}], \texttt{getRecVar(stmt)}, [\![\texttt{stmt}]\!](\texttt{s}))$

Above lists the symbolic evaluation rules of simple assignment(AS), statement sequences(SQ). Rule (I1) is used in order to evaluate if-statements without else-branches. Similarly, rule (I2) is applied to if-statements with else-branches. Rule (WH) is used to evaluate a while loop statement. Rule (WA), (WI1), (WI2) and (WS) are applied for the inside statements of the while loop, in order to generate recurrence equations. Rule (FR) is used to evaluate a for loop.

Assignment Statements

For all statements of the program, our symbolic analysis deduces `StateRep` that describes the variable values that the program point is reached.

StateRep is a representation of a state described by δ,γ, μ, η and $\perp$.

For evaluating an expression, we define function `evalExpr` which symbolically evaluates the value of expression expr for a specific program `StateRep` as follows:

$$[\![\ ]\!] : \texttt{Statement} \rightarrow (\texttt{StateRep} \rightarrow \texttt{StateRep})$$

(AS) $[\![\texttt{var:=expr}]\!] = \lambda \texttt{s} \in \texttt{StateRep}.\delta(\texttt{s},\texttt{var=evalExpr(expr, s)})$

(SQ) $[\![\texttt{stmt}_1; \texttt{stmt}_2]\!] = \lambda \texttt{s} \in \texttt{StateRep}.[\![\texttt{stmt}_2]\!]([\![\texttt{stmt}_1]\!](s))$

$$\texttt{evalExpr} : \texttt{Expr} \rightarrow \texttt{StateRep} \rightarrow \texttt{OutputExpr}$$

(E1) $\texttt{evalExpr(constant, s)} = \texttt{constant}$

(E2) $\texttt{evalExpr(var, s)} =$

$$
\begin{cases}
\texttt{var}, & \texttt{if s=}\perp \\
\texttt{expr}, & \texttt{if s=}\delta\texttt{(s', var=expr)} \\
\texttt{evalExpr(var,s')}, & \texttt{if s=}\delta\texttt{(s',var'=expr')} \bigwedge \texttt{var'}\neq \texttt{var} \\
\texttt{evalExpr(var, s}_1\texttt{)} & \texttt{if s=}\gamma\texttt{(cond,s}_1\texttt{,s}_2\texttt{)} \bigwedge \texttt{cond} \\
\texttt{evalExpr(var, s}_2\texttt{)} & \texttt{if s=}\gamma\texttt{(cond,s}_1\texttt{,s}_2\texttt{)} \bigwedge \neg\texttt{(cond)} \\
\texttt{var} & \texttt{if (s=}\mu\texttt{(\_,getRecVar(stmt),\_)} \\
 & \bigvee \texttt{s=}\eta\texttt{(\_,getRecVar(stmt),\_))} \\
 & \bigwedge \texttt{var} \notin \texttt{getRecVar(stmt)} \\
\texttt{var(t+1)} & \texttt{if (s=}\mu\texttt{(\_,getRecVar(stmt),\_)} \\
 & \bigvee \texttt{s=}\eta\texttt{(\_,getRecVar(stmt),\_))} \\
 & \bigwedge \texttt{var} \in \texttt{getRecVar(stmt)} \\
 & \texttt{where t is under the loop stopping condition,} \\
 & \texttt{the loop iteration times.}
\end{cases}
$$

(E3) $\texttt{evalExpr(expr}_1 \texttt{ op expr}_2\texttt{, s)=evalExpr(expr}_1\texttt{,s) op evalExpr(expr}_2\texttt{,s)}$

Where $\texttt{expr}_1, \texttt{expr}_2 \in$ Expr and stmt $\in$ Statement

The above lists the symbolic evaluation rules of simple assignment (AS), statement sequence(SQ), and the evaluation function evalExpr expressed as denotational semantic rules.

The rule (AS) is applied for an assignment var:=expr under the stateRep. It is a function that maps the assignment with an input StateRep to a new StateRep. The new StateRep uses δ constructor to keep track of the change of the variable var.

Rules $(E_1) - (E_3)$ describe the symbolic evaluation of expression. Rule (E_1) de-

R	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$
$\neg R$	$\geq$	$>$	$\leq$	$<$	$\neq$	$=$

Table 4.1: Boolean symbols and their opposite values

scribes the constant evaluation. Rule (E_2) is used to extract the symbolic value of a variable from state s and rule (E_3) is used for translating an operation to its symbolic domain. In general, function `evalExpr` transforms expressions of the input program to symbolic expressions based on a given StateRep. For statement sequences, according to Rule(SQ) we symbolically analyze the first statement with the given StateRep, then the resulting StateRep is taken as the StateRep for the remaining statements.

Conditional Statements

If a conditional statement is encountered, usually the conditional expression can not be statically determined whether for all input data sets either the true or the false branch has to be followed. So the symbolic analysis has to consider both branches. We use constructor γ to evaluate the conditional statements. Constructor γ is defined as:

$$\texttt{reverse} : \texttt{BooleanExpr} \rightarrow \texttt{BooleanExpr}$$

$$\gamma \quad : \texttt{BooleanExpr} \rightarrow \texttt{StateRep} \rightarrow \texttt{StateRep} \rightarrow \texttt{StateRep}$$

$$\texttt{Reduce}(\ \gamma(\texttt{cond}, [\![\texttt{stmt}_1]\!](\texttt{s}), [\![\texttt{stmt}_2]\!](\texttt{s}))) =$$

$$\begin{cases} [\![\texttt{stmt}_1]\!](\texttt{s}) & \texttt{if evalExpr(cond,s)} \\ [\![\texttt{stmt}_2]\!](\texttt{s}) & \texttt{if evalExpr(reverse(cond),s)} \end{cases}$$

$$\texttt{reverse}(\texttt{expr}_1\ \texttt{op}\ \texttt{expr}_2) = \texttt{expr}_1\ \neg(\texttt{op})\ \texttt{expr}_2$$

$$\texttt{where expr}_1, \texttt{expr}_2 \in \texttt{Expr}, \texttt{op} \in \texttt{Op}$$

The symbolic semantics rules for symbolical evaluation if-statements are defined as:

(I1) $[\![\texttt{if cond then stmt end if}]\!] = \lambda\ \texttt{s} \in \texttt{StateRep}.\ \gamma(\texttt{cond}, [\![\texttt{stmt}]\!](\texttt{s}), \texttt{s})$

(I2) $[\![\texttt{if cond then stmt}_1 \texttt{ else stmt}_2 \texttt{ end if}]\!] =$

$\lambda\ \texttt{s} \in \texttt{StateRep}.\ \gamma(\texttt{cond}, [\![\texttt{stmt}_1]\!](\texttt{s}), [\![\texttt{stmt}_2]\!](\texttt{s}))$

Rule (I1) is used to evaluate if-statements without else branches. In fact, the constructor γ is equivalent to the "piecewise"function in Maple. Before evaluating the branches, the condition cond of the if-statement is evaluated. if the condition cond is true, then goes to evaluate statements stmt under StateRep s, otherwise the state keeps unchanged. Similarly Rule (I2) is applied to if-statements with else branches. After evaluating the branches, if the condition cond is true, then evaluate statements $\texttt{stmt}_1$ under StateRep s, otherwise evaluate statements $\texttt{stmt}_2$ under StateRep s.

While Loop Statements

Assume the following pattern for the while loop: `while cond do stmt`.

We use constructor μ to describe the while loop statements. The constructor μ is similar to the fixed point function, which is defined as:

$$[\![\ ]\!] : \texttt{Statement} \rightarrow (\texttt{StateRep} \rightarrow \texttt{StateRep})$$

$$[\![\![\ ]\!]\!] : \texttt{Statement} \rightarrow (\texttt{StateRep} \rightarrow \texttt{StateRep})$$

$$\mu : \texttt{BooleanExpr} \rightarrow \{\texttt{Identifier}\} \rightarrow \texttt{StateRep} \rightarrow \texttt{StateRep}$$

The denotational semantics of while loop is given in the followings:

(WH) $[\![\texttt{while cond do stmt end do}]\!] =$

$\lambda \texttt{s} \in \texttt{StateRep}.\mu(\texttt{cond}, \texttt{getRecVar(stmt)}, [\![\![\texttt{stmt}]\!]\!](\texttt{s}))$

(WA) $[\![\texttt{var:=expr}]\!](\texttt{s}) = \delta(\texttt{s},\texttt{var(t+1)=evalRecExpr(expr,s)})$

(WI1) $[\![\texttt{if cond then stmt end if}]\!](\texttt{s}) = \gamma(\texttt{cond}, [\![\texttt{stmt}]\!](\texttt{s}), \texttt{s})$

(WI2) $[\![\texttt{if cond then stmt}_1 \texttt{ else stmt}_2 \texttt{ end if}]\!](\texttt{s}) =$

$\gamma(\texttt{cond}, [\![\texttt{stmt}_1]\!](\texttt{s}), [\![\texttt{stmt}_2]\!](\texttt{s}))$

(WS) $[\![\texttt{stmt}_1; \texttt{stmt}_2]\!](\texttt{s}) = [\![\texttt{stmt}_2]\!]([\![\texttt{stmt}_1]\!](\texttt{s}))$

