ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 33 за 2009 г.

Климов Ю. А.

Специализатор CILPE: доказательство корректности

Рекомендуемая форма библиографической ссылки: Климов Ю. А. Специализатор CILPE: доказательство корректности // Препринты ИПМ им. М.В.Келдыша. 2009. № 33. 32 с. URL: http://library.keldysh.ru/preprint.asp?id=2009-33

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

Ю.А. Климов

Специализатор CILPE:
доказательство корректности

Москва
2009

Ю.А. Климов

Специализатор CILPE: доказательство корректности

Аннотация

Специализатор CILPE преобразует программы, написанные на языке SOOL (Stack Object-Oriented Language) методом частичных вычислений. В работе доказана корректность CILPE: для любой корректной разметки исходной программы генератор остаточных программ строит остаточную программу, эквивалентную исходной при заданных значениях S-аргументов и любых значениях D-аргументов.

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

Yu.A. Klimov

Specializer CILPE: correctness proof

Abstract

The specializer CILPE, based on partial evaluation, transforms programs written in SOOL, a Stack Object-Oriented Language. In the paper the correctness of CILPE is proven: if the annotations in a source program are correct, the residual program generator produces a residual program that is equivalent to the source one for the specified values of the S-arguments and any values of D-arguments.

Содержание

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 объектно-ориентированных 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.

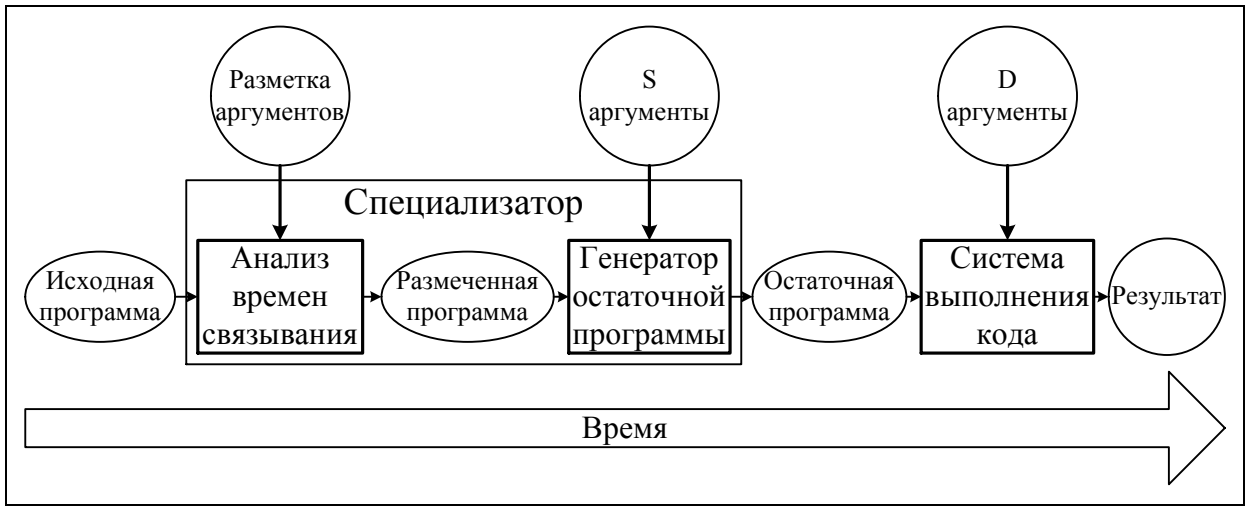

Рис. 1. Общепринятая структура специализатора, основанного на методе частичных вычислений.

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* (Residual 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, удовлетворяющий требованиям, изложенным в [Klim08a].

В данной работе доказывается корректность генератора остаточных программ на языке SOOL [Klim08c]. В доказательстве используются фор-

мальные описания корректной разметки программы [Klim09a] и генератора остаточной программы [Klim09b].

2. Доказательство корректности

Специализатор на вход принимает размеченную программу и значения S-аргументов. В результате специализации получается новая программа.

