2013

Program analysis : termination proofs for Linear Simple Loops

Hongyi Chen
Louisiana State University and Agricultural and Mechanical College

PROGRAM ANALYSIS:
TERMINATION PROOFS FOR LINEAR SIMPLE LOOPS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Hongyi Chen
B.S., Sichuan University, China
M.S., Louisiana State University
May 2013

ACKNOWLEDGMENTS

At this opportunity the author wishes to express her most sincere gratitude and appreciation to Dr. Supratik Mukhopadhyay of the Computer Science Department for his valuable guidance and genuine interest as research advisor and chairman of the examination committee. His encouragements have kept me going even in difficult times, when the research seemed to go nowhere.

This dissertation would not have been possible without Dr. Sukhamay Kundu, who continually and enthusiastically conveyed the spirit and attitude of a rigorous researcher. Dr. Kundu had spent enormous amount of time on reviewing this dissertation, he made many important suggestions which greatly enhanced this dissertation. With Dr. Kundu's help, I was able to view my work in a broader background and gain a better understanding of it.

Deep appreciation is also extended to Dr. Konstantin Busch, Dr. Rajgopal Kannan of the Computer Science Department, Dr. Jimmie Lawson of the Math Department, and Dr. Luis A Escobar of the Experimental Statistics Department for serving as my committee members. Their comments have improved the quality of this research.

Finally, I am forever grateful for the supports from my parents, my boyfriend and all my college friends throughout my study at LSU. They have all contributed to this dissertation through ideas, encouragement, and support. Without their help, my study will not be as enjoyable and memorable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

LSL - Linear Simple Loop

BRC - Binary Reachability Check

LinearTermCheck - Procedure of termination check for LSL

DRR - Procedure of disjunctive ranking relation generation

DRRDetInt - Procedure of disjunctive ranking relation generation for deterministic integer case

ABSTRACT

Termination proof synthesis for simple loops, *i.e.*, loops with only conjoined constraints in the loop guard and variable updates in the loop body, is the building block of termination analysis, as well as liveness analysis, for large complex imperative systems. In particular, we consider a subclass of simple loops which contain only linear constraints in the loop guard and linear updates in the loop body. We call them *Linear Simple Loops* (LSLs). LSLs are particularly interesting because most loops in practice are indeed linear; more importantly, since we allow the update statements to handle nondeterminism, LSLs are expressive enough to serve as a foundational model for non-linear loops as well. Existing techniques can successfully synthesize a linear ranking function for an LSL if there exists one. When a terminating LSL does not have a linear ranking function, these techniques fail. In this dissertation we describe an automatic method that generates proofs of (universal) termination for LSLs based on the synthesis of disjunctive ranking relations. The method repeatedly finds linear ranking functions on parts of the state space and checks whether the transitive closure of the transition relation is included in the union of the ranking relations. We have implemented the method and have shown experimental evidence of the effectiveness of our method.

CHAPTER 1
INTRODUCTION

1.1 Background

In the 1900's, David Hilbert, a preeminent German mathematician, published a list of problems aiming at solving the fundamental questions of mathematics. Hilbert states: "Once a logical formalism is established one can expect that a systematic, so-to-say computational, treatment of logic formulas is possible, which would somewhat correspond to the theory of equations in algebra". This computational mechanism gave rise to the so-called decision problem, also known as "Entscheidungsproblem", which asked whether there exists an algorithm that takes an input of any first-order logic formula and checks the universal validity of that formula. After Hilbert's challenge, Alonzo Church [9] and Alan Turing [28] proved separately that a general solution to the Entscheidungsproblem is impossible, *i.e.*, there exists no such algorithm that checks the validity of any first-order logic formula. To prove this theorem, Turing reduced the "Halting problem" - given a Turing machine and an input check whether the Turing machine will eventually halt - to Entscheidungsproblem. Turing then proved the undecidability of Halting problem.

Nowadays, the Church-Turing Theorem can be viewed as the root of theoretical computer science. It is must-cover topic in any introductory course of theoretical computer science. Because of the popularization of Church-Turing Theorem, many undergraduate and graduate computer science students, or even some savvy computer science faculties, become to hold this misconception that it is impossible to prove termination of any program. Consequently the research topic of automatically proving program termination has rarely been explored for the past century.

Termination analysis has now found its new life in the past 10 years. Many pioneer computer scientists have come to realize the true consequence of Church-Turing Theorem – it is not that we are always unable to prove termination, rather we are unable to always prove termination. In fact, our current techniques show we can potentially prove termination in most practical cases. It is as Moshe Vardi puts it [30]: "most real-life programs, if they terminate, do so for rather simple reasons, because programmers almost never conceive of very deep and sophisticated reasons for termination", therefore it should be possible to prove termination for most programs in real cases. This is not a contradiction to Church-Turing Theorem in that no matter how sophisticated a technique we develop, there will always be at least one terminating program that this particular technique will fail to prove or disprove termination.

As of the writing of this dissertation, many new tools of termination checker have emerged and we can automatically prove or disprove termination for industrial-sized programs, for example, Windows OS device drivers. We have also proved termination for some notoriously difficult examples, such as Ackermann's function, McCarthy's 91 function, etc.

Termination is at the foundation of one type of important program properties called liveness property. A liveness property typically takes the form of "something good will eventually happen". To check this liveness property all we need to do is to prove the termination of the code before the occurrence of "something good".

In this dissertation, our research aims to solve the *Universal Termination Problem*, or the *Uniform Halting Problem*, which can be stated as follows:

> Given a program P, determine whether it will always finish running or could potentially execute forever.

```
int x;
while (x>0) {
    x := x - 1;
}
```

Figure 1.1: Simple Loop-program

1.2 Problem Statement

For the purpose of this dissertation, it is convenient to model the semantics of a computer program in the form of three elements: "state space", "transition relation", and "initial states". A program execution can be thought of as a path starting from one of the initial states followed by changes of states in the execution. We say a program is **terminating** if all of its execution paths are finite. A program is called **non-terminating** if there exists at least one infinite execution path. When trying to prove termination, formally we are trying to prove that the program's "transition relation" is "well-founded". Before attempting to automate the search for proofs of termination we first must ground ourselves with some basic concepts and notations.

Consider the following simple loop program in Figure 1.1. We will use this example to illustrate the concepts after we give a formal description.

Definition 1 (State, State space)**.** Given a program P, a state s of P is an assignment of a value to each variable of P, including program counter (the next statement to be executed). State space S is the set of all such states.

Let $X = (x_1, \ldots, x_n)$ be a vector composed of all the variables of a program P. We will denote a state of P as a valuation of X. That is, the valuation of X, $(v_1, \ldots, v_n)$, is the state of P where $x_1 = v_1, \ldots, x_n = v_n$.

Example 1.1. The simple loop program in Figure 1.1 has one variable, thus $X = (x)$. We can write the states as $(0), (1), (-1), (2), (-2), \ldots$. Or simply as $\ldots, (-2), (-1), (0), (1), (2), \ldots$. The state space S of the program is $\mathbb{Z}$. Throughout this dissertation we ignore program counter because we view the loop body as a single simulta-

neous assignment and the the "next statement" always refers to the true evaluation of loop condition and a simultaneous assignment, or the termination of loop and halting of program.

Definition 2 (Transition Relation)**.** Given a program P with its state space S, the transition relation of P is the binary relation $R \subseteq S \times S$ such that $(s_1, s_2) \in R$ if and only if when P is in state s_1 it reaches state s_2 in one step of execution.

Example 1.2. For the simple loop program in Figure 1.1, if the program is in state (2) it will reach the state (1) in one step of execution. In general, for any state (v) where $v > 0$, the program will reach the state $(v - 1)$ in one step of execution. Therefore the transition relation of the program is $R = \{((i), (j)) \in \mathbb{Z}^2 \mid i > 0 \text{ and } j = i - 1\}$.

Definition 3 (Transition System)**.** Given a program P, its transition system is a 3-tuple $\langle S, I, R\rangle$ where S is the state space of P, $I \subseteq S$ is the set of initial states for P, and $R \subseteq S \times S$ is its transition relation.

Example 1.3. The transition system of the simple loop program in Figure 1.1 is

$$\langle \mathbb{Z}, \mathbb{Z}, \{((i), (j)) \in \mathbb{Z}^2 \mid i > 0 \text{ and } j = i - 1\}\rangle$$

As depicted in Figure 1.2, nodes are the states, the links are the transitions. Note we assume each state in S can be an initial state.

Now that we have defined a model, transition system, for programs. Let us discuss the mathematical property, well-foundedness, corresponding to termination.

Definition 4 (Paths Permitted by a Relation)**.** Given a set S, a (finite or infinite) sequence $p = \langle s_1, s_2, s_3, \dots\rangle$ is called a path if $s_i \in S$ for all i, where s_i's need not be distinct. p is finite if it is of finite length, otherwise p is infinite. Given a relation $R \subseteq S \times S$, we say a finite path p is permitted by R if $(s_i, s_{i+1}) \in R$ for $1 \leq i < last$, where $last$ denotes the last index of p. We say an infinite path p is permitted by R if $(s_i, s_{i+1}) \in R$ for all $i \geq 1$.

