А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский

Учебный курс “CIL и системное программирование в Microsoft .NET”

Лекция 17.

Генерация линейных участков кода для стековой машины

Введение

- В этой лекции мы рассмотрим специфику генерации линейных участков кода на языке CIL
- Линейными участками мы будем называть участки кода, не содержащие развилок и защищенных блоков

17.1. Генерация кода для выражений

- Линейные участки кода получаются, главным образом, в процессе генерации кода для выражений
- Выражения бывают разные, и заранее неизвестно, какого именно типа выражения придется генерировать в различных задачах, связанных с динамической генерацией кода
- Поэтому мы рассмотрим некоторый тип выражений, наиболее часто встречающийся на практике, и покажем, как для выражений этого типа порождать линейные последовательности инструкций CIL

17.1.1. Абстрактный синтаксис выражений

- Пусть наши выражения оперируют с константами, переменными и параметрами
- Разрешим использование в выражениях как значений примитивных типов, так и объектов (в частности, массивов)
- В качестве операций мы будем использовать:
  - четыре арифметические бинарные операции
  - унарный минус
  - обращение к полю объекта
  - доступ к элементу массива
  - вызов экземплярного метода объекта

Правила БНФ для выражений

```
Expr       ::= const c | local x | arg x
             | Expr index Expr
             | Expr field f
             | minus Expr
             | Expr BinOp Expr
             | local x assign Expr
             | arg x assign Expr
             | Expr index Expr assign Expr
             | Expr field f assign Expr
             | Expr call s ArgList
ArgList    ::= Expr ArgList
             | пусто
BinaryOp   ::= plus | minus | mul | div.
```

17.1.2. Отображение абстрактного синтаксиса выражений в CIL

- Определим набор функций, отображающих различные поддеревья абстрактного синтаксиса в соответствующие им последовательности инструкций CIL
- Будем считать, что каждая функция принимает в качестве параметра дерево абстрактного синтаксиса (оно записывается в квадратных скобках) и возвращает последовательность инструкций (при этом запятые обозначают операцию объединения последовательностей)

Константы, переменные и параметры, поля объектов и элементы массивов

GenExpr[const c] = нужный вариант инструкции ldc

```
GenExpr[local x] = ldloc x;

GenExpr[arg x] = ldarg x;

GenExpr[Expr1 index Expr2] =
    GenExpr[Expr1],
    GenExpr[Expr2],
    ldelem.нужный тип;

GenExpr[Expr field f] =
    GenExpr[Expr],
    ldfld f;
```

Арифметические операции

```
GenExpr[minus Expr] =
    GenExpr[Expr],
    neg;

GenExpr[Expr1 BinOp Expr2] =
    GenExpr[Expr1],
    GenExpr[Expr2],
    GenBinOp[BinOp];

GenBinOp[plus] = add;
GenBinOp[minus] = sub;
GenBinOp[mul] = mul;
GenBinOp[div] = div;
```

Присваивание переменным и аргументам

```
GenExpr[local x assign Expr] =
    GenExpr[Expr],
    dup,
    stloc x;

GenExpr[arg x assign Expr] =
    GenExpr[Expr],
    dup,
    starg x;
```

Присваивание элементам массивов и полям объектов

```
GenExpr[Expr1 index Expr2 assign Expr3] =
    GenExpr[Expr1],
    GenExpr[Expr2],
    GenExpr[Expr3],
    dup,
    stloc временная переменная,
    stelem.нужный тип,
    ldloc временная переменная;
GenExpr[Expr1 field f assign Expr2] =
    GenExpr[Expr1],
    GenExpr[Expr2],
    dup,
    stloc временная переменная,
    stfld f,
    ldloc временная переменная;
```

Вызов методов

```
GenExpr[Expr call s ArgList] =
    GenExpr[Expr],
    GenArgList[ArgList],
    call(callvirt) s;

GenArgList[Expr ArgList] =
    GenExpr[Expr],
    GenArgList[ArgList];

GenArgList[пусто] = ;
```