Необходимо доказать, что исходная и специализированная программы эквивалентны: работают одинаково и возвращают один и тот же результат при заданных значениях S-аргументов и произвольных значениях D-аргументов.

Интерпретация исходной размеченной программы, генерация остаточной программы и интерпретация остаточной программы происходят по шагам. Будем одновременно проводить (1) интерпретацию размеченной программы, (2) генерацию остаточной программы и (3) интерпретацию остаточной программы при произвольных значениях D-аргументов. На каждом шаге имеем:

1. инструкцию размеченной программы и
   a. состояние интерпретатора перед ее выполнением;
   b. BT-состояние, описанное в разметке программы;
   c. состояние генератора остаточной программы перед ее обработкой;
2. инструкцию остаточной программы и состояние интерпретатора перед ее выполнением.

Состояние и элементы состояния интерпретатора размеченной программы будем записывать без индексов. Состояние генератора остаточной программы – с индексом S (например, $state_S$). Состояние интерпретатора остаточной программы – с индексом D (например, $state_D$).

Доказательство будем вести индукцией по количеству таких шагов, сравнивая на каждом шаге состояние интерпретатора размеченной программы и соединенное состояние.

2.1. Соединение состояний

Во время генерации остаточной программы обращения к D-полям S-объектов и D-элементам S-массивов заменяются обращением к локальным переменным остаточной программы. Для такого преобразования используется сопоставление ptr2var, ставящее в соответствие ссылкам на поля или на элементы локальные переменные.

При интерпретации остаточной программы в локальных переменных хранятся значения. Используя сопоставление ptr2var можно соединить состояние генератора и интерпретатора остаточной программы в одно состояние: для этого нужно перенести значения переменных остаточной программы в поля или элементы, которые им соответствуют согласно ptr2var.

В результате получим состояние, которое, как доказано ниже, совпада-

ет с состоянием интерпретатора исходной программы!

Определим соединение состояний. Рассмотрим какой-нибудь шаг. Тогда имеем:

1. размеченное состояние btState = (btStack, btEnv) и BT-кучу btHeap из размеченной программы;
2. состояние rpgState = ($state_S$, ptr2var) генератора остаточной программы;
3. состояние $state_D$ интерпретатора остаточной программы.

Чтобы соединить эти состояния необходимо проверить согласованность состояний с размеченным состоянием btState. А именно:

1. количество и типы S-элементов в стеке btStack совпадают с количеством и типами элементов стека в $state_S$ (или стек в $state_S$ больше);
2. количество и типы D-элементов в стеке btStack совпадают с количеством и типами элементов стека в $state_D$ (или стек в $state_D$ больше);
3. для каждой локальной переменной размеченной программы, которая имеет разметку D согласно btEnv, для каждого поля объекта или элемента массива, которые имеют значение DYNVALUE, определено отображение ptr2var из состояния генератора остаточной программы. Значение этого отображения – это объявленная локальная переменная в остаточной программе. Отметим, что поле объекта или элемент массива имеют значение DYNVALUE тогда и только тогда, когда объект размечен S, а поле или элемент – D.

Описанную согласованность будем проверять по шагам одновременно с доказательством корректности.

Если состояния согласованы, то соединим их в одно. Соединение происходит на основе BT-состояния btState и отображения ptr2var (рис. 2):

1. Стеки соединяются согласно размеченному стеку в btState: на место S-значений записываются значения из rpgState стека, а на место D-значений – из stateD стека.
2. Окружения соединяются согласно размеченному окружению btEnv: если разметка S, то берется значения из env_S; если разметка D, то берется значение из env_D, согласно отображению ptr2var.
3. Кучи соединяются согласно значению DYNVALUE: если значение поля объекта и элемента массива равно DYNVALUE, то его следует заменить на значение переменной, которая определяется по отображению ptr2var. Затем объединить кучи.

   В результате получим объединенное состояние $state_{S+D}$.

