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

© Copyright by Yun Zhai, May 2006
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.
I would like to express my sincere appreciations to my supervisors, Dr. Jacques Carette and Dr. Ryszard Janicki, for their inspirations, invaluable and patient guid-
ances, 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
### CONTENTS

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: Binomials Coefficients ................................. 50  
  6.1.5 Example 4: Bessel .................................................. 52  
  6.1.6 Example 5: A Recursive Call Function ............................ 54
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 \texttt{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 \texttt{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 \texttt{chebyshev} ............................. 50  
6.5 Recursive and initial functions for \texttt{Binomial Coefficients} ................. 51  
6.6 Recursive and initial functions for \texttt{Bessel} ................................. 53
<table>
<thead>
<tr>
<th></th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7</td>
<td>Recursive and initial functions for chebyshev1</td>
<td>55</td>
</tr>
<tr>
<td>6.8</td>
<td>Translation of program into the input of generating invariants [29] for GCD</td>
<td>57</td>
</tr>
<tr>
<td>6.9</td>
<td>Translation of program into the input of generating invariants [29] for LCM</td>
<td>59</td>
</tr>
</tbody>
</table>
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.
1. Introduction

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. Introduction

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 compiler 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. Introduction

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]:

```maple
procedure factorial(n)
    i:=1;
    fac:=1;
    while i < n do
        begin
            i:=i+1;
            fac:=fac*i;
        end;
    fac;
end proc;
```

5
2. Intuition and Motivation

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 $\text{fac}$. Define $D = \mathbb{Z} \times \mathbb{Z}$ and denote the elements of $D$ as $(i, \text{fac})$. Each instruction can be modeled by a function $F_i : D \rightarrow D$, $i = 1, 2, 4, 5$, in the following manner:

- "$i:=1$" corresponds to $F_1(i, \text{fac}) = (1, \text{fac})$,
- "$\text{fac}:=1$" corresponds to $F_2(i, \text{fac}) = (i, 1)$,
- "$i:=i+1$" corresponds to $F_4(i, \text{fac}) = (i + 1, \text{fac})$,
- and "$\text{fac}:=\text{fac} \cdot i$" to $F_5(i, \text{fac}) = (i, \text{fac} \cdot i)$.

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

- "$i<1$" corresponds to $I_3(i, \text{fac})$, and
- "$i \geq 1$" corresponds to $\bar{I}_3(i, \text{fac})$, where

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

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

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 $S$, $S_1$, $S_2$, 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.
2. Intuition and Motivation

<table>
<thead>
<tr>
<th>Statement</th>
<th>Modeled By</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1;S2</td>
<td>$R_1 \circ R_2$</td>
</tr>
<tr>
<td>if T then S1 else S2</td>
<td>$(I_T \circ R_1) \cup (I_T \circ R_2)$</td>
</tr>
<tr>
<td>while T do S</td>
<td>$(I_T \circ R)^* I_T$</td>
</tr>
</tbody>
</table>

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:

$$F = F_1 \circ F_2 \circ (I_3 \circ F_4 \circ F_5)^* \circ I_3$$

Calculating “$\circ$” is easy in this case, but calculating “$^*$” is not. Let $G = I_3 \circ F_4 \circ F_5$.

Then we have:

$$G(i, fac) = (I_3 \circ F_4 \circ F_5)(i, fac)$$

$$= F_5(F_4(I_3(i, fac)))$$

$$= \begin{cases} 
(i + 1, fac \cdot (i + 1)) & \text{if } i < n \\
\bot & \text{if } i \geq n 
\end{cases}$$

Similarly:

$$G^2(i, fac) = G(G(i, fac))$$

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

Hence:

$$G^k(i, fac) = \begin{cases} 
(i + k, fac \cdot (i + 1) \cdot (i + 2) \ldots (i + k)) & \text{if } i + k - 1 < n \\
\bot & \text{if } i + k - 1 \geq n 
\end{cases}$$
2. Intuition and Motivation

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

$$(i, \text{fac}) G^k(i', \text{fac'}) \iff i' = i + k \land \text{fac'} = \text{fac} \ast (i + 1) \ast \ldots \ast (i + k) \land i + k - 1 < n.$$ 

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

$$(i, \text{fac}) G^* (i', \text{fac'})$$

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

$$\iff \exists k, 0 \leq k < n - i + 1 \land i' = i + k \land \text{fac'} = \text{fac} \ast (i + 1) \ast \ldots \ast (i + k).$$

We may now make some simplification:

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

This means:

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

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

$$\iff \exists k, 0 \leq k < n - i + 1 \land i' = k + 1 \land \text{fac'} = (k + 1)!$$

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

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

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

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

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

and

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

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

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

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

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:

$$(\exists k, 0 \leq k < n \land i = k + 1 \land \text{fac} = (k + 1)!) \land (i \geq n \land i = i' \land \text{fac} = \text{fac'})$$

is reduced to:
\[ i' = k + 1 = n - 1 + 1 = n \land fac' = n! \]

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 \texttt{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 \textit{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 \texttt{factorial} presented in chapter 2. Every time when the loop is executed, the value of \(i\) is incremented by one and the value of \(\text{fac}\) is incremented \(i\) times. We may express this change in a form of recurrences. For this example, the recurrence relation is following

\[
i(t + 1) = i(t) + 1, \quad \text{fac}(t + 1) = \text{fac}(t) \cdot i(t + 1),
\]

where \(i(t + 1)\) and \(\text{fac}(t + 1)\), for \(t \geq 0\) are the values of variable \(i\) and \(\text{fac}\) at the end of iteration \(t + 1\). In this sense, \(t\) represents \textit{time}, which is the important new concept in this representation. Because \textit{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 \(\text{fac}\) is 1, so we initialize the recurrence by \(i(0) = 1\) and \(\text{fac}(0) = 1\). Table 3.1 describes the meaning of our loop.
Loops and Recurrence Equations