$\texttt{getRecVar} : \texttt{Statement} \rightarrow \{\texttt{Identifier}\}$

(G1) $\texttt{getRecVar(var:=expr)} = \{\texttt{var}\}$

(GI1) $\texttt{getRecVar(if cond then stmt end if)} = \texttt{getRecVar(stmt)}$

(GI2) $\texttt{getRecVar(If cond then stmt}_1 \texttt{ else stmt}_2 \texttt{ end if)} =$

$\texttt{getRecVar(stmt}_1\texttt{)} \cup \texttt{getRecVar(stmt}_2\texttt{)}$

(GW) $\texttt{getRecVar(while cond do stmt end do)} = \texttt{getRecVar(stmt)}$

(GS) $\texttt{getRecVar(stmt}_1\texttt{;stmt}_2\texttt{)} = \texttt{getRecVar(stmt}_1\texttt{)} \cup \texttt{getRecVar(stmt}_2\texttt{)}$

We use `getRecVar` function to obtain the recurrence variables in the loop body. Let `RecVar` denotes the recurrence variables in the loop body, then we have:

$$\text{RecVar} = \text{getRecVar(stmt)}$$

$\texttt{evalRecExpr} : \texttt{Expr} \rightarrow \texttt{StateRep} \rightarrow \texttt{OutputExpr}$

(ER1) $\texttt{evalRecExpr(constant,s)=constant}$

(ER2) $\texttt{evalRecExpr(var,s)=}$

$$\begin{cases} \texttt{var(t+1)}, & \texttt{if s=}\delta\texttt{(s',var(t+1)=expr)} \\ \texttt{evalRecExpr(var,s')}, & \texttt{if s=}\delta\texttt{(s',var'(t+1)=expr)} \\ \texttt{var(t)}, & \texttt{if s=}\delta\texttt{(}\perp\texttt{,var'(t+1)=expr)} \bigwedge \texttt{var}\in \texttt{RecVar} \\ \texttt{evalExpr(var, s)}, & \texttt{otherwise if var} \notin \texttt{RecVar} \end{cases}$$

(ER3) $\texttt{evalRecExpr(expr}_1 \texttt{ op expr}_2\texttt{,s)}$

$= \texttt{evalRecExpr(expr}_1\texttt{,s) op evalRecExpr(expr}_2\texttt{,s)}$

$\texttt{evalConExpr : Expr} \rightarrow \texttt{StateRep} \rightarrow \texttt{OutputExpr}$

(EC1) $\texttt{evalConExpr(constant,s)= constant}$

(EC2) $$\texttt{evalConExpr(var,s)=} \begin{cases} \texttt{var(t)}, & \texttt{if var} \in \texttt{s} \\ \texttt{var}, & \texttt{otherwise} \end{cases}$$

(EC3) $\texttt{evalConExpr(expr}_1 \texttt{ op expr}_2\texttt{, s)}$

$= \texttt{evalConExpr(expr}_1\texttt{,s) op evalConExpr(expr}_2\texttt{,s)}$

Loop Stop Condition

we can decide the loop stop condition on the basis of the condition *cond*. In general, this condition reads

$$\min_{t \geq 0} \{t \mid \neg cond\}$$

where *cond* depends on the state variables at time t. In general, this is a Diophantine equation, and thus well-known to be unsolvable. But in many practical situations, the actual equations are simple. We draw attention to three such cases.

• *Case 1* $cond = v_i\ R\ c$, where $v_1, \ldots, v_m$ are the recurrence variables, c is constant with respect to the v_i's, and R is a relational operator from Table 4.1. The converse of R can easily be computed explicitly, also shown in Table 4.1. Assuming that the expression for $v_i = F_i(t)$ is simple enough (in terms of t), this can be solved in closed form.

• *Case 2* $cond = v_i\ R\ \phi(v_j)$, where $v_1, \ldots, v_m$ are the recurrence variables, and some of them occur in the expression ϕ, with R as before.

$$\begin{aligned} z &= \min_{t \geq 0}\{t | \neg(v_i\ R\ \phi(v_j)\} \\ &= \min_{t \geq 0}\{t | F_i(t)\ S\ \phi(F_j(t))\} \end{aligned}$$

where $S = \neg R$.

• *Case 3* *cond* is a conjunction of terms which satisfy *Case 1* or *Case 2*. Then we can simply take the minimum of all the conjuncts.

For Loop Statements

`For loop` statement is similar to the `while loop`, the difference is that the for loop iteration times can be decided more easily than the while loop.

$[\![\ ]\!] : \texttt{Statement} \rightarrow (\texttt{StateRep} \rightarrow \texttt{StateRep})$

$\eta : \texttt{[Variable, Int, Int, Int]} \rightarrow \{\texttt{Identifier}\} \rightarrow \texttt{StateRep} \rightarrow \texttt{StateRep}$

(FR) $[\![\texttt{for i from s to e by step do stmt end do}]\!] =$

$\lambda \texttt{s} \in \texttt{StateRep}.\ \eta([\texttt{i, s, e, step}], \texttt{getRecVar(stmt)}, [\![\texttt{stmt}]\!](\texttt{s}))$

$\texttt{getRecVar} : \texttt{Statement} \rightarrow \texttt{StateRep}$

(G1) $\texttt{getRecVar(var:=expr)=\{var(n)\}}$

(GS) $\texttt{getRecVar(stmt}_1\texttt{;stmt}_2\texttt{)} = \texttt{getRecVar(stmt}_1\texttt{)} \cup \texttt{getRecVar(stmt}_2\texttt{)}$

$$\texttt{evalFor : [Variable, Int, Int, Int]} \rightarrow \texttt{StateRep} \rightarrow \texttt{OutputExpr}$$

(EF) $$\texttt{evalFor([i, s, e, step], s}_1\texttt{)} = \left\lfloor \frac{\texttt{e} - \texttt{s} + 1}{\texttt{step}} \right\rfloor \texttt{if i} \notin \texttt{s}_1$$

We use `getRecVar` to get recurrence variable in the loop body, which is same as the definition in the while loop. Rule (EF) describes the for loop iteration evaluation. `[i, s, e, step]` is the representing of the `for loop` control condition by using loop variable, initial value, ending value and step value to describe.

We can reduce η function into the following:

$$\texttt{Reduce}(\eta(\texttt{[i, s, e, step]},\texttt{getRecVar(stmt)}, [\![\texttt{stmt}]\!](\texttt{s})) =$$
$$\texttt{let } t = \texttt{evalFor([i, s, e, step], s) in}$$
$$([\![\texttt{stmt}]\!](\texttt{s}))^{\texttt{t}}$$

4.1.6 Code Examples

Example 1

Example 1 is to implement assignment statements and conditional statements to get symbolic interpretation of these statements.

$l_1 : x := x_1;$

$l_2 : y := y_1;$

$l_3 :$ **if** $y < 0$ **then** $x := -2 * y;$ **else** $x := 2 * y$ **end if**

$s = \perp$

$$
\begin{aligned}
[\![x := x_1]\!](s) &= \delta(s, x = evalExpr(x_1, s)) \\
&= \delta(\perp, x = x_1) = s_1 \\
[\![y := y_1]\!](s_1) &= \delta(s_1, y = evalExpr(y_1, s_1)) \\
&= \delta(\delta(\perp, x = x_1), y = y_1) = s_2 \\
[\![\ \texttt{if}\ ...]\!](s_2) &= \gamma(y < 0, [\![x := -2 * y]\!](s_2), [\![x := 2 * y]\!](s_2)) \\
&= \gamma(y < 0, \delta(\delta(\delta(\perp, x = x_1), y = y_1), x = -2 * y_1) \\
&\qquad\qquad , \delta(\delta(\delta(\perp, x = x_1), y = y_1), x = 2 * y_1))
\end{aligned}
$$

$$
\begin{aligned}
&\text{Reduce}(\gamma(y < 0, s_3, s_4)) \\
= &\begin{cases}
\delta(\delta(\delta(\perp, x = x_1), y = y_1), x = -2 * y_1) & \text{if } y_1 < 0 \\
\delta(\delta(\delta(\perp, x = x_1), y = y_1), x = 2 * y_1) & \text{if } y_1 \geq 0
\end{cases}
\end{aligned}
$$

Example 2: Factorial

We choose Factorial function as an example with `while loop` to get symbolic interpretation of these statements.

$$l_1 : i := 1;$$

$$l_2 : fac := 1;$$

$$l_3 : \textbf{while} \quad i < n \quad \textbf{do} \quad i := i + 1; fac := fac * i \quad \textbf{end do}$$

$s = \perp$

$$
\begin{aligned}
[\![i := 1]\!](s) &= \delta(s, i = evalExpr(1, s)) \\
&= \delta(s, i = 1) = s_1 \\
[\![fac := 1]\!](s_1) &= \delta(s_1, fac = evalExpr(1, s_1)) \\
&= \delta(\delta(\perp, i = 1), fac = 1) = s_2
\end{aligned}
$$

$$[\![\texttt{while i<n do i:=i+1; fac:=fac*i end do}]\!](s_2)$$
$$= \mu(i < n, getRecVar(i := i+1; fac := fac * i), [\![i := i+1; fac := fac * i]\!](s_2))$$
$$= \mu(i < n, \{i, fac\}, \delta(\delta(\delta(\delta(\bot, i = 1), fac = 1),$$
$$i(t+1) = i(t) + 1), fac(t+1) = fac(t) * i(t+1))) = s_3$$

Reduce s_3 to the following:

Reduce(s_3)

$$= (\delta(\delta(\delta(\delta(\bot, i = 1), fac = 1), i(t+1) = i(t) + 1),$$
$$fac(t+1) = fac(t) * i(t+1)))^{\min_{t \geq 0}\{t | i(t) \geq n\}}$$

Example 3: Chebyshev Polynomials

Chebyshev polynomial is an example implementing `for loop` statement to get symbolic interpretation of these statements.