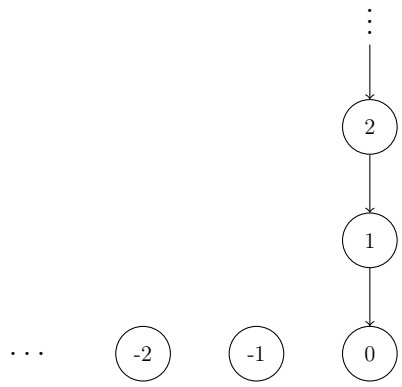

Figure 1.2: Transition System for the Program in Figure 1.1

Example 1.4. For the simple loop program in Figure 1.1, an example path permitted by R is $p = \langle(5), (4), (3), (2), (1), (0)\rangle$. Notice that however large value the initial x takes, the path is always of finite length.

Definition 5 (Well-founded Relation)**.** A binary relation $R \subseteq S \times S$ is well-founded iff it does not permit infinite path.

As mentioned earlier the transitive relation $R = \{(i, j) \in \mathbb{Z} \times \mathbb{Z} \mid i > 0 \text{ and } j = i - 1\}$, with $S = \mathbb{Z}$, is well-founded because it permits no infinite path. The next two theorems state the relation between termination and well-foundedness. Proving termination amounts to proving the corresponding transition relation well-founded. And since the structure of the transition relation could be complicated, we normally prove the well-foundedness of an "abstracted" relation instead.

Theorem 1.2.1. *A program P represented by transition system $\langle S, I, R\rangle$, where $I = S$, is terminating iff R is a well-founded relation.*

Theorem 1.2.2. *Let T be a well-founded binary relations. For any binary relation R, if $R \subseteq T$ then R is well-founded (i.e. any subset of a well-founded relation is also a well-founded relation).*

1.3 Disjunctive Ranking Relation

Before we can define disjunctive ranking relation we need to lay down some building blocks. The basic mathematical concept at the base of ranking is the well-ordered set.

Definition 6 (Well-Ordered Set)**.** A set D is well-ordered with respect to a linear order relation $\leq$ if there is no infinite decreasing sequence $d_0 > d_1 > d_2 > \dots$ of elements in D.

Example 1.5. $\mathbb{N} = \{0, 1, 2, \dots\}$ and $D = \{-2.5, -1.5, 0.5, 1.5, 2.5, \dots\}$ with the usual $\leq$ ("less than equal to") ordering are well-ordered set. Note that D is isomorphic to $\mathbb{N}$. The set $\{0, 1, 2, \dots, \omega, \omega + 1, \omega + 2, \dots\}$ is also well-ordered set but not isomorphic to $\mathbb{N}$.

Definition 7 (Ranking Function)**.** Given a transition relation $R \subseteq S \times S$, a function $r : S \to D$ is a ranking function for R, if D is a well-ordered set isomorphic to $\mathbb{N}$ and for every $(s_1, s_2) \in R$ we have $r(s_1) > r(s_2)$.

Note that for a path $p = \langle s_1, \dots, s_i, \dots, s_j, \dots \rangle$ permitted by R, we also have $r(s_i) > r(s_j)$ for $i < j$.

Definition 8 (Ranking Relation)**.** Given a ranking function $r : S \to D$ (w.r.t. a transition relation R) we define the corresponding ranking relation on S by

$$\tau(r) = \{(s_1, s_2) \in S \times S \mid r(s_1) > r(s_2)\}$$

Let us note two important points. First, $\tau(r)$ is transitive, *i.e.*, $\tau(r) = \tau(r)^+$, where $\tau(r)^+$ is the transitive closure of $\tau(r)$. Second, $R \subseteq \tau(r)$.

A fundamental observation is that for any program with ranking relation R and ranking function r, the ranking relation $\tau(r)$ is a well-founded relation and $R \subseteq \tau(r)$. By theorems 1.2.2 and 1.2.1 we have that if a program has a ranking function it terminates.

Theorem 1.3.1. *A program P represented by transition system $\langle S, I, R \rangle$, where $I = S$, is terminating iff exists a ranking function r for R (i.e. $R \subseteq \tau(r)$).*

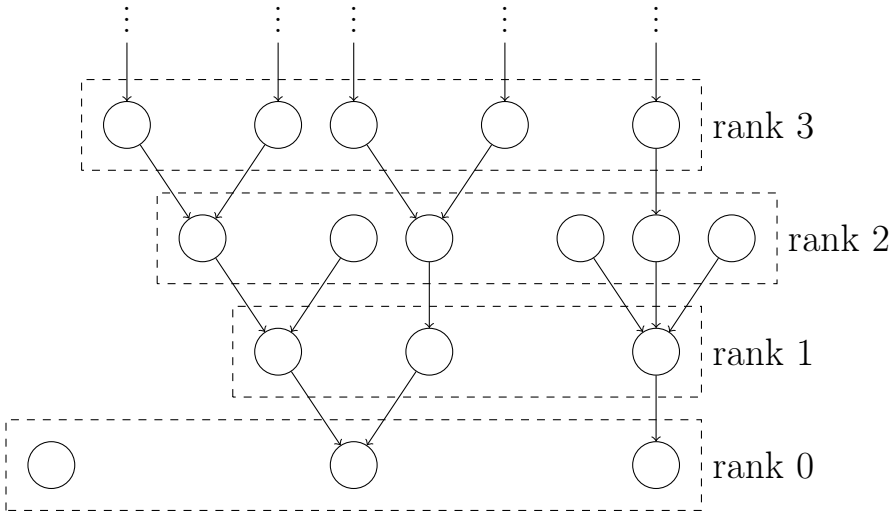

Figure 1.3: Ranking Function Example for a Terminating Transition System

Definition 9 (Disjunctive Ranking Relation)**.** A disjunctive ranking relation T is a finite union of ranking relations, arising from functions $r_i : S \to D$ and their related R_i.

$$T = T_1 \cup \cdots \cup T_n$$

where $T_i = \tau(r_i)$

The relation between disjunctive ranking relations and termination has been established in [21] using Ramsey's theorem [23]. Let P be a program, R be the corresponding transition relation induced by P, R^+ be the (non-reflexive) transitive closure of R, then P is terminating if and only if $R^+ \subseteq T$ for some disjunctive ranking relation T.

Theorem 1.3.2. *A program P represented by transition system $\langle S, I, R\rangle$, where $I = S$, is terminating iff exists disjunctive ranking relation T such that $R^+ \subseteq T$.*

We say a ranking function is linear if the function $r : S \to D$ is linear. Here, we assume S is one of $\mathbb{Z}^n$, $\mathbb{Q}^n$ or $\mathbb{R}^n$ for some $n > 1$ and D is a well-ordered set isomorphic to $\mathbb{N}$. Similarly, we say a ranking relation is linear if the underlying ranking function is linear. Finally, we say a disjunctive ranking relation is linear if all the individual ranking relations are linear.

Consider the transition system illustrated in Figure 1.4. The state space and initial states are both $\{1, 2\}$, the transition relation is $R = \{(1, 2), (2, 1)\}$. We separate the transition

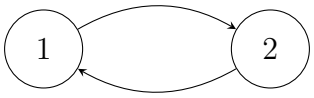

Figure 1.4: Transition System of a Non-terminating program

relation R into two smaller transition relations, $R_1 = \{(1, 2)\}$ and $R_2 = \{(2, 1)\}$. For R_1 we have a ranking function $r_1(x) = -x$, and for R_2 we have a ranking function $r_2(x) = x$. Neither of them is a ranking function for R. The corresponding ranking relations are $T_1 = \tau(r_1) = \{(1, 2)\}$ and $T_2 = \tau(r_2) = \{(2, 1)\}$. Observe that $R \subseteq T_1 \cup T_2$, but it is insufficient to prove termination of the transition relation (Figure 1.4 is obviously non-terminating). The ranking function r_1 can only prove the well-foundedness of sub-relation R_1. For the original relation R, r_1 does not has the property that for any path $\langle \dots s_i \dots s_j \dots \rangle$ permitted by R $r_1(s_i) > r_1(s_j)$. Similarly for r_2. To check whether the disjunctive ranking relation $T = T_1 \cup T_2$ suffices to prove termination of transition relation R in Figure 1.4, we have to show $R^+ \subseteq T$. But since it is not the case, this T is no good. This example shows the need for the stronger condition $R^+ \subseteq T$ instead of $R \subseteq T$ when T a union of more than one ranking relation. Theorem 1.3.2 states the condition of $R^+ \subseteq T$ is necessary and sufficient to prove termination. For example in Figure 1.4, we will see there does not exist a ranking function r such that $(1, 1) \in \tau(r)$, because it's impossible to make $r(1) > r(1)$. Therefore by extension, there does not exist a disjunction ranking relation T such that $R^+ \subseteq T$.

CHAPTER 2
LITERATURE REVIEW

In this chapter, we go over the existing techniques that address termination problem. We build our method upon some of them.

2.1 Ranking Function

Turing [29] proposed finding a mapping from a transition system to some known well-ordered set, thus proving the map is homomorphism.

2.1.1 Linear Ranking Function

The state of art of linear ranking function generation is that Poldelski and Rybalchenko provide us a complete method [20] to generate the linear ranking functions if there exists one. The method is based on Farkas's lemma [26], a mathematical mechanism to derive the hidden constraints from a system of linear inequalities.