Input program

\[
\text{factorial} := \text{proc}(n) \\
\quad \text{local } i, \text{ fac}; \\
\quad i := 1; \quad i(0) := 1 \\
\quad \text{fac} := 1; \quad \text{fac}(0) := 1 \\
\quad \text{while } i < n \text{ do} \\
\quad \quad i := i+1; \quad i(t+1) := i(t) + 1 \\
\quad \quad \text{fac} := \text{fac} \times i; \quad \text{fac}(t+1) := \text{fac}(t) \times i(t+1) \\
\quad \text{end do}; \\
\text{end proc:}
\]

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 \(\text{fac}(t)\), but \(\text{fac}(t + 1)\) 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 \(\text{fac}\), we get \(\text{fac}(t + 1) = \text{fac}(t) \cdot (t + 2)\), which is also solvable. Putting the solution together, we get

\[
\begin{align*}
\text{(3.1)} & \quad i(t) = t + 1 \\
\text{(3.2)} & \quad \text{fac}(t) = (t + 1)!
\end{align*}
\]

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
4. Theoretical Background

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:

\[
\begin{align*}
\text{Type} & : \text{Int} | \text{Bool} | \text{Symbol} \\
\text{Variable} & : \text{Identifier} : \text{Type} \\
\text{ArithmeticOp} & : + | - | * | / \\
\text{ArithmeticExpr} & : \text{int} | \text{symbol} | \text{constant} | \text{var(ArithmeticExpr)} \\
& \quad | \text{ArithmeticExpr} \text{ArithmeticOp} \text{ArithmeticExpr}
\end{align*}
\]

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:
4. Theoretical Background

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:


  Op : BinaryOp | UnaryOp

4.1.2 Grammar

We will illustrate the abstract interpretation using the language described by the following simple grammar:
4. Theoretical Background

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:
\[
\delta : \text{StateRep} \rightarrow (\text{Identifier, Expr}) \rightarrow \text{StateRep}
\]
\[
\gamma : \text{BooleanExpr} \rightarrow \text{StateRep} \rightarrow \text{StateRep} \rightarrow \text{StateRep}
\]
\[
\mu : \text{BooleanExpr} \rightarrow \{\text{Identifier}\} \rightarrow \text{StateRep} \rightarrow \text{StateRep}
\]
\[
\eta : [\text{Variable, Int, Int, Int}] \rightarrow \{\text{Identifier}\} \rightarrow \text{StateRep} \rightarrow \text{StateRep}
\]
\[
[ ] : \text{Statement} \rightarrow (\text{StateRep} \rightarrow \text{StateRep})
\]
\[
[] : \text{Statement} \rightarrow (\text{StateRep} \rightarrow \text{StateRep})
\]
\[
\text{evalExpr} : \text{Expr} \rightarrow \text{StateRep} \rightarrow \text{OutputExpr}
\]
\[
\text{getRecVar} : \text{Statement} \rightarrow \{\text{Identifier}\}
\]
\[
\text{Reduce} : \text{StateRep} \rightarrow \text{OutputExpr}
\]

The constructor \(\delta, \gamma, \mu\) and \(\eta\) describe the updating \text{StateRep} to a new \text{StateRep}. The function \([\ ]\) and \([],\[]\) describe the mapping from one \text{StateRep} to a new \text{StateRep} under the state commands. The function \text{evalExpr} evaluates an expression with the \text{StateRep} to the \text{outputExpr}. Function \text{Reduce} can evaluate \(\delta, \gamma, \mu\) and \(\eta\) when possible.

Let \text{id} and \text{expr} be the type of \text{Identifier} and \text{Expression} individually, then \((\text{id, expr})\) can be denoted by \text{id=expr} in \(\delta\).
4. Theoretical Background

4.1.5 Semantics Equations

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

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

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

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

(WH) \[[\text{while cond do stmt end do}]\] = \(\lambda s \in \text{StateRep}. \eta(\text{cond}, \text{getRecVar(\text{stmt})), \text{getRecVar(\text{stmt}])}\)

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

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

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

(WS) \[[\text{stmt}_1 ; \text{stmt}_2]\](s) = [\text{stmt}_2][[\text{stmt}_1](s)]\)

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

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-branckes. 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.
4. Theoretical Background

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 $\delta, \gamma, \mu, \eta$ 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:

\[
\begin{align*}
\text{evalExpr : Expr} & \rightarrow \text{StateRep} \rightarrow \text{OutputExpr} \\
\text{(E1)} & \quad \text{evalExpr(constant, s)} = \text{constant} \\
\text{(E2)} & \quad \text{evalExpr(var, s)} =
\end{align*}
\]
4. Theoretical Background

\[
\begin{cases}
\text{var,} & \text{if } s=\bot \\
\text{expr,} & \text{if } s=\delta(s', \text{var}=\text{expr}) \\
\text{evalExpr(var,s'),} & \text{if } s=\delta(s', \text{var}=\text{expr}') \land \text{var}' \neq \text{var} \\
\text{evalExpr(var, s_1)} & \text{if } s=\gamma(\text{cond}, s_1, s_2) \land \text{cond} \\
\text{evalExpr(var, s_2)} & \text{if } s=\gamma(\text{cond}, s_1, s_2) \land \neg(\text{cond}) \\
\text{var} & \text{if } (s=\mu(\_, \text{getRecVar(stmt)}, \_)) \\
& \lor \ s=\eta(\_, \text{getRecVar(stmt)}, \_)) \\
& \land \ \text{var} \notin \text{getRecVar(stmt)} \\
\text{var(t+1)} & \text{if } (s=\mu(\_, \text{getRecVar(stmt)}, \_)) \\
& \lor \ s=\eta(\_, \text{getRecVar(stmt)}, \_)) \\
& \land \ \text{var} \in \text{getRecVar(stmt)} \\
\end{cases}
\]

where \( t \) is under the loop stopping condition, the loop iteration times.

