Санкт-Петербургский государственный университет

Кафедра системного программирования

Оптимизация модели символьной памяти в проекте V#

Пожарский Роман Николаевич, группа 21.Б07

Санкт-Петербургский государственный университет

5 января 2024 г.

Научный руководитель: к. ф-м. н. Д.А. Мордвинов, доцент кафедры системного программирования

Консультант: М.П. Костицын, магистр СПбГУ по направлению «Программная инженерия»

План презентации

1. Символьное исполнение
2. Модель памяти V#
3. Алгоритм

Символьное исполнение

Техника анализа программного обеспечения, которая позволяет исполнить код функции с символьными переменными, которые подставляются вместо аргументов.

Пример

```
public int Abs(int x) {
  int result = x;
  if (x < 0) {
    result = -x;
  }
  return result;
}
```

V# --- платформа для генерации тестов для программ на платформе .NET, .NET Core, .NET Framework.

Символьное исполнение

Память в .NET

Память, используемая .NET программой делится на 4 типа

1. стек --- для примитивных типов, аргументов и локальных переменных функций
2. динамическая (куча) --- для ссылочных типов
3. статическая --- для статических полей
4. пул интернирования строк --- область для оптимизации памяти при работе со строками

Символьное исполнение

Динамическая память

Наиболее сложным разделом является динамическая память, потому что формально разные ссылочные параметры функции могут быть одним объектом

```
public int Foo(int[] a, int[] b)
{
  a[0] = 1;
  if (b[0] != 1)
  {
    return 1;
  }
  return 0;
}
int[] a = new int[] {1, 2};
var r = Foo(a, a);
```

Постановка задачи

Целью работы является оптимизация модели памяти в проекте V#. Для её выполнения были поставлены следующие задачи:

1. изучить литературу по моделированию символьной памяти
2. разработать алгоритмы оптимизации символьной памяти
3. реализовать алгоритмы оптимизации символьной памяти
4. протестировать влияние проведённых оптимизаций на крупных проектах на платформе .NET.

Модель памяти V#

Модель памяти с регионами

- части составных объектов хранятся в состоянии в специальных словарях, ключами которых являются некоторые идентификаторы , а значения --- *регионы памяти*
- регионы памяти и соответствующие им идентификаторы имеются для
  1. полей классов (имя, тип, класс, в котором объявлено поле)
  2. элементов массивов (тип элементов, размерность)
  3. длинн массивов (тип элементов, размерность)
  4. нижних границ массивов (тип элементов, размерность)
  5. конкретных объектов (адрес)
  6. статических полей (тип)
  7. буферов, выделенных на стеке при помощи *stackalloc* (специальный ключ)
- для идентификации объектов внутри региона памяти используются *ключи* (например, $a[i]$)
- в рамках работы регион памяти можно отождествить с *деревом обновлений*