Given an LSL $L(X, Y)$, convert it in the form $(AA') \binom{X}{Y} \leq b$. The method work by solving the linear programming problem of finding two vectors λ_1, λ_2 over the rational such that

- $\lambda_1, \lambda_2 \geq 0$ and,
- $\lambda_1 A' = 0$ and,
- $(\lambda_1 - \lambda_2) A = 0$ and,
- $\lambda_2 (A + A') = 0$ and,
- $\lambda_2 b < 0$

If such vectors do not exist the method report the program does not have a linear ranking function. If such vectors are found the method construct the following function which is a suitable linear ranking function for the input program

$$\rho(X) \stackrel{\text{def}}{=} \begin{cases} rX & \text{if exists } Y \text{ such that } (AA') \binom{X}{Y} \leq b, \\ \delta_0 - \delta & \text{otherwise} \end{cases}$$

where $r \stackrel{\text{def}}{=} \lambda_2 A'$, $\delta_0 \stackrel{\text{def}}{=} -\lambda_1 b$ and $\delta \stackrel{\text{def}}{=} -\lambda_2 b$.

2.1.2 Lexicographical-ordered Ranking Function

There are also work on lexicographical-ordered ranking functions. Consider a program represented as n transition relations, $\rho_1, \ldots, \rho_n$, over some state space. A *linear lexicographic ranking function* (LLRF) for the n relations is a sequence of n linear function, $f_1, \ldots, f_n$, such that for every $1 \leq i \leq n$, f_i is a ranking function over ρ_i (i.e., for every $(s,t) \in \rho_i$ we have $f_i(s) > f_i(t)$ and $f_i(s) \geq 0$) and for every $j < i$, f_j does not increase over ρ_i (*i.e.*, for every $(s,t) \in \rho_i$ we have $f_j(s) \geq f_j(t)$). There are quite many works devoted on lexicographical-ordered ranking function generation, for example [1, 6, 16].

2.2 Disjunctive Ranking Relation

Recent years, the discovery of disjunctive ranking relation as termination proof [21] have led to new termination check technologies. The validity of disjunctive ranking relation as termination proof is based Ramsey's Theorem [23]. Many tools [2, 14, 17, 22] have moved away from single ranking functions and towards disjunctive ranking relations.

Berdine *et al.* [2] propose a termination analysis using a combination of fixpoint-based abstract interpretation and an abstract domain of disjunctively ranking relations. Their

method performs an iterative computation of the set of abstract values and has a fixpoint detection of the form $T \subseteq R^+$.

Cook *et al.* [13] use relational predicates to extend the framework of Reps *et al.* [24] to support termination properties during computation of inter-procedural program summaries.

Kroening *et al.* [17] use the Octagon and Polyhedra abstract domains to discover invariance constraints sufficient to ensure termination. Well-foundedness checks are left to iterative verification by an external procedure as in the Terminator algorithm [22] and CTA [18].

2.2.1 Binary Reachability Check

Another technique we are benefited from is the method of binary reachability check developed by the Terminator team [14]. Given a disjunctive ranking relation T, binary reachability check is to prove or disprove $R^+ \subseteq T$. Note that the regular reachability check is over the state space S other than $S \times S$. The general idea of performing binary reachability check is to syntactically transform the original program, hence reduce the binary reachability check to the regular reachability check, which can be carried out on any temporal safety checker. Moreover, if the validity of $R^+ \subseteq T$ is not satisfied, the construct of the transformed program will enable the safety checker to generate a counterexample, an error path to be specific, that violates the inclusion. This counterexample is particularly helpful to expand and refine the disjunctive ranking relation T.

Given a transition relation R and a disjunctive ranking relation T, the goal of binary reachability check is to verify whether $R^+(L) \subseteq T$ is true. If yes, the procedure returns "false". Otherwise, the procedure returns an execution path $\langle s_1, \ldots, s_n \rangle$ such that $(s_1, s_n) \notin T$. The input and output of procedure BRC is described as follows (check [14] for details of the algorithm):

input: a transition relation R, and a disjunctive ranking relation T

output: if $R^+ \subseteq T$ return "true", else a counter example (s_1, s_n) such that $(s_1, s_n) \in R^+$ and $(s_1, s_n) \notin T$

2.3 Polyranking Principle

Another related topic is the termination principle for polyranking functions raised by Bradley *et al.* A polyranking function needs not always decrease but decreases eventually. It is a generalization of the regular lexicographical-ordered ranking function. In [5], the authors show a method for finding bounded expressions that are eventually negative over loops with parallel transitions. In [6], the authors demonstrate a method for synthesize lexicographic linear polyranking functions with supporting linear invariants over linear loops.

2.4 Size Change Termination

One of the directions of research that contributed to the efficacy of termination provers in practice is the size change termination principle presented by Lee, Jones and Ben-Amram [19]. Termination analysis based on the size change principle usually involves a two-stage structure. In first stage, construction of an abstract model of the original program in the form of size change graphs. Size change graph is an instance of a system which is much simpler than a programming language. The graphs contain abstract program values as nodes and use two types of edges, along which values of variables must decrease, or decrease or stay the same. No edge between nodes means that none of the relations can be ensured.. In second stage, analysis of the size change graphs for termination. The goal is to infer program termination via the following reasoning: if in any infinite execution of the program, some size must descend unboundedly, the program must always terminate, since infinite descent of a natural number is impossible.

CHAPTER 3
PRELIMINARIES

In this chapter, we introduce the technical terms specific to our research. We also give details of the methods that we are using as subroutines.

3.1 Loop Model and Semantics

Throughout this dissertation, we will study a subclass of programs called *Linear Simple Loops* (LSLs). Before we formally define LSL, we first show the origin, *Linear Simple Loop-program*, of LSL. Then we show how we generalize Linear Simple Loop-programs to a broader class LSL.

Definition 10 (Linear Simple Loop-program)**.** A Linear Simple Loop-program is a single loop-program such that

- the loop guard is a conjunction of linear constraints, and
- the loop body consists of only assignments for which the right-hand side is a linear expression (in particular, there are no if-then-else statements or loops in the loop body).

Figure 3.1 gives an example of Linear Simple Loop-program. Since we want the transition relation model consecutive iterations of the loop, instead of execution of individual assignment statements in the loop body, we will convert the sequential assignments into

```
int x, y, z;
while (x ≥ 0) {
    x := x + y;
    y := z;
    z := -y - 1;
}
```

Figure 3.1: Linear Simple Loop-program with Sequential Assignments

```
x := x + y;
y := z;
z := -y - 1;
```

$$x^{new} = x^{old} + y^{old}$$
$$y^{new} = z^{old}$$
$$z^{new} = -y^{new} - 1 = -z^{old} - 1$$

Figure 3.2: Linear Simple Loop-program with with Simultaneous Assignments

simultaneous assignments. Thus in each iteration, all assignments can be viewed as being executed simultaneously in one step. This kind of conversion is always possible, and we will demonstrate the conversion process with the example in Figure 3.2. We first use variables such as x^{old} and x^{new} (and possibly x^{newnew}) to replaces the variables in each assignment. Then we get a relation between the initial values of the variables and the values of the variables after all three assignments.

Since the assignments has no order any more, from this point on we can express a loop-program as a set of constraints. We will use two copies of the variables. The first copy, $X^0 = (x^0, y^0, z^0)$, stands for the initial values of the variables. The second copy, $X^1 = (x^1, y^1, z^1)$, stands for the values after all three assignments. X^1 is expressed only in terms of X^0.

$$L_{simul} = \{x^0 \geq 0, x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1\}$$

For each iteration, X^0 and X^1 must satisfy all the constraints in the set L. On the other hand, all X^0 and X^1 that satisfies all the constraints must constitute a valid loop iteration. This is true because we assume no initial condition before the loop, therefore the initial states is the same as the state space. For this example, the initial states and the state space is both $\mathbb{Z}^3$.

It is sometimes necessary to unroll a loop. L_{unroll} shows how we unroll L_{simul} once.

$$L_{unroll} = \{x^0 \geq 0, x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1,$$
$$x^1 \geq 0, x^2 = x^1 + y^1, y^2 = z^1, z^2 = -z^1 - 1\}$$

We make a copy of all the constraints and increase the variable superscripts by 1. Now L_{unroll} involves three set of values of variables, X^0, X^1, and X^2. Notice that we assume X^2 will always have value, that is the same to say, when we unroll a loop once we assume the second iteration will always be entered and executed.

Besides unrolling, we sometimes add extra constraints to the set as well. For instance, L_{extra} adds an constraint $x^1 < x^0$ to L_{simul}.

$$L_{extra} = \{x^0 \geq 0, x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1, x^1 < x^0\}$$

At this point, what we call "loop" is not necessarily associated with a concrete loop-program any more. It simply denotes a transition relation, or a set of paired states that satisfy all the constraints of the loop. For the loop of L_{extra}, the transition relation is between the state described by X^0 and the state described by X^1.

But what is the transition relation for the loop L_{unroll}? Is it between the state described by X^0 and the state described by X^1, or between the state described by X^0 and the state described by X^2, or even between the state described by X^1 and the state described by X^2? We define the transition relation to be between the initial state X^0 and some reachable state X^k where $1 \leq k \leq n$ and n denotes the largest superscript value.

As one can see, both the constraints and the index k are important for the description of the transition relation. Thus we put both elements into the formal description of the

transition relation of a loop. For example, we have

$$
\begin{aligned}
L &= \langle \{x^0 \geq 0, x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1\}, 1\rangle \\
L_1 &= \langle \{x^0 \geq 0, x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1, \\
&\qquad x^1 \geq 0, x^2 = x^1 + y^1, y^2 = z^1, z^2 = -z^1 - 1\}, 1\rangle \\
L_2 &= \langle \{x^0 \geq 0, x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1, \\
&\qquad x^1 \geq 0, x^2 = x^1 + y^1, y^2 = z^1, z^2 = -z^1 - 1\}, 2\rangle \\
L_3 &= \langle \{x^0 \geq 0, x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1, x^1 < x^0\}, 1\rangle
\end{aligned}
$$

L and L_3 denote the transition relations of L_{simul} and L_{extra}. L_1 and L_2 are two possible transition relations associated with L_{unroll}, where L_1 denotes a set of pairs (X^0, X^1), and L_2 denotes a set of pairs (X^0, X^2).