Отметим, что стеки и окружения соединяются согласно разметкам btStack и btEnv. И если, например, в соединяемых стеках находится больше элементов, чем указано в разметке, то данные элементы не соединяются. Аналогично с окружением – соединяются только присутствующие в btEnv пере-

менные. Это позволяет соединять и сравнивать состояния при вызове функции, о чем рассказано ниже.

Аналогично соединяются входные состояния: аргументы генератора остаточной программы и интерпретатора остаточной программы. В этом случае необходимо объединить только аргументы согласно BT-видам btArg аргументов. И аналогично соединяются состояния при вызове методов.

Соединенное состояние, а так же его элементы, будем обозначать с индексом S+D: $state_{S+D}$.

```
JoinState_btHeap btState rpgState state_D = state_S+D
   where  (btStack, btEnv) = btState
          (state_S, ptr2var) = rpgState
          (stack_S, env_S, heap_S) = state_S
          (stack_D, env_D, heap_D) = state_D
          stack_S+D = JoinStack_btHeap btStack stack_S stack_D
          env_S+D = JoinEnv_btHeap btEnv env_S env_D
          heap_S+D = JoinHeap ptr2var env_D heap_S heap_D
          state_S+D = (stack_S+D, vars_S+D, heap_S+D)
```

```
JoinStack_btHeap [] _ _ = []
JoinStack_btHeap (btVal::btStack) stack_S stack_D =
   if GetBTKind_btHeap(btVal) == S
   then (head stack_S)::(JoinStack btStack (tail stack_S) stack_D)
   else (head stack_D)::(JoinStack btStack stack_S (tail stack_D))
```

```
JoinEnv_btHeap btEnv ptr2var env_S env_D = [var↦ val | (var↦btVal) ← btEnv,
       val = if GetBTKind_btHeap(btVal) == S then env_S(var) else env_D(ptr2var(var))]
```

```
JoinHeap ptr2var env_D heap_S heap_D =
                              [ref↦(EditObjectOrArray ref) | (ref↦_) ← heap_S]++heap_D
   where
      EditObjectOrArray oref = (class, obj_S+D)
         where  (class, obj_S) = heap_S(oref)
                obj_S+D = [fld↦val' | (fld↦val) ← obj_S,
                 if val == DYNVALUE then val' = env_D(ptr2var(oref, fld)) else val]
      EditObjectOrArray aref = (type[], (n, arr_S+D))
         where  (type[], (n, arr_S)) = heap_S(aref)
                arr_S+D = [i↦val' | (i↦val) ← obj_S,
                   if val == DYNVALUE then val' = env_D(ptr2var(aref, i)) else val]
```

Рис. 2. Соединение состояния генератора остаточной программы и интерпретатора остаточной программы.

2.2. Сравнение состояний

В языке нет операций над адресами – только доступ к полям объектов или элементам массивов. Поэтому конкретное значение адреса, получаемое объектом или массивом при создании, несущественно. Единственное требова-

ние – чтобы адрес был «новым», никакой ранее созданный объект или массив в рамках данной интерпретации или специализации программы не должен иметь такой же адрес.

Поэтому будем создавать новые адреса таким образом, чтобы при одновременном выполнении инструкции создания объекта или массива интерпретатором размеченной программы и генератором остаточной программы, созданный объект или массив получал одинаковый «новый» адрес.

```
CompareStates (stack1, env1, heap1) (stack2, env2, heap2) = (stack1 == stack2) &&
                                (CompareMaps env1 env2) && (CompareHeap [] refs)
  where  refs = (FindRef stack1) ++ (FindRef env1)
         CompareHeap crefs [] = TRUE
         CompareHeap crefs (ref:refs) | ref `elem` crefs =
                                                        CompareHeap crefs refs
         CompareHeap crefs (ref:refs) =
                                        (CompareObjOrArr heap1(ref) heap2(ref)) &&
                                    (CompareHeap crefs (refs++(FindRef(heap2(ref)))))
         CompareObjOrArr (class, obj1) (class, obj2) = (CompareMaps obj1 obj2)
         CompareObjOrArr (type[], (n, arr1)) (type[], (n, arr2)) =
                                                        (CompareMaps arr1 arr2)
         FindRef stack = [ref | ref ← stack]
         FindRef env = [ref | (_↦ref) ← env]
         FindRef (class, obj) = [ref | (_↦ref) ← obj]
         FindRef (type[], (n, arr)) = [ref | (_↦ref) ← arr]
         CompareMaps map1 map2 = Domain(map1) == Domain(map2) &&
                                (all [map1(var) == map2(var) | var ← Domain(map1)])
         Domain map = sort [key | (key↦_) ← map]
```

