Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 307 (2014) 33–45

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

An Abstract Domain to Infer Symbolic Ranges over Nonnegative Parameters

Xueguang Wu[1], Liqian Chen[2] and Ji Wang[3]

National Laboratory for Parallel and Distributed Processing,
National University of Defense Technology,
Changsha 410073, China

Abstract

The value range information of program variables is useful in many applications such as compiler optimization and program analysis. In the framework of abstract interpretation, the interval abstract domain infers numerical bounds for each program variable. However, in certain applications such as automatic parallelization, symbolic ranges are often desired. In this paper, we present a new numerical abstract domain, namely the abstract domain of parametric ranges, to infer symbolic ranges over nonnegative parameters for each program variable. The new domain is designed based on the insight that in certain contexts, program procedures often have nonnegative parameters, such as the length of an input list and the size of an input array. The domain of parametric ranges seeks to infer the lower and upper bounds for each program variable where each bound is a linear expression over nonnegative parameters. The time and memory complexity of the domain operations of parametric ranges is $O(nm)$ where n is the number of program variables and m is the number of nonnegative parameters. On this basis, we show the application of parametric ranges to infer symbolic ranges of the sizes of list segments in programs manipulating singly-linked lists. Finally, we show preliminary experimental results.

Keywords: Abstract interpretation, Abstract domains, Intervals, Symbolic ranges.

1 Introduction

The range information of variable values, that is the lower and upper bound of the values that a variable may take, is quite useful in many applications including compiler optimization, automatic parallelization, bug detection, etc. Value range analysis seeks to automatically infer a range $[a, b]$ for each program variable x at compile time, which denotes the constraint $a \leq x \leq b$. The theory of abstract interpretation [8] provides a general framework to compute statically approximate but sound value ranges for program variables. The interval abstract domain [7] can discover lower and upper bounds on the values of program variables but the

[1] Email: xueguangwu@sina.cn
[2] Email: lqchen@nudt.edu.cn
[3] Email: wj@nudt.edu.cn

http://dx.doi.org/10.1016/j.entcs.2014.08.004

found bounds are numerical constants. However, in certain applications (such as automatic parallelization [3], symbolic bounds analysis [14] and bitwidth analysis [17]), symbolic ranges are desired. Symbolic range means that the bounds a and b of variable x are symbolic expressions over program variables except x.

In this paper, we present an abstract domain, namely *parametric ranges*, for deriving symbolic ranges over nonnegative parameters for each program variable. The new domain is designed based on the insight that in certain applications, the parameters of program procedures are often nonnegative, such as the initial length of an input list, the initial size of an input array (or memory region), and the starting address of a memory region. The domain of parametric ranges seeks to infer the lower and upper bounds for each program variable where each bound is a linear expression over nonnegative parameters, i.e., $x \in [\Sigma_i^m a_i p_i + c, \Sigma_i^m b_i p_i + d]$ where p_i denotes the symbolic value of the ith nonnegative parameter of the program procedure, $a_i, b_i \in \mathbb{R}, c \in \mathbb{R} \cup \{-\infty\}, d \in \mathbb{R} \cup \{+\infty\}$. The time and space complexity of the domain operations of parametric ranges is $O(nm)$ where n is the number of program variables and m is the number of nonnegative parameters. On this basis, we show the application of parametric ranges to infer symbolic ranges of list segment sizes for programs manipulating singly-linked lists. Moreover, we show how to combine the domain of parametric ranges with the affine equality domain to infer more complicated relations. Finally, we show preliminary experimental results.

We illustrate the domain of parametric ranges for invariant generation using a motivating example shown in Fig. 1 which has a nonnegative parameter n of type unsigned int. The interval domain [7] can only infer $x \in [0, +\infty]$ at the loop head ①. However, our domain of parametric ranges infers interesting symbolic ranges for x. Moreover, for this example, the invariants generated by the domain of parametric ranges are as precise as those given by the polyhedra domain [9].

```
void foo(unsigned int n) {
    unsigned int x;
    x := n;
  ① while (x ≤ 2n) do {
  ②     if (?) then  x := x + 2;
        else         x := 2 * x + 1;
  ③ } od
}
```

Loc	Intervals	Polyhedra	Parametric Ranges
①	$x \in [0, +\infty]$	$x \in [n, 4n + 2]$	$x \in [n, 4n + 2]$
②	$x \in [0, +\infty]$	$x \in [n, 2n]$	$x \in [n, 2n]$
③	$x \in [1, +\infty]$	$x \in [n + 1, 4n + 2]$	$x \in [n + 1, 4n + 2]$

Fig. 1. A motivating example

The rest of the paper is organized as follows. Section 2 discusses some related work. Section 3 presents the new abstract domain of parametric ranges. In Section 4, we show the application of parametric ranges to infer symbolic ranges of list segment sizes in list-manipulating programs. Section 5 presents our prototype implementation together with preliminary experimental results. Finally, conclusions as well as suggestions for future work are given in Section 6.