Before we go into the formal definition, we want to generalize LSL to be able to handle nondeterminism. To be specific, we allow linear expressions on both sides of an "assignment" statement, and inequalities instead of equal relation. This gives us more flexibility to model nondeterministic inputs or non-linear operations. For example, if we have an assignment $x := x - y^2$; we can abstract it to be $x^1 < x^0$.

Now that we allow linear inequalities as our constraints, we will standardize all constraints to be expressed using the $\leq$ relation. As one can see, an equal relation $\psi = b$ can be replaced by two inequalities $\psi \leq b$ and $-\psi \leq -b$.

We are finally ready to give the formal definition of LSLs. Here all variables range over one of the domain $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$.

Definition 11 (Linear Simple Loops)**.** A Linear Simple Loop L over m program variables, denoted as a column vector $X^0 = (x_1^0, x_2^0, \ldots, x_m^0)^T$, is a pair $L = \langle \textsc{Cons}_L, k\rangle$ where

- $\textsc{Cons}_L$ is a set of linear constraints, each of the form $A_0X^0 + \cdots + A_nX^n \leq b$, where $A_0, \ldots, A_n$ are row vectors in $\mathbb{Z}^m$, $b \in \mathbb{Z}$, $X^i = (x_1^i, x_2^i, \ldots, x_m^i)^T, 1 \leq i \leq n$, is a copy

of variables X^0, for at least for one of the constraints $A_k \neq 0$. We call $\textsc{Cons}_L$ loop constraints.

- and k is an integers, $0 < k \leq n$.

Note that our model (*i.e.* LSL) does not include an initial condition. In other words, we only model the transition relation and we assume any state is reachable.

For the purpose of a clearer presentation, instead of using $\leq$ for all constraints, we may use other relations $<, =, \geq$ or $>$, and we do not necessarily keep only integer on the right-hand side. The conversions from $=$ and $\geq$ to $\leq$ are rather obvious. It's more tricky to handle $<$ and $>$. In particular, when the variables range over $\mathbb{Z}$, and we have $\psi < b$ (ψ is a linear expression with integer coefficients, b is an integer), the standard form for the constraint should be $\psi \leq b - 1$. When the variables range over $\mathbb{Q}$ or $\mathbb{R}$, and we have $\psi < b$, we will use the abstracted standard form $\psi \leq b$. This might change the program behavior, we will discuss the limitation of this conversion at the end of this section.

The formal semantics of LSL is defined as follows. Let $L = \langle \textsc{Cons}_L, k \rangle$ be an LSL over variables X^0 and its $n (n \geq k)$ copies $X^1, X^2, \ldots, X^n$. We denote by $\overline{R}(L)$ the set of all tuples $(s_0, s_1, \ldots, s_n)$ such that all the constraints in $\textsc{Cons}_L$ are satisfied simultaneously when assigning s_0 to X^0, s_1 to X^1, $\ldots$, s_n to X^n. The transition relation of L is:

$$R(L) = \{(s_0, s_k) \mid (s_0, s_1, \ldots, s_n) \in \overline{R}(L)\}$$

It is sometimes convenient to put the LSL constraints in a matrix form. Given an LSL $L = \langle \textsc{Cons}_L, k \rangle$ over variables vector X^0 and its n copies. Let $\phi_1, \phi_2, \ldots, \phi_t$ be some enumeration of $\textsc{Cons}_L$. We put L in a matrix form $AX \leq b$ where X is a vector composed from the concatenation of the vectors $X^0, \ldots, X^n$ and the i'th row of the matrix A and the i'th element in the vector b are the coefficients of ϕ_i (the matrix coefficients correspond to the coefficients of $X^0, \ldots, X^n$).

In the followings we will define two useful operations. We start by $\textsc{Shift}_i$ which is the process of transforming variables in a constraint to a higher copy. The subscript i denotes how much value to increase for each superscript of the variables. For example, $\textsc{Shift}_1(x^0 - x^1 < 1) = x^1 - x^2 < 1$, and $\textsc{Shift}_2(x^0 - x^1 < 1) = x^2 - x^3 < 1$.

Definition 12 ($\textsc{Shift}_i$)**.** Given a linear expression $\psi : A_0X^0 + A_1X^1 + \cdots + A_nX^n$ (where $A_0, \ldots, A_n$ are row vectors in $\mathbb{Z}^m$), we define $\textsc{Shift}_k(\psi)$ inductively,

$$\textsc{Shift}_1(\psi) \triangleq A_0X^1 + A_1X^2 + \cdots + A_nX^{n+1}$$
$$\textsc{Shift}_{i+1}(\psi) \triangleq \textsc{Shift}_i(\textsc{Shift}_1(\psi))$$

We extend $\textsc{Shift}_i$ for a linear constraints $\varphi : \psi \leq b$ (where $b \in \mathbb{Z}$), and for a set of linear constraints C,

$$\textsc{Shift}_i(\varphi) \triangleq \textsc{Shift}_i(\psi) \leq b$$
$$\textsc{Shift}_i(C) \triangleq \{\textsc{Shift}_i(\varphi) \mid \varphi \in C\}$$

Next we define operation $\textsc{Unroll}_{i,j}$. This operation adds more constraints over higher copies of X.

Definition 13 ($\textsc{Unroll}_{i,j}$)**.** Given a set of linear constraints C and integer $j > 0$ we define $\textsc{Unroll}_{i,j}(C)$ inductively,

$$\textsc{Unroll}_{1,j}(C) \triangleq C$$
$$\textsc{Unroll}_{i+1,j}(C) \triangleq C \cup \textsc{Shift}_j(\textsc{Unroll}_{i,j}(C))$$

We extend $\textsc{Unroll}$ for LSLs, $L = \langle \textsc{Cons}_L, k \rangle$ by,

$$\textsc{Unroll}_i(L) \triangleq \langle \textsc{Unroll}_{i,k}(\textsc{Cons}_L), k \rangle$$

Before we move on to the next section, we want to state the significance of studying LSL and the limitations of LSL model. We choose to work on the LSL subclass because when we work on complex loops (loops with control statements such as if-then-else or inner loops in the loop body), we found that a single path of execution of a complex loop is in fact composed of a finite path followed by a simple loop [15]. And if the simple loop has non-linear expressions or nondeterministic inputs, we can often times abstract them to be linear. Hence Linear Simple Loops can serve as building blocks for the study of complex loops. Our model suffers from some limitations. When variables range over the reals (or rationals), we abstract update statements with strict inequality using a non-strict inequality. This is due to the ranking function synthesis method (described in Section 3.2) is based on Farkas's Lemma which cannot handle strict inequalities. As this abstraction is over-approximation our method becomes incomplete in this case, but it is still sound. Another limitation is that, when variables range over the integers, we cannot model precisely the truncation in integer division. For example, the assignment x := x/2 will be modeled as $2x^1 = x^0$ and it will permit transitions (x^0, x^1) like $(2, 1)$ and $(6, 3)$ but not $(3, 1)$.

3.2 Extension of the Linear Ranking Function Synthesis

In this section, we give the algorithm for linear ranking function synthesis. It is an extension of the Podelski algorithm [20] described in Section 2.1, and in our LSL notations. The algorithm given by Podelski can only handle transitions over X^0 and X^1. Our algorithm extends it by handling transitions over any number of copies of X^0.

Let $\vec{A}$ denote the row vector $(A^0 \dots A^n)$, $\vec{A}_i$ denote the i-th element A^i, $\vec{A}_{-i}$ denote the row vector with all but the i-th element $(A^0 \dots A^{i-1} A^{i+1} \dots A^n)$. Similarly we define column vectors $\vec{X}$, $\vec{X}_i$, and $\vec{X}_{-i}$.

Given LSL $L = \langle \textsc{Cons}_L, k \rangle$ in matrix form $\vec{A}\vec{X} \leq b$ ($\vec{X}_i$ is the X^i copy of X^0) we are interested if exist non-negative vectors λ_1 and λ_2 over rationals such that the following

constraints are satisfied

$$\lambda_1 \vec{A}_{-0} = 0 \tag{3.1}$$

$$\lambda_2 \vec{A}_{-0,-k} = 0 \tag{3.2}$$

$$(\lambda_1 - \lambda_2)\vec{A}_0 = 0 \tag{3.3}$$

$$\lambda_2(\vec{A}_0 + \vec{A}_k) = 0 \tag{3.4}$$

$$\lambda_2 b < 0 \tag{3.5}$$

Then following theorem states that if such λ_1, λ_2 exist then L has a linear ranking function (soundness). The theorem following right after states that is L has a linear ranking function then such λ_1, λ_2 exist (completeness).