(E3) \[ \text{evalExpr(expr}_1 \text{ op expr}_2, s) = \text{evalExpr(expr}_1, s) \text{ op evalExpr(expr}_2, s) \]

Where \( \text{expr}_1, \text{expr}_2 \in \text{Expr} \) and \( \text{stmt} \in \text{Statement} \)

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

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

Rules \((E_1) - (E_3)\) describe the symbolic evaluation of expression. Rule \((E_1)\) de-
4. Theoretical Background

<table>
<thead>
<tr>
<th>$R$</th>
<th>$&lt;$</th>
<th>$\leq$</th>
<th>$&gt;$</th>
<th>$\geq$</th>
<th>$=$</th>
<th>$\neq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg R$</td>
<td>$\geq$</td>
<td>$&gt;$</td>
<td>$\leq$</td>
<td>$&lt;$</td>
<td>$\neq$</td>
<td>$=$</td>
</tr>
</tbody>
</table>

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 $\text{evalExpr}$ 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 $\gamma$ to evaluate the conditional statements. Constructor $\gamma$ is defined as:

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

$$\gamma : \text{BooleanExpr} \rightarrow \text{StateRep} \rightarrow \text{StateRep} \rightarrow \text{StateRep}$$

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

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

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

where $\text{expr}_1, \text{expr}_2 \in \text{Expr}, \text{op} \in \text{Op}$
The symbolic semantics rules for symbolical evaluation if-statements are defined as:

(I1) $\llbracket \text{if } \text{cond} \text{ then } \text{stmt} \text{ end if} \rrbracket = \lambda s \in \text{StateRep}. \gamma(\text{cond}, \llbracket \text{stmt} \rrbracket(s), s)$

(I2) $\llbracket \text{if } \text{cond} \text{ then } \text{stmt}_1 \text{ else } \text{stmt}_2 \text{ end if} \rrbracket = \lambda s \in \text{StateRep}. \gamma(\text{cond}, \llbracket \text{stmt}_1 \rrbracket(s), \llbracket \text{stmt}_2 \rrbracket(s))$

Rule (I1) is used to evaluate if-statements without else branches. In fact, the constructor $\gamma$ is equivalent to the "piecewise" function in Maple. Before evaluating the branches, the condition $\text{cond}$ of the if-statement is evaluated. If the condition $\text{cond}$ is true, then goes to evaluate statements $\text{stmt}$ 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 $\text{cond}$ is true, then evaluate statements $\text{stmt}_1$ under StateRep $s$, otherwise evaluate statements $\text{stmt}_2$ under StateRep $s$.

**While Loop Statements**

Assume the following pattern for the while loop: while $\text{cond}$ do $\text{stmt}$.

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

$\llbracket \text{[ } \text{ ]} \rrbracket : \text{Statement} \rightarrow (\text{StateRep} \rightarrow \text{StateRep})$

$\llbracket \text{[ } \text{ ]} \rrbracket : \text{StateRep} \rightarrow (\text{StateRep} \rightarrow \text{StateRep})$

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

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

(WH) $\llbracket \text{while } \text{cond} \text{ do } \text{stmt} \text{ end do} \rrbracket = \lambda s \in \text{StateRep}. \mu(\text{cond}, \text{getRecVar}(\text{stmt}), \llbracket \text{stmt} \rrbracket(s))$
4. Theoretical Background

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

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

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

\( \gamma(\text{cond}, [\text{stmt} \_1] (s), [\text{stmt} \_2] (s)) \)

(WS) \( [\text{stmt} \_1;\text{stmt} \_2] (s) = [\text{stmt} \_2] ([\text{stmt} \_1] (s)) \)

getRecVar : Statement \( \rightarrow \) \{Identifier\}

(G1) \( \text{getRecVar(var:=expr)} = \{\text{var}\} \)

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

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

\( \text{getRecVar(stmt} \_1) \cup \text{getRecVar(stmt} \_2) \)

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

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

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

\[ \text{RecVar} = \text{getRecVar(stmt)} \]

evalRecExpr : \( \text{Expr} \rightarrow \text{StateRep} \rightarrow \text{OutputExpr} \)

(ER1) \( \text{evalRecExpr(constant}, s) = \text{constant} \)

(ER2) \( \text{evalRecExpr(var}, s) = \)
4. Theoretical Background

\[
\begin{cases}
\text{var}(t+1), & \text{if } s = \delta(s', \text{var}(t+1) = \text{expr}) \\
\text{evalRecExpr}(\text{var}, s'), & \text{if } s = \delta(s', \text{var}'(t+1) = \text{expr}) \\
\text{var}(t), & \text{if } s = \delta(\bot, \text{var}'(t+1) = \text{expr}) \land \text{var} \in \text{RecVar} \\
\text{evalExpr}(\text{var}, s), & \text{otherwise if } \text{var} \notin \text{RecVar}
\end{cases}
\]

\(\text{(ER3)}\) \quad \text{evalRecExpr}(\text{expr}, \text{op} \text{expr}_2, s) \\
= \text{evalRecExpr}(\text{expr}_1, s) \text{ op } \text{evalRecExpr}(\text{expr}_2, s)

\[
\text{ evalConExpr : Expr } \to \text{ StateRep } \to \text{ OutputExpr }
\]

\(\text{(EC1)}\) \quad \text{evalConExpr}(\text{constant}, s) = \text{constant}

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

\(\text{(EC3)}\) \quad \text{evalConExpr}(\text{expr}, \text{op} \text{expr}_2, s) \\
= \text{evalConExpr}(\text{expr}_1, s) \text{ op } \text{evalConExpr}(\text{expr}_2, s)

Loop Stop Condition

we can decide the loop stop condition on the basis of the condition \textit{cond}. In general, this condition reads

\[
\min_{t \geq 0} \{ t \mid -\text{cond} \}
\]

where \textit{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.
4. Theoretical Background

- **Case 1** \( \text{cond} = v_i \ R \ c \), 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** \( \text{cond} = v_i \ R \ \phi(v_j) \), where \( v_1, \ldots, v_m \) are the recurrence variables, and some of them occur in the expression \( \phi \), with \( R \) as before.

\[
\begin{align*}
z &= \min_{t \geq 0} \{ t \mid \neg(v_i \ R \ \phi(v_j)) \} \\
&= \min_{t \geq 0} \{ t \mid F_i(t) \ S \ \phi(F_j(t)) \}
\end{align*}
\]

where \( S = \neg R \).