2 Related Work

Value range analysis has received much attention in compilation optimization, automatic parallelization and program analysis. In the framework of abstract interpretation, Cousot and Cousot [7] present the interval abstract domain to perform interval analysis using widening and narrowing. The interval domain can only infer numerical bounds of variable values, but scales to large-scale software in practice due to its linear time and space complexity.

Blume and Eigenmann [3] present symbolic range propagation technique to compute symbolic ranges in the context of parallelizing compilers. In their approach, symbolic range of variable x can be non-linear expressions with max/min operators over arbitrary variables except x. Hence, their approach is of exponential complexity in the worst case and requires heuristics to derive polynomial time operations. Our domain of parametric ranges differs from theirs in the following respect: parametric ranges of program variables are linear expressions over nonnegative parameters. In our approach, the set of program variables and the set of nonnegative parameters are disjoint. These restrictions greatly simplify the algorithms for implementing domain operations of parametric ranges. Thanks to these restrictions, the domain operations of parametric ranges are $O(nm)$ in time and space complexity.

Rugina and Rinard [14] present a framework for symbolic bounds analysis of pointers, array indices and accessed memory regions. They utilize symbolic polynomial expressions to bound the ranges of the pointers and array indices used to access memory. Instead of using fixed-point algorithms, their approach formulates the symbolic bound analysis problem as a system of constraints over symbolic bound polynomials and then reduces the constraint system into a linear program. The step of reduction to linear program is incomplete. Moreover, the soundness of the reduction to linear programs requires that all the variables in the symbolic polynomial expressions should be nonnegative. Our approach also requires that all the parameters in symbolic ranges are nonnegative. However, our approach only allows linear expressions as parametric ranges and is designed as an abstract domain. And our analysis does not need linear programming.

Sankaranarayanan et al. [15] present an abstract domain of symbolic range constraints. They assume a linear ordering among program variables and restrict the range for a variable x to involve variables of order strictly higher than x. Our domain restricts the range for a program variable to involve a separate set of nonnegative parameters rather than other program variables. In general, our domain is less expressive than their domain, since one could always assign nonnegative parameters with higher order than program variables in their domain. Moreover, their domain allows explicit relations among program variables and does not restrict parameters involved in the ranges to be nonnegative. However, with respect to time complexity, our domain is cheaper than their domain.

Our domain of parametric ranges is closest to the gauge abstract domain proposed by Venet [16] which is able to efficiently infer linear inequality invariants between each program variable and all loop counters in the scope of the loops. In

the gauge domain, each program variable is approximated by a pair of symbolic ranges that are linear expressions with integer coefficients over loop counters. Our domain of parametric ranges restricts the parameters to be nonnegative but not necessarily integers. Also we do not restrict the coefficients in the parametric ranges to be integers. The analysis based on the gauge domain will introduce a loop counter for each level of loops during the analysis, and thus could infer symbolic ranges for each program variable in terms of loop counters. Those loop counters can not be considered as parameters in our domain of parametric ranges, since they are initialized by zero and increase during the analysis. However, the loop bounds can be considered as parameters in our domain, and then we could infer parametric ranges for program variables in terms of loop bounds.

3 Symbolic Ranges over Nonnegative Parameters

3.1 Domain Representation

Let's consider a program procedure with n program variables $x_1, \ldots, x_j, \ldots, x_n$, together with m nonnegative parameters $p_1, \ldots, p_i, \ldots, p_m$. In practice, those parameters can be chosen from formal parameters of program procedures, global variables that are only read but never written by the considered program procedure, inputs from I/O devices, etc.

We maintain parametric ranges for each program variable x_j in the abstract domain, denoted as

$$x_j \in [\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d]$$

where $a_i, b_i \in \mathbb{R}, c \in \mathbb{R} \cup \{-\infty\}, d \in \mathbb{R} \cup \{+\infty\}$ and p_i is a non-negative parameter. The concretization is defined by

$$\gamma([\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d]) = \{x_j \in \mathbb{R} \mid \Sigma_i a_i p_i + c \leq x_j \leq \Sigma_i b_i p_i + d\}$$

which represents the set of possible values of x_j.

Relaxation of the non-negativity restriction. Note that the non-negativity restriction of parameters can be relaxed. If we know one of the numerical lower bound c or the numerical upper bound d for a parameter p_i that may take both negative or positive values, we could introduce a new auxiliary nonnegative parameter p'_i such that

$$p'_i \stackrel{\text{def}}{=} p_i - c$$

or

$$p'_i \stackrel{\text{def}}{=} d - p_i$$

It is easy to see that it always holds that $p'_i \geq 0$. Then we could replace all the appearances of p_i by $p'_i + c$ (or $d - p'_i$) in the whole program syntactically.