$l_1 : u_0 := 1;$

$l_2 : u_1 := x;$

$l_3 :$ for i from 2 to n do

$v = u_1;$

$u_1 = -u_0 + 2 * x * u_1;$

$u_0 = v;$

end do;

$s = \bot$

$$[\![u_0 := 1]\!](s) = \delta(s, u_0 = evalExpr(1, s))$$
$$= \delta(s, u_0 = 1) = s_1$$
$$[\![u_1 := x]\!](s_1) = \delta(s_1, u_1 = evalExpr(x, s_1))$$
$$= \delta(\delta(\perp, u_0 = 1), u_1 = 1) = s_2$$

$$[\![\texttt{ for i from 2 to n do } v= u_1\texttt{; } u_1\texttt{=-}u_0\texttt{ + 2*x*}u_1\texttt{; } u_0\texttt{=}v\texttt{ end do}]\!](s_2)$$
$$= \eta([i, 2, n-1, 1], getRecVar(v := u_1; u_1 := -u_0 + 2 * x * u_1; u_0 := v),$$
$$[\![v := u_1; u_1 := -u_0 + 2 * x * u_1; u_0 := v]\!](s_2))$$
$$= \eta([i, 2, n-1, 1], \{v, u_0, u_1\}, \delta(\delta(\delta(\delta(\delta(\perp, u_0 = 1), u_1 = x), v(t+1) = u_1(t)),$$
$$u_1(t+1) = -u_0(t+1) + 2 * x * u_1(t)), u_0(t+1) = v(t+1)))$$
$$= s_3$$

Reduce s_3 to the following:

Reduce(s_3)

$$= (\delta(\delta(\delta(\delta(\delta(\perp, u0 = 1), u1 = x), v(t+1) = u1(t)),$$
$$u1(t+1) = -u0(t+1) + 2 * x * u1(t)), u_0(t+1) = v(t+1))^{(n-1)}$$

The above two examples show how to extract the meaning from the input function to generate initial equations, recurrence equations and loop iteration times.

Chapter 5

System Analysis

Before describing the system analysis, it is convenient to discuss the syntax and semantics of the input language for programs that our system can handle.

5.1 Input Language

The language for programs to be input by our symbolic system was chosen as a subset of Maple [27]. Representation language contains usual mathematical operators (+, *, <, $\leq$) but also $\sum$ (it is used to compute a closed form for an indefinite or definite sum), $\prod$ (which is is used to compute a formula for an indefinite or definite product), recurrences, etc., and it is a combination of the following statements:

1. `var:=expr`
2. `if cond then` $\texttt{stmt}_1$ `else` $\texttt{stmt}_2$
3. `while cond do stmt end do`
4. `for i from s to e by step do stmt end do`
5. *Recursive Function Calls*

Program Statements	Relations Generated
`var:=expr`	`StateTransition`
`if-then-else`	`Piecewise`
`while C do`	`Fixedpoint([C],...)`
`for-from-to-by-do`	`Fixedpoint(For(),...)`
Recursion	`RecursionCall`

Table 5.1: Rules for program transformation

5.2 Relation Generator

In our system, we have two modules: *Relation Generator* and *Relation Solver*, both written in Maple [27]. This is made especially easy since Maple has some very powerful reflection capabilities through its `ToInert` function, which gives an accurate AST representation for any Maple program (or expression). The first one generates a series of appropriate (recurrence, state transition, etc.) for the given input program, while the second one solves the relations and produces an output expression, either in an explicit form, or, if an implicit form can not be found, then implicit forms (like invariants) are returned. Figure 5.1 shows our symbolic execution system.

The *Relation Generator* is a total function – it translates the given input program into a sequence of appropriate relations. We transform the original programs into a sequence of relations according to the rules shown in Table 5.1.

Table 5.2 shows what is generated for our program `factorial`. Table 5.3 shows the mapping relation between the generated relations in our system and in the semantic functions.

For the input program, how can our system make difference between different types of statement? In the *Relation Generator*, we call `ToInert(eval(f))`, which

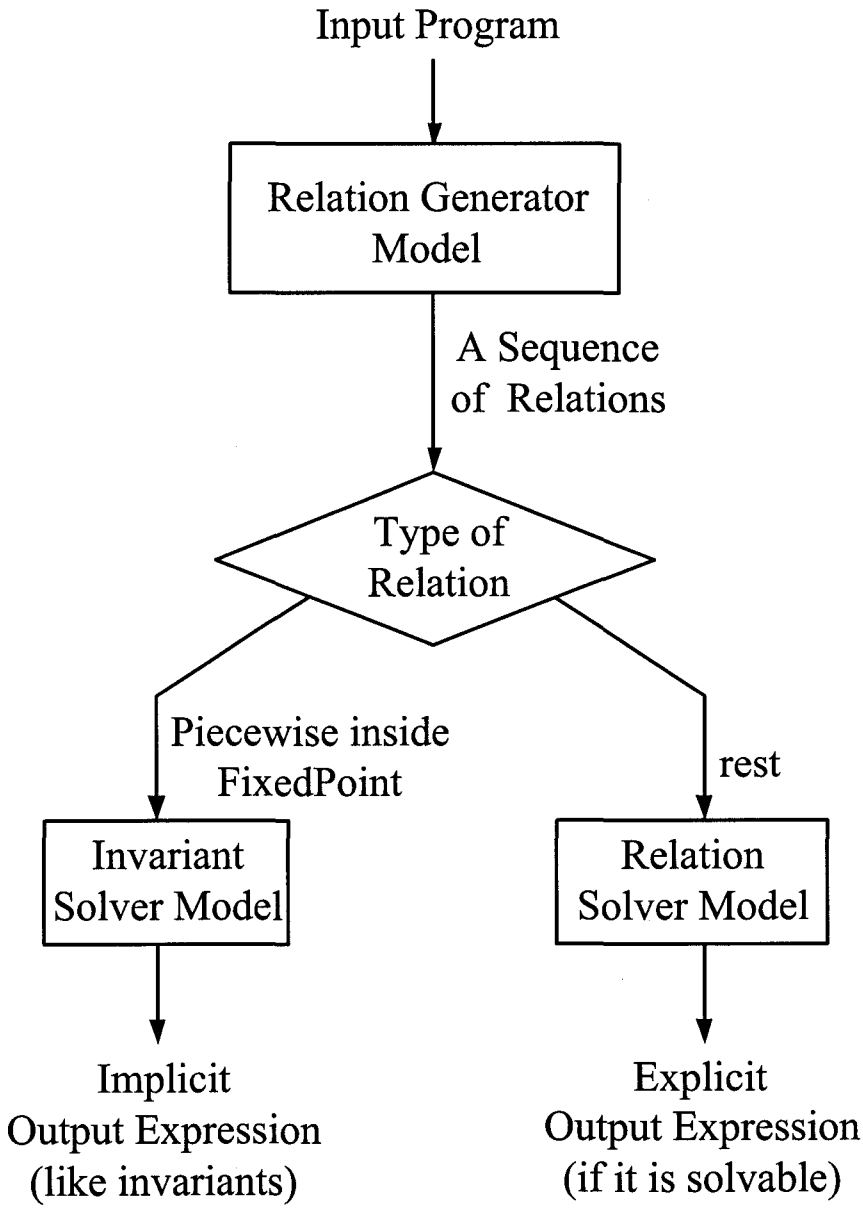

Figure 5.1: Symbolic execution system

Input Program	Relations
`factorial:=proc(n)`	
`i:=1`	`F1:StateTransition(i,1)`
`fac:=1`	`F2:StateTransition(fac,1)`

`while i < n do`
`  i:=i+1`
`  fac:=fac*i`
`end do`
	`F3:FixedPoint([i < n],`
	`[StateTransition(i,i+1),`
	`StateTransition(fac,fac*i)]`
	`)`

`fac`	`F4:fac`
`end proc;`	

Table 5.2: Translation of `factorial` into the set of appropriate relations

Generated Relation(GenRelation)	Semantic Function
StateTransition(var, expr)	δ(var, expr)
Piecewise(cond, transition1, transition2)	γ(cond,$[\![\text{stmt}_1]\!]$, $[\![\text{stmt}_2]\!]$)
Fixedpoint([cond], transition1)	μ(cond,{var}, $[\![\text{stmt}]\!]$)
Fixedpoint(For[i, s, e, step], transition1)	η([i, s, e, step], {var}, $[\![\text{stmt}]\!]$)