Модель памяти V#

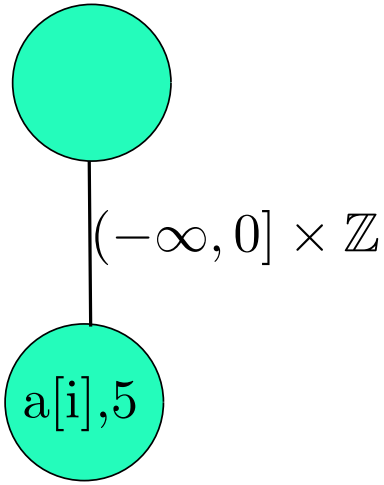
Дерево обновлений

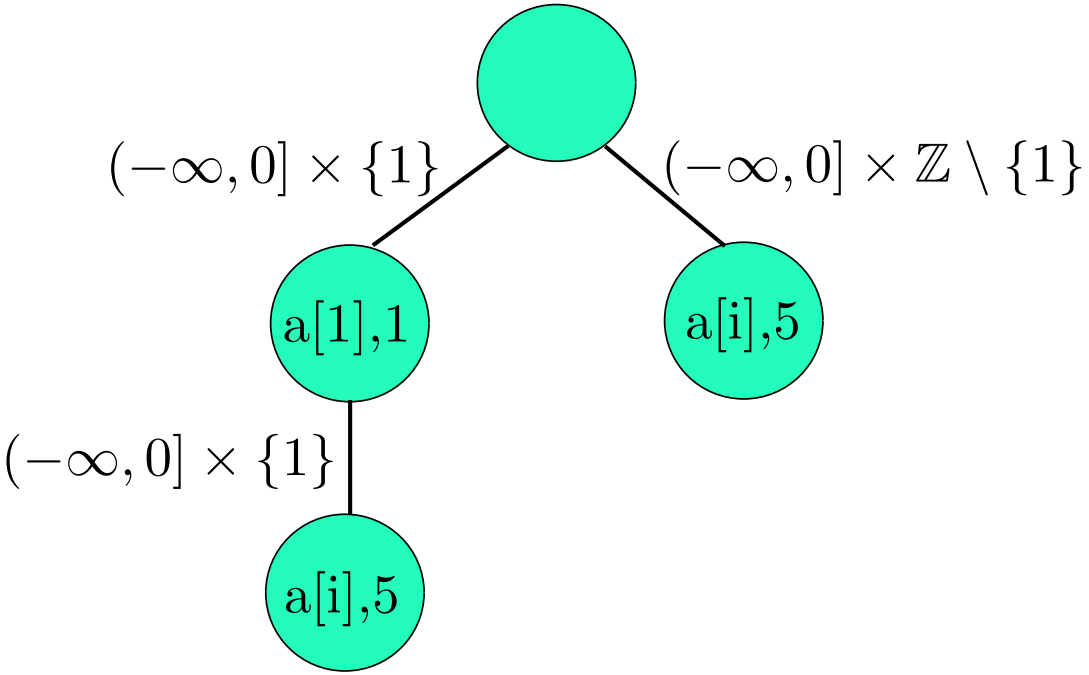
- в узлах хранится ключ, по которому делалась запись, и значение
- индексируется *регионами*, которые определяют множество возможных значений ключей
- поддерживает 2 инварианта:
  1. регионы поддеревьев включены в регион дерева
  2. регионы, индексирующие поддеревья одного дерева, не пересекаются между собой

Модель памяти V#

Дерево обновлений, пример

```
public object Foo(object[] a, int i, int j) {
  a[i] = 5;
  a[1] = 1;
  return a[j];}
```