3.2 Domain Operations

Before we show domain operations over parametric ranges, we first define the following ordering $\sqsubseteq_e$ ($\sqsubset_e$) on linear expressions over nonnegative parameters such

that

$$\Sigma_i a_i p_i + c \sqsubseteq_e \Sigma_i b_i p_i + d \quad \text{if and only if} \quad \forall p \in [\underline{p}, \overline{p}], \Sigma_i (b_i - a_i) p_i + (d - c) \geq 0$$

$$(\quad \Sigma_i a_i p_i + c \sqsubset_e \Sigma_i b_i p_i + d \quad \text{if and only if} \quad \forall p \in [\underline{p}, \overline{p}], \Sigma_i (b_i - a_i) p_i + (d - c) > 0 \quad)$$

where $[\underline{p}, \overline{p}]$ denotes numerical ranges for parameters p. Note that it always holds that $[\underline{p_i}, \overline{p_i}] \subseteq [0, +\infty]$ (where $0 \leq i \leq m$). In our implementation, we first check whether it holds that

$$\Sigma_i (b_i - a_i) p'_i + (d - c) \geq 0$$

$$(\quad \Sigma_i (b_i - a_i) p'_i + (d - c) > 0 \quad)$$

where

$$p'_i = \begin{cases} \overline{p_i} & \text{if} \quad a_i \geq b_i \\ \underline{p_i} & \text{otherwise} \end{cases}$$

which implies $\Sigma_i (b_i - a_i) p_i + (d - c) \geq 0$ (or $\Sigma_i (b_i - a_i) p_i + (d - c) > 0$). However, note that $\Sigma_i (b_i - a_i) p_i + (d - c) \geq 0$ does not necessarily imply $\Sigma_i (b_i - a_i) p'_i + (d - c) \geq 0$. We utilize $\bot_p$ to denote the empty parametric range when $\Sigma_i (b_i - a_i) p_i + (d - c) < 0$.

Now we describe the implementation of the most common domain operations that are used in numerical static analysis.

Inclusion Test. Inclusion test $\sqsubseteq_p$ between two parametric ranges for the same program variable x_j, is defined as

$$[\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d] \sqsubseteq_p [\Sigma_i a'_i p_i + c', \Sigma_i b'_i p_i + d']$$

$$\stackrel{\text{def}}{=} \Sigma_i a'_i p_i + c' \sqsubseteq_e \Sigma_i a_i p_i + c \ \wedge \ \Sigma_i b_i p_i + d \sqsubseteq_e \Sigma_i b'_i p_i + d'$$

Meet. The intersection $\sqcap_p$ of two parametric ranges for the same program variable x_j, is defined as

$$[\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d] \sqcap_p [\Sigma_i a'_i p_i + c', \Sigma_i b'_i p_i + d']$$

$$\stackrel{\text{def}}{=} \begin{cases} \bot_p & \text{if} \quad \Sigma_i b_i p_i + d \sqsubset_e \Sigma_i a'_i p_i + c' \vee \Sigma_i b'_i p_i + d' \sqsubset_e \Sigma_i a_i p_i + c \\ [lexp, lexp'] & \text{otherwise} \end{cases}$$

where

$$lexp \stackrel{\text{def}}{=} \begin{cases} \Sigma_i a_i p_i + c & \text{if } \Sigma_i a'_i p_i + c' \sqsubseteq_e \Sigma_i a_i p_i + c \\ \Sigma_i a'_i p_i + c' & \text{else if} \quad \Sigma_i a_i p_i + c \sqsubseteq_e \Sigma_i a'_i p_i + c' \\ \Sigma_i a_i p_i + c & \text{else if} \quad \Sigma_i a_i + c \geq \Sigma_i a'_i + c' \\ \Sigma_i a'_i p_i + c' & \text{otherwise} \end{cases}$$

and

$$lexp' \stackrel{\text{def}}{=} \begin{cases} \Sigma_i b_i p_i + d & \text{if } \Sigma_i b_i p_i + d \sqsubseteq_e \Sigma_i b'_i p_i + d' \\ \Sigma_i b'_i p_i + d' & \text{else if } \Sigma_i b'_i p_i + d' \sqsubseteq_e \Sigma_i b_i p_i + d \\ \Sigma_i b_i p_i + d & \text{else if } \Sigma_i b_i + d \leq \Sigma_i b'_i + d' \\ \Sigma_i b'_i p_i + d' & \text{otherwise} \end{cases}$$