Table 5.3: Mapping relations between the generated relations and the semantic functions

converts the input Maple function f into a sequence of inert form. Table 5.4 shows the meanings of Maple `Inert` functions. For example, for the `factorial` function given in chapter 2, calling `ToInert(eval(factorial))` inside the *Relation Generator*, we obtain the following:

```
_Inert_PROC(_Inert_PARAMSEQ(_Inert_NAME("n")),
_Inert_LOCALSEQ(_Inert_NAME("i"), _Inert_NAME("fac")),
_Inert_OPTIONSEQ(), _Inert_EXPSEQ(),
_Inert_STATSEQ(_Inert_ASSIGN(_Inert_LOCAL(1),
_Inert_INTPOS(1)), _Inert_ASSIGN(_Inert_LOCAL(2), _Inert_INTPOS(1)),
_Inert_FORFROM(_Inert_EXPSEQ(),_Inert_INTPOS(1), _Inert_INTPOS(1),
_Inert_EXPSEQ(), _Inert_INEQUAT(_Inert_LOCAL(1), _Inert_PARAM(1)),
_Inert_STATSEQ(_Inert_ASSIGN(_Inert_LOCAL(1),
_Inert_SUM(_Inert_LOCAL(1), _Inert_INTPOS(1))),
_Inert_ASSIGN(_Inert_LOCAL(2),
_Inert_PROD(_Inert_LOCAL(2), _Inert_LOCAL(1)))))), _Inert_LOCAL(2)),
_Inert_DESCRIPTIONSEQ(), _Inert_GLOBALSEQ(), _Inert_LEXICALSEQ(),
_Inert_EOP(_Inert_EXPSEQ()))
```

The above sequence of `Inert` functions exactly corresponds to the input program `factorial`. Table 5.5 shows the rules to translate the `Inert` functions into our symbolic system relations.

5.3 Solving Relations: Overview

The method for solving relations depends on the kind of relation that generated. The technique described in this and almost all remaining chapters is a refinement and generalization of many results from [7, 20, 30]. Of course, if code does not contain

Inert Function	Meaning
`_Inert_PARAMSEQ`	Providing the parameters of the procedure
`_Inert_LOCALSEQ`	Providing the local variables of the procedure
`_Inert_LOCAL`	Mapping to the local variable
`_Inert_STATSEQ`	Providing the statement sequence of the procedure
`_Inert_ASSIGN`	Corresponding to the Assignment statement
`_Inert_FORFROM`	Corresponding to the Loop statement
`_Inert_IF`	Corresponding to the If conditional statement

Table 5.4: Inert functions and their meanings

Inert Function	Generated Relation	Rule
`_Inert_ASSIGN`	**StateTransition**	T1
`_Inert_IF`	**Piecewise**	T2
`_Inert_FORFROM(_Inert_EXPSEQ...)`	**Fixedpoint**	T3
`_Inert_FORFROM(_Inert_LOCAL...)`	**Fixedpoint(For(), ...)**	T4

Table 5.5: Transformation and rules

either loops nor recursion, from a symbolic point of view such straight-line code is completely trivial, and we can simply compute the result. The only drawback is that such an answer can be exponentially larger than the input program.

For the case where we have either a `while` or `for` loop whose body is straight-line code, we generate a system of recurrence equations, which we try to solve in closed-form, using whatever triangular structure we may find. Using similar ideas, we can also generate systems of recurrences for programs containing recursion.

When loops contain branches (i.e. `if-then-else`), the resulting system of recurrences essentially **never** falls within a class of solvable recurrence. At present, we immediately shift to generate implicit results, in the form of polynomial invariants [20, 29].

We classified the generated relations into five classes according to the types of relations.

1. *FixedPoint.* It maps to the input procedure including `while-do` structure. The recursive functions and initial functions will be generated and solved based on the loop stop condition, which can be calculated in this case.

2. *FixedPoint(For).* It maps to the input procedure including `for-do` structure. The recursive functions and initial functions will be generated and solved based on the number of times the loop `for-do` iterates.

3. *Piecewise inside FixedPoint.* It happens when we have `if-then-else` inside `while-do`. It is usually very difficult to decide the number of iterations, so we usually cannot generate explicit output. We might generate implicit output, i.e. an invariant instead, more or less in the style of [29].

4. *RecursionCall.* We can generate some recursive functions and their initial functions and then solve them(but not always).

5. *Neither Fixedpoint nor RecursionCall.* It maps to the input procedure is the sequential procedure without `while-do` or `for-do` in it. We do not need to generate recursive functions or invariants. We can easily get the explicit output by simply calculating relation composition.

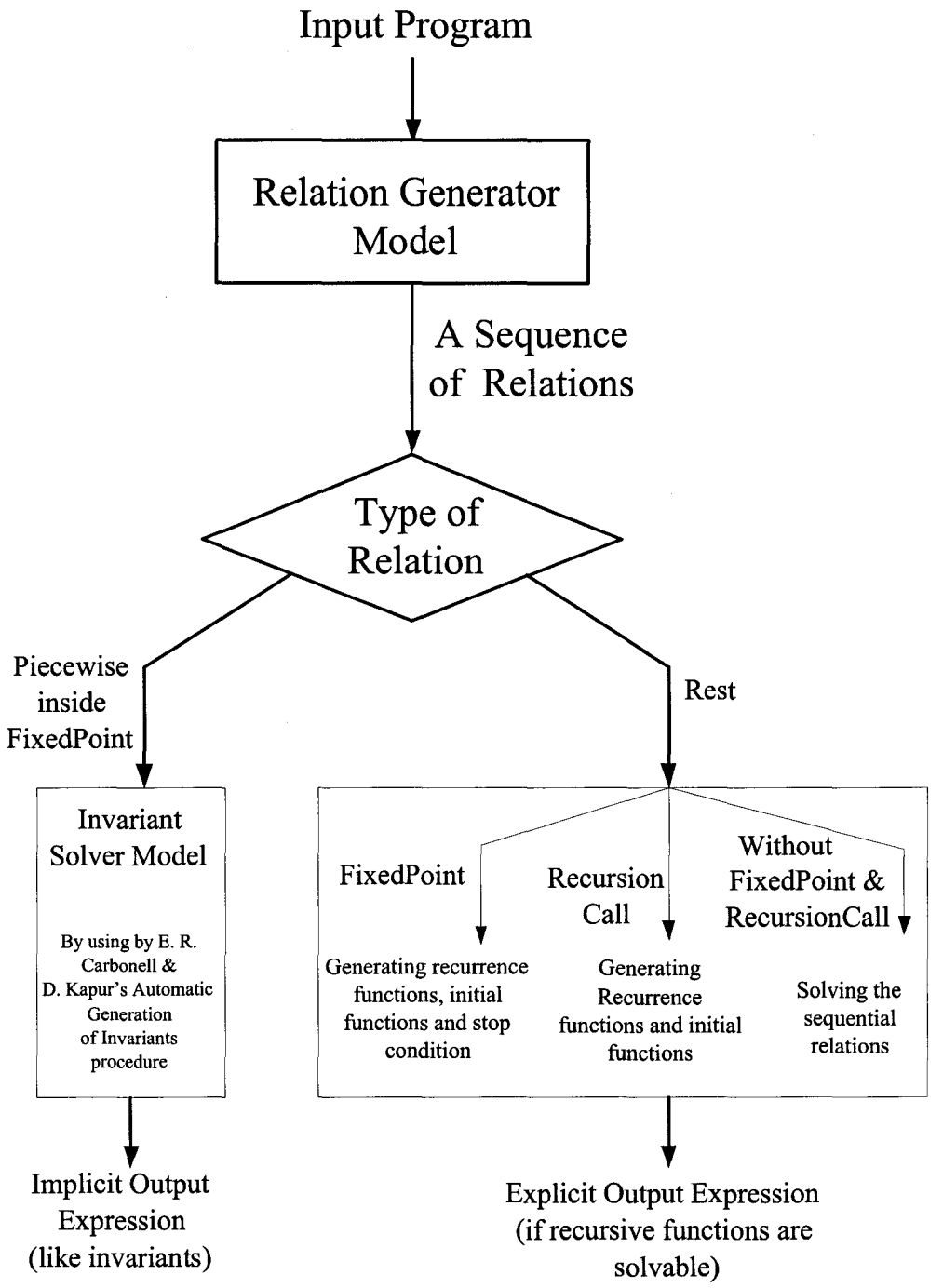

Figure 5.2: Symbolic execution system in detail

Relations	Recurrence and Initial Condition
`StateTransition(i,1)`	*Initial Condition:* $i(0) = 1$
`StateTransition(fac,1)`	*Inital Condition:* $fac(0) = 1$
`FixedPoint(i< n,`	*Loop Termination:* $lt = min\{t \geq 0 \mid i(t) \geq n\}$
`StateTransition(i,i+1),`	*Recurrence:* $i(t+1) = i(t) + 1$
`StateTransition(fac,fac*i),` `)`	*Recurrence:* $fac(t+1) = fac(t) \cdot i(t+1)$
`return fac`	$fac(lt)$

Table 5.6: Recurrence equations and initial conditions for `factorial`.

5.4 From Code to Recurrences: while

If the input program is a simple `while` loop, without `if-then-else` statements inside the `while` loop, the core relation we generate will be `FixedPoint`. Table 5.6 shows the results for our `factorial` program. In this case our system [28] will produce the output formula "$n!$".

5.4.1 Generating Recurrence Relations

In this case all we have inside the loop are assignment statements which are represented by `StateTransition` relations. These relations might however be *mixed*, in other words a variable at time $t + 1$ might occur on both the left and right hand sides. This occurs in our `factorial` code, where *fac* depends on $i(t + 1)$ rather than $i(t)$. However, a simple program transformation related to *Static Single Assignment (SSA)* form [11] takes care of this issue.