Рис. 3. Сравнение состояний.

Данное ограничение позволит сравнивать состояния (рис. 3):

1. Необходимо проверить равенство значений элементов на стеке.
2. Необходимо проверить равенство значений локальных переменных.
3. В кучах нужно сравнить достижимые из стека или окружения объекты или массивы. Если объект или массив недостижим, то значения полей или элементов нельзя получить в программе. Поэтому состояния считаем одинаковы, если они отличаются только в частях, относящихся к недостижимым объектам и массивам. В реальных системах недостижимые объекты и массивы удаляются сборщиком мусора.

3. База индукции

Будем доказывать, что на каждом шаге состояние интерпретатора размеченной программы совпадает с соединенным состоянием.

Доказательство будем проводить индукцией по количеству шагов ин-

терпретатора размеченной программы. Каждому шагу интерпретатора размеченной программы будет соответствовать один шаг генератора остаточной программы, а так же ноль, один или несколько шагов интерпретатора остаточной программы (в зависимости от количества инструкций, которые генератор добавил в остаточную программу на соответствующем шаге).

Пусть нам даны BT-виды аргументов btArg и значения S-аргументов arg_S и D-аргументов arg_D. Рассмотрим начальные состояния (1) интерпретатора размеченной программы, (2) генератора остаточной программы и (3) интерпретатора остаточной программы.

На вход интерпретатора размеченной программы подаются и аргументы arg_S, и аргументы arg_D. Причем ровно в том порядке, как описано в разметке btArg. Легко видеть, что соединенное состояние совпадает с состоянием интерпретатора размеченной программы.

4. Шаг индукции: программа

Рассмотрим первый шаг – начало выполнения размеченной и остаточной программ и начало генерации остаточной программы. Известно, что аргументы arg_S и arg_D согласованы с разметкой аргументов btArg.

$$\frac{\text{btInstrs} = [\text{NewObject}^D\ \text{MAIN}, \text{CallMethod}^X\ \text{Main}, \text{Leave}^X] \quad \text{btProg} \vdash_i (\text{btInstrs}, 0) : (\text{arg}, [], []) \Rightarrow (\text{res}, [], \text{heap})}{\vdash_i \text{btProg} : \text{arg} \Rightarrow \text{res}}$$

$$\frac{\begin{array}{l}\ldots \\ \exists \lambda_0 : \text{Integer} \to \text{BTState} : \\ \text{instrs} = [\text{NewObject}^D\ \text{MAIN}, \text{CallMethod}^X\ \text{Main}, \text{Leave}^X], \\ \text{btArg} = \text{fst}\ \lambda_0(0), \quad \text{btRes} = \text{fst}\ \lambda_0(2), \\ \forall \text{btVal} \in \text{btArg} : \ (\text{primType}, \_) = \text{btVal}, \\ \forall \text{btVal} \in \text{btRes} : \ \text{BTKind}(\text{btVal}) = D, \\ \forall n\ 0 \le n < 3 : \ \text{btProg}, \text{btHeap}, \lambda_0, \text{btRes} \vdash_{bt} (\text{instrs}, n) \\ \ldots\end{array}}{\text{btHeap} \vdash_{bt} \text{btProg}}$$