Theorem 3.2.1. *An LSL $L = \langle \textsc{Cons}_L, k \rangle$ in matrix form $\vec{A}\vec{X} \leq b$ is terminating if there exist non-negative vectors λ_1, λ_2 over rationals such that* (3.1)-(3.5) *are satisfied. More over, a linear ranking function is given by*

$$\rho(X^0) = \begin{cases} rX^0 & \text{if exists } X^k \text{ such that } (X^0, X^k) \in R(L) \\ \delta_0 - \delta & \text{otherwise} \end{cases}$$

where $r \triangleq \lambda_2 \vec{A}_k$, $\delta_0 \triangleq -\lambda_1 b$, and $\delta \triangleq -\lambda_2 b$.

Proof. Let the pair of nonnegative vectors λ_1 and λ_2 be a solution of the system (3.1)-(3.5). For every $\vec{X}$ satisfying $\vec{A}\vec{X} \leq b$, by assumption $\lambda_1 \geq 0$, we have $\lambda_1 \vec{A}\vec{X} \leq \lambda_1 b$. We carry out the following sequence of transformations.

$$\begin{aligned}
\lambda_1 \vec{A}_{-0}\vec{X}_{-0} + \lambda_1 \vec{A}_0\vec{X}_0 &\leq \lambda_1 b \\
\lambda_1 \vec{A}_0\vec{X}_0 &\leq \lambda_1 b && \text{by (3.1)} \\
\lambda_2 \vec{A}_0\vec{X}_0 &\leq \lambda_1 b && \text{by (3.3)} \\
-\lambda_2 \vec{A}_k\vec{X}_0 &\leq \lambda_1 b && \text{by (3.4)}
\end{aligned}$$

From the assumption $\lambda_2 \geq 0$, we have $\lambda_2 \vec{A}\vec{X} \leq \lambda_2 b$. We carry out the following sequence of transformations.

$$\begin{aligned}
\lambda_2 \vec{A}_{-0,-k}\vec{X}_{-0,-k} + \lambda_2 \vec{A}_0\vec{X}_0 + \lambda_2 \vec{A}_k\vec{X}_k &\leq \lambda_2 b \\
\lambda_2 \vec{A}_0\vec{X}_0 + \lambda_2 \vec{A}_k\vec{X}_k &\leq \lambda_2 b && \text{by (3.2)} \\
-\lambda_2 \vec{A}_k\vec{X}_0 + \lambda_2 \vec{A}_k\vec{X}_k &\leq \lambda_2 b && \text{by (3.4)}
\end{aligned}$$

We define $r \triangleq \lambda_2 \vec{A}_k$, $\delta_0 \triangleq -\lambda_1 b$, $\delta \triangleq -\lambda_2 b$ and then we have,

$$r\vec{X}_0 \geq \delta_0 \qquad \text{(a)}$$

$$r\vec{X}_0 - r\vec{X}_k \geq \delta \qquad \text{(b)}$$

Lastly by (3.5), we have

$$\delta > 0 \qquad \text{(c)}$$

We define a function $\rho(X^0)$ as stated in the theorem. By (a), (b), and (c) we conclude that $\rho(X^0)$ must be a linear ranking function for L. □

Theorem 3.2.2. *Let $L = \langle \textsc{Cons}_L, k\rangle$ be an LSL in matrix form $\vec{A}\vec{X} \leq b$ ($\vec{X}_i$ is the X^i copy of X^0) over rationals variables. If L has a linear ranking function then exist non-negative vectors λ_1, λ_2 over rationals such that* (3.1)-(3.5) *are satisfied.*

Proof. Let the vector r together with the scalars δ_0 and δ define a linear ranking function for L. Then, for all $\vec{X}$ such that $\vec{A}\vec{X} \leq b$ we have $rX^0 \geq \delta_0$ and $rX^k \leq rX^0 - \delta$ and $\delta > 0$.

By the non-emptiness of the transition relation, the system $\vec{A}\vec{X} \leq b$ has at least one solution. Hence, we can apply the 'affine' form of Farkas' lemma, from which follows that there exists δ_0' and δ' such that $\delta_0' \geq \delta_0$, $\delta' \geq \delta$, and each of the inequalities $-rX^0 \leq -\delta_0'$ and$-rX^0 + rX^k \leq -\delta'$ is a nonnegative linear combination of the inequalities of the system $\vec{A}\vec{X} \leq b$. This means that there exist nonnegative rational-valued vectors λ_1 and λ_2 such that

$$\lambda_1\vec{A}\vec{X} = -rX^0$$
$$\lambda_1 b = -\delta_0'$$

and

$$\lambda_2\vec{A}\vec{X} = -rX^0 + rX^k$$
$$\lambda_2 b = -\delta'$$

After multiplication and simplification we obtain

$$\lambda_1\vec{A}_0 = -r \qquad \lambda_1\vec{A}_{-0} = 0$$
$$\lambda_2\vec{A}_0 = -r \qquad \lambda_2\vec{A}_k = r$$
$$\lambda_2\vec{A}_{-0,-k} = 0$$

from which equations (3.1)-(3.4) follow directly. Since $\delta' \geq \delta > 0$, we have $\lambda_2 b < 0$, *i.e.*, the equation (3.5) holds. □

3.3 Binary Reachability Check

As stated above, our termination argument (DRR) is composed from a set of ranking functions, not a single ranking function. With a single ranking function one must show that the rank decreases from the pre- to post-state after each single transition step (*i.e.* $R \subseteq \tau(r)$). In our settings it is not sufficient to look at a single transition step. Instead, we must consider all finite sequences of transitions. We must show that, for every sequence, one of the ranking functions decreases between the pre- and post-state. In other words: we must first find all pairs of states s_1 and s_2 such that s_2 is reachable from s_1; and we must then show that the value of one of the ranking functions decreases from s_1 to s_2. We call this task *binary reachability check*.

Given an LSL L and a disjunctive ranking relation T, the goal of binary reachability check is to verify whether $R^+(L) \subseteq T$ is true. If yes, the procedure returns "false". Otherwise, the the procedure returns a new LSL L' such that L' is an unrolling of L and $L'^+ \nsubseteq T$. Notice the unrolling of L is still an LSL, and it serves as a counterexample to the current disjunctive ranking relation T. The input and output of procedure BRC is described as follows, it is tailored specifically to LSL inputs:

input: an LSL L, and a disjunctive ranking relation T

output: if $R^+(L) \subseteq T$ return "false", else return LSL L' such that L' is an unrolling of L and $L'^+ \nsubseteq T$

CHAPTER 4
FORMAL ALGORITHM

4.1 Overview

In the following section we will illustrate the basic idea at the core of this dissertation. Here we will give an informal description of the principal inner workings of our method, a concrete example is given in section 4.2 and a formal description is given in section 4.3.

Consider a program over integer variables, represented by its transition relation which is given as a set of linear constraints over primed and unprimed variables, where primed variables stand for the state before the transition and the primed variables stand for the state after the transition.

Let $C \geq 0$ be one of the program constraints and assume C involves only unprimed variables. We write C' to mean the same expression as C with every variable replaced with its primed copy. We can separate the transition relation into two cases. In the first case we have $C' < C$ and in the second case we have $C' \geq C$. A key point is the observation that $C' < C$ and $C \geq 0$ are sufficient to conclude that the linear expression C is a ranking function for the first case. $C' < C$ means C will decrease in every transition, and $C \geq 0$ means C is bounded from bellow.

To handling the second case we will first rewrite $C' \geq C$ as $C' - C \geq 0$. Note the similarity to the constraint we started with $C \geq 0$. We will apply the same trick to this constraint. We separate the transition relation into two more cases, $C'' - C' < C' - C$ and $C'' - C' \geq C' - C$. Observant readers will notice we introduced a new variant of C. The C'' expression is the same expression as C' with the exception that every variable is replaced with its doubly primed copy. The doubly primed variables stand for the state after two transitions from the unprimed state and one transition from the primed state. For the

```
int x, y, z;
while (x ≥ 0) {
    x := x + y;
    y := z;
    z := -y - 1;
}
```

Figure 4.1: Example

first case, the expression $C' - C$ can be used as ranking function since it decreases in every transition ($C'' - C' < C' - C$) and it is bounded from bellow ($C' - C \geq 0$). The second case is rewritten again as $C'' - 2C' + C \geq 0$ and the process continues. The rewriting of the next second case will be $C''' - 3C'' + 3C'' - C \geq 0$, and so on. Termination of this process is achieved by inspecting the second case of each step before going further down. We try to synthesize a linear ranking function for the transition relation as it is and if the synthesis succeeds the process terminates.

Finally we check if the collection of the ranking functions generated by the above process is a valid termination condition. We either find a counter example for its validity which is given as a new transition relation which does not adhere to the termination condition, and then we run the process again on this counter example to strengthen the termination condition. Or we determine the termination condition is valid.

4.2 Example

We first demonstrate our technique with a simple deterministic LSL over the integers. Then we will extend our technique for nondeterministic updates and rational / real variables.

4.2.1 Deterministic Updates over Integer Domain

Consider the while loop in Figure 4.1. It has only 3 simple assignments, but it is not obvious whether it is terminating. It is easy to see that the traces of z are composed of two alternating numbers, one negative the other non-negative, and that the negative number has

Table 4.1: Sample Trace

	0	1	2	3	4	5	6	7	8	9
z:	10	-11	10	-11	10	-11	10	-11	10	-11
y:	3	10	-11	10	-11	10	-11	10	-11	10
x:	5	8	18	7	17	6	16	5	15	4

a higher value. The variable y always gets assigned to value of z from the previous state. Hence it behaves like z, except being one step behind. The variable x increments itself with y. Therefore x will alternatively increase (or stay unchanged) and decrease. Moreover the decrease is larger than the increase, hence x will eventually become negative and the loop will terminate. Table 4.1 illustrates the beginning of a trace of the program. Observe the value of X on the odd cycles, clearly it is decreasing and will eventually reach -1.

