Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Российской академии наук

Ю.А. Климов

Специализатор CILPE:
генерация остаточной программы

Москва
2009

Ю.А. Климов

Специализатор CILPE: генерация остаточной программы

Аннотация

В методе частичных вычислений генератор остаточной программы (Residual Program Generator, RPG) по размеченной программе строит результат специализации. В данной работе описан генератор остаточной программы для размеченных программ на стековом объектно-ориентированном языке SOOL (Stack Object-Oriented Language). Описанный генератор остаточной программы используется в специализаторе CILPE.

Работа поддержана проектами РФФИ № 08-07-00280-а и № 09-01-00834-а.

Yu.A. Klimov

Specializer CILPE: residual program generation

Abstract

In a program specializer, the Residual Program Generator (RPG) takes as input a source program decorated with annotations provided by its binding time analyzer and produces a specialized version of the source program. CILPE is a program specializer dealing with programs written in SOOL, a simple Stack Object-Oriented Language. The paper describes the residual program generator that is a part of CILPE.

Содержание

1. Введение ... 4
2. Генератор остаточной программы ... 5
   2.1. Преобразование размеченной программы ... 6
   2.2. Состояние ... 6
   2.3. Остаточная программа ... 9
3. Обработка программы ... 10
4. Обработка метода ... 12
5. Обработка последовательности инструкций ... 14
   5.1. Правила обработки ... 15
   5.2. Обработка инструкции CallMethod^{X} mthd ... 17
6. Обработка инструкций ... 20
   6.1. S-инструкции ... 20
   6.2. D-инструкции ... 21
   6.3. X-инструкции ... 21
7. Заключение ... 25
8. Литература ... 25

1. Введение

Оптимизация программ на основе использования априорной информации о значении части переменных называется *специализацией*. Одним из широко используемых методов специализации является метод ч*астичных вычислений* (Partial Evaluation, PE) [Jone93].

Метод частичных вычислений основан на разделении операций и других программных конструкций на *статические* (S) и *динамические* (D). При этом понятие «статические операции и конструкции» не следует путать с понятием «статические методы и классы» (static), которое используется в языках программирования, например, Ява и C#.

В процессе частичных вычислений операции над известными данными исполняются, а над неизвестными — переносятся в остаточную программу. Остаточная программа зависит только от неизвестной (на стадии специализации) части аргументов.

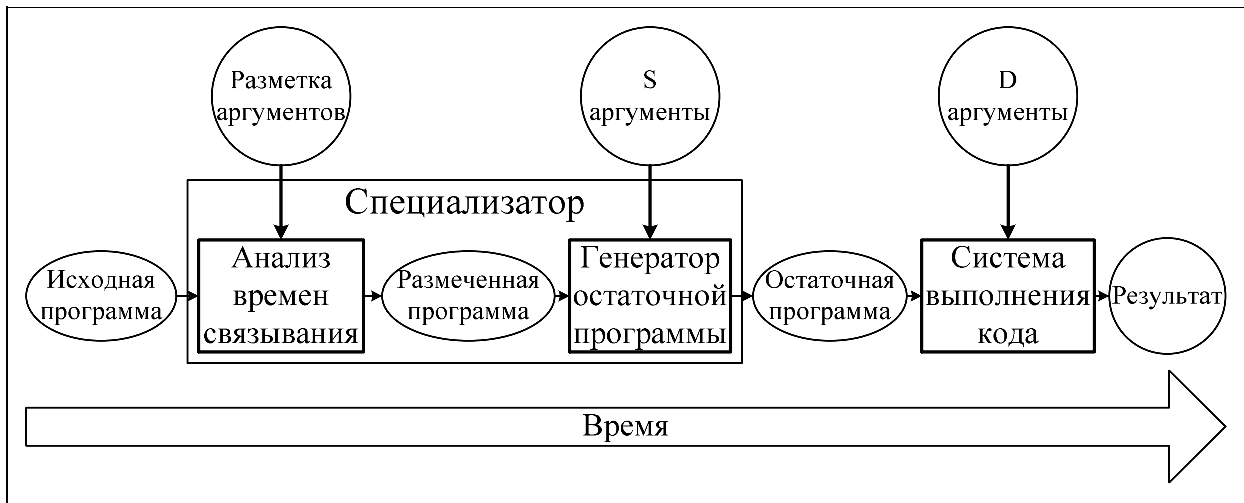

Рис. 1. Общепринятая структура специализатора, основанного на методе частичных вычислений.

Часть метода специализации, отвечающая за разделение операций и данных, называется *анализом времен связывания* (Binding Time Analysis, BTA, BT-анализ) (рис. 1). Вторая часть метода специализации, отвечающая за вычисление статической части программы и выделение динамической части в отдельную программу, называется *генератором остаточной программы* (Re-

sidual Program Generator, RPG). В этой части, собственно, и происходят *частичные вычисления*.

На основе метода частичных вычислений создан специализатор CILPE [Chep03,Klim08b] для языка CIL платформы Microsoft .NET [CLI,MS.NET], удовлетворяющий требованиям, изложенным в [Klim08a].

В данной работе формально описывается генератор остаточной программы [Klim06] для языка SOOL [Klim08c], который используется в специализаторе CILPE. Генератор принимает на вход размеченную программу [Klim09] и значения статических аргументов и строит остаточную программу.

2. Генератор остаточной программы

На последнем этапе частичных вычислений происходит генерация остаточной программы: по размеченной программе и значениям S-аргументов строится остаточная программа. На этом этапе происходят *частичные вычисления*: часть инструкций выполняется, а другая часть переходит в остаточную программу.

На вход генератору остаточной программы поступает размеченная программа [Klim09] и значения ее S-аргументов. По этим данным генератор остаточной программы строит новый метод (подпрограмму), аргументами которого являются только D-аргументы исходной программы, а также, если необходимо, вспомогательные методы для классов исходной программы.

Генератор остаточной программы производит обобщенное выполнение размеченной программы. Основной информацией для обобщенного выполнения является разметка инструкции: S-инструкции выполняются, D-инструкции как есть переходят в остаточную программу, а X-инструкции в преобразованном виде переходят в остаточную программу.

В отличие от BT-анализа [Klim09], правила которого, как и правила типизации, задают условия корректной разметки, правила генератора остаточной программы, как и правила интерпретации, однозначно предписывают необходимые вычисления.

2.1. Преобразование размеченной программы

На вход генератора остаточной программы поступают размеченная программа btProg и BT-куча btHeap.