5.4.2 Generating Initial Conditions

The initial conditions are easily determined: they are the values of each of the loop variables (i.e. those which change) right before the loop starts. These can be determined by unwinding the stack of `StateTransition` calls preceding the loop. This is always possible, though might again generate very large answers.

5.4.3 Stopping Conditions

If we want to find the actual stopping condition for a loop, we need to solve (symbolically) the recurrence equations (with known initial functions) just generated. Suppose the solution is:

$$v_1(t) = F_1(t), v_2(t) = F_2(t), \ldots, v_m(t) = F_m(t).$$

Now, we can decide the loop stop condition on the basis of the condition *Cond* in `FixedPoint([Cond],...)`. In general, this condition reads

$$\min_{t \geq 0}\{t \mid \neg Cond\}$$

where *Cond* depends on the state variables at time t. We have discussed it in chapter 4.

5.5 Solving with for Loops

Since a `for` loop is a special case of a `while` loop, this case is very similar to the previous. Generating recurrences is exactly the same. Assume the following pattern for the `for` loop:

`for i from` s `to` e `by step do S.`

Generating Initial Conditions

Initial functions are generated from the stack of `StateTransitions` preceding the loop for all variables, with the addition of $i(0) = s$. Assuming [StateTransition(v_1, $expr_1$), StateTransition(v_2, $expr_2$),..., StateTransition(v_m, $expr_m$)] precedes the loop, where $v_1, v_2, \ldots, v_m$ are recursive variables, then the generated initial functions are in the followings:

$$
\begin{aligned}
v_1(0) &= expr_1, \\
v_2(0) &= expr_2[v_1 := v_1(0)], \\
\ldots &= \ldots, \\
v_m(0) &= expr_m[v_1 := v_1(0), v_2 := v_2(0), \ldots, v_{m-1} := v_{m-1}(0)] \\
i(0) &= s
\end{aligned}
$$

where expr[var:=expr1] means the expression expr with variable var in expr replaced with expr1. It follows the chain of δ that we discussed in chapter 4.

5.5.1 Number of Loop Iterations

Now, we can decide the for loop iteration times based on the For relation inside the FixedPoint relation. We have to consider two cases:

- *Case 1* The variable i, i.e. loop counter, *is not modified by* S. In this case the number of iterations z can be solved explicitly and uniformly for all cases, and is given by

$$
z = \begin{cases} \left\lceil \frac{e-s+1}{step} \right\rceil & \text{if } \left\lceil \frac{e-s+1}{step} \right\rceil > 0 \\ 0 & \text{if } \left\lceil \frac{e-s+1}{step} \right\rceil \leq 0 \end{cases}
$$

• *Case 2* The variable i, i.e. loop counter, *is modified by* S. In this case we have to transform the `for` loop into a new `for` loop with the i update inside the loop, and using a new independent variable to control the loop. Then it transforms into the form of case 1, therefore we can decide the loop iteration times. In chapter 6, the Binomial Coefficients example shows how to solve this kind of problem.

• *Case 3* The variable i, i.e. loop counter, *is not only modified by s, but also starts from 0.* We choose to shift loop starting point and ending point by step times. Then applying to case 2, we can decide the loop iteration times. In chapter 6, the Bessel function example explains how to solve this kind of problem in detail.

5.5.2 Solving Relations Involving Recursion Call

If a recursive function is correctly defined, it defines both recurrence functions and initial conditions in quite natural way.

For example, some recursive function generates the relations: StateTransition(u, piecewise(n=0, $expr_1$, f(RecursionCall(n-1))), then we can get the initial and Recurrence functions in the followings:

$$u(0) = expr_1$$
$$u(t) = f(u(t-1))$$

Based on the above that generated initial and recurrence functions, we can solve them and get the implicit output.

However, we can not always solve (symbolically) the recurrence equations thus generated (for instance we cannot do it for Ackerman function).

Note that it is important here to assume that we have a *meaningful* program, as otherwise a recursively defined function might come equipped with naturally defined

initial conditions.

5.5.3 The Case of Branches in Loops

When we have `if-then-else` inside a `while`, we usually are not able to generate explicit symbolic output. We can often generate implicit output, or invariants, in a way similar to that described in [20, 29]. From the generated relations, we can translate them into the input of Rodríguez-Carbonell and Kapur's method[29] to automatic generation of polynomial loop invariants. We will discuss how to automatically generate the input of Rodríguez-Carbonell and Kapur's method[29] from the relations.

Let us give an example to see how to obtain input polynomials from relations.

```
e:= [StateTransform(a, x), StateTransform(b, y), StateTransform(p1, 1),
StateTransform(q1, 0), StateTransform(r, 0), StateTransform(s1, 1),
Loop([a <> b], [[StateTransform(a, 'piecewise'(b < a, a-b, a)),
StateTransform(b, 'piecewise'(b < a, b, b-a)),
StateTransform(p1, 'piecewise'(b < a, p1-q1, p1)),
StateTransform(q1, 'piecewise'(b < a, q1, q1-p1)),
StateTransform(r, 'piecewise'(b < a, r-s1, r)),
StateTransform(s1, 'piecewise'(b < a, s1, s1-r))]]), a]
```

The above relation is generated from GCD function(given in chapter 6). The set of initial polynomial can be generated from the stack of `StateTransitions` preceding the loop for all variables, i.e. $[a = x, b = y, p1 = 1, q1 = 0, r = 0, s1 = 1]$. The sets of Conditional polynomial can be generated from the inside of `Loop` relation, grouping by the condition inside the piecewise, i.e. $[a = a - b, b = b, p1 = p1 - q1, q1 =$

$q1, r = r - s1, s1 = s1]$ and $[b = b - a, p1 = p1, q1 = q1 - p1, r = r, s1 = s1 - r]$. Transformation from the generated relations into the input polynomial for generating invariants is quite straightforward.

Let us have a brief introduction to Rodríguez-Carbonell and Kapur's method [29]. What is their module's theory background and why it works. Rodríguez-Carbonell and Kapur in [29] have proved that the set of polynomials serving as loop invariants has the algebraic structure of an ideal. Using this connection, it is proved that the procedure for finding invariants can be expressed using operations on ideals, for which Gröbner basis constructions can be employed.

For a given loop, the set $\{p\}$ of polynomials such that p=0 is invariant, i.e,. p evaluates to 0 at the header whenever the loop body is executed, is a polynomial ideal. This ideal is henceforth called the *invariant polynomial ideal* of the loop. Any conjunction of polynomial equations such that the polynomials are a basis of this ideal is shown to be *inductive*, i.e., it holds when entering the loop and is preserved by every iteration of the loop. Moreover, such formula formula is strongest among all the inductive invariants of the loop when invariants are conjunctions of polynomial equations. Using Hilbert's basis theorem, they also establish the existence of such an inductive invariant for a given loop. If a loop does not have any polynomial invariant, the procedure will generate the polynomial 0 (which is equivalent to *true*) as the invariant.

In [29], they show that how the procedure for computing the invariant polynomial ideal can be approximated using Gröbner bases computations. Moreover, for solvable mappings with rational positive eigenvalues, this approximation is exact, i.e. the algorithm computes the invariant ideal. More details are discussed in [29] about how to generate polynomial invariants [29].

In our system, one of module generating the invariants comes from Rodríguez-Carbonell and Kapur's method [29] source code. However since Rodríguez-Carbonell and Kapur's system [29] could not start to work from the given program point, our system has a module to generate their system input from the given program and based on their system to generate invariants automatically.

Chapter 6

Examples of Using Symbolic Execution Tool

This chapter gives some examples that show the use of symbolic execution tool. It starts off with the computation of $\sum_{i=0}^{n} \frac{1}{i!}$ example, which shows how the relations are get and how the initial functions, recursive functions and number of loop iteration can be decided from the relations. We also give the following examples: computation of $\sum_{i=0}^{n} i!$, computation of the binomial coefficients, computation of Chebyshev polynomials and the computation of values of Bessel function from Taylor series, to show how to get the symbolic explicit output. We also give two examples to explain how to get the implicit output.

6.1 Examples of Generating Explicit Output

6.1.1 Example 1: $\sum_{i=0}^{n} \frac{1}{i!}$

This is an example to compute $\sum_{i=0}^{n} \frac{1}{i!}$

```
sig_prod:=proc(n)
   local i, s, w;
   (s,w):=(1,1);
   for i from 1 to n  do
      w:=w/i;
      s:=s+w;
   end do;
   s;
end proc:
```

By observing the above function, we can see that the loop controller i not only controls the loop, but also shows up in the loop body (we only dealt with the case that i shows up on the right hand side of the expressions in the loop body). For this kind of input function, our system transforms the input function into a new function such that the loop controller is independent to control the loop (which means that it does not show up in the loop body). There are three steps:

1. Choose a variable, which does not belong to the input function, as the loop controller, i.e. ii.

2. Preceding the loop execution, adding a new assignment statement to initialize the original loop controller according to the loop start point.

3. At the loop bottom, adding an assignment statement to update the original loop controller by the sum of it and the step.