Let us first convert the while loop above to an LSL. The sequential update $x := x+y; y := z; z := -y - 1;$ is first translated to a simultaneous update constraints over (x^0, y^0, z^0) and (x^1, y^1, z^1). To do so we need to replace the concurrence of y in the update of z with the update of y, $z := -z - 1$. Then we replace all the variables in the right hand side of := with their 0 copy and all variables in the left hand side of := to their 1 copy, $x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1$. We get the following LSL,

$$L = \langle \{x^0 \geq 0, x^1 = x^0 + y^0,\ y^1 = z^0,\ z^1 = -z^0 - 1\}, 1 \rangle$$

If we apply the method of Section 2.4 to L, it will return failure since L does not have a linear ranking function. As mentioned earlier, we want to construct multiple linear ranking functions, each of them over a restrained input space. We do this by adding constraints to L such that the new LSL is guaranteed to have a linear ranking function. From $\textsc{Cons}_L$ we see that we already have the linear expression x^0 that is bounded, *i.e.*, $x^0 \geq 0$. If we add to L a constraint $x^0 > x^1$, then we know x^0 can serve as a ranking function for the restrained LSL because x^0 has a lower bound and is strictly decreasing, which is a sufficient condition for x^0 to become a ranking function over the integer domain.

We break L into two LSLs $L_{1.1}$ and $L_{1.2}$ such that $L_{1.1}$ is obtained by combining L with constraint $x^0 > x^1$, and $L_{1.2}$ is obtained by combining L with the negation of the constraint, namely $x^0 \leq x^1$.

$$L_{1.1} = \langle \textsc{Cons}_L \cup \{x^0 > x^1\}, 1\rangle \qquad \text{(trivial case)}$$

$$L_{1.2} = \langle \textsc{Cons}_L \cup \{x^0 \leq x^1\}, 1\rangle \qquad \text{(synthesis case)}$$

We call $L_{1.1}$ the *trivial case* since we immediately obtain a linear ranking function from it.

$$\rho_1(X^0) = \begin{cases} x^0 & \text{if } \exists X^1 \text{ such that } X^0, X^1 \text{ satisfies } L_{1.1} \\ -1 & \text{otherwise} \end{cases}$$

We call $L_{1.2}$ the *synthesis case* since it needs further examination. We call $x^0 \geq 0$ the *Seed* for partitioning L.

From this point onwards, we only need to take care of $L_{1.2}$. First we check whether $L_{1.2}$ has a linear ranking function already. In this particular case we find out that this is not true. Next, we would like to repeat the earlier process on $L_{1.2}$, *i.e.*, adding constraints to $L_{1.2}$ such that a linear ranking function must exist. Since $L_{1.2}$ already includes $x^0 \leq x^1$, using $x^0 \geq 0 \wedge x^0 > x^1$ again will no longer make sense. However observe that the new constraint in $L_{1.2}$ gives a new linear expression that is bounded below, *i.e.*, $x^1 \geq x^0$. This constraint will become our new *Seed*, and we can use it to partition $L_{1.2}$. This time we partition with the constraint $(x^1 - x^0) > (x^2 - x^1)$ and its negation.

At this point a new issue arises, x^2 is introduced to denote the value of x after one transition from x^1. However from $L_{1.2}$ alone, there is no such information about x^2. To remedy this situation, we first need to unroll L so that the unrolled transition involves x^2. We do this by making a copy of all the loop constraints in L, then changing X^1 to X^2, X^0

to X^1 (the process is formally described by $\textsc{Unroll}(L)$ in Section 4). We get L_2 as follows.

$$\begin{aligned}
L' &= \textsc{Unroll}_2(L) \\
&= \langle \{x^0 \geq 0, x^1 = x^0 + y^0,\ y^1 = z^0,\ z^1 = -z^0 - 1, \\
&\quad x^1 \geq 0,\ x^2 = x^1 + y^1,\ y^2 = z^1,\ z^2 = -z^1 - 1\}, 1\rangle \\
L_2 &= \langle \textsc{Cons}_{L'} \cup \mathit{Seed}, 1\rangle \\
&= \langle \{x^0 \geq 0, x^1 = x^0 + y^0,\ y^1 = z^0,\ z^1 = -z^0 - 1, \\
&\quad x^1 \geq 0,\ x^2 = x^1 + y^1,\ y^2 = z^1,\ z^2 = -z^1 - 1\} \cup \{x^1 \geq x^0\}, 1\rangle
\end{aligned}$$

Now we can partition L_2 using the constraint mentioned above.

$$L_{2.1} = \langle \textsc{Cons}_{L_2} \cup \{(x^1 - x^0) > (x^2 - x^1)\}, 1\rangle \qquad \text{(trivial case)}$$

$$L_{2.2} = \langle \textsc{Cons}_{L_2} \cup \{(x^1 - x^0) \leq (x^2 - x^1)\}, 1\rangle \qquad \text{(synthesis case)}$$

$L_{2.1}$ is again the trivial case, where a linear ranking function is guaranteed

$$\rho_2(X^0) = \begin{cases} x^1 - x^0 = y^0 & \text{if } \exists X^1, X^2 \text{ such that } X^0, X^1, X^2 \text{ satisfies } L_{2.1} \\ 0 & \text{otherwise} \end{cases}$$

Now we check whether the synthesis case has a linear ranking function. Notice that this time we can not use the method described in Section 2.4 any more, since now the synthesis case LSL involves X^2. In Section 4, we describe a general ranking function synthesis method which can handle this general form of LSLs. If we feed $L_{2.2}$ to the method in Section 4, we get the following linear ranking function.

$$\rho_3(X^0) = \begin{cases} 2x^0 + z^0 & \text{if } \exists X^1, X^2 \text{ such that } X^0, X^1, X^2 \text{ satisfies } L_{2.2} \\ -1 & \text{otherwise} \end{cases}$$

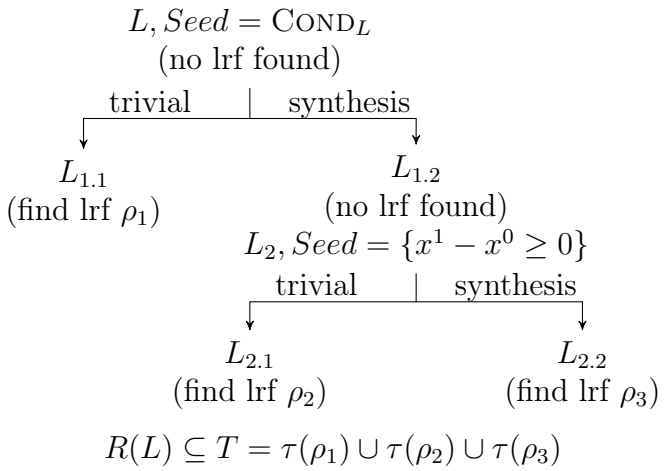

Figure 4.2: Execution of $L \triangleq \langle \{x^0 \geq 0, x^1 = x^0 + y^0, y^1 = z^0, z^1 = -z^0 - 1\}, 1 \rangle$

As shown in Figure 2, up to this point we have divided L to three LSLs, $L_{1.1}$, $L_{2.1}$, and $L_{2.2}$. Each of these three has a linear ranking function. Let $T = \tau(\rho_1) \cup \tau(\rho_2) \cup \tau(\rho_3)$. Theorem 4.4.2 in Section 4 shows us that $R(L) \subseteq T$. That is, any two consecutive states form a pair that belongs to T.

Recall that our goal is to find a T such that $R^+(L) \subseteq T$. We first check whether the T we found already satisfies $R^+(L) \subseteq T$. As it turns out for this particular case, it is not. BRC gives an error path that executes L twice. Therefore we get a new LSL L'' that unrolls L twice and L'' describes a relation from X^0 to X^2.

$$L'' = \textsc{Unroll}_2(L) \text{ with } i_{L''} = 0, j_{L''} = 2$$

Note L'' has the same set of constraints as L', but has different backedge. We feed L'' to the method described in Section 4.1. It shows that L'' has a linear ranking function already.

$$\rho_4(X^0) = \begin{cases} x^0 + y^0 & \text{if } \exists X^1, X^2 \text{ such that } X^0, X^1, X^2 \text{ satisfies } L' \\ -1 & \text{otherwise} \end{cases}$$

Again we update T by $T = T \cup \tau(\rho_4)$ and this time the test $R^+(L) \subseteq T$ succeeds, *i.e.*, we have successfully found a disjunctive ranking relation T for the original LSL L.

4.2.2 Variables over $\mathbb{Q}$ or $\mathbb{R}$ and Nondeterministic Updates

Notice that when variables range over $\mathbb{Q}$ or $\mathbb{R}$, the two constraints $\varphi \geq b$ and $\varphi > \textsc{Shift}(\varphi)$ can no longer guarantee φ to be a linear ranking function. One way to remedy that is to pick a small positive value c and partition the state space by $\varphi - \textsc{Shift}(\varphi) > c$ and its negation $\varphi - \textsc{Shift}(\varphi) \leq c$. Similar to the integer example, the former constraint will generate the trivial case, and the latter constraint will generate the synthesis case.

Another way is to still partition with $\varphi > \textsc{Shift}(\varphi)$ and its negation $\varphi \leq \textsc{Shift}(\varphi)$. However since the former can no longer generate a trivial case, we need to continue the partition process on the trivial case as well.