В действительности генератору достаточно размеченной программы btProg (т.е. размеченных инструкций) практически во всех случаях за исключением двух: описание разметки аргументов и результатов метода и создание объекта. Для сокращения записи перенесем часть информации из btHeap в btProg (рис. 2). А именно:

1. В описании каждого метода изменим каждый BT-тип (type, btValue) на пару (type, btKind), где btKind = $GetBTKind_{btHeap}$(btValue).
2. В описании каждой инструкции $NewObject^{X}$ $class^{btOref}$ заменим описание BT-значения создаваемого объекта btOref на отображение btFld : Field → BTKind для полей объекта класса class, построенного следующим образом:

   btFld(fld) = $GetBTKind_{btHeap}$(btObj(fld)) where (_,_,btObj)=btHeap(btOref)

```
Для всех описаний аргументов и результатов:
f1 ((type, btValue)) = (type, GetBTKind_btHeap(btValue))

Для всех BT-инструкций NewObject^X class^btOref
f2 (NewObject^X class^btOref) = NewObject^X class^btFld
   where  btFld(fld) = GetBTKind_btHeap(btObj(fld))
              (_, _, btObj) = btHeap(btOref)
```

Рис. 2. Преобразование BT-программы.

В дальнейшем будем использовать только измененную BT-программу.

2.2. Состояние

Генератор остаточной программы производит частичные вычисления, поэтому его состояние близко к состоянию интерпретатора: оно состоит из состояния интерпретатора и отображения Pointer2Var (рис. 3).

Во время частичных вычислений часть данных вычислена, а часть неизвестна. В тех местах, в которых значение неизвестно (во время генерации остаточной программы), вместо обычного значения записывается специальное

значение DYNVALUE.

Отображение Pointer2Var в состоянии генератора остаточной программы показывает место, где неизвестная (во время генерации остаточной программы) часть данных будет находиться во время выполнения остаточной программы.

```
RPG-State = (State, Pointer2Var)

Value' = Value | DYNVALUE

Pointer2Var = Pointer → TypedVar

Pointer = Var | (Address, Field) | (Address, Integer)

TypedVar = (Type, Var)
```

Рис. 3. Состояние генератора остаточной программы.

Pointer2Var задает отображение указателей: имен переменных, полей объектов (пара адрес объекта и имя поля) и элементов массивов (пара адрес массива и номер элемента) в типизированные локальные переменные остаточной программы (значение отображения — тип и имя переменной). Это отображение пополняется при создании новых объектов и массивов и используется при обработке X-инструкций доступа к переменным, полям объектов и элементам массивов, а также при вызове методов. Для краткости тип и имя переменных (type, var) будем записывать в виде $\mathrm{var}^{\mathrm{type}}$.

Для создания и использования Pointer2Var будем использовать вспомогательные функции (рис. 4)

При вызове метода все D-части S-объектов или S-массивов должны быть переданы в генерируемый метод. Для этого используется функция Vars2Stack, которая генерирует инструкции загрузки на стек всех значений переменных, которые соответствуют D-значениям, доступным из аргументов.

После завершения вызова метода необходимо загрузить со стека все значения обратно в локальные переменные. Для этого используется функция Stack2Vars.

```
Vars2Stack : ([Value], Heap, Pointer2Var) → [RPG-Instruction]
Vars2Stack (arg, heap, ptr2var) =
                    [LoadVar (ptr2var ptr) | ptr ← FindDynValues(arg, heap)]

Stack2Vars : ([Value], Heap, Pointer2Var) → [RPG-Instruction]
Stack2Vars (arg, heap, ptr2var) =
              reverse [StoreVar (ptr2var ptr) | ptr ← FindDynValues(arg, heap)]

MkNewVars_prog : ([Value], Heap) → Pointer2Var
MkNewVars_prog (arg, heap) = [ptr↦var^type | ptr ← FindDynValues(arg, heap),
                         var^type — новая локальная переменная типа type,
                                            type = TypeOf_heap,prog(ptr)]

GetTypes_prog : ([Value], Heap) → [Type]
GetTypes_prog (arg, heap) = [TypeOf_heap,prog(ptr)] |
                                         ptr ← FindDynValues(arg, heap)]

AddVars :: ([VarDec], RPG-State) → RPG-State
AddVars (varDecs, (stack_1, env_1, heap_1), ptr2var_1)) =
                                     ((stack_1, env_2, heap_1), ptr2var_2)
   where  env_2 = env_1++[var↦DefValue(type) | (var, type) ← varDecs]++
          ptr2var_2 = ptr2var_1++[var↦var^type | (var, type) ← varDecs,
                               var^type — новая переменная типа type]

FindDynValues : ([Value], Heap) → [Pointers]
FindDynValues (arg, heap) = nub (map FindDynValues' arg)
   where  FindDynValues' (addr : Address) | heap(addr) == (type[], (length, arr)) =
                             [(addr, n) | 0 ≤ n < length, arr(n) == DYNVALUE]
          FindDynValues' (addr : Address) | heap(addr) == (type, obj) =
                        [(addr, fld) | fld ← Domain(obj), obj(fld) == DYNVALUE]
          FindDynValues' _ = []

TypeOf_heap,prog : Pointer → Type
TypeOf_heap,prog (addr, n) = type  where (type[], _) = heap(addr)
TypeOf_heap,prog (addr, fld) = FieldType_prog(fld)
```

Рис. 4. Вспомогательные функции для работы с Pointer2Var.

При обработке метода необходимо для D-значений завести новые переменные. Для этого используется функция $MkNewVars_{prog}$.

Функция $GetTypes_{prog}$ используется, чтобы собрать воедино типы D-полей S-объектов и D-элементов S-массивов.

Функция AddVars используется для пополнения состояния (окружения и отображения Pointer2Var) новыми локальными переменными.

2.3. *Остаточная программа*

При генерации остаточной программы последовательно строятся различные части программы. Для независимого построения частей программы тело метода будет немного изменено (рис. 5).

Во-первых, добавлена инструкция LABEL str (n, rpgState), описывающая метку и состояние генератора остаточной программы. Метка будет использоваться в инструкциях условного и безусловного перехода вместо номеров строк. А сохраненное состояние генератора остаточной программы будем использовать для сравнения обработанных состояний и нового состояния.

Во-вторых, вместо инструкций Goto n и Branch n с параметром целое число, используются инструкции Goto str и Branch str с параметром строка.

В-третьих, вместо инструкций LoadVar var и StoreVar var с параметром имя переменной используются инструкции LoadVar (var, type) и StoreVar (var, type) с параметрами тип и имя переменной. Для краткости эти инструкции будем записывать в виде LoadVar var^{type} и StoreVar var^{type}.

```
MethodBody = ([VarDec], [RPG-Instruction])

RPG-Instruction = Instruction | Goto String | Branch String |
      LoadVar (VarName, Type) | StoreVar (VarType, VarName) |
                                   Label String (N, RPG-State)
N = Integer
```

Рис. 5. Тело генерируемого метода.