Алгоритм

Разделение результатов чтения

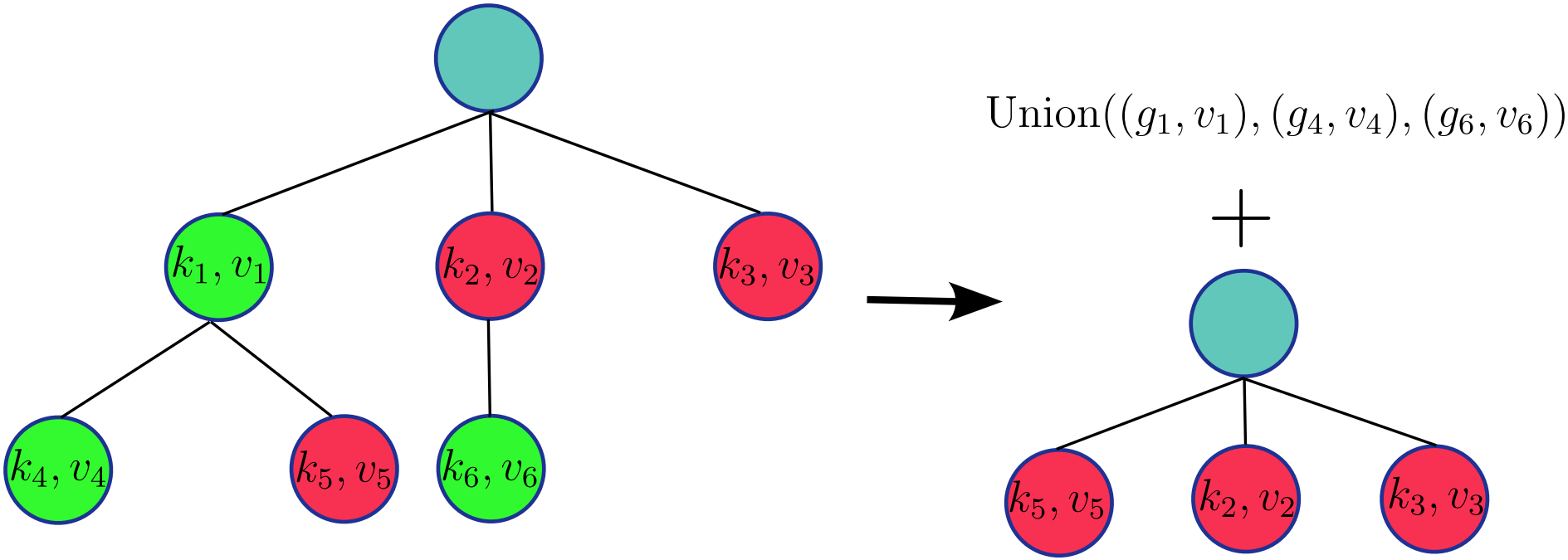
Если в дереве обновлений нет ключа, по которому происходит чтение, возвращается символьная константа с сохранённым для решателя деревом обновлений, хотя результат чтения можно сделать множеством возможных значений, защищённых условиями, с которыми дальше можно работать в символьном исполнении.

```
public object Foo(object[] a, int i, int j) {
  a[i] = 5;
  a[1] = 1;
  return a[j];}

Foo(a, i, j);
if (j == 1 && a[j] != 1) { .. } // недостижимая ветка
```

Алгоритм

По определённому предикату отбираются наиболее интересные записи, из которых составится Union. Подграф из оставшихся записей преобразуется в дерево, и, как и ранее, останется символьной константой с сохранённым для решателя деревом



Алгоритм

Псевдокод

```
func splitRead(tree, readKey, predicate, exploringPath):
 symblicCaseTree = new()
 union = new()
 notMatchPathCond = Conjuction(exploringPath.map(writeKey => !MC(readKey, writeKey, writeRegion)))
 foreach (reg, (utKey, st)) in tree:
   matchCond = MC(readKey, utKey.key, reg)
   g = (Conjuction(notMatchPathCond, matchCond])
   recUnion, rec symbolicTree = splitRead(st, readKey, predicate, exploringPath + [st.key])
   union.add(recUnion)
   if (predicate(utKey) and g != false)
     v = utKey.value
     connectSubtree(symbolicCaseTree, rec symbolicTree)
   union.add((g,v))
   else if (g != false)
     connectSubtreeByNode(symbolicCaseTree, reg, utKey, rec symbolicTree)
   else
     connectSubtree(symbolicCaseTree, rec symbolicTree)
 return union, symbolicCaseTree

func connectSubtree(tree, subtree):
 foreach (reg, (utKey, subtree)) in subtree:
   tree.Add(reg, (utKey, subtree))

func connectSubtreeByNode(tree, nodeReg, nodeUtKey, subtree):
 newUpdateTree = new()
 foreach (reg, (utKey, subtree)) in subtree:
   newUpdateTree.Add(reg, (utKey, subtree))
 tree.Add(nodeReg, (nodeUtKey, newUpdateTree))
```

Алгоритм

Результаты

Внедрение алгоритма разделения результатов чтения позволило

- выявить места, где в качестве результата чтения не ожидался Union;
- провести нагрузочное тестирование на виртуальную машину

На следующий семестр запланирована реализация модифицированного алгоритма записи.

В ходе первого семестра

1. изучена литература по моделированию символьной памяти
2. частично разработаны алгоритмы оптимизации символьной памяти
3. частично реализованы алгоритмы оптимизации символьной памяти