Nondeterministic updates are also an issue. If we look at ranking function ρ_2 above, the expression y^0 originates from the expression $x^1 - x^0$. We cannot use $x^1 - x^0$ directly because the ranking functions need to be expressed in terms of X^0. With deterministic updates, we can get rid of x^1 by substituting it with $x^0 + y^0$. With nondeterministic updates, we may not be able to simplify the expression in this manner. Therefore we need to apply Theorem 1 in Section 4.1 to generate ranking functions on X^0 only, and when we fail to find one, we need to partition the trivial case further. In our algorithm shown in Figure 4.4, this is the approach we take in all situations.

4.3 Algorithm for Synthesizing Disjunctive Ranking Relations

4.3.1 Formal Description

Lastly we define function Diff. This function creates new constraints that we use to partition the original LSL. It does so by taking constraint, shifting it and then binding the constraint

and its shift with $>$ or $\leq$. For instance, for constraint $\varphi : x^0 \leq 0$ we have $\textsc{Diff}_{1,>}(\varphi) = x^1 > x^0$ and $\textsc{Diff}_{1,\leq}(\varphi) = x^1 \leq x^0$.

Definition 14 ($\textsc{Diff}_i$)**.** Given a linear integer constraint $\varphi : \psi \leq b$ and set of linear integer constraints $Seed$, we define

$$\textsc{Diff}_{i,\sim}(\varphi) \triangleq \textsc{Shift}_i(\psi) \sim \psi$$
$$\textsc{Diff}_{i,\sim}(Seed) \triangleq \{\textsc{Diff}_{i,\sim}(\varphi) \mid \varphi \in Seed\}$$

where $\sim$ is one of $\{>, \leq\}$.

Now we give two procedures LinearTermCheck and DRR (for "Disjunctive Ranking Relation"). DRR, described in Figure 4.4, is a recursive procedure, that given an LSL L returns a disjunctive ranking relation T such that $R(L) \subseteq T$. The procedure DRRDetInt, described in Figure 4.5, is an optimization of the DRR procedure for the case where the constraints constitute deterministic updates and the variables are over $\mathbb{Z}$. Procedure LinearTermCheck, described in Figure 4.3, repeatedly calls DRR while $R^+(L) \not\subseteq T$, each time feeding DRR with an unrolling of the original L.

Suppose that DRR is called recursively i times with inputs $(L_1, Seed_1), (L_2, Seed_2), \ldots,$ $(L_i, Seed_i)$. If the linear ranking function synthesis for L_i succeeds and return the parameters r, δ_0, δ the following is the ranking function we use,

$$\rho_i(X) = \begin{cases} r(X) & \text{if exists } X^1 \text{ such that } (X, X^1) \in R(L_i) \text{ [case-0]} \\ \delta_0 - \delta & \text{else if exists } X^1 \text{ such that } (X, X^1) \in R(L_{i-1}) \text{ [case-1]} \\ \delta_0 - 2\delta & \text{else if exists } X^1 \text{ such that } (X, X^1) \in R(L_{i-2}) \text{ [case-2]} \\ \vdots & \vdots \\ \delta_0 - (i-1)\delta & \text{else if exists } X^1 \text{ such that } (X, X^1) \in R(L_1) \text{ [case-(i-1)]} \\ \delta_0 - i\delta & \text{otherwise [case-i]} \end{cases} \quad (\sharp)$$

procedure LinearTermCheck // may not terminate
input: LSL $L_{original} = \langle \text{CONS}_L, 1 \rangle$
output: disjunction linear ranking relation T or "fail"
begin
  $L \leftarrow L_{original}$;
  $T \leftarrow \emptyset$;
  do
    let *Seed* be the constraints associated with the loop guard of L
    if (DRR(L,*Seed*) terminates and returns the disjunctive linear ranking relation T') {
      $T \leftarrow T \cup T'$;
      if BRC($L_{original}, T$) terminates, $L \leftarrow BRC(L_{original}, T)$;
      // L is the counter example BRC found or $L = null$ (no counter example)
    }
    else // DRR(L, *Seed*) does not terminate and LinearTermCheck does not terminate
  while ($L \neq null$) // Another way LinearTermCheck may not terminate
  return T;
end.

Figure 4.3: Procedure LinearTermCheck

4.4 Correctness Proof

Theorem 4.4.2 insures the disjunctive ranking relation returned by DRR is large enough to contain the transition relation of the input LSL. This, in turn, insures that BRC will give a new counterexample for each iteration (until $R^+(L) \subseteq T$) and the termination condition converges towards a solution. The proof of theorem 4.4.2 relies on lemma 4.4.1. Lastly theorem 4.4.3 asserts the correctness of the algorithm.

Lemma 4.4.1. *Let $(s_0, \ldots, s_k, \ldots, s_{m \cdot k})$ be an $(m \cdot k + 1)$-trace of some $L = \langle \text{CONS}_L, k \rangle$ and let Seed be over $X^0, \ldots, X^{m \cdot (k-1)}$. If DRR is called with L and Seed as input and it succeeds then (s_0, s_k) is contained in the return set of DRR.*

Proof. We prove the claim by induction on the recursion depth.

Base: No recursion, *i.e.* the synthesizer synthesized a ranking function for L. By the correctness of the ranking function synthesizer it follows that $(s_0, s_k) \in T$

Step: We separate the proof into two cases:

procedure DRR // may not terminate
input: LSL $L = \langle \textsc{Cons}_L, k \rangle$, and $Seed$ a subset of $\textsc{Cons}_L$.
output: disjunction linear ranking relation T'
begin
  if linear ranking function synthesis on L succeeds with function ρ
    return $\tau(\rho)$;
  else {
    $T' \leftarrow \emptyset$
    $L_{unroll} \leftarrow \textsc{Unroll}_2(L)$
    for each $\varphi \in Seed$
      $Seed_{triv} \leftarrow \{\textsc{Diff}_{k,>}(\varphi)\}$
      $L_{triv} \leftarrow \langle \textsc{Cons}_{L_{unroll}} \cup Seed_{triv}, k \rangle$
      $T' \leftarrow T' \cup \text{DRR}(L_{triv}, Seed_{triv})$
    $Seed \leftarrow \textsc{Diff}_{k,\leq}(Seed)$
    $L \leftarrow \langle \textsc{Cons}_{L_{unroll}} \cup Seed, k \rangle$
  return $T' \cup \text{DRR}(L, Seed)$
  }
end.

Figure 4.4: Procedure DRR

procedure DRRDetInt // may not terminate
input: LSL $L = \langle \textsc{Cons}_L, k \rangle$, and $Seed$ a subset of $\textsc{Cons}_L$.
output: disjunction linear ranking relation T'
begin
  if linear ranking function synthesis on L succeeds with function ρ
    return $\tau(\rho)$;
  else {
    $T' \leftarrow \emptyset$
    $L_{unroll} \leftarrow \textsc{Unroll}_2(L)$
    for each $\varphi \in Seed$
      $Seed_{triv} \leftarrow \{\textsc{Diff}_{k,>}(\varphi)\}$
      $L_{triv} \leftarrow \langle \textsc{Cons}_{L_{unroll}} \cup Seed_{triv}, k \rangle$
      let φ' be the simplification of φ over X^0 // this simplification is possible since the
                                                // program is deterministic
      let ρ be the ranking function for L_{triv} with $r = \varphi', \delta = 1, \delta_0 = -1$
      $T' \leftarrow T' \cup \tau(\rho)$
    $Seed \leftarrow \textsc{Diff}_{k,\leq}(Seed)$
    $L \leftarrow \langle \textsc{Cons}_{L_{unroll}} \cup Seed, k \rangle$
  return $T' \cup \text{DRR}(L, Seed)$
  }
end.

Figure 4.5: Procedure DRRDetInt

1. Exists $s_{m\cdot k}+1,\ldots,s_{m\cdot(k+1)}$ such that $s_0,\ldots,s_{m\cdot(k+1)}$ is an $(m\cdot(k+1)+1)$-trace of $\textsc{Unroll}_k(L)$. This case is further refined into two sub-cases:
   (a) Exists $\varphi \in \mathit{Seed}$ such that $s_0,\ldots,s_{m\cdot(k+1)}$ satisfy $\textsc{Diff}_{k,>}(\varphi)$. In this case, by the induction hypothesis, (s_0,s_k) will be in the return value of the recursive call to DRR for φ and therefore $(s_0,s_k)\in T$.
   (b) $s_0,\ldots,s_{m\cdot(k+1)}$ satisfy $\textsc{Diff}_{k,\leq}(\varphi)$ for all $\varphi \in \mathit{Seed}$. In this case, by the induction hypothesis, (s_0,s_k) will be in the return value of the last recursive call to DRR and therefore $(s_0,s_k)\in T$.
2. There does not exist $s_{m\cdot k}+1,\ldots,s_{m\cdot(k+1)}$ such that $s_0,\ldots,s_{m\cdot(k+1)}$ is an $(m\cdot(k+1)+1)$-trace of $\textsc{Unroll}_k(L)$. In this case s_0 is assigned a value by the ranking function labeled ($\sharp$) from a case that supersede the case the function assign to s_k and therefore s_0 gets a higher ranking. It follows that $(s_0,s_k)\in T$. □

Theorem 4.4.2. *Suppose that DRR is called with input $L=\langle \textsc{Cons}_L, k\rangle$ and Seed where L is over $X^0,\ldots,X^k$. If DRR terminates successfully with return value T', then $R(L)\subseteq T'$.*