При генерации тела метода необходимо проверять, встречалось ли данное состояние ранее. Это производится с помощью функции FindState (рис. 6).. Если состояние ранее не встречалось, то возвращается NOTHING, а если встречалось — возвращается тело метода и инструкция безусловного перехода.

При добавлении новых инструкций, необходимо также запомнить со-

стояние. Для этого используется функция AddInstrs.

После завершения генерации, по [RPG-Instrution] восстанавливается тело метода с помощью функции MkMethodBody: удаляются все инструкции Label, а инструкции Goto, Branch, LoadVar и StoreVar приводятся к виду, как описано в языке SOOL.

```
FindState : ([RPG-Instruction], Integer, RPG-State) → Maybe [RPG-Instruction]
FindState (rpgInstrs, n, (state, _)) =
   case [str | Label str (n, (state', _)) ← rpgInstrs, state == state'] of
      []        → NOTHING
      str::_    → JUST (rpgInstrs++[Goto str])

AddInstrs : ([RPG-Instruction], Integer, RPG-State, [RPG-Instruction]) →
                                                            [RPG-Instruction]
AddInstrs (rpgInstrs, n, rpgState, []) = rpgInstrs
AddInstrs (rpgInstrs, n, rpgState, rpgInstrs') = rpgInstrs++
                                        [Label str (n, rpgState)]++rpgInstrs'
   where str — имя новой метки

MkMethodBody : [RPG-Instruction] → MethodBody
MkMethodBody rpgInstrs = (vars, instrs)
   where  vars = nub ([(var, type) | LoadVar (var, type) ← rpgInstrs]++
                          [(var, type) | StoreVar (var, type) ← rpgInstrs])
          instrs = concatMap simplInstr rpgInstrs
          simplInstr (Label _ _) = []
          simplInstr (LoadVar (var, type)) = [LoadVar var]
          simplInstr (StoreVar (var, type)) = [StoreVar var]
          simplInstr (Goto str) = [Goto (getNumber str rpgInstrs)]
          simplInstr (Branch str) = [Branch (getNumber str rpgInstrs)]
          getNumber str (Label str _)::_ = 0
          getNumber str (Label _ _)::rpgInstrs = getNumber str rpgInstrs
          getNumber str _::rpgInstrs = 1+(getNumber str rpgInstrs)
```

Рис. 6. Вспомогательные функции для генерации тела метода.

3. Обработка программы

Обработка размеченной программы, как и в случае интерпретатора, начинается с обработки начальной последовательности инструкций btInstrs (рис. 7). Объект, у которого вызывается метод Main, размечен D, и инструкция создания перейдет в остаточную программу. Вызов метода Main —

NOINLINE. Поэтому в результате будет сгенерирован новый метод Main, зависящий только от D-аргументов.

В результате обработки начального набора инструкций или методов возвращается построенный метод и список методов, которые необходимо построить. Этот список возникает из NOINLINE вызовов методов внутри обрабатываемого метода.

Разметка NOINLINE методов требует, чтобы все возвращаемые значения (как через результаты, так и через аргументы) были размечены D. Поэтому такие вызовы при генерации остаточной программы значений не возвращают и не изменяют текущее состояние, следовательно можно продолжить генерацию текущего метода, отложив генерацию вызываемого метода.

btInstrs = [NewObject^D MAIN, CallMethod^X Main, Leave^X]
rgpState = ((sArg, [], []), [])
btProg $\vdash_{rpg}$ (btInstrs, 0) : (rpgState, []) $\Rightarrow$ (_, $mthds_2$)
btProg' = btProg
Пока $mthds_2$ не пусто:
  ($mthd_2$, mthdArg) = head $mthds_2$
  1. Если первый аргумент S, то
    (oref::_, heap) = mthdArg
    (class, _) = head(oref)
    $mthdDef_2$ = MethodBTDefinition($mthd_2$, class)
  2. Если первый аргумент D, то для всех определений $mthdDef_2$
    методов с именем $mthd_2$
  Если метод $mthdDef_2$ с аргументами mthdArg еще не обрабатывался
    btProg $\vdash_{rpg}$ $mthdDef_2$: mthdArg $\Rightarrow$ ($mthdDef_3$, $mthds_3$)
    $mthds_2$= (tail $mthds_2$)++$mthds_3$
    добавить метод $mthdDef_3$ в программу btProg'
  иначе
    $mthds_2$= tail $mthds_2$

$\vdash_{rpg}$ btProg : sArg $\Rightarrow$ btProg'

Рис. 7. Правило обработки программы.

В правиле обработки программы написан цикл, последовательно обрабатывающий методы. В результате обработки одного исходного метода получается метод остаточной программы и список методов, которые необходимо обработать.

Методы, которые необходимо обработать, задаются именем метода и значениями S-аргументов. Перед генерацией проверяется, не обрабатывался ли данный метод с данными значениями S-аргументов прежде. Если уже обрабатывался, то этот метод повторно не обрабатывается, и мы переходим к следующему методу.

Если первый аргумент имеет разметку S, то обрабатывается метод, соответствующий типу этого аргумента. Если имеет разметку D, то обрабатываются все методы с данным именем (нельзя определить точный тип первого метода, поэтому неизвестно, какой метод необходимо обработать).

Когда список методов на обработку пуст, генерация остаточной программы завершается.

Для построения имен новых методов используется функция MakeName (рис. 8), которая по имени метода и его статическим аргументам строит новое имя. Эта функция для разных аргументов должна выдавать разные имена, а для одинаковых — одинаковые.

MakeName :: (String, ([Value], Heap)) → String

Рис. 8. Вспомогательная функция для обработки методов.

4. Обработка метода

Обработка метода генератором остаточной программы начинается так же, как и интерпретатором: созданием начального состояния (рис. 9). Но в отличие от интерпретатора, генератору остаточной программы необходимо сгенерировать инструкции для передачи D-данных через стек. В исходной программе эти D-данные передавались по ссылке: как поля объектов или элементы массивов.

Для этого используются функции $MkNewVars_{prog}$ и Stack2Vars. Функция $MkNewVars_{prog}$ по аргументам arg и куче heap, находит все D-поля объектов и D-элементы массивов. Затем для каждого такого поля или элемента генерирует переменную и строит отображение $ptr2var_1$ соответствующего указателя на эту переменную. Функция Stack2Vars загружает со стека значения в

переменные, построенные функцией $\text{MkNewVars}_{\text{btProg}}$.