Note that for $lexp$ we try to choose the smaller one (with respect to $\sqsubseteq_e$) between $\Sigma_i a_i p_i + c$ and $\Sigma_i a'_i p_i + c'$. When $\Sigma_i a_i p_i + c$ and $\Sigma_i a'_i p_i + c'$ are not comparable, we use a heuristic strategy to pick one from them as the new lower bound, by comparing the sums of the coefficients. It is worth noting that it is sound to choose either $\Sigma_i a_i p_i + c$ or $\Sigma_i a'_i p_i + c'$ as the resulting lower bound, according to the concrete semantics of intersection, i.e.,

$$\begin{aligned} &\gamma([\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d]) \cap \gamma([\Sigma_i a'_i p_i + c', \Sigma_i b'_i p_i + d']) \\ \subseteq\ &\gamma([\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d] \sqcap_p [\Sigma_i a'_i p_i + c', \Sigma_i b'_i p_i + d']). \end{aligned}$$

The same idea applies to $lexp'$ for upper bound.

Join. To abstract the control-flow join in the program, we need to compute the union of two parametric ranges for the same program variable x_j. In this paper, we compute an over-approximation $[lexp, lexp']$ of the union such that both the two input parametric ranges are included in $[lexp, lexp']$. The join operation $\sqcup_p$ is defined as follows:

$$[\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d] \sqcup_p [\Sigma_i a'_i p_i + c', \Sigma_i b'_i p_i + d'] \stackrel{\text{def}}{=} [lexp, lexp']$$

where

$$lexp \stackrel{\text{def}}{=} \begin{cases} \Sigma_i a_i p_i + c & \text{if } \Sigma_i a_i p_i + c \sqsubseteq_e \Sigma_i a'_i p_i + c' \\ \Sigma_i a'_i p_i + c' & \text{else if } \Sigma_i a'_i p_i + c' \sqsubseteq_e \Sigma_i a_i p_i + c \\ \Sigma_i \min(a_i, a'_i) p_i + \min(c, c') & \text{otherwise} \end{cases}$$

and

$$lexp' \stackrel{\text{def}}{=} \begin{cases} \Sigma_i b_i p_i + d & \text{if } \Sigma_i b'_i p_i + d' \sqsubseteq_e \Sigma_i b_i p_i + d \\ \Sigma_i b'_i p_i + d' & \text{else if } \Sigma_i b_i p_i + d \sqsubseteq_e \Sigma_i b'_i p_i + d' \\ \Sigma_i \max(b_i, b'_i) p_i + \max(d, d') & \text{otherwise} \end{cases}$$

Note that since for all i we have $p_i \geq 0$, it always holds that

$$(\Sigma_i \min(a_i, a'_i) p_i + \min(c, c')) \sqsubseteq_e (\Sigma_i a_i p_i + c)$$

and

$$(\Sigma_i b_i p_i + d) \sqsubseteq_e (\Sigma_i \max(b_i, b'_i) p_i + \max(d, d'))$$

Hence, the result of the join operation $\sqcup_p$ always provides an overapproximation of the two input parametric ranges.

Example 3.1 Consider the program shown in Fig. 1. When the fixed point iteration becomes stable, at ②, we have $x \in [n, 2n]$. At the program point after applying the assignment transfer function of the `then` branch, we get $x \in [n+2, 2n+2]$. And at the program point after applying the assignment transfer function of the `else` branch, we get $x \in [2n+1, 4n+1]$. Then, at the control-flow join point ③, we need to compute the join $[n+2, 2n+2] \sqcup_p [2n+1, 4n+1]$, which will result in $[n+1, 4n+2]$.

Test Transfer Function. Any conditional test involving program variables (x_j) and parameters (p_i) can be converted to a series of conditional tests of the form $x_j \leq \Sigma_i a_i p_i + c$ or $x_j \geq \Sigma_i b_i p_i + d$, by substituting each variable x_k (where $k \neq j$) in the conditional test constraint by its parametric range. Then we define the test transfer function as follows:

$$[\![x_j \leq \Sigma_i a_i p_i + c]\!]^{\#}(\rho_{x_j}) \stackrel{\text{def}}{=} \rho_{x_j} \sqcap_p [-\infty, \Sigma_i a_i p_i + c]$$

$$[\![x_j \geq \Sigma_i b_i p_i + d]\!]^{\#}(\rho_{x_j}) \stackrel{\text{def}}{=} \rho_{x_j} \sqcap_p [\Sigma_i b_i p_i + d, +\infty]$$

where ρ denotes the abstract environment that maps each program variable to its symbolic range before applying the transfer function and ρ_{x_j} denotes the symbolic range of x_j.