- **Case 3** \( \text{cond} \) 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.

\[
[ ] : \text{Statement} \rightarrow (\text{StateRep} \rightarrow \text{StateRep})
\]

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

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

getRecVar : \text{Statement} \rightarrow \text{StateRep}

(G1) \text{getRecVar(var:=expr)=\{var(n)} \}
4. Theoretical Background

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

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

(EF) \( \text{evalFor}([i, s, e, \text{step}], s) = \left[ \frac{e - s + 1}{\text{step}} \right] \) if \( i \notin s_1 \)

We use \text{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. \([i, s, e, \text{step}]\) is the representing of the \textbf{for} loop control condition by using loop variable, initial value, ending value and step value to describe.

We can reduce \( \eta \) function into the following:

\[
\text{Reduce}(\eta([i, s, e, \text{step}], \text{getRecVar(\text{stmt}), } \llbracket \text{stmt} \rrbracket(s))) = \]

\[
\text{let } t = \text{evalFor}([i, s, e, \text{step}], s) \text{ in } (\llbracket \text{stmt} \rrbracket(s))^t
\]

4.1.6 Code Examples

Example 1

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

\[
l_1 : x := x_1;
\]

\[
l_2 : y := y_1;
\]

\[
l_3 : \text{if } y < 0 \text{ then } x := -2 \times y; \text{ else } x := 2 \times y \text{ end if}
\]
4. Theoretical Background

\[ s = \perp \]
\[ \lbrack x := x_1 \rbrack (s) = \delta(s, x = evalExpr(x_1, s)) \]
\[ = \delta(\perp, x = x_1) = s_1 \]

\[ \lbrack y := y_1 \rbrack (s_1) = \delta(s_1, y = evalExpr(y_1, s_1)) \]
\[ = \delta(\delta(\perp, x = x_1), y = y_1) = s_2 \]

\[ \lbrack \text{if ...} \rbrack (s_2) = \gamma(y < 0, \lbrack x := -2 * y \rbrack (s_2), \lbrack x := 2 * y \rbrack (s_2)) \]
\[ = \gamma(y < 0, \delta(\delta(\perp, x = x_1), y = y_1), x = -2 * y_1) \]
\[ , \delta(\delta(\perp, x = x_1), y = y_1, x = 2 * y_1) \]

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

Example 2: Factorial

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

\[ l_1 : i := 1; \]
\[ l_2 : fac := 1; \]
\[ l_3 : \text{while } i < n \text{ do } i := i + 1; fac := fac * i \text{ end do} \]

\[ s = \perp \]
\[ \lbrack i := 1 \rbrack (s) = \delta(s, i = evalExpr(1, s)) \]
\[ = \delta(s, i = 1) = s_1 \]

\[ \lbrack fac := 1 \rbrack (s_1) = \delta(s_1, fac = evalExpr(1, s_1)) \]
\[ = \delta(\delta(\perp, i = 1), fac = 1) = s_2 \]
4. Theoretical Background

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

Reduce s_3 to the following:

Reduce(s_3) \\
= (\delta(\delta(\delta(\bot, i = 1), \text{fac} = 1), i(t + 1) = i(t) + 1), \\
\text{fac}(t + 1) = \text{fac}(t) \times i(t + 1)))^{\min_{t \geq 0}(t | i(t) \geq n)}

Example 3: Chebyshev Polynomials

Chebyshev polynomial is an example implementing \textbf{for} loop statement to get symbolic interpretation of these statements.

\begin{align*}
  l_1 & : u_0 := 1; \\
  l_2 & : u_1 := x; \\
  l_3 & : \text{for } i \text{ from 2 to } n \text{ do} \\
  & \quad v = u_1; \\
  & \quad u_1 = -u_0 + 2 \times x \times u_1; \\
  & \quad u_0 = v; \\
  & \text{end do;}
\end{align*}

s = \bot
4. Theoretical Background

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

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 (+, *, <, ≤) 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. \texttt{var:=expr}
2. \texttt{if cond then stmt\_1 else stmt\_2}
3. \texttt{while cond do stmt end do}
4. \texttt{for i from s to e by step do stmt end do}
5. \texttt{Recursive Function Calls}
5. System Analysis

### Table 5.1: Rules for program transformation

<table>
<thead>
<tr>
<th>Program Statements</th>
<th>Relations Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>var:=expr</td>
<td>StateTransition</td>
</tr>
<tr>
<td>if-then-else</td>
<td>Piecewise</td>
</tr>
<tr>
<td>while C do</td>
<td>Fixedpoint([C],...)</td>
</tr>
<tr>
<td>for-from-to-by-do</td>
<td>Fixedpoint(For(),...)</td>
</tr>
<tr>
<td>Recursion</td>
<td>RecursionCall</td>
</tr>
</tbody>
</table>

#### 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 cannot 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
5. System Analysis

Figure 5.1: Symbolic execution system
### Table 5.2: Translation of factorial into the set of appropriate relations

<table>
<thead>
<tr>
<th>Input Program</th>
<th>Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial:=proc(n)</td>
<td></td>
</tr>
<tr>
<td>i:=1</td>
<td>F1:StateTransition(i,1)</td>
</tr>
<tr>
<td>fac:=1</td>
<td>F2:StateTransition(fac,1)</td>
</tr>
<tr>
<td>while i &lt; n do</td>
<td>F3:FixedPoint([i &lt; n], [StateTransition(i,i+1), StateTransition(fac,fac*i)])</td>
</tr>
<tr>
<td>end do</td>
<td></td>
</tr>
<tr>
<td>fac</td>
<td>F4:fac</td>
</tr>
<tr>
<td>end proc;</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Generated Relation(GenRelation)</th>
<th>Semantic Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateTransition(var, expr)</td>
<td>$\delta$(var, expr)</td>
</tr>
<tr>
<td>Piecewise(cond, transition1, transition2)</td>
<td>$\gamma$(cond,[stmt$_1$], [stmt$_2$])</td>
</tr>
<tr>
<td>Fixedpoint([cond], transition1)</td>
<td>$\mu$(cond,{var}, [stmt])</td>
</tr>
<tr>
<td>Fixedpoint(For[i, s, e, step], transition1)</td>
<td>$\eta$([i, s, e, step], {var}, [stmt])</td>
</tr>
</tbody>
</table>