Затем инициализируется окружение с помощью функции AddVar и обрабатывается тело метода.

$$\text{(mthd, _, _, _, (varDecs, btInstrs))} = \text{mthdDef}$$

$$\text{ptr2var}_1 = \text{MkNewVars}_{\text{btProg}}(\text{arg}_1, \text{heap}_1)$$
$$\text{rpgState}_1 = ((\text{arg}_1, [\,], \text{heap}_1), \text{ptr2var}_1)$$
$$\text{rpgInstrs}_1 = \text{Stack2Vars}(\text{arg}_1, \text{heap}_1, \text{ptr2var}_1)$$

$$\text{rpgState}_2 = \text{AddVars}(\text{varDecs}, \text{rpgState}_1)$$

$$\text{prog} \vdash_{\text{rpg}} (\text{btInstrs}, 0) : (\text{rpgState}_2, \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_3, \text{mthds})$$

$$\text{rpgInstrs}_4 = \text{AddBeforeLeave}(\text{rpgInstrs}_3, \text{Vars2Stack}(\text{arg}_1, \text{heap}_1, \text{ptr2var}_1))$$

$$\text{mthd}' = \text{MakeName}(\text{mthd}, (\text{arg}_1, \text{heap}_1))$$
$$\text{tArg} = [\text{type} \mid (\text{type}, \text{D}) \leftarrow \text{btArg}]\text{++}\text{GetTypes}_{\text{btProg}}(\text{arg}_1, \text{heap}_1)$$
$$\text{tRes} = [\text{type} \mid (\text{type}, \text{D}) \leftarrow \text{btRes}]\text{++}\text{GetTypes}_{\text{btProg}}(\text{arg}_1, \text{heap}_1)$$
$$\text{mthdBody}' = \text{MkMethodBody}(\text{rpgInstrs}_4)$$
$$\text{mthdDef}' = (\text{mthd}', \text{tArg}, \text{tRes}, \text{mthdBody}')$$

$$\overline{\text{btProg} \vdash_{\text{rpg}} \text{mthdDef} : (\text{arg}_1, \text{heap}_1) \Rightarrow (\text{mthdDef}', \text{mthds})}$$

Рис. 9. Правило обработки метода.

По завершению обработки тело метода модифицируется (вызов функции AddBeforeLeave, рис. 10) — перед каждой инструкцией Leave вставляются инструкции загрузки переменных на стек. Это необходимо для возвращения данных через стек, которые в исходной программе передавались по ссылке через D-поля S-объектов или D-элементы S-массивов.

```
AddBeforeLeave :: ([RPG-Instructions], [RPG-Instructions]) → [RPG-Instructions]
AddBeforeLeave (rgpInstrs, instrs) = AddBeforeLeave' rgpInstrs
   where  AddBeforeLeave' [] = []
          AddBeforeLeave' Leave::rgpInstrs = instrs++Leave::
                                                  (AddBeforeLeave' rgpInstrs)
          AddBeforeLeave' instr::rgpInstrs = instr::(AddBeforeLeave' rgpInstrs)
```

Рис. 10. Вспомогательная функция для обработки метода.

Затем строится сигнатура метода. Типы метода получаются из типов D-аргументов исходного метода добавлением типов переменных, созданных

функцией $MkNewVars_{btProg}$. Типы результатов также получаются из типов результатов исходного метода (они все D) и из типов переменных, созданных функцией $MkNewVars_{btProg}$. Объявления локальных переменных метода восстанавливаются функцией MkMethodBody по инструкциям чтения/записи переменных в теле метода.

5. Обработка последовательности инструкций

При обработке инструкций для каждого метода для обеспечения конечности программы используются метки Label str (n, rpgState) с записанными номерами инструкций исходной программы n и состоянием генератора остаточной программы rpgState, при котором эта инструкция была сгенерирована.

При обработке инструкции проверяется, было ли такое состояние генератора остаточной программы при обработке этой же инструкции ранее с помощью функции FindState (рис. 11).

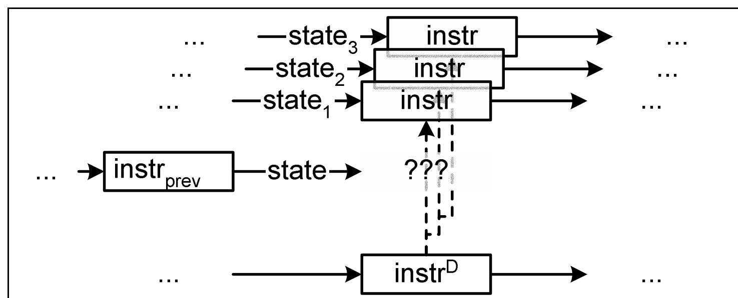

Рис. 11. Процесс генерации остаточной программы: D-инструкция.

Если такая ситуация встречалась раньше, то генерируется переход на инструкцию, которая была сгенерирована в аналогичной предыдущей ситуации (рис. 12).

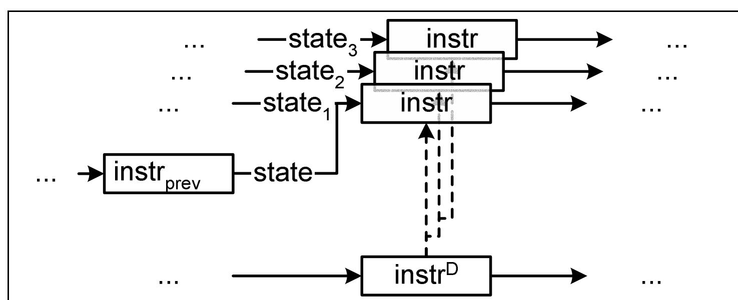

Рис. 12. Процесс генерации остаточной программы: состояние повторилось.

Если такая ситуация ранее не встречалась, то инструкция обрабатывается по ниже приведенным правилам, а в генерируемое тело метода добавляется инструкция Label с состоянием генератора остаточной программы (рис. 13).

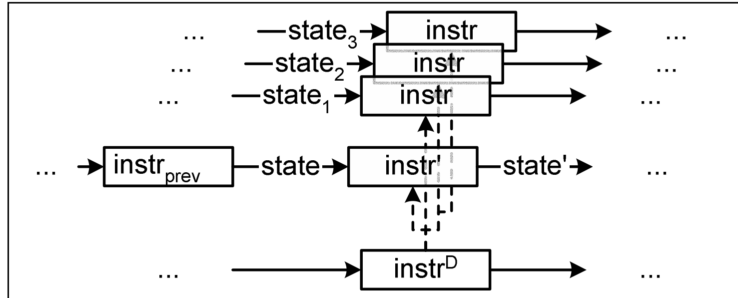

Рис. 13. Процесс генерации остаточной программы: новое состояние.