Table 6.1 shows a direct translation between the program components and the recurrence, initial conditions, and number of loop iterations. Our system [28] produces the

$\sum_{i=0}^{n} \frac{1}{i!}$	Recurrence and Initial Condition
`(s, w) := (1, 1);`	*Initial Condition:* $s(0) = 1, w(0) = 1, i(0) = 1$
`for i from 1 to n do`	*Number of Loop Iteration:* $z = \lceil \frac{n-1+1}{1} \rceil = n$ if $n \geq 1$
`w := w/i;` `s := s+w`	*Recurrence Equations:* $w(t+1) = w(t)/i(t)$ $s(t+1) = s(t) + w(t+1)$ $i(t+1) = i(t) + 1$
`s`	$s(z)$

Table 6.1: Recursive and initial functions for computing $\sum_{i=0}^{n} \frac{1}{i!}$

following output for the above program :

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

It is an approximation to $exp(1)$. The incomplete Γ function is defined as:

$$\Gamma(a, z) = \Gamma(a) - z^a/a1F1(a, 1+a, -z)$$

where $1F1$ is the confluent hypergeometric function (in Maple notation, $1F1(a, 1+a, -z) = \text{hypergeom}([a], [1+a], -z)$).

6.1.2 Example 2: $\sum_{i=0}^{n} i!$

This is an example to compute $\sum_{i=0}^{n} i!$. Table 6.2 shows a direct translation between the program components and the recurrence, initial conditions, and number of loop iterations. Our system [28] produces the following output for the above program :

$\sum_{i=0}^{n} i!$	Recurrence and Initial Condition
`(s, w) := (1, 1);`	*Initial Condition:* $s(0) = 1, w(0) = 1, i(0) = 1$
`for i from 1 to n do`	*Number of Loop Iteration:* $z = \lceil \frac{n-1+1}{1} \rceil = n$ if $n \geq 1$
`w := w*i;` `s := s+w`	*Recurrence Equations:* $w(t+1) = w(t) * i(t)$ $s(t+1) = s(t) + w(t+1)$ $i(t+1) = i(t) + 1$
`s`	$s(z)$

Table 6.2: Recursive and initial functions for computing $\sum_{i=0}^{n} i!$

$$-KummerU(1,1,-1) - n! * KummerU(n+1, n+1, -1) * (-1)^{(n+1)} + n! \quad (6.1)$$

Where KummerU(μ, ν, z) solves the differential equation $z*y'' + (\nu - z)*y' - \mu*y = 0$ [1].

There is another function to compute $\sum_{i=0}^{n} i!$. Table 6.3 shows the translation between the program components and the recurrence, initial conditions, and number of loop iterations. Our system [28] produces the following output for the above program :

$$\frac{e^{-1}\Gamma(-n,-1)}{\Gamma(-n)} - \frac{\Gamma(-1,-1)e^{-1}}{\Gamma(-n+1)\Gamma(-1)} - \frac{(-1)^n}{\Gamma(-n+1)} + 1 \quad (6.2)$$

Even though I believe that (6.1) and (6.2) are equal, it is challenge for me to prove it.

$\sum_{i=0}^{n} i!$	Recurrence and Initial Condition
`s := n ;`	*Initial Condition:* $s(0) = n, i(0) = n - 1$
`for i from n-1 by -1 to 1 do`	*Number of Loop Iteration:* $z = n - 1$ if $n \geq 2$
`s := i*(1+s);`	*Recurrence Equations:* $s(t+1) = i(t) * (1 + s(t))$ $i(t+1) = i(t) - 1$
`s+1`	$s(z) + 1$

Table 6.3: Another example to compute $\sum_{i=0}^{n} i!$

6.1.3 Example 3: Chebyshev Polynomials

Here we show a simple example with `for` loops.

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

chebyshev Polynomials	Recurrence and Initial Condition
`(u0,u1) := (1,x);`	*Initial Condition*: $u0(0) = 1, u1(0) = x$
`for i from 2 to n do`	*Number of Loop Iteration*: $z = \lceil \frac{n-2+1}{1} \rceil = n - 1$ if $n \geq 2$
	Recurrence Equations:
`v := u1;`	$v(t+1) = u1(t)$
`u1 := -u0 + 2*x*u1`	$u1(t+1) = -u0(t) + 2 \cdot x \cdot u1(t)$
`u0 := v`	$u0(t+1) = v(t+1)$
`u1`	$u1(z)$

Table 6.4: Recursive and initial functions for `chebyshev`

Translation of the program into set of appropriate relations is quite straightforward and is omitted. Table 6.4 shows a direct translation between the program components and the recurrence, initial conditions, and stopping condition.

Our system [28] produces the following output for the above program `chebyshev`:

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

better known as the closed-form for the Chebyshev polynomial $T_n(x)$ for $n \geq 2$ [1].

6.1.4 Example 3: Binomial Coefficients

The binomial coefficient $\binom{u}{k}$ is the number of ways of picking k unordered outcomes from u possibilities, also known as a combination or combinatorial number. The computation of binomial coefficients is given by the followings:

```
binomial:=proc(u, k)
```

Binomial Coefficients	Recurrence and Initial Condition
`res:= 1`	*Initial Condition*: $res(0) = 1$, $i(0) = 1$
`for i from 1 to k do`	*Number of Loop Iterations*: $z = \lceil \frac{k-1+1}{1} \rceil = k$ if $k \geq 1$
`res := res * (u-i+1)/i;`	*Recurrence Equations* $res(t+1) = res(t) * (u - i(t) + 1)/i(t)$ $i(t+1) = i(t) + 1$
`res`	$res(z)$

Table 6.5: Recursive and initial functions for `Binomial Coefficients`

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

Transformation of `binomial` into an equivalent new `binomial` function in which the loop controller is independent is quite straitforward and is omitted. Table 6.5 shows a direct translation between the binomial coefficient program and the recurrence, initial conditions, and number of loop iterations.

Our system [28] produces the following output for the above program `chebyshev`:

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

Proposition 1.

$$\frac{(-1)^k\Gamma(-u+k)}{\Gamma(-u)\Gamma(k+1)} \quad \textit{is equivalent to binomial coefficient } \binom{u}{k}.$$

Proof. In order to prove

$$\frac{(-1)^k\Gamma(-u+k)}{\Gamma(-u)\Gamma(k+1)} = \frac{\Gamma(u+1)}{\Gamma(u-k+1)\Gamma(k+1)}$$

we only need to prove

$$\frac{(-1)^k\Gamma(-u+k)}{\Gamma(-u)} = \frac{\Gamma(u+1)}{\Gamma(u-k+1)}$$

Since

$$\frac{\Gamma(u+1)}{\Gamma(u-k+1)} = u*(u-1)\dots(u-k+1)$$

So, we need to show

$$\frac{(-1)^k\Gamma(-u+k)}{\Gamma(-u)} = u*(u-1)\dots(u-k+1)$$

let $-u = t-k+1$, then:

$$\begin{aligned}\frac{(-1)^k\Gamma(-u+k)}{\Gamma(-u)} &= \frac{(-1)^k\Gamma(t+1)}{\Gamma(t-k+1)} \\ &= (-1)^k * t*(t-1)*\cdots*(t-k+1) \\ &= (-1)^k * [-(u-k+1)*\cdots*(-(u-1))*(-u)] \\ &= (u-k+1)*\cdots*(u-1)*u\end{aligned}$$

Q.E.D. □

6.1.5 Example 4: Bessel

This is an example to show that in the for loop, the loop controller is not independent to control the loop, it also happens in the loop body statement. At the same time, the loop controller i starts from 0.

Bessel Function	Recurrence and Initial Condition
(res, u) = (0, 1)	*Initial Condition*: $res(0)=0,\ u(0)=1,\ i(0)=0$
for i from 0 to m-1 do	*Number of Loop Iterations*: $z = \lceil \frac{m-1+1}{1} \rceil = m$ if $m \geq 1$
res := res + u	*Recurrence Equations* $res(t+1) = res(t) + u(t)$
u := -u * z^2/(4*(i+nu+1)(i+1))	$u(t+1) = -u(t) * z^2/(4 * (i(t) + nu + 1)(i(t) + 1))$
	$i(t+1) = i(t) + 1$
res	$res(z)$

Table 6.6: Recursive and initial functions for `Bessel`

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

Since the loop controller starts from 0, we shift the loop starting and ending point by step. Transformation of the program into a new equivalent *bessel* is omitted and

the generation of recursive functions, initial functions, and loop termination condition are given in table 6.6.

From table 6.6, we get the following recurrence functions:

$$\begin{aligned} res(t+1) &= res(t) + u(t) \\ u(t+1) &= -u(t) \cdot z^2/(4 * (i(t) + nu + 1) \cdot (i(t) + 1)) \\ i(t+1) &= i(t) + 1 \end{aligned} \tag{6.3}$$

As s system, Maple can not solve (6.1) directly. However, if we solve the system in data-dependency order, i.e.

$$\begin{aligned} i(t+1) &= i(t) + 1 \\ u(t+1) &= -u(t) \cdot z^2/(4 * (i(t) + nu + 1) \cdot (i(t) + 1)) \\ res(t+1) &= res(t) + u(t) \end{aligned} \tag{6.4}$$

then the system is solvable by steps, and the result is:

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

where J_ν is the Bessel function of the first kind, while $s^{(+)}$ is known as Lommel's s function [1]. What is interesting about this 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.