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

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 \( \text{ToInert(eval(factorial))} \) inside the Relation Generator, we obtain the following:

\[
\begin{align*}
&\_\text{Inert\_PROC}(\_\text{Inert\_PARAMSEQ}(\_\text{Inert\_NAME}("n")), \\
&\_\text{Inert\_LOCALSEQ}(\_\text{Inert\_NAME}("i"), \_\text{Inert\_NAME}("fac")), \\
&\_\text{Inert\_OPTIONSEQ}(), \_\text{Inert\_EXPSEQ}(), \\
&\_\text{Inert\_STATSEQ}(\_\text{Inert\_ASSIGN}(\_\text{Inert\_LOCAL}(1), \\
&\_\text{Inert\_INTPOS}(1)), \_\text{Inert\_ASSIGN}(\_\text{Inert\_LOCAL}(2), \_\text{Inert\_INTPOS}(1)), \\
&\_\text{Inert\_FORFROM}(\_\text{Inert\_EXPSEQ}(), \_\text{Inert\_INTPOS}(1), \_\text{Inert\_INTPOS}(1), \\
&\_\text{Inert\_EXPSEQ}(), \_\text{Inert\_INEQUAT}(\_\text{Inert\_LOCAL}(1), \_\text{Inert\_PARAM}(1)), \\
&\_\text{Inert\_STATSEQ}(\_\text{Inert\_ASSIGN}(\_\text{Inert\_LOCAL}(1), \\
&\_\text{Inert\_SUM}(\_\text{Inert\_LOCAL}(1), \_\text{Inert\_INTPOS}(1))), \\
&\_\text{Inert\_ASSIGN}(\_\text{Inert\_LOCAL}(2), \\
&\_\text{Inert\_PROD}(\_\text{Inert\_LOCAL}(2), \_\text{Inert\_LOCAL}(1)))), \_\text{Inert\_LOCAL}(2)), \\
&\_\text{Inert\_DESCRIPTIONSEQ}(), \_\text{Inert\_GLOBALSEQ}(), \_\text{Inert\_LEXICALSEQ}(), \\
&\_\text{Inert\_EOP}(\_\text{Inert\_EXPSEQ}())
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Inert Function</th>
<th>Generated Relation</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Inert_ASSIGN</td>
<td>StateTransition</td>
<td>T1</td>
</tr>
<tr>
<td>_Inert_IF</td>
<td>Piecewise</td>
<td>T2</td>
</tr>
<tr>
<td>_Inert_FORFROM(_Inert_EXPSEQ...)</td>
<td>Fixedpoint</td>
<td>T3</td>
</tr>
<tr>
<td>_Inert_FORFROM(_Inert_LOCAL...)</td>
<td>Fixedpoint(For(), ...)</td>
<td>T4</td>
</tr>
</tbody>
</table>

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.
5. System Analysis

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.
A Sequence of Relations

Type of Relation

Piecewise inside FixedPoint

Invariant Solver Model

By using by E. R. Carbonell & D. Kapur’s Automatic Generation of Invariants procedure

Implicit Output Expression (like invariants)

FixedPoint

Generating recurrence functions, initial functions and stop condition

Explicit Output Expression (if recursive functions are solvable)

Recursion Call

Generating Recurrence functions and initial functions

Without FixedPoint & RecursionCall

Solving the sequential relations

Input Program

Relation Generator Model

Figure 5.2: Symbolic execution system in detail
5. System Analysis

### Relations

<table>
<thead>
<tr>
<th>Relations</th>
<th>Recurrence and Initial Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateTransition(i,1)</td>
<td>Initial Condition: $i(0) = 1$</td>
</tr>
<tr>
<td>StateTransition(fac,1)</td>
<td>Initial Condition: $fac(0) = 1$</td>
</tr>
<tr>
<td>FixedPoint(i&lt; n,)</td>
<td>Loop Termination: $\text{lt} = \min{t \geq 0 \mid i(t) \geq n}$</td>
</tr>
<tr>
<td>StateTransition(i,i+1),</td>
<td>Recurrence: $i(t+1) = i(t) + 1$</td>
</tr>
<tr>
<td>StateTransition(fac,fac*i),</td>
<td>Recurrence: $\text{fac}(t+1) = \text{fac}(t) \cdot i(t+1)$</td>
</tr>
<tr>
<td>return fac</td>
<td>$\text{fac}(\text{lt})$</td>
</tr>
</tbody>
</table>

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. System Analysis

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 \( \text{Cond} \) in \( \text{FixedPoint}([\text{Cond}], \ldots) \). In general, this condition reads

\[ \min_{t \geq 0} \{ \text{eval} [\text{Cond}] \} \]

where \( \text{Cond} \) 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:

\[ \text{for } i \text{ from } s \text{ to } e \text{ by step } \text{ do } S. \]
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 $[\text{StateTransition}(v_1, expr_1), \text{StateTransition}(v_2, expr_2), \ldots, \text{StateTransition}(v_m, expr_m)]$ precedes the loop, where $v_1, v_2, \ldots, v_m$ are recursive variables, then the generated initial functions are in the followings:

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

where $expr[var:=expr1]$ means the expression $expr$ with variable $var$ in $expr$ replaced with $expr1$. It follows the chain of $\delta$ 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

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

- **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:

\[
\begin{align*}
    u(0) &= expr_1 \\
    u(t) &= f(u(t - 1))
\end{align*}
\]

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
5. System Analysis

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.