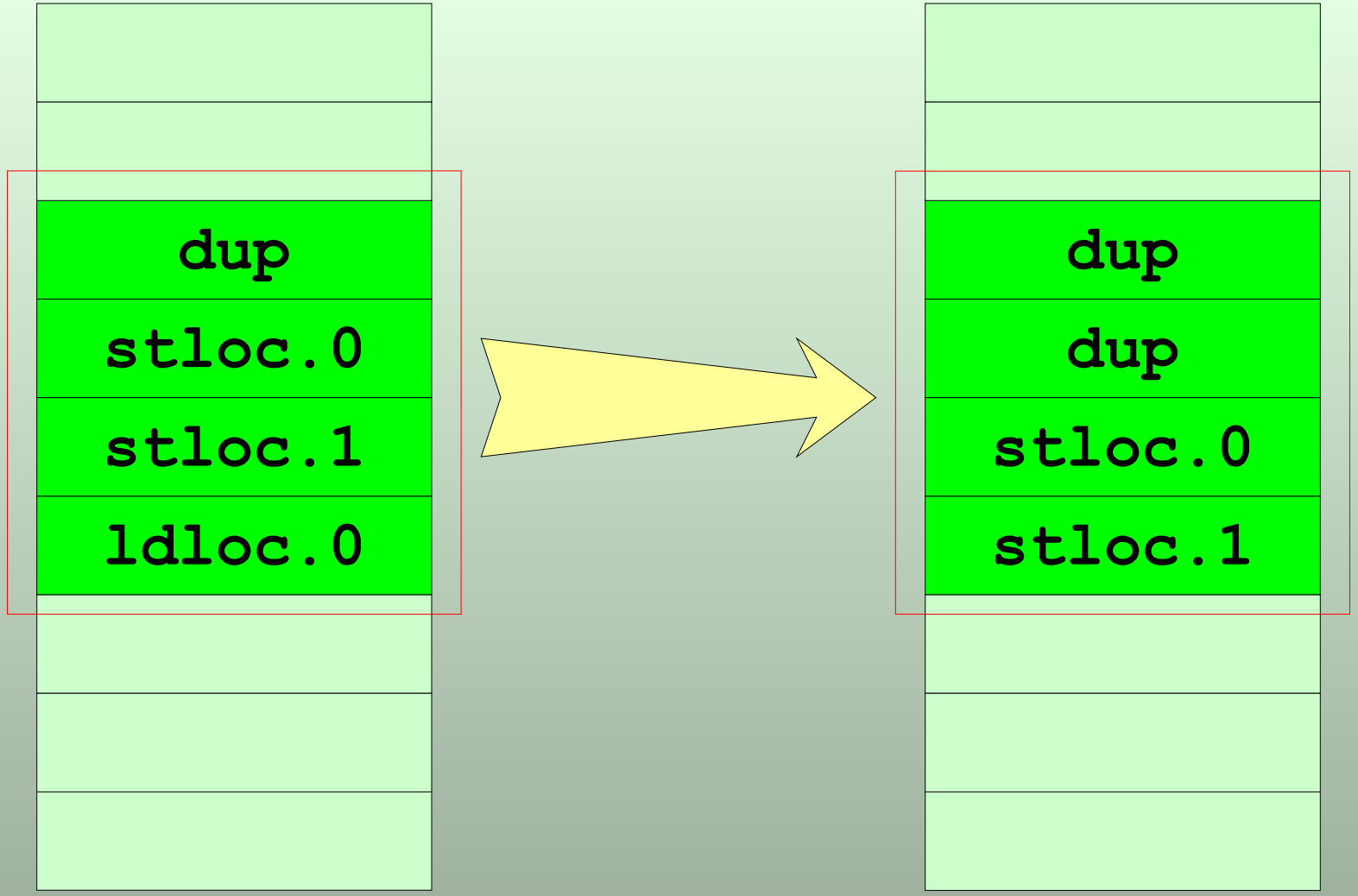
17.2. Оптимизация линейных участков кода

- Зачастую бывает удобно разделить фазы генерации и оптимизации кода. Это позволяет существенно упростить генератор, сделать его независимым от оптимизатора, и, кроме того, повторно использовать оптимизатор с другими генераторами
- Существует большое количество методов оптимизации, но в контексте динамической генерации кода, требующей быстрой работы оптимизатора, не все эти методы применимы. Поэтому мы рассмотрим один из самых простых методов – так называемую peephole-оптимизацию

Peephole-оптимизация (слайд 1)

- Суть peephole-оптимизации заключается в том, что оптимизатор ищет в коде метода сравнительно короткую последовательность инструкций, удовлетворяющую некоторому образцу, и заменяет ее более эффективной последовательностью инструкций
- Алгоритм peephole-оптимизации использует понятие фрейма. Фрейм можно представить как окошко, двигающееся по коду метода. Содержимое фрейма сравнивается с образцом, и в случае совпадения выполняется преобразование

Peephole-оптимизация (слайд 2)



Peephole-оптимизация (слайд 3)

- Peephole-оптимизация линейного участка кода должна выполняться многократно до тех пор, пока на очередном проходе фрейма по этому участку кода не будет найден ни один образец
- С другой стороны, алгоритм peephole-оптимизации может быть остановлен в любой момент, что позволяет добиться любой требуемой скорости работы оптимизатора

Некоторые образцы и замены для peephole-оптимизации CIL-кода (слайд 1)

Образец	Замена
`stloc(starg) x` `ldloc(ldarg) x`	`dup` `stloc(starg) x`
`ldloc (ldarg) x` `ldloc (ldarg) x`	`ldloc (ldarg) x` `dup`
`ckfinite` `ckfinite`	`ckfinite`
`not(neg)` `pop`	`pop`

Некоторые образцы и замены для peephole-оптимизации CIL-кода (слайд 2)

Образец	Замена
add(sub,mul,div,…) **pop**	**pop** **pop**
ldc.i4.0 **add(sub)**	**–**
ldloca(ldarga) x **initobj int32**	**ldc.i4.0** **stloc(starg) x**
stloc(starg) x **ldloc(ldarg) y** **ldloc(ldarg) x** **add(*любая коммутативная бинарная операция*)**	**dup** **stloc(starg) x** **ldloc(ldarg) y** **add(*любая коммутативная бинарная операция*)**