6.1.6 Example 5: A Recursive Call Function

This is an example to show the recursive call.

```
chebyshev1:=proc(n)
local res;
```

chebyshev1 Function	Recurrence and Initial Condition
`if n=0 then res:=1`	*Initial Condition: res(0)=0*
`elif n=1 then res:=x`	*Initial Condition: res(1)=x*

`else`
`res := 2*x*chebyshev1(n-1)+`
`chebyshev1(n-2)`
	Recurrence Equations
	$res(t) = 2 * x * res(t-1)+$
	$res(t-2)$

`res`	$res(t)$

Table 6.7: Recursive and initial functions for `chebyshev1`

```
if n=0 then
   res:=1;
elif n=1 then
   res:=x;
else
   res:=2*x*chebyshev1(n-1)+chebyshev1(n-2);
end if;
   res;
end proc;
```

From the generated relation, our system can tell it is a recursive function call. Translation of the input *chebyshev1* into its equivalent relations is omitted and the generation of recursive functions and initial functions are given in table 6.7. For the recursive call function, we do not need to generate loop stop condition.

Our system produces the same output as the example 3 *chebyshev* polynomial function.

6.2 Examples of Generating Implicit Output

6.2.1 Example 1: GCD

The GCD example is to show how to deal with the single *while* loop with *if* condition in our symbolic execution system. The computation of GCD function is given by the followings:

```
GCD:=proc(x, y)
local a, b, p1, q1, r, s1;
    a:=x; b:=y; p1:=1; q1:=0; r:=0; s1:=1;
    while a <> b do
    if a > b then
       a:=a-b; p1:=p1-q1; r:=r-s1;
    else
       b:=b-a; q1:=q1-p1; s1:=s1-r;
    end if;
    end do;
    a;
end proc:
```

In our symbolic execution system, we use Rodríguez-Carbonell and Kapur's method [29] to get the invariant which denotes the implicit output. Table 6.8 shows to translate GCD program into the input of generating invariants [29].

Our system [28] produces the following output for the above program `GCD`:

$$\{-x - b*r + a*s1 = 0,\ y + a*q1 - b*p1 = 0,\ s1*y + q1*x - b = 0,$$
$$x*p1 + y*r - a = 0,\ \ 1 + q1*r - s1*p1 = 0\}$$

GCD program	Initial input of generating invariants [29]
a:=x; b:=y ; p1:=1; q1:=0; r:=0; s1:=1	*[a=x, b=y, p1=1, q1=0, r=0, s1=1],*
while $a <> b$ do	*[*
if $a > b$ then	*[*
a:=a-b; p1:=p1-q1; r:=r-s1;	*a=a-b, p1=p1-q1, r=r-s1],*
else	*[*
b:=b-a; q1:=q1-p1; s1:=s1-r;;	*b=b-a, q1=q1-p1, s1=s1-r*
end if; end do	*]]*

Table 6.8: Translation of program into the input of generating invariants [29] for GCD

The above is a set of polynomial invariants expressed in terms of ideal. Rodríguez-Carbonell and Kapur's [29] prove that the set of invariant polynomials of a loop has the algebraic structure of an ideal. Using this connection, they prove that the procedure for finding invariants can be expressed using operations on ideals. Moreover, for any finite basis of this ideal, the corresponding conjunction of polynomial equation is the strongest possible inductive invariant for the loop expressible as a conjunction of polynomial equations [29].

A loop invariant is a relation among program variables that is true when control enters a loop, remains true each time the program executes the body of the loop, and is still true when control exits the loop. Understanding loop invariants can help us analyze programs, check for errors, and derive programs from specifications [22].

6.2.2 Example 2: LCM

LCM program is an example shows how to deal with the nested loop in our symbolic execution system. The computation of LCM function is given by the followings:

```
LCM:=proc(a,b)
   local x, y, u, v;
   x:=a; y:=b; u:=b; v:=0;
   while x<>y do
      while x>y do
         x:=x-y; v:=u+v;
      end do;
      while x<y do
         y:=y-x; u:=u+v;
      end do;
   end do;
   return u+v;
end proc:
```

Table 6.9 shows to translate LCM program into the input of generating invariants [29]. [29] did not mention that if their method also works for nested while loop. It seems that their algorithm works for some nested while loop.

Our system [28] produces the following output for the above program `LCM`:

$$\{-u * x - y * v + a * b = 0\}$$

The above is also a set of polynomial invariants expressed in terms of ideal [29]. In our example, since we have nested loop, the polynomial invariant should be true on

LCM program	Initial input of generating invariants [29]
`x:=a; y:=b; u:=b; v:=0;`	*[x=a, y=b, u=b, v=0],*
`while` $x <> y$ `do`	*[*
`while` $x > y$ `do`	*[*
`x:=x-y; v:=u+v;`	*x=x-y, v=u+v*
`end do`	*],*
`while` $x < y$ `do`	*[*
`y:=y-x; u:=u+v;`	*y=y-x, u=u+v*
`end do`	*]])*

Table 6.9: Translation of program into the input of generating invariants [29] for LCM

each iteration before executing the outer loop, between executing the outer loop and inner loop, after executing the inner loop body, and after executing the outer loop body, which have more restricts than the single loop. This is one of the reason that why we got much less invariants expression from this example than from the GCD example.

Chapter 7

Related Work

Symbolic execution has been studied since seventies, however with different goals than ours. King [19] in 1976 has developed EFFIGY, a symbolic execution system with a fixed number of integers.

Kemmerer and Eckmann [18] have presented an approach to symbolic execution based on the concept of path expressions and path conditions.

DISSECT [15] and SELECT [5] are also symbolic execution systems that use the path conditions concept. DISSECT can be used to symbolically execute some simple FORTRAN programs. The main purpose of SELECT [5] is to complement mechanical program verification and debug programs.

Rodríguez-Carbonell and Kapur [29] have recently developed some interesting techniques for automatically finding loop invariants. E. Rodríguez-Carbonell and D. Kapur [29] introduce that conjunctions of polynomial equations as loop invariants. It shows that the set of invariant polynomials of a loop has the algebraic structure of an ideal, which immediately suggests that polynomial ideal theory and algebraic geometry can give insight into the problem of finding loop invariants. E. Rodríguez-

Carbonell and D. Kapur [29] also present the procedure for finding polynomial invariants, expressed in terms of ideals. E. Rodríguez-Carbonell and D. Kapur [29] also shows how to implement this procedure using Gröbner bases. The implementation has been used to automatically discover nontrival invariants for several examples to illustrate the power of the techniques. However, the input parameters still need to be extracted manually.

Fahringer and Scholz [30] have presented a comprehensive and compact control and data flow analysis information, called program context for solving program analysis problems. Program contexts include three conmponets: variable values, assumptions about and constrains between variable values, and path condition. Their approach targets linear and non-linear symbolic expression and the program analysis information is represented as symbolic expression defined over the program's problem size. Fahringer and Scholz [30] introduce an algorithm for generating program contexts based on control flow graphs. The alogrithm comprises accurate modeling of assignment and input/out statements, branches, lops, recurrences, arrays, dynamic records and procedures.

Chapter 8

Contributions 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*. The system [28] can handle assignment statements, `if-then-else` statements, `for-do` statements, and `while-do` statements, the latter two with some restrictions.

Despite the restrictions, it can be used for a huge variety of programs, including programs to compute binomial coefficients, Bessel functions, orthogonal polynomials, and so on. In the case of finding invariants, we mainly follow [29] and can cover similar programs.

For the 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 also would like to be able to produce explicit symbolic solutions in some cases where now we can only produce invariants. We also would like to enrich the programming

model to consider some more complicate data structures such as arrays.

Appendix A

Specification by Maude

Maude is a high-level language and is an equationally-base, algebraic language with a term rewriting implementation. We use maude to define the specification for our symbolic execution system.

The following model describes the Arithmetic Expression type. It corresponds to the `ArithmeticExpression` and `ArithmeticOp` in the semantics syntax definition.

```
fmod ARITHMETICEXPRESSION is
  protecting QID .
  protecting INT .
  sort Variable ArithmeticExpression .
  subsorts Variable < ArithmeticExpression .
  subsort Qid < Variable .
  subsorts Int < ArithmeticExpression .

  op _+_ : ArithmeticExpression ArithmeticExpression ->
           ArithmeticExpression [ditto] .
```

```
    op _-_ : ArithmeticExpression ArithmeticExpression ->
             ArithmeticExpression [ditto] .
    op _*_ : ArithmeticExpression ArithmeticExpression ->
             ArithmeticExpression [ditto] .
    op _/_ : ArithmeticExpression ArithmeticExpression ->
             ArithmeticExpression .
  endfm
```

The following is an example to test ARITHEMETICEXPRESSION module:

```
reduce in ARITHMETICEXPRESSION : 'a + 'b .
result ArithmeticExpression: 'a + 'b
```