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

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 = \ldots] \]
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 \emph{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 \emph{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 \emph{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
6. Examples of Using Symbolic Execution Tool

<table>
<thead>
<tr>
<th>( \sum_{i=0}^{n} \frac{1}{i!} )</th>
<th>Recurrence and Initial Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s, w) := (1, 1);)</td>
<td><strong>Initial Condition:</strong></td>
</tr>
<tr>
<td></td>
<td>(s(0) = 1, w(0) = 1, i(0) = 1)</td>
</tr>
<tr>
<td><strong>for i from 1 to n do</strong></td>
<td><strong>Number of Loop Iteration:</strong></td>
</tr>
<tr>
<td>(w := w/i;)</td>
<td>(z = \lfloor \frac{n-1+1}{1} \rfloor = n ) if (n \geq 1)</td>
</tr>
<tr>
<td>(s := s+w)</td>
<td><strong>Recurrence Equations:</strong></td>
</tr>
<tr>
<td></td>
<td>(w(t + 1) = w(t)/i(t))</td>
</tr>
<tr>
<td></td>
<td>(s(t + 1) = s(t) + w(t + 1))</td>
</tr>
<tr>
<td></td>
<td>(i(t + 1) = i(t) + 1)</td>
</tr>
<tr>
<td>(s)</td>
<td>(s(z))</td>
</tr>
</tbody>
</table>

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

following output for the above program:

\[
\frac{e \ast \Gamma(n + 1, 1)}{\Gamma(n + 1)}
\]

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

\[
\Gamma(a, z) = \Gamma(a) - z^{-a}/a \! 1 \! F \! 1(a, 1 + a, -z)
\]

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

**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:
6. Examples of Using Symbolic Execution Tool

<table>
<thead>
<tr>
<th>( \sum_{i=0}^{n} i! )</th>
<th>Recurrence and Initial Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s, w) := (1, 1);)</td>
<td>Initial Condition: s(0) = 1, w(0) = 1, i(0) = 1</td>
</tr>
<tr>
<td>for i from 1 to n do</td>
<td>Number of Loop Iteration: z = ([n-1+1]) = n if (n \geq 1)</td>
</tr>
<tr>
<td>w := w*i;</td>
<td>Recurrence Equations: w(t + 1) = w(t) * i(t)</td>
</tr>
<tr>
<td>s := s+w</td>
<td>s(t + 1) = s(t) + w(t + 1)</td>
</tr>
<tr>
<td>s</td>
<td>i(t + 1) = i(t) + 1</td>
</tr>
</tbody>
</table>

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.
6. Examples of Using Symbolic Execution Tool

<table>
<thead>
<tr>
<th>$\sum_{i=0}^{n} i!$</th>
<th>Recurrence and Initial Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s := n$ ;</td>
<td>$Initial Condition:$</td>
</tr>
<tr>
<td></td>
<td>$s(0) = n, i(0) = n - 1$</td>
</tr>
<tr>
<td>for $i$ from $n-1$ by $-1$ to $1$ do</td>
<td></td>
</tr>
<tr>
<td>$s := i*(1+s)$;</td>
<td>$Number of Loop Iteration:$</td>
</tr>
<tr>
<td></td>
<td>$z = n - 1$ if $n \geq 2$</td>
</tr>
<tr>
<td>$s+1$</td>
<td>$Recurrence Equations:$</td>
</tr>
<tr>
<td></td>
<td>$s(t + 1) = i(t) \ast (1 + s(t))$</td>
</tr>
<tr>
<td></td>
<td>$i(t + 1) = i(t) - 1$</td>
</tr>
<tr>
<td></td>
<td>$s(z) + 1$</td>
</tr>
</tbody>
</table>

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.

```maple
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;
```

6. Examples of Using Symbolic Execution Tool

<table>
<thead>
<tr>
<th>chebyshev Polynomials</th>
<th>Recurrence and Initial Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u0,u1) := (1,x);</td>
<td>Initial Condition: u0(0) = 1, u1(0) = x</td>
</tr>
</tbody>
</table>

for i from 2 to n do

<table>
<thead>
<tr>
<th>v := u1;</th>
<th>Number of Loop Iteration:</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1 := -u0 + 2<em>x</em>u1</td>
<td>z = ( \left[ \frac{n-2+1}{2} \right] = n - 1 ) if ( n \geq 2 )</td>
</tr>
<tr>
<td>u0 := v</td>
<td>Recurrence Equations:</td>
</tr>
<tr>
<td>u1</td>
<td>v(t + 1) = u1(t)</td>
</tr>
<tr>
<td></td>
<td>u1(t + 1) = -u0(t) + 2 \cdot x \cdot u1(t)</td>
</tr>
<tr>
<td></td>
<td>u0(t + 1) = v(t + 1)</td>
</tr>
</tbody>
</table>

Table 6.4: Recursive and initial functions for \textit{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 \textit{chebyshev}:

\[
\frac{(x - \sqrt{x^2 - 1})^{-n} + (x + \sqrt{x^2 - 1})^{-n}}{2}
\]

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:

\[
\text{binomial} := \text{proc}(u, k)
\]
6. Examples of Using Symbolic Execution Tool

Binomial Coefficients

<table>
<thead>
<tr>
<th>Recurrence and Initial Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>res:= 1</strong></td>
</tr>
<tr>
<td><strong>Initial Condition:</strong></td>
</tr>
<tr>
<td>res(0) = 1, i(0) = 1</td>
</tr>
</tbody>
</table>

for i from 1 to k do

<table>
<thead>
<tr>
<th>Number of Loop Iterations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>z = ([k-1+1]_i) = k if k ≥ 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recurrence Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>res(t + 1) = res(t) * (u - i(t) + 1)/i(t)</td>
</tr>
<tr>
<td>i(t + 1) = i(t) + 1</td>
</tr>
</tbody>
</table>

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 straightforward 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.
\[
\frac{(-1)^k \Gamma(-u + k)}{\Gamma(-u) \Gamma(k + 1)} \text{ is equivalent to binomial coefficient } \binom{u}{k}.
\]