Assignment Transfer Function. Any assignment $x_j := e$ wherein e is an expression involving program variables (x_j) and parameters (p_i) can be converted to an assignment of the form $x_j := [\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d]$, by substituting each program variable in e by its parametric range. Then we define the assignment transfer function as follows:

$$[\![x_j := [\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d]]\!]^{\#}(\rho_{x_j}) \stackrel{\text{def}}{=} [\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d]$$

Widening with Thresholds. Widening with thresholds [2] $\triangledown^T$ is a widening parameterized by a finite set of threshold values T, including $-\infty$ and $+\infty$. Widening with thresholds for the parametric ranges domain is defined as:

$$\begin{aligned}&[\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d] \nabla_p^T [\Sigma_i a'_i p_i + c', \Sigma_i b'_i p_i + d'] \\ &\stackrel{\text{def}}{=} [\Sigma_i a''_i p_i + c'', \Sigma_i b''_i p_i + d'']\end{aligned}$$

where

$$\begin{cases} a''_i \stackrel{\text{def}}{=} a_i \leq a'_i \ ?\ a_i : \max\{\ell \in T \mid \ell \leq a'_i\} \\ c'' \stackrel{\text{def}}{=} c \leq c' \ ?\ c : \max\{\ell \in T \mid \ell \leq c'\} \\ b''_i \stackrel{\text{def}}{=} b_i \geq b'_i \ ?\ b_i : \min\{h \in T \mid h \geq b'_i\} \\ d'' \stackrel{\text{def}}{=} d \geq d' \ ?\ d : \min\{h \in T \mid h \geq d'\} \end{cases}$$

where T is a finite set of threshold values, including $-\infty$ and $+\infty$. Recall that ? : denotes the conditional operator.

Intuitively, $\triangledown_p^T$ utilizes element-wise the interval widening with thresholds for each coefficient of parameters as well as the constant term. Furthermore, when a program variable x_j is also known to be always nonnegative, e.g., of type unsigned int, the widening for the parametric ranges of x_j can be improved as:

$$[\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d] \nabla_p^T [\Sigma_i a'_i p_i + c', \Sigma_i b'_i p_i + d']$$

$$\stackrel{\text{def}}{=} \begin{cases} [\Sigma_i a''_i p_i + c'', \Sigma_i b''_i p_i + d''] & \text{if } 0 \sqsubseteq_e \Sigma_i a''_i p_i + c'' \\ [0, \Sigma_i b''_i p_i + d''] & \text{otherwise} \end{cases}$$

where a''_i, c'', b''_i, d'' are defined the same as above.

Example 3.2 Consider the procedure shown in Fig. 2 which is adapted from [6]. After the first iteration, the input arguments of the widening at ① are $\rho_x : [0.75n + 1, 0.75n + 1]$ and $\rho'_x : [0.6875n + 1, n + 1.25]$. If we use $\{0, 0.5, 1, 1.5\}$ together with $+\infty$ and $-\infty$ as the threshold set T, $\rho_x \nabla_p^T \rho'_x$ will result in $[0.5n + 1, n + 1.5]$, which will be stable in the subsequent increasing iterations.

```
void foowiden(unsigned int n){
    real x;
    x := 0.75 * n + 1;
    while (true) do {
①   if (?)
        then   x := n + 1;
        else   x := 0.25 * x + 0.5 * n + 1;
    } od
}
```

Fig. 2. An example to show widening of parametric ranges.

4 Application to Infer Symbolic Ranges of List Segment Sizes

In our previous work [4][5], we have shown an approach to derive a numerical program with nonnegative integer variables from a program that manipulates singly-linked lists. The main idea is as follows: a singly-linked list can be divided into a set of non-overlapping list segments, according to reachability of pointer variables to list nodes. For each list segment, we introduce an auxiliary nonnegative integer variable, called counter variable, to denote the size of that list segment (that is the number of the list nodes contained in that list segment). In list-manipulating program procedures, there often exist nonnegative parameters that represent the initial lengths of the input lists. Hence, in the derived numerical programs from list-manipulating programs, counter variables together with those numerical variables appeared in the

original list-manipulating programs can be considered as program variables, while the initial lengths of the input lists can be considered as nonnegative parameters.

In the derived numerical programs from list-manipulating programs, there often exist affine equality relations between program variables and parameters. Hence, we combine the domain of parametric ranges with the affine equality abstract domain [12] to perform the analysis. We use parametric ranges to track symbolic range information of each program variable, and use affine equalities to track the affine equality relations among program variables and parameters. Assume that there are n program variables and m nonnegative parameters. Then each domain element in the combined domain is described by an affine equality system $Ay = b$ where $y = [x\ p]^T$ and $A \in \mathbb{R}^{(n+m)\times(n+m)}, b \in \mathbb{R}^{n+m}$, together with symbolic ranges for program variables $x_j \in [\Sigma_i a_i p_i + c, \Sigma_i b_i p_i + d]$ (where $a_i, b_i \in \mathbb{R}, c \in \{\mathbb{R}, -\infty\}, d \in \{\mathbb{R}, +\infty\}$) and numerical ranges for nonnegative parameters $p_i \in [c', d']$ (where $0 \le c' \le d'$).