5.1. *Правила обработки*

При обработке инструкции сначала проверяется, обрабатывалась ли данная инструкция при том же состоянии генератора остаточной программы. Данная проверка делается вызовом функции FindState. Если такая инструкция уже обрабатывалась, то вызов возвращает список инструкций — результат генерации остаточной программы (рис. 14).

$$\frac{\text{JUST rpgInstrs}_2 = \text{FindState}(\text{rpgInstrs}_1, \text{n}, \text{rpgState})}{\text{btProg} \vdash_{\text{rpg}} (\text{btInstrs}, \text{n}) : (\text{rpgState}, \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_2\ [])}$$

Рис. 14. Обработка последовательности инструкций, отображение π(n, state) определено.

Если обработка еще не проводилась (то есть функция FindState вернула NOTHING), то в зависимости от инструкции, обработка происходит следующим образом.

1. Если n-ная инструкция в списке instrs — это инструкция Leave^X, то генерируется инструкция Leave (рис. 15).

$$\frac{\text{Leave}^X = \text{btInstrs}[\text{n}] \qquad \text{NOTHING} = \text{FindState}(\text{rpgInstrs}_1, \text{n}, \text{rpgState}) \quad \text{rpgInstrs}_2 = \text{AddInstrs}(\text{rpgInstrs}_1, \text{n}, \text{rpgState}, [\text{Leave}])}{\text{btProg} \vdash_{\text{rpg}} (\text{btInstrs}, \text{n}) : (\text{rpgState}, \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_2, [])}$$

Рис. 15. Обработка последовательности инструкций, инструкция Leave^X.

2. Если n-ная инструкция в списке instrs — это инструкция S-инструкция Goto^S m, то выполнение продолжается для m-ной инструкции в списке instrs (рис. 16).

$$\frac{\text{Goto}^S\ m = \text{btInstrs}[n] \qquad \text{NOTHING} = \text{FindState}(\text{rpgInstrs}_1, n, \text{rpgState}) \qquad \text{btProg} \vdash_{rpg} (\text{btInstrs}, m) : (\text{rpgState}, \text{instrs}_1, \pi_1) \Rightarrow (\text{instrs}_2, \pi_2, \text{mthds}_2)}{\text{btProg} \vdash_{rpg} (\text{btInstrs}, n) : (\text{rpgState}, \text{instrs}_1, \pi_1) \Rightarrow (\text{instrs}_2, \pi_2, \text{mthds}_2)}$$

Рис. 16. Обработка последовательности инструкций, инструкция Goto^S m.

3. Если n-ная инструкция в списке instrs — это инструкция Branch^S m, то проверяется значение на вершине стека (рис. 17). Это значение должно быть целым числом. Если оно равно 0, то выполнение продолжается для (n+1)-ой инструкции, иначе — для m-ной.

$$\frac{\begin{array}{c}\text{Branch}^S\ m = \text{btInstrs}[n] \qquad \text{NOTHING} = \text{FindState}(\text{rpgInstrs}_1, n, \text{rpgState}) \\ ((\text{val::stack}, \text{env}, \text{heap}), \text{ptr2var}) = \text{rpgState}, \text{где val — целое число} \\ \text{rpgState}' = ((\text{stack}, \text{env}, \text{heap}), \text{ptr2var}) \qquad k = \text{if val==0 then n+1 else m} \\ \text{btProg} \vdash_{rpg} (\text{btInstrs}, k) : (\text{rpgState}', \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_2, \text{mthds}_2)\end{array}}{\text{btProg} \vdash_{rpg} (\text{btInstrs}, n) : (\text{rpgState}, \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_2, \text{mthds}_2)}$$

Рис. 17. Обработка последовательности инструкций, инструкция Branch^S m.

4. Если n-ная инструкция в списке instrs — это инструкция Branch^D m, то генерируется инструкция условного перехода Branch str (str — имя новой метки) и последовательно обрабатываются две ветви (рис. 18).

$$\frac{\begin{array}{c}\text{Branch}^D\ m = \text{btInstrs}[n] \qquad \text{NOTHING} = \text{FindState}(\text{rpgInstrs}_1, n, \text{rpgState}) \\ \text{rpgInstrs}_2 = \text{AddInstrs}(\text{rpgInstrs}_1, n, \text{rpgState}, [\text{Branch str}]) \\ \text{str — имя новой метки} \\ \text{btProg} \vdash_{rpg} (\text{btInstrs}, n+1) : (\text{rpgState}, \text{rpgInstrs}_2) \Rightarrow (\text{rpgInstrs}_3, \text{mthds}_3) \\ \text{rpgInstrs}'_3 = \text{rpgInstrs}_3\text{++}[\text{Label str _}] \\ \text{btProg} \vdash_{rpg} (\text{btInstrs}, m) : (\text{rpgState}, \text{rpgInstrs}'_3) \Rightarrow (\text{rpgInstrs}_4, \text{mthds}_4)\end{array}}{\text{btProg} \vdash_{rpg} (\text{btInstrs}, n) : (\text{rpgState}, \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_4, \text{mthds}_2\text{++}\text{mthds}_4)}$$

Рис. 18. Обработка последовательности инструкций, инструкция Branch^D m.

5. Если n-ная инструкция в списке instrs — это инструкция вызова метода CallMethod^x mthd, то она обрабатывается специальным образом, который описан ниже.
6. В остальных случаях, применяется правило обработки одной n-ой ин-

струкции, и далее, начиная с нового состояния, продолжается обработка последовательности инструкций для (n+1)-ой инструкции (рис. 19).

$$\frac{\begin{array}{c}\text{NOTHING} = \text{FindState}(\text{rpgInstrs}_1, \text{n}, \text{rpgState}) \\ \text{btProg} \vdash_{\text{rpg}} \text{btInstrs[n]} : \text{rpgState}_1 \rightarrow (\text{rpgState}_2, \text{instrs}) \\ \text{rpgInstrs}_2 = \text{AddInstrs}(\text{rpgInstrs}_1, \text{n}, \text{rpgState}_1, \text{instrs}) \\ \text{btProg} \vdash_{\text{rpg}} (\text{btInstrs}, \text{n+1}) : (\text{rpgState}_2, \text{rpgInstrs}_2) \Rightarrow (\text{rpgInstrs}_3, \text{mthds}_3)\end{array}}{\text{btProg} \vdash_{\text{rpg}} (\text{btInstrs}, \text{n}) : (\text{rpgState}_1, \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_3, \text{mthds}_3)}$$

Рис. 19. Обработка последовательности инструкций.