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
\[
\frac{\Gamma(u + 1)}{\Gamma(u - k + 1)} = u * (u - 1) \ldots (u - k + 1)
\]
So, we need to show
\[
\frac{(-1)^k \Gamma(-u + k)}{\Gamma(-u)} = u * (u - 1) \ldots (u - k + 1)
\]

let \(-u = t - k + 1\), then:
\[
\frac{(-1)^k \Gamma(-u + k)}{\Gamma(-u)} = \frac{(-1)^k \Gamma(t + 1)}{\Gamma(t - k + 1)}
\]
\[
= (-1)^k * t * (t - 1) \cdots * (t - k + 1)
\]
\[
= (-1)^k * [-((u - k + 1) \cdots * (-u) * (-u)]
\]
\[
= (u - k + 1) \cdots * (u - 1) * u
\]
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.
<table>
<thead>
<tr>
<th>Bessel Function</th>
<th>Recurrence and Initial Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(res, u) = (0, 1)</td>
<td>Initial Condition:</td>
</tr>
<tr>
<td></td>
<td>res(0)=0, u(0)=1, i(0)=0</td>
</tr>
<tr>
<td>for i from 0 to m-1 do</td>
<td>Number of Loop Iterations:</td>
</tr>
<tr>
<td></td>
<td>z = \left\lfloor \frac{m-1+1}{1} \right\rfloor = m if m \geq 1</td>
</tr>
<tr>
<td>res := res + u</td>
<td>Recurrence Equations</td>
</tr>
<tr>
<td>u := -u * z^2/(4*(i+nu+1)(i+1))</td>
<td>res(t + 1) = res(t) + u(t)</td>
</tr>
<tr>
<td>i(t + 1) = i(t) + 1</td>
<td>u(t + 1) = -u(t) * z^2/</td>
</tr>
<tr>
<td></td>
<td>(4 * (i(t) + nu + 1)(i(t) + 1))</td>
</tr>
<tr>
<td>res</td>
<td>res(z)</td>
</tr>
</tbody>
</table>

Table 6.6: Recursive and initial functions for Bessel

```maple
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:

\[
\begin{align*}
\text{res}(t+1) &= \text{res}(t) + u(t) \\
u(t+1) &= -u(t) \cdot \frac{z^2}{(4 \cdot (i(t) + nu + 1) \cdot (i(t) + 1))} \\
i(t+1) &= i(t) + 1
\end{align*}
\] (6.3)

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

\[
\begin{align*}
i(t+1) &= i(t) + 1 \\
u(t+1) &= -u(t) \cdot \frac{z^2}{(4 \cdot (i(t) + nu + 1) \cdot (i(t) + 1))} \\
\text{res}(t+1) &= \text{res}(t) + u(t)
\end{align*}
\] (6.4)

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

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

where \( J_\nu \) 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;
```
6. Examples of Using Symbolic Execution Tool

<table>
<thead>
<tr>
<th>chebyshev1 Function</th>
<th>Recurrence and Initial Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>if n=0 then res:=1</td>
<td>Initial Condition: res(0)=0</td>
</tr>
<tr>
<td>elif n=1 then res:=x</td>
<td>Initial Condition: res(1)=x</td>
</tr>
<tr>
<td>else</td>
<td>Recurrence Equations</td>
</tr>
<tr>
<td>res := 2<em>x</em>chebyshev1(n-1)+chebyshev1(n-2)</td>
<td>res(t) = 2 * x * res(t - 1)+res(t - 2)</td>
</tr>
<tr>
<td>res</td>
<td>res(t)</td>
</tr>
</tbody>
</table>

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:

\[
\begin{align*}
-x - b * r + a * s1 &= 0, \\
y + a * q1 - b * p1 &= 0, \\
s1 * y + q1 * x - b &= 0, \\
x * p1 + y * r - a &= 0, \\
1 + q1 * r - s1 * p1 &= 0
\end{align*}
\]
6. Examples of Using Symbolic Execution Tool

<table>
<thead>
<tr>
<th>GCD program</th>
<th>Initial input of generating invariants [29]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=x; b:=y; p1:=1; q1:=0; r:=0; s1:=1</td>
<td>([a=x, b=y, p1=1, q1=0, r=0, s1=1]),</td>
</tr>
<tr>
<td>while (a &lt;&gt; b) do</td>
<td></td>
</tr>
<tr>
<td>if (a &gt; b) then</td>
<td></td>
</tr>
<tr>
<td>(a:=a-b; p1:=p1-q1; r:=r-s1;)</td>
<td>(a=a-b, p1=p1-q1, r=r-s1),</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>(b:=b-a; q1:=q1-p1; s1:=s1-r;)</td>
<td>(b=b-a, q1=q1-p1, s1=s1-r)</td>
</tr>
<tr>
<td>end if; end do</td>
<td></td>
</tr>
</tbody>
</table>

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. 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:

\[
\text{LCM} := \begin{array}{l}
\text{proc} (a, b) \\
\quad \text{local } x, y, u, v; \\
\quad x := a; y := b; u := b; v := 0; \\
\quad \text{while } x \neq y \text{ do} \\
\quad \quad \text{while } x > y \text{ do} \\
\quad \quad \quad x := x - y; v := u + v; \\
\quad \quad \quad \text{end do}; \\
\quad \quad \text{while } x < y \text{ do} \\
\quad \quad \quad y := y - x; u := u + v; \\
\quad \quad \quad \text{end do}; \\
\quad \text{end do}; \\
\quad \text{return } u + v;
\end{array}
\]

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
### 6. Examples of Using Symbolic Execution Tool

<table>
<thead>
<tr>
<th>LCM program</th>
<th>Initial input of generating invariants [29]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x:=a; y:=b; u:=b; v:=0;</code></td>
<td><code>[x=a, y=b, u=b, v=0],</code></td>
</tr>
<tr>
<td><code>while x &lt;&gt; y do</code></td>
<td><code>/</code></td>
</tr>
<tr>
<td><code>while x &gt; y do</code></td>
<td><code>/</code></td>
</tr>
<tr>
<td><code>- x:=x-y; v:=u+v;</code></td>
<td><code>x=x-y, v=u+v</code></td>
</tr>
<tr>
<td><code>end do</code></td>
<td><code>,</code></td>
</tr>
<tr>
<td><code>while x &lt; y do</code></td>
<td><code>/</code></td>
</tr>
<tr>
<td><code>- y:=y-x; u:=u+v;</code></td>
<td><code>y=y-x, u=u+v</code></td>
</tr>
<tr>
<td><code>end do</code></td>
<td><code>)</code></td>
</tr>
</tbody>
</table>