Bound tightening. In the combined domain, the symbolic ranges of each program variable are maintained by the domain of parametric ranges. However, the parametric ranges may be changed during domain operations of the affine equality domain. E.g., when an affine equality is added, the parametric ranges of variables can be tightened. In this paper, we adapt the bound prorogation technique that is a kind of constraint propagation widely used in constraint programming [1], to tighten the parametric ranges.

In fact, each affine equality of the combined domain can be used to tighten the parametric ranges for those program variables occurring in it. Let $[\underline{x}_j, \overline{x}_j]$ denote the parametric range of program variable x_j. Then, given an equality $\Sigma_i a_i x_i + \Sigma_j a_j p_j = b$, if $a_i \neq 0$, a new candidate parametric lower bound for x_i comes from: $\underline{x}'_i = (b - \Sigma_j a_j p_j - \Sigma_{j\neq i} a_j \overline{\underline{x}}_j)/a_i$ where $\overline{\underline{x}}_j = a_j > 0\,?\,\overline{x}_j : \underline{x}_j$, and a new candidate parametric upper bound for x_i comes from: $\overline{x}'_i = (b - \Sigma_j a_j p_j - \Sigma_{j\neq i} a_j \overline{\underline{x}}'_j)/a_i$ where $\overline{\underline{x}}'_j = a_j > 0\,?\,\underline{x}_j : \overline{x}_j$. If the new parametric ranges are tighter, then x_i's ranges are updated. This process can be repeated with each variable in that equality and with each equality in the system.

Example 4.1 Consider the list-manipulating program procedure *copy_and_delete*() shown in Fig. 3 (a). It has a nonnegative parameter n as the initial length of the input list x. The program first reversely copies list x to list p, and then delete both lists simultaneously. To prove the memory safety of this program, a key invariant is needed to show that the length of list p is equal to the length of list x after line 11. Moreover, if a statement accesses the *next* field of a list variable p (e.g., the statements in Lines 6, 8, 13, 14), we need to show that the length of the list segment pointed to by p is greater than or equal to 1. Fig. 3 (b) shows a numerical program derived from the list-manipulating program *copy_and_delete*(), in which t^{pq} denotes the size of the list segment that can be reached by list variables p and q. For example, the list assignment statement $\{y := x;\}$ on Line 3 in Fig. 3 (a) means that after the assignment both x and y point to the list segment that used to be pointed to by only x. And after the assignment, it is easy to see that there will be not any more list segment that is pointed to by only x (without y). Hence, according to the semantics of list

```
void copy_and_delete(List* x, uint n){
1:   List* y, p, q;
2:   assume \length(x)==n;
3:   y := x;
4:   q := p := null;
5:   while (y != null) do {
6:     y := y → next;
7:     q := malloc();
8:     q → next := pList;
9:     p := q;
10:  } od
11:   y := x;
12:  while (y != null) do {
13:    y := y → next;
14:    q := q → next;
15:    free(x);
16:    free(p);
17:    x := y;
18:    p := q;
19:  } od
}
```

(a)

```
void copy_and_delete_num(uint n){
  uint t^x, t^y, t^xy, t^p, t^q, t^pq;
   t^x := n;
  t^xy := t^x; t^x := 0;
    t^p := 0; t^q := 0; t^pq := 0;
    while (t^xy ≥ 1) do {
      t^x := t^x+1; t^xy := t^xy -1;
      t^p := t^pq; t^pq := 0; t^q := 1;
      t^pq := t^p; t^p := 0;
      t^pq := t^q+t^pq; t^q := 0;
  } od
  t^xy := t^x; t^x := 0;
    while (t^xy ≥ 1) do {
      t^x := t^x+1; t^xy := t^xy -1;
      t^p := t^p+1; t^pq := t^pq -1;
      t^y := t^xy; t^xy := 0; t^x := 0;
      t^q := t^pq; t^pq := 0; t^p := 0;
      t^xy := t^y; t^y := 0;
      t^pq := t^q; t^q := 0;
  } od
}
```

(b)

Fig. 3. A list-manipulating program (a) together with its numerical version (b). uint denotes unsigned int.

assignment, we can derive the numerical statements $\{t^{xy} := t^x; t^x := 0;\}$ shown on Line 3 in Fig. 3 (b). We refer the reader to [4][5] for details about how to derive a numerical program from a list-manipulating program. Using the combined domain of parametric ranges and affine equalities, after Line 12, we infer invariants: $\{t^{xy} - t^{pq} == 0, t^{xy} \in [1, n], t^{pq} \in [1, n], n \in [1, +\infty]\}$. These invariants are sufficient to prove the memory safety of the second loop (i.e., from Line 12 to Line 19).