5.2. Обработка инструкции CallMethod^X mthd

INLINE метод

Если метод Inline, т.е. он должен быть раскрыт при генерации остаточной программы, то обработка происходит следующим образом (рис. 20, 21):

1. Со стека текущего состояния достается oref — ссылка на объект, вычисляется тип объекта, и по этому типу определяется вызываемый метод;
2. В состояние генератора остаточной программы добавляется информация про локальные переменные вызываемого метода: в окружение добавляется значение по умолчанию для этих переменных, а в отображение ptr2var — сопоставление этих переменных новым локальным переменным в остаточной программе.
3. Далее вызываемый метод обрабатывается обычным образом.
4. По завершению обработки вызываемого метода, необходимо восстановить состояние, при котором происходила генерация инструкций Leave. Заменить инструкции Leave на инструкции безусловного перехода на новую метку. Запомнить пары состояние-имя метки.
5. Для каждой пары состояние-имя метки продолжить обработку текущего метода с состояния, при котором генерировалась инструкция Leave (удалив перед этим в окружении информацию про локальные переменные вызываемого метода).

CallMethod^X mthd = btInstrs[n]
NOTHING = FindState($rpgInstrs_1$, n, rpgState)

((oref::_, _, _), _) = $rpgState_1$ oref — ссылка на объект
(class, _) = heap(oref)

(_, INLINE, _, _, (varDecs, btInstrs')) = $MethodBTDefinition_{btProg}$(mthd, class)
$rpgState_2$ = AddVars(varDecs, $rpgState_1$)
prog $\vdash_{rpg}$ (btInstrs', 0) : ($rpgState_2$, []) $\Rightarrow$ ($rpgInstrs_3$, $mthds_3$)

($rpgInstrs_4$, $rpgStates_4$) = ReplaceLeave($rpgInstrs_3$)
$rpgInstrs_5$ = AddInstrs($rpgInstrs_1$, n, rpgState, $rpgInstrs_4$)

$rpgInstrs_6$ = $rpgInstrs_5$
$mthds_6$ = $mthds_3$
Для каждой пары (str_4, $rpgState_4$) из $rpgStates_4$:
  $rpgState_5$ = RemoveVars(varDecs, $rpgState_4$)
  btProg $\vdash_{rpg}$ (btInstrs, n+1) : ($rpgState_5$, $rpgInstrs_6$++[Label str_4 _]) $\Rightarrow$
    ($rpgInstrs_7$, $mthds_7$)
  $rpgInstrs_6$ = $rpgInstrs_7$
  $mthds_6$ = $mthds_6$++$mthds_7$

---

btProg $\vdash_{rpg}$ (btInstrs, n) : ($rpgState_1$, $rpgInstrs_1$) $\Rightarrow$ ($rpgInstrs_7$, $mthds_7$)

Рис. 20. Обработка последовательности инструкций, инструкция CallMethod^X mthd, INLINE.

RemoveVars :: ([VarDec], RPG-State) → RPG-State
RemoveVars (varDecs, ($stack_1$, env_1, $heap_1$), $ptr2var_1$)) =
  (($stack_1$, env_2, $heap_1$), $ptr2var_2$)
  where env_2 = filter p env_1
    $ptr2var_2$ = filter p env_1 $ptr2var_1$
    p (var↦_) = var \`notelem\` (map fst varDecs))

ReplaceLeave :: [RPG-Instruction] → ([RPG-Instruction], [(String, RPG-State)])
ReplaceLeave [] = ([], [])
ReplaceLeave (Label str rpgState)::(Leave)::$rpgInstrs_1$ = ($rpgInstrs_3$, $rgpStates_3$)
  where $rpgInstrs_3$ = (Label str rpgState)::(Goto str')::$rpgInstrs_2$
    $rgpStates_3$ = (str', rpgState)::$rgpStates_2$
    ($rpgInstrs_2$, $rgpStates_2$) = ReplaceLeave $rpgInstrs_1$
    str' — новая метка

Рис. 21. Вспомогательные функции для обработки последовательности инструкций, инструкция CallMethod^X mthd, INLINE.

NOINLINE метод

Если вызываемый метод NOINLINE, то разметка аргументов и результатов метода такая, что дальнейшая обработка исходного метода возможна без обработки вызываемого метода. Т.е. все результаты и все поля S-аргументов размечены D.

Метод может возвращать значения и через поля объектов-аргументов. D-полям S-аргументов в остаточной программе соответствуют переменные, значения которых необходимо передать новому методу, а после выполнения интерпретатором остаточного нового метода, необходимо записать эти значения обратно в переменные.

CallMethod^X mthd = btInstrs[n]
NOTHING = FindState(rpgInstrs$_1$, n, rpgState)

(NOINLINE, btArg, _) = MethodBTSignature$_{btProg}$(mthd)
((arg$_1$++stack$_1$, env$_1$, heap$_1$), ptr2var$_1$) = rpgState$_1$
Length arg$_1$ = Length (filter (S ==) (map snd btArg))
rpgState$_2$ = ((stack$_1$, env$_1$, heap$_1$), ptr2var$_1$)

mthdArg = (arg$_1$, heap$_1$)
mthd' = (mthd, mthdArg)

instrs = Vars2Stack(arg$_1$, heap$_1$, ptr2var$_1$)++[CallMethod MakeName(mthd')]++ Stack2Vars(arg$_1$, heap$_1$, ptr2var$_1$)
rpgInstrs$_2$ = AddInstrs(rpgInstrs$_1$, n, rpgState$_1$, instrs)
btProg $\vdash_{rpg}$ (btInstrs, n+1) : (rpgState$_2$, rpgInstrs$_2$) $\Rightarrow$ (rpgInstrs$_3$, mthds$_3$)

---

btProg $\vdash_{rpg}$ (btInstrs, n) : (rpgState$_1$, rpgInstrs$_1$) $\Rightarrow$ (rpgInstrs$_3$, mthd'::mthds$_3$)

Рис. 22. Обработка последовательности инструкций,
инструкция CallMethod^X mthd, NOINLINE.

Поэтому для обработки NOINLINE инструкции CallMethod^X mthd необходимо (рис. 22):

1. Снять со стека статические аргументы метода.
2. По этим аргументам построить запрос на специализацию метода mthd с указанными аргументами.
3. Сгенерировать инструкции загрузки на стек переменных, соответст-

вующих D-полям S-объектов и D-элементам S-массивов, которые передаются методу (вызов функции Vars2Stack).