Proof. Assume that DRR terminates successfully. Let $(s_0,s_k)\in R(L)$. By the definition of $R(L)$, there exists $s_1,\ldots,s_{k-1}$ such that $s_0,\ldots,s_k$ be a $(k+1)$-trace of L. It follows immediately from Lemma 4.4.1 that (s_0,s_k) is contained in the return set of DRR. □

Theorem 4.4.3. *If procedure LinearTermCheck terminates successfully on a program P, then P terminates and has a disjunctive linear ranking relation T.*

Proof. If procedure LinearTermCheck terminates successfully on program P then we have found a disjunctive ranking relation T such that $R^+(P)\subseteq T$. As shown in [21], this is a sufficient condition for the termination of P and therefore P terminates. □

4.4.1 Termination and Complexity of the Algorithm

The procedures LinearTermCheck and DRR as given in this section may not always terminate, in particular when the input LSL is not terminating. When implemented we need to

bound the recursion depth of DRR and the number of iterations of the main loop. When the input LSL is deterministic and the variables range over $\mathbb{Z}$, the recursive calls to DRR for each $\varphi \in \mathit{Seed}$ will succeed with no further calls and therefore the number of calls to DRR will be linear in the depth bound. When the LSL is non-deterministic or the variables range over $\mathbb{Q}$ or $\mathbb{R}$, the number of calls to DRR in the worst case is exponential in the depth bound. Finally we note that the LSL $\textsc{Unroll}_i(L)$ has i times as many constraints and variables as in L.

CHAPTER 5
EXPERIMENTS

We created a test suite of LSL loops. To our knowledge it is the first LSL test suite. The loops are collected from other research work [3–5, 7, 10–12, 20, 21, 27] and real code. The test suite is still growing. At the time of our submission, it contains 38 LSL loops. Among them 11 are non-terminating loops, 7 are terminating with linear ranking functions, 20 are terminating with non-linear ranking functions. Moreover, 6 are non-deterministic, 32 are deterministic, 5 have 1 variable, 22 have 2 variables, 10 have 3 variables, and one has 4 variables. All loops are executed over domain $\mathbb{Z}$. The test suite as well as the implementation are available at [8].

We compared our method to linear ranking function synthesis method [20] using the implementation found in [25], and the polyranking method [6] using the implementation found in [3]. Detailed experimental results are provided in Table 5.1. The "Terminating" columns indicate whether the LSL terminates. The columns of "Linear", "Polyrank", and "Ours" indicate whether the methods of Podelski *et al.*'s linear ranking function synthesis method [20], Bradley *et al.*'s polyranking method [6], and ours, respectively, have successfully found a termination proof or run overtime. The "BRC" column states the number of times procedure BRC was called and the "DRR" column states the accumulative depth of DRR recursion. The "Failed Proc" column indicates which procedure, TermChech or DRR, failed terminating if the whole process failed to terminate. Since the runtime for all three methods was in the magnitude of a few milliseconds we omitted it from the table.

As shown in the table, our method considerably outperformed the other two methods. We succeed for all 7 loops with a linear ranking function. Out of the 20 terminating loops that have no linear ranking function we are successful for 17. For all non-terminating loops, the execution needs to be manually terminated. Except for one loop, all the proof searches

Table 5.1: Experiment results

#	Terminating	Linear	Polyrank	Ours	BRC	DRR	Failed Proc
1	yes	no	no	no	-	-	DRR
2	yes	yes	yes	yes	0	1	-
3	yes	yes	yes	yes	0	1	-
4	yes	yes	yes	yes	0	1	-
5	yes	yes	no	yes	0	1	-
6	yes	no	no	yes	0	2	-
7	no	-	-	-	-	-	DRR
8	no	-	-	-	-	-	DRR
9	no	-	-	-	-	-	DRR
10	no	-	-	-	-	-	DRR
11	yes	no	no	no	-	-	DRR
12	yes	no	no	yes	0	2	-
13	yes	no	no	yes	0	2	-
14	yes	no	no	yes	0	2	-
15	yes	yes	no	yes	0	1	-
16	no	-	-	-	-	-	DRR
17	no	-	-	-	-	-	DRR
18	yes	no	no	yes	0	2	-
19	no	-	-	-	-	-	DRR
20	no	-	-	-	-	-	DRR
21	yes	no	no	yes	0	2	-
22	no	-	-	-	-	-	DRR
23	yes	no	no	yes	0	2	-
24	yes	no	no	yes	0	2	-
25	yes	yes	yes	yes	0	1	-
26	yes	no	no	yes	0	2	-
27	yes	no	no	yes	0	2	-
28	yes	no	no	yes	0	2	-
29	no	-	-	-	-	-	DRR
30	yes	no	no	no	∞	3	BRC
31	yes	no	no	yes	0	2	-
32	yes	no	no	yes	0	2	-
33	no	-	-	-	-	-	DRR
34	yes	no	no	yes	1	3	-
35	yes	no	no	yes	0	2	-
36	yes	no	no	yes	0	2	-
37	yes	yes	yes	yes	0	1	-
38	yes	no	no	yes	0	2	-

fail in procedure DRR. In comparison, the linear ranking function synthesis method [20] succeeds for all the 7 loops with a linear ranking function; it fails to find a termination proof for all the 20 examples among the rest that were terminating. The polyranking method [6] succeeds in proving termination for 5 out of the 7 examples with a linear ranking function; it fails to find a termination proof for all the 20 examples among the rest that were terminating. We set the tree depth to be 100 for the polyranking method.

CHAPTER 6
CONCLUSIONS

This paper describes an automatic method for generating disjunctive ranking relations for Linear Simple Loops. The method repeatedly finds linear ranking functions on restricted state space until it reaches an over-approximation of the transitive closure of the transition relation. As demonstrated experimentally we largely expanded the scope of LSLs that can be solved. We also extended an existing technique for linear ranking function synthesis. The extended method can handle more general form of LSLs. Another contribution is that we created the first LSL test suite.

REFERENCES

[1] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In Radhia Cousot and Matthieu Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2010.

[2] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 211–224, New York, NY, USA, 2007. ACM.

[3] Aaron R. Bradley. polyrank: Tools for termination analysis. `http://theory.stanford.edu/~arbrad/software/polyrank.html`, 2005.

[4] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *In CAV*, pages 491–504. Springer, 2005.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In *In VMCAI2005: Verification, Model Checking, and Abstract Interpretation, volume 3385 of LNCS*, pages 113–129. Springer, 2005.

[6] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *In ICALP*, pages 1349–1361. Springer Verlag, 2005.

[7] Mark Braverman. Termination of Integer Linear Programs. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 372–385. Springer, 2006.

[8] Hong Yi Chen, Shaked Flur, and Supratik Mukhopadhyay. LSL Test Suite. `https://tigerbytes2.lsu.edu/users/hchen11/lsl/`.

[9] Alonzo Church. A Note on the Entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

[10] Michael Colón and Henny Sipma. Synthesis of Linear Ranking Functions. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 67–81, London, UK, 2001. Springer-Verlag.

[11] Michael A. Colon and Tomas E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304. Springer, 1998.

[12] Byron Cook, Sumit Gulwani, Tal Lev-ami, Andrey Rybalchenko, and Mooly Sagiv. Proving Conditional Termination, 2008.

[13] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Summarization For Termination: No Return!

[14] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 415–426, New York, NY, USA, 2006. ACM.

[15] Byron Cook and Andrey Rybalchenko. Proving that programs eventually do something good. In *In POPL06: Principles of Programming Languages*, pages 265–276. Springer, 2007.

[16] Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. Lexicographic Termination Proving. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2013, 2013.

[17] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Loopfrog: A Static Analyzer for ANSI-C Programs.

[18] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Loop Summarization using Abstract Transformers. In *6th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 5311, pages 111–125, Seoul, South Korea, 2008. Springer, Springer.

[19] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The Size-change Principle for Program Termination. In *POPL*, pages 81–92, 2001.

[20] Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, pages 239–251, 2004.

[21] Andreas Podelski and Andrey Rybalchenko. Transition Invariants. In *LICS*, pages 32–41, 2004.

[22] Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *In PADL*. Springer, 2007.

[23] Frank P. Ramsey. On a Problem of Formal Logic. In *Proc. London math. Soc.*, volume 30, pages 491–504, 1930.

[24] Thomas Reps, Mooly Sagiv, and Susan Horwitz. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[25] Andrey Rybalchenko. RankFinder. http://www.mpi-sws.org/ rybal/rankfinder/.

[26] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[27] A. Tiwari. Termination of linear programs. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV*, volume 3114 of *LNCS*, pages 70–82. Springer, July 2004.

[28] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58:345–363, 1936.

[29] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1948.

[30] Moshe Y. Vardi. Solving the unsolvable. *Commun. ACM*, 54(7):5, 2011.

VITA

Hongyi Chen was born in Chengdu, China. She received the degree of Bachelor of Engineering in Computer Science from Sichuan University, China. She then worked as a C/C++ programmer in China for three years before she joined the graduate program at Department of Computer Science of Louisiana State University, Baton Rouge. She received the degree of Master of Science in System Science in 2008. She went on to work as a Java programmer for a year before she returned to LSU to pursue her PhD degree in computer Science, which is to be awarded at the commencement of Spring 2013. Her research interests are Formal Verification, Static Analysis, Model Checking and Theorem Proving.