5 Implementation and Experiments

We have implemented our prototype domain of parametric ranges PaRa, inside the Apron [11] numerical abstract domain library which provides a common interface for numerical abstract domains. Our experiments were conducted using the Interproc [13] static analyzer.

In order to assess the precision and efficiency of PaRa, we compare the obtained invariants as well as the performance of PaRa with the interval domain Box and the polyhedra domain NewPolka in Apron. Our experiments were conducted on two sets of examples. The results are summarized in Figs. 4-5. The column "#Vars" gives the number of program variables and "#Pars" gives the number of nonnegative parameters. "PaRa+AffineEqs" denotes the combined domain of parametric ranges and affine equalities. "Inv." compares as a whole the invariants obtained by two domains. A ">" ("<", "=", "≠") indicates that the left-side

analysis outputs stronger (weaker, equivalent, incomparable) invariants than the right-side analysis. The analysis times are given when the analyzer is run on a 2.5GHz PC with 2G of RAM running Fedora 12.

Program			Analysis Results						
Name	#Vars	#Pars	Box	Inv.	PARA	Inv.	PARA+AffineEqs	Inv.	NEWPOLKA
foo	1	1	0.006s	$<$	0.007s	$=$	0.008s	$=$	0.012s
foowiden	1	1	0.006s	$<$	0.007s	$=$	0.008s	$>$	0.012s
ex_ipps95 [3]	1	1	0.004s	$<$	0.006s	$=$	0.007s	$=$	0.011s
ex_sas07 [15]	2	2	0.005s	$<$	0.006s	$<$	0.006s	$=$	0.012s
ex_toplas05 [14]	2	1	0.006s	$<$	0.008s	$<$	0.010s	$=$	0.016s
ex_cav09_1 [10]	3	2	0.007s	$<$	0.010s	$<$	0.015s	$=$	0.021s
ex_cav09_2 [10]	2	2	0.007s	$<$	0.007s	$<$	0.010s	$=$	0.017s
ex_cav09_3 [10]	4	1	0.008s	$<$	0.011s	$<$	0.017s	$=$	0.021s
ex_cav12_1 [16]	2	1	0.003s	$<$	0.004s	$=$	0.004s	$<$	0.010s
ex_cav12_2 [16]	2	0	0.001s	$=$	0.002s	$<$	0.004s	$=$	0.006s
all_above	20	12	0.023s	$<$	0.053s	$<$	0.092s	$\neq$	0.335s

Fig. 4. Experimental results on numerical programs.

Fig. 4 shows the results on a collection of small numerical programs. foo and foowiden respectively correspond to the programs shown in Fig. 1 and Fig. 2. Programs with prefix "ex_" are taken from related work of symbolic ranges analysis [3,14,15,16], symbolic complexity bound analysis [10]. The program all_above is constructed by concatenating all the other programs listed in Fig. 4. For all programs listed in Fig. 4 except ex_cav12_2 (which involves no parameters), PARA gives more precise invariants than the interval domain Box. Compared with the polyhedra domain NEWPOLKA, PARA generates less precise invariants in most cases. However, for the program foowiden, PARA generates more precise invariants than NEWPOLKA due to the widening with thresholds used in PARA which tries thresholds on coefficients of parameters. For all programs with prefix "ex_" except ex_ipps95 and ex_cav12_1, although PARA is less precise than NEWPOLKA, "PARA+AffineEqs" is as precise as NEWPOLKA, since these programs involve affine equality relations between program variables, which cannot be expressed in PARA but can be expressed in "PARA+AffineEqs". For ex_cav12_1, "PARA+AffineEqs" is less precise than NEWPOLKA, since this program involves inequality invariants among program variables which cannot be expressed in "PARA+AffineEqs" but can be captured by NEWPOLKA.

Fig. 5 shows the results on a set of numerical programs which are manually derived from list-manipulating programs. These list-manipulating programs contain commonly used operations over singly-linked lists such as create, traverse, reverse, merge, copy, delete and dispatch. The program list_all_above is constructed by concatenating all the other programs listed in Fig. 5. This kind of numerical programs derived from list-manipulating programs often involve affine equality relations among program variables and nonnegative parameters. Hence, we utilize the combined domain of parametric ranges and affine equalities to analyze these programs. On these programs, "PARA+AffineEqs" gives as precise invariants as those given by NEWPOLKA. And these invariants are precise enough to prove the memory safety of the original list-manipulating programs.