4. Сгенерировать инструкцию вызова метода CallMethod mthd'.
5. Сгенерировать инструкции записи значений со стека обратно в переменные (вызов функции Stack2Vars).
6. Продолжить обработку последовательности инструкций

Вызываемый метод будет обработан в цикле обработки методов. Причем если вызов виртуальный, т.е. на этапе генерации остаточной программы нельзя определить, какой конкретно метод будет вызван, то обработке будут подвержены все возможные методы, а методы-результаты таких обработок будут образовывать новые виртуальные методы, на которые возможен переход в сгенерированной инструкции.

6. Обработка инструкций

Для описания правил обработки отдельных инструкций используются сокращенные правила (рис. 23).

$$\text{context} \vdash_{rpg} \text{instruction} : \text{rpgState}_1 \rightarrow (\text{rpgState}_2, \text{instructions})$$

Рис. 23. Правило обработки инструкции.

Данные правила описывают преобразования RPG-состояния и список инструкций, которые должны перейти в остаточную программу.

6.1. S-инструкции

S-инструкции — это те инструкции, которые должны быть вычислены генератором остаточной программы.

Такие инструкции зависят только от S-данных, поэтому эти данные будут вычислены при выполнении предыдущих инструкций генератором остаточной программы. И S-инструкция может быть вычислена во время генерации остаточной программы. Результатом S-инструкции также являются S-данные.

S-инструкцией может быть любая инструкция языка SOOL, за исклю-

чением инструкции конца метода Leave, инструкции создания объекта NewObject и инструкции вызова метода CallMethod.

Обработка S-инструкции заключается в интерпретации этой инструкции (рис. 24).

$$\frac{\text{btProg} \vdash_{\text{i}} \text{instr} : \text{state}_1 \rightarrow \text{state}_2}{\text{btProg} \vdash_{\text{rpg}} \text{instr}^{\text{S}} : (\text{state}_1, \text{ptr2var}) \rightarrow ((\text{state}_2, \text{ptr2var}), [])}$$

Рис. 24. Правило обработки S-инструкций.

6.2. D-инструкции

D-инструкции не преобразуются генератором остаточной программы, а без изменения переносятся в остаточную программу.

Такие инструкции зависят только от D-данных, и результатом являются тоже только D-данные, поэтому перенос такой инструкции в остаточную программу разрешен (рис. 25).

$$\text{btProg} \vdash_{\text{rpg}} \text{instr}^{\text{D}} : \text{rpgState} \rightarrow (\text{rpgState}, [\text{instr}])$$

Рис. 25. Правило обработки D-инструкций.

D-инструкцией может быть любая инструкция языка SOOL, за исключением инструкции конца метода Leave, инструкции безусловного перехода Goto m и инструкции вызова метода CallMethod.

6.3. X-инструкции

Если инструкция размечена X, то она специальным образом обрабатывается генератором остаточной программы. X-инструкции можно разделить на четыре группы: инструкции создания (NewObject и NewArray), инструкции доступа (LoadVar, StoreVar, LoadField, StoreField, LoadElement и StoreElement), инструкция Lifting и инструкции передачи управления (CallMethod и Leave). Инструкции передачи управления рассмотрены выше. Рассмотрим оставшиеся три группы.

Инструкции создания

Инструкции NewObject и NewArray создают новый объект и новый массив соответственно. Если эти инструкции размечены X, то значит созда-

ваемый объект или массив размечен S, а его поля (или часть полей) или элементы размечены как D.

Если поля или элементы размечены как D, то специализатор в остаточной программе должен сгенерировать локальные переменные для каждого такого поля или элемента и инициировать их (рис. 26, 27). А в последствии заменять инструкции доступа к этим полям или элементам на инструкции доступа к локальным переменным.

$$\frac{\begin{array}{l} oref \text{ — новый адрес} \\ obj = [fld \mapsto value \mid (fld, type) \leftarrow GetFieldDecs_{btProg}(class), \\ \qquad value = \text{if } btFld(fld) == S \text{ then } DefaultValue(type) \text{ else } DYNVALUE] \\ \qquad\qquad \text{— новый объект класса } class \\ heap' = heap[oref \mapsto (class, obj)] \\ ptr2var' = ptr2var[(oref, fld) \mapsto var_{fld}^{type} \mid (fld, type) \leftarrow GetFieldDecs_{btProg}(class), \\ \qquad btFld(fld) = D, var_{fld}^{type} \text{ — новая переменная типа } type] \end{array}}{\begin{array}{l} btProg \vdash_{rpg} NewObject^{X}\ class^{btFld} : ((stack, env, heap), ptr2var) \rightarrow \\ \qquad (((oref::stack, env, heap'), ptr2var'), []) \end{array}}$$

Рис. 26. Правило специализации инструкции $NewObject^{X}$ $class^{btFld}$.

$$\frac{\begin{array}{l} aref \text{ — новый адрес} \\ arr = [i \mapsto DYNVALUE \mid i \leftarrow [0..n-1]] \text{ — новый массив} \\ heap' = heap[aref \mapsto (type[], (length, arr))] \\ ptr2var' = ptr2var[(aref, i) \mapsto var_{i}^{type} \mid i \leftarrow [0..n-1], \\ \qquad var_{i}^{type} \text{ — новая переменная типа } type] \end{array}}{\begin{array}{l} btProg \vdash_{rpg} NewArray^{X}\ type : ((n::stack, env, heap), ptr2var) \rightarrow \\ \qquad (((aref::stack, env, heap'), ptr2var'), []) \end{array}}$$

Рис. 27. Правило специализации инструкции $NewArray^{X}$ type.

При специализации инструкции $NewObject^{X}$ генератор остаточной программы, во-первых, выполняет эту инструкцию: создает объект obj класса class. И, во-вторых, для каждого поля fld класса class смотрит на BT-значение этого поля в BT-классе $btClass^{s}$. Если BT-значение D, то создается новая локальная переменная остаточной программы var_{fld}^{type}, а указатель на это поле (object, field) заносится в отображение ptr2var и связывается с новой локальной переменной var_{fld}^{type}.

При специализации инструкции $NewArray^{X}$ генератор остаточной про-

граммы, как и в случае инструкции $NewObject^X$, во-первых, выполняет эту инструкцию и создает массив array с элементами типа type. И, во-вторых, для каждого индекса элемента массива i заводится новая локальная переменная остаточной программы var_i^{type}, а указатель на этот элемент (array, i) заносится в отображение ptr2var и связывается с новой локальной переменной var_i^{type}.