In the following, we define the Boolean Expression type for the input domain. It corresponds to the BooleanExpression and RelationOp definition in the semantics syntax.

```
fmod BOOLEANEXPRESSION is
  protecting ARITHMETICEXPRESSION .
  sort BooleanExpression .
  subsorts Bool < BooleanExpression .
  op _and_ :  BooleanExpression BooleanExpression -> BooleanExpression [ditto
  op _or_ :  BooleanExpression BooleanExpression -> BooleanExpression [ditto]
  op _xor_ :  BooleanExpression BooleanExpression -> BooleanExpression [ditto
  op _<=_ : ArithmeticExpression ArithmeticExpression -> BooleanExpression [d
  op _<_ : ArithmeticExpression ArithmeticExpression -> BooleanExpression [di
endfm
```

The followings test BOOLEANEXPRESSION module:

```
reduce in BOOLEANEXPRESSION : 2 + 3 <= 4 + 5 .
result Bool: true

reduce in BOOLEANEXPRESSION : 2 + 2 < 2 + 3  xor 1 < 3 .
result Bool: false
```

We define Expression type in the following:

```
fmod EXPRESSION is
  protecting ARITHMETICEXPRESSION .
  protecting BOOLEANEXPRESSION .
  sort Expression .
  subsorts BooleanExpression < Expression .
  subsorts ArithmeticExpression < Expression .
  op _==_ : Expression Expression -> BooleanExpression [comm] .
endfm
```

We define Statement in the following:

```
fmod STATEMENT is
   protecting EXPRESSION .
   sorts Statement .
   op _;_ : Statement Statement -> Statement [assoc prec 50] .
   op while_do_od : BooleanExpression Statement -> Statement
     [format (nir! o r! o++ --nir! o)] .
   op _:=_ : Variable Expression -> Statement
     [format (ni d d d)] .
   op if_then_else_fi : BooleanExpression Statement Statement -> Statement
```

```
      [format(nir! o r! o r! o nir! o)] .
    op return_ : Expression -> Statement
      [format (nir! o d)] .
  endfm
```

The followings test STATEMENT module:

```
reduce in STATEMENT : while 'a + 'b + 'c < 3 + 5 do 'x := 'x + 2 od .
result Statement:
while 'a + 'b + 'c < 8 do
      'x := 'x + 2
od

reduce in STATEMENT : return ('x + 'a < 'y + 'z) .
result Statement: return ('a + 'x < 'y + 'z)
```

The following model describes the defining of the `StateRep` type that we defined in section 4. It could be `FixedPoint` μ, `Gamma` γ or `StateTransition` δ function.

```
  fmod STATEREP is
    protecting EXPRESSION .
    sort StateRep .
    subsort Variable < StateRep .
    subsort Expression < StateRep .

    op _._ : StateRep StateRep -> StateRep .
    op Gamma : BooleanExpression StateRep StateRep -> StateRep .
    op FixedPoint : BooleanExpression StateRep -> StateRep .
```

```
    op StateTransition : StateRep StateRep -> StateRep .

  endfm
```

The following is part of GEN_FUNCTION module. It translates statements into functions which have the `stateRep` type. So for any input statement given from the user, this module can substitute the input statements by using `stateRep` function. For example, if we have x:=a assignment statement, by calling this module, we can get StateTransition('x, 'a), which has `StateRep` type.

```
fmod GEN-FUNCTION is
   protecting STATEMENT .
   protecting STATEREP .
   var 'x : Variable .
   var 'a : Expression .
   var b1 : BooleanExpression .
   var s1 : Statement .
   var s2 : Statement .
   op geneqns_ : Statement -> StateRep .
   eq geneqns('x := 'a) = StateTransition('x , 'a) .
   eq geneqns(if b1 then s1 else s2 fi) = Gamma(b1, geneqns(s1), geneqns(s2))
   eq geneqns(while b1 do s1 od) = FixedPoint(b1 , geneqns(s1)) .
   eq geneqns(s1 ; return 'x) = StateTransition('x, geneqns(s1)) .
   eq geneqns(s1 ; s2 ) = geneqns(s1) . geneqns(s2) .
endfm
```

The followings test the GEN-FUNCTION module:

```
reduce in GEN-FUNCTION : geneqns( if 2  + 'f < 1 + 'g
                                then 'x := 2 < 3 else 'x := 4 < 3 fi) .

result StateRep: Gamma('f + 2 < 'g + 1, StateTransition('x, true),
    StateTransition('x, false))

reduce in GEN-FUNCTION : geneqns ('x := 'a ; while 't < 'r do 'x := 'v ;
                                  'y := 'd od ; return 'x ) .

result StateRep:  StateTransition('x, StateTransition('x, 'a) . FixedPoint('t <
    'r, StateTransition('x, 'v) . StateTransition('y, 'd)))
```

The following is OUTPUTEXP module. It defines the `OutputExpr` data type for the output expression. It corresponds to the `OutputExpr` type defined in the semantics syntax.

```
 fmod OUTPUTEXPR is
    protecting STATEREP .
    sort OutputExpr .
    subsort StateRep < OutputExpr .
    op Empty : -> OutputExpr .
    op soleqns_ : StateRep -> OutputExpr .
    op SolRecursive_ : StateRep -> OutputExpr .
    op Inv : StateRep StateRep -> OutputExpr .
    op _._ : OutputExpr OutputExpr -> OutputExpr .
  endfm
```

The following is SOLVE_FUNCTION module. It solves `StateRep` type Function to get output expression which has `OutputExpr` type. for some case, which is very hard to get an explicit final output, we will use function `solrecursive` or function Inv, which has the `StateRep` type, to represent it.

```
fmod SOLVE-FUNCTION is
   protecting OUTPUTEXPR .
   protecting GEN-FUNCTION .
   var 'x : Variable .
   var 'y : Variable .
   var s1 : StateRep .
   var s2 : StateRep .
   var s3 : StateRep .
   var b1 : BooleanExpression .
   var b2 : BooleanExpression .
   eq soleqns(StateTransition('x, StateTransition('x, s1))) = s1 .
   eq soleqns(StateTransition('x, StateTransition('y, s1))) = Empty .
   eq soleqns(StateTransition('x, s1 . s2 )) = soleqns(StateTransition('x, s1))
                                               soleqns(StateTransition('x, s2)
   eq soleqns(FixedPoint(b1, Gamma(b2, s1, s2))) = Inv(s1, s2) .
   eq soleqns(FixedPoint(b1, s1 )) = SolRecursive(s1) .
endfm
```

Bibliography

[1] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. Dover Publications, New York 1965.

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

[3] J. Blieberger. Data Flow Frameworks for Worst-Case Execution Time Analysis. *Real Time System Journal*, 2001.

[4] A. Blikle, An anlysis 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] R. S. Boyer, B. Elspas and K. N. Levitt. SELECT-A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices, 10(6)*, pages 234–245, June 1975.

[6] J. Carette, R. Janicki and Y. Zhai, Program Verification by Calculating Relations, *Conf. of Applied Simulation and Modelling*, 2006.

[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] Lori A. Clarke, A System to generate test data and symbolically execute programs, *IEEE Transaction on Software Engineering*, 1976, September, pp. 215-222

[9] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze, Using symbolic execution for verifying safety-critical systems, *Communications of the ACM*, 2001, 142-151.

[10] A. Coen-Porisini, F. D. Paoli, C. Ghezzi, and D. Mandrioli. Software Specialization Via Symbolic Execution. *IEEE Transactions on software Engineering*,17(9):884-899, Sept. 1991.

[11] 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.

[12] T. Fahringer and B. Scholz. A Unified Symbolic Evaluation Framework for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1106-1126, 2000.

[13] T. Fahringer and B. Scholz. Advanced Symbolic Analysis for Compilers. *Lecture Notes in Computer Science (LNCS)*, Vol. 2628, Springer Press, 2003.

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

[15] W. E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Trans. on Software Engineering SE-3, 4*, pages 266–278, July 1977.

[16] R. Janicki, Analysis of coroutines by means of vector coroutines, *Fundamenta informaticae*, 2, 2 (1979), 289-316

[17] A. Kaldewaij, *Programming. The Derivation of Algorithms*, Prentice-Hall 1990.

[18] R. A. Kemmerer, S. T. Eckmann. UNISEX: A UNix-based symbolic Executor for Pascal. *Softw. Pratt. Exper. 15,5*, pages 439–457, May 1985.

[19] J. C. King. Symbolic Execution and program testing. *Communications of the ACM*, pages 385–394, July 1976.

[20] 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).

[21] D. Kozen, A completeness theorem for Kleene algebras and the algebra of regular events, *Information and Computation* 110 (1994), 366-390.

[22] *Loop Invariants*, http://academic.evergreen.edu/curricular/dsa01/loops.html

[23] B. Luca, S. Andrei, H. Anderson and S. Khoo, Program Transformation by Solving Recurrences, *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2006, 121 - 129

[24] B. D.Martino. Algorithmic Concept Recognition Support for Automatic Parallelization: A case Study for Loop Optimization and Parallelization. *Journal of Information Science and Engineering, Special Issue on Compiler Techniques for High-Performance Computing*, March 1998

[25] *The Maude System*, http://maude.cs.uiuc.edu

[26] A. Mazurkiewicz, Proving algorithms by tail function, *Information and Control*, 18 (1971) 793-798.

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

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

[29] 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.

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

[31] J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The M.I.T. Series in Computer Science, M.I.T. Press, Cambridge MA, 1977.

[32] N. Tawbi. Estimation of nested loop execution time by integer arithmetic in convex polyhedra. In *Proc. of the 1994 International Parallel Processing Symposium*, April 1994.

[33] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

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