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 components: 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 algorithm comprises accurate modeling of assignment and input/out statements, branches, loops, 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 $\text{ArithmeticExpression}$ and $\text{ArithmeticOp}$ in the semantics syntax definition.

\[
\text{fmod ARITHMETICEXPRESSION is}
\]
\[
\text{protecting QID .}
\]
\[
\text{protecting INT .}
\]
\[
\text{sort Variable ArithmeticExpression .}
\]
\[
\text{subsorts Variable < ArithmeticExpression .}
\]
\[
\text{subsort Qid < Variable .}
\]
\[
\text{subsorts Int < ArithmeticExpression .}
\]

\[
\text{op } \_+\_ : \text{ArithmeticExpression ArithmeticExpression } \to \text{ArithmeticExpression [ditto] .}
\]
A. Specification by Maude

\[ \text{op } _-_- : \text{ArithmeticExpression ArithmeticExpression } \rightarrow \text{ArithmeticExpression [ditto].} \]

\[ \text{op } _*_- : \text{ArithmeticExpression ArithmeticExpression } \rightarrow \text{ArithmeticExpression [ditto].} \]

\[ \text{op } _/_- : \text{ArithmeticExpression ArithmeticExpression } \rightarrow \text{ArithmeticExpression.} \]

endfm

The following is an example to test ARITHMETICEXPRESSION module:

\[
\text{reduce in ARITHMETICEXPRESSION : 'a + 'b .} \\
\text{result ArithmeticExpression: 'a + 'b}
\]

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.

\[
\text{fmod BOOLEANEXPRESSION is} \\
\text{protecting ARITHMETICEXPRESSION .} \\
\text{sort BooleanExpression .} \\
\text{subsorts Bool < BooleanExpression .} \\
\text{op } _\text{and}_- : \text{BooleanExpression BooleanExpression } \rightarrow \text{BooleanExpression [ditto].} \\
\text{op } _\text{or}_- : \text{BooleanExpression BooleanExpression } \rightarrow \text{BooleanExpression [ditto].} \\
\text{op } _\text{xor}_- : \text{BooleanExpression BooleanExpression } \rightarrow \text{BooleanExpression [ditto].} \\
\text{op } _\leq_- : \text{ArithmeticExpression ArithmeticExpression } \rightarrow \text{BooleanExpression [ditto].} \\
\text{op } _<_- : \text{ArithmeticExpression ArithmeticExpression } \rightarrow \text{BooleanExpression [ditto].} \\
\text{endfm}
\]

The followings test BOOLEANEXPRESSION module:
reduce in BOOLEANEXPRESSION : \(2 + 3 \leq 4 + 5\).
result Bool: true

reduce in BOOLEANEXPRESSION : \(2 + 2 < 2 + 3 \text{ xor } 1 < 3\).
result Bool: false

We define Expression type in the following:

\[
\text{fmod EXPRESSION is}
\]
\[
\text{protection ARITHMETICEXPRESSION}.
\]
\[
\text{protection BOOLEANEXPRESSION}.
\]
\[
\text{sort Expression}.
\]
\[
\text{subsorts BooleanExpression < Expression}.
\]
\[
\text{subsorts ArithmeticExpression < Expression}.
\]
\[
\text{op }\_==\_ : \text{Expression Expression -> BooleanExpression}\ [\text{comm}].
\]
\[
\text{endfm}
\]

We define Statement in the following:

\[
\text{fmod STATEMENT is}
\]
\[
\text{protection EXPRESSION}.
\]
\[
\text{sorts Statement}.
\]
\[
\text{op }\_;\_ : \text{Statement Statement -> Statement}\ [\text{assoc prec 50}].
\]
\[
\text{op while do od : BooleanExpression Statement -> Statement}
\[
\text{[format (nir o r! o++ --nir! o)]}.
\]
\[
\text{op }\_:=\_ : \text{Variable Expression -> Statement}
\[
\text{[format (ni d d d)]}.
\]
\[
\text{op if then else fi : BooleanExpression Statement Statement -> Statement}
\]
A. Specification by Maude

[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 .

A. Specification by Maude

\[
\text{op StateTransition : StateRep StateRep -> StateRep .}
\]
\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.

\[
\text{fmod GEN-FUNCTION is}
\]
\[
\quad \text{protecting STATEMENT .}
\]
\[
\quad \text{protecting STATEREP .}
\]
\[
\quad \text{var 'x : Variable .}
\]
\[
\quad \text{var 'a : Expression .}
\]
\[
\quad \text{var b1 : BooleanExpression .}
\]
\[
\quad \text{var s1 : Statement .}
\]
\[
\quad \text{var s2 : Statement .}
\]
\[
\quad \text{op geneqns_ : Statement -> StateRep .}
\]
\[
\quad \text{eq geneqns('x := 'a) = StateTransition('x , 'a) .}
\]
\[
\quad \text{eq geneqns(if b1 then s1 else s2 fi) = Gamma(b1, geneqns(s1), geneqns(s2)) .}
\]
\[
\quad \text{eq geneqns(while b1 do s1 od) = FixedPoint(b1 , geneqns(s1)) .}
\]
\[
\quad \text{eq geneqns(s1 ; return 'x) = StateTransition('x, geneqns(s1)) .}
\]
\[
\quad \text{eq geneqns(s1 ; s2 ) = geneqns(s1) . geneqns(s2) .}
\]
\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 Inv : StateRep StateRep -> OutputExpr .
    op _._ : OutputExpr OutputExpr -> OutputExpr .
endfm
The following is SOLVE_FUNCTION module. It solves \texttt{StateRep} type \texttt{Function} to get output expression which has \texttt{OutputExpr} type. for some case, which is very hard to get an explicit final output, we will use function \texttt{solrecursive} or function \texttt{Inv}, which has the \texttt{StateRep} type, to represent it.

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


BIBLIOGRAPHY


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


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