Инструкции доступа

Если инструкция доступа размечена X, то она обращается к D-полю S-объекта, D-элементу S-массива или к D локальной переменной исходного метода. Генератор остаточной программы все D-поля S-объектов, D-элементы S-массивов или D локальные переменные исходного метода преобразует в локальные переменные остаточного метода. Поэтому такие инструкции преобразуются в операции доступа к локальной переменной остаточного метода.

Инструкции LoadVar и StoreVar читают и записывают в локальную переменную исходной программы. Эти инструкции заменяются инструкциями LoadVar или StoreVar, соответственно, для переменной остаточной программы (рис. 28, 29). Эта переменная определяется по переменной исходной программы и отображению ptr2var.

$$\text{btProg} \vdash_{rpg} \text{LoadVar}^X\ \text{var} : (\text{state}, \text{ptr2var}) \rightarrow \\ ((\text{state}, \text{ptr2var}), [\text{LoadVar ptr2var(var)}])$$

Рис. 28. Правило специализации инструкции $LoadVar^X$ var.

$$\text{btProg} \vdash_{rpg} \text{StoreVar}^X\ \text{var} : (\text{state}, \text{ptr2var}) \rightarrow \\ ((\text{state}, \text{ptr2var}), [\text{StoreVar ptr2var(var)}])$$

Рис. 29. Правило специализации инструкции $StoreVar^X$ var.

Инструкции LoadField и StoreField читают и записывают в поле объекта значение. Поле задается в параметре инструкции, а объект — в операнде инструкции на стеке. Поэтому генератор остаточной программы читает со стека ссылку на объект object и в остаточную программу добавляет инструкцию LoadVar или StoreVar, соответственно, для переменной остаточной программы, определенной по указателю на поле объекта и отображению ptr2var

(рис. 30, 31).

btProg $\vdash_{rpg}$ LoadField^X fld : ((oref::stack, env, heap), ptr2var) → (((stack, env, heap), ptr2var), [LoadVar ptr2var(oref, fld)])

Рис. 30. Правило специализации инструкции LoadField^X fld.

btProg $\vdash_{rpg}$ StoreField^X fld : ((oref::stack, env, heap), ptr2var) → (((stack, env, heap), ptr2var), [StoreVar ptr2var(oref, fld)])

Рис. 31. Правило специализации инструкции StoreField^X fld.

Инструкции LoadElement и StoreElement читают и записывают в элемент массива. Ссылка на массив и номер элемента массива задаются в операндах инструкции на стеке. Поэтому генератор остаточной программы читает со стека номер элемента и ссылку на массив, а в остаточную программу добавляет инструкцию LoadVar или StoreVar, соответственно, для переменной остаточной программы, определенной по указателю на элемент массива и отображению ptr2var (рис. 32, 33).

btProg $\vdash_{rpg}$ LoadElement^X : (n::aref::stack, env, heap, ptr2var) → (((stack, env, heap), ptr2var), [LoadVar ptr2var(aref, n)])

Рис. 32. Правило специализации инструкции LoadElement^X.

btProg $\vdash_{rpg}$ StoreElement^X : (n::aref::stack, env, heap, ptr2var) → (((stack, env, heap), ptr2var), [StoreVar ptr2var(aref, n)])

Рис. 33. Правило специализации инструкции StoreElement^X.

Инструкция Lifting^X

Инструкция Lifting переносит примитивное данное из состояния генератора остаточной программы в остаточную программу. Она читает значение, расположенное на вершине стека, и генерирует инструкцию загрузки константы на стек, в качестве константы выступает исходное значение (рис. 34).

btProg $\vdash_{rpg}$ Lifting^X : ((val::stack, env, heap), ptr2var) → (((stack, env, heap), ptr2var), [LoadConst val])

Рис. 34. Правило специализации инструкции Lifting^X.

7. Заключение

В работе описан генератор остаточной программы для языка SOOL. По корректной размеченной программе и значениям S-переменных, он строит результат специализации. Можно показать, что при условии корректности размеченной программы, результат специализации также будет корректен.

На основе описанных правил построен генератор остаточной программы [Klim06] для языка SOOL [Klim08c] который используется в специализаторе CILPE [Chep03,Klim08b] для платформы Microsoft .NET [CIL, MS.NET].

8. Литература

[Chep03] A.M.Chepovsky, An.V.Klimov, Ar.V.Klimov, Yu.A.Klimov, A.S.Mishchenko, S.A.Romanenko, S.Yu.Skorobogatov "Partial Evaluation for Common Intermediate Language" // Manfred Broy and Alexandre V. Zamulin (eds.), Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI'2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003. LNCS 2890, Springer-Verlag, December 2003, pp.171-177.

[CLI] Common Language Infrastructure (CLI) // http://www.ecma-international.org/publications/standards/Ecma-335.htm, http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx.

[Jone93] N.D.Jones, C.K.Gomard, P.Sestoft "Partial Evaluation and Automatic Compiler Generation" // C.A.R. Hoare Series, Prentice-Hall, 1993.

[Klim05a] Ю.А.Климов "Поливариантный анализ времен связывания в специализаторе CILPE для Common Intermediate Language платформы Microsoft.NET" // Труды Всероссийской конференции "Технологии Microsoft в теории и практике программирования", Москва, 17-18 февраля 2005 г., М: Изд-во МГТУ им. Н.Э. Баумана, 2005, с.128.

[Klim05b] Ю.А.Климов "О поливариантном анализе времен связывания в специализаторе объектно-ориентированного языка" // Труды Всероссийской научной конференции "Научный сервис в сети Интернет: технологии распределенных вычислений", Новороссийск, 19-24 сентября 2005 г., М: Изд-во

МГУ, с.89-91.

[Klim06] Ю.А.Климов "Генератор остаточной программы и корректность специализатора объектно-ориентированного языка" // Труды Всероссийской научной конференции "Научный сервис в сети Интернет: технологии параллельного программирования", Новороссийск, 18-23 сентября 2006 г., М: Изд-во МГУ, с.137-140.

[Klim08a] Ю.А.Климов "Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных" // Препринт Института прикладной математики им. М.В. Келдыша РАН, 2008 г., №12.

[Klim08b] Ю.А.Климов "Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках" // Препринт Института прикладной математики им. М.В. Келдыша РАН, 2008 г., №30.

[Klim08c] Ю.А.Климов "SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ" // Препринт Института прикладной математики им. М.В. Келдыша РАН, 2008 г., №44.

[Klim09] Ю.А.Климов "Специализатор CILPE: анализ времен связывания" // Препринт Института прикладной математики им. М.В. Келдыша РАН, 2009 г., №7.