Program			Analysis Results				
Name	#Vars	#Pars	Box	Inv.	PaRa+AffineEqs	Inv.	NewPolka
list_create	4	1	0.008s	$<$	0.011s	$=$	0.018s
list_traverse	3	1	0.006s	$<$	0.008s	$=$	0.016s
list_reverse	5	1	0.009s	$<$	0.015s	$=$	0.025s
list_length_equal	4	1	0.007s	$<$	0.010s	$=$	0.017s
list_merge	5	2	0.009s	$<$	0.018s	$=$	0.027s
list_copy_and_delete	6	1	0.007s	$<$	0.024s	$=$	0.032s
list_dispatch	7	1	0.011s	$<$	0.029s	$=$	0.040s
list_all_above	34	8	0.036s	$<$	0.181s	$=$	0.826s

Fig. 5. Experimental results on numerical programs derived from list-manipulating programs.

6 Conclusion

We have presented an abstract domain of parametric ranges to capture the symbolic ranges of each program variable, wherein the bounds of value ranges of each program variable are described as linear expressions over nonnegative parameters. This domain is more powerful than the interval abstract domain. The time and space complexity of its domain operations is $O(nm)$ where n is the number of program variables and m is the number of nonnegative parameters. As an example to illustrate the usefulness of the new domain, we have shown its application to infer symbolic ranges of list segment sizes for analyzing list-manipulating programs.

It remains for future work to consider experiments on larger programs and the usage of parametric ranges in more applications such as buffer overflow analysis. Another direction of work is to consider using nonlinear expressions over nonnegative parameters as symbolic ranges. In this case, a nonlinear template could be chosen for each program variable.

Acknowledgement

We would like to thank the anonymous reviewers for their constructive comments. This work is supported by the 973 Program under Grant No. 2014CB340703, the NSFC under Grant Nos. 61202120, 61120106006, 91318301, and the SRFDP under Grant No. 20124307120034.

References

[1] Bessiere, C., *Constraint propagation*, in: F. Rossi, P. van Beek and T. Walsh, editors, *Handbook of Constraint Programming*, Elsevier, 2006 .

[2] Blanchet, B., P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, *A static analyzer for large safety-critical software*, in: *PLDI* (2003), pp. 196–207.

[3] Blume, W. and R. Eigenmann, *Symbolic range propagation*, in: *IPPS* (1995), pp. 357–363.

[4] Chen, L., R. Li, X. Wu and J. Wang, *Static analysis of list-manipulating programs via bit-vectors and numerical abstractions*, in: *SAC* (2013), pp. 1204–1210.

[5] Chen, L., R. Li, X. Wu and J. Wang, *Static analysis of lists by combining shape and numerical abstractions*, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.06.004.

[6] Chen, L., A. Miné, J. Wang and P. Cousot, *An abstract domain to discover interval linear equalities*, in: *VMCAI*, LNCS **5944** (2010), pp. 112–128.

[7] Cousot, P. and R. Cousot, *Static determination of dynamic properties of programs*, in: *Proc. of the 2nd International Symposium on Programming* (1976), pp. 106–130.

[8] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *POPL* (1977), pp. 238–252.

[9] Cousot, P. and N. Halbwachs, *Automatic discovery of linear restraints among variables of a program*, in: *POPL* (1978), pp. 84–96.

[10] Gulwani, S., *Speed: Symbolic complexity bound analysis*, in: *CAV*, LNCS **5643** (2009), pp. 51–62.

[11] Jeannet, B. and A. Miné, *Apron: A library of numerical abstract domains for static analysis*, in: *CAV*, LNCS **5643** (2009), pp. 661–667.

[12] Karr, M., *Affine relationships among variables of a program*, Acta Inf. **6** (1976), pp. 133–151.

[13] Lalire, G., M. Argoud and B. Jeannet, *Interproc*, http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/.

[14] Rugina, R. and M. C. Rinard, *Symbolic bounds analysis of pointers, array indices, and accessed memory regions*, ACM Trans. Program. Lang. Syst. **27** (2005), pp. 185–235.

[15] Sankaranarayanan, S., F. Ivancic and A. Gupta, *Program analysis using symbolic ranges*, in: *SAS*, LNCS **4634** (2007), pp. 366–383.

[16] Venet, A., *The gauge domain: Scalable analysis of linear inequality invariants*, in: *CAV*, LNCS **7358** (2012), pp. 139–154.

[17] Zaks, A., Z. Yang, I. Shlyakhter, F. Ivancic, S. Cadambi, M. K. Ganai, A. Gupta and P. Ashar, *Bitwidth reduction via symbolic interval analysis for software model checking*, IEEE Trans. on CAD of Integrated Circuits and Systems **27** (2008), pp. 1513–1517.