$$\frac{\begin{array}{l}\text{btInstrs} = [\text{NewObject}^D\ \text{MAIN}, \text{CallMethod}^X\ \text{Main}, \text{Leave}^X] \\ \text{rgpState} = ((\text{arg}_S, [], []), []) \\ \text{btProg} \vdash_{rpg} (\text{btInstrs}, 0) : (\text{rpgState}, []) \Rightarrow (\_, \text{mthds}_2) \\ \ldots\end{array}}{\vdash_{rpg} \text{btProg} : \text{arg}_S \Rightarrow \text{btProg}'}$$

$$\frac{\text{rInstrs} = [\text{NewObject MAIN}, \text{CallMethod Main}, \text{Leave}] \quad \text{rProg} \vdash_i (\text{rInstrs}, 0) : (\text{arg}_D, [], []) \Rightarrow (\text{res}_D, [], \text{heap}_D)}{\vdash_i \text{rProg} : \text{arg}_D \Rightarrow \text{res}_D}$$

Рис. 4. Правила для программы.

Рассмотрим правило выполнения размеченной программы, правило обработки размеченной программы генератором остаточной программы и правило выполнения остаточной программы (рис. 4).

$$\frac{\begin{array}{l}(tArg, \_) = MethodSignature_{prog}(mthd) \qquad Length(arg) = Length(tArg) \\ type = TypeOf_{heap}(Head(arg)) \qquad type <_{prog} Head(tArg) \qquad type \neq NULLTYPE \\ prog \vdash_I MethodDefinition_{prog}(mthd, type) : (arg, heap_1) \Rightarrow (res, heap_2)\end{array}}{prog \vdash_i CallMethod^X\ mthd : (arg{+}{+}stack, env, heap_1) \rightarrow (res{+}{+}stack, env, heap_2)}$$

$$\frac{(btArg, btRes) = MethodBTSignature_{btProg}(mthd)}{prog, btHeap \vdash_{bt} CallMethod^X\ mthd : (btArg{+}{+}btStack, btEnv) \rightarrow (btRes{+}{+}btStack, btEnv)}$$

Рис. 5. Правила для инструкции $CallMethod^X$ mthd.

$$\frac{\begin{array}{l}(\_, tArg, tRes, (varDecs, instrs)) = mthdDef \qquad TypeOf_{heap1}(arg) <_{prog} tArg \\ env_1 = [var \mapsto DefaultValue(type) \mid (var, type) \leftarrow varDecs] \\ prog \vdash_i (instrs, 0) : (arg, env_1, heap_1) \Rightarrow (res, env_2, heap_2) \\ TypeOf_{heap2}(res) <_{prog} tRes\end{array}}{prog \vdash_i mthdDef : (arg, heap_1) \Rightarrow (res, heap_2)}$$

$$\frac{\begin{array}{l}(\_, btTypeArg, btTypeRes, (\_, btInstrs)) = btMthdDef \\ \forall btType \in btTypeArg : \\ \quad \bullet \text{ либо } (btType \text{ — ссылочный BT-тип}) \\ \qquad (type, btOref) = btType,\ (\_, types, \_) = btHeap(btOref),\ type \in types \\ \quad \bullet \text{ либо } (btType \text{ — примитивный BT-тип}) \\ \qquad (type_1, (type_2, btKind)) = btType,\ type_1 = type_2 \\ \forall btType \in btTypeRes : \\ \quad \bullet \text{ либо } (btType \text{ — ссылочный BT-тип}) \\ \qquad (type, btOref) = btType,\ (\_, types, \_) = btHeap(btOref),\ type \in types \\ \quad \bullet \text{ либо } (btType \text{ — примитивный BT-тип}) \\ \qquad (type_1, (type_2, btKind)) = btType,\ type_1 = type_2 \\ fst\ \lambda_{btMthdDec}(0) = btArg = map\ snd\ btTypeArg \qquad btRes = map\ snd\ btTypeRes \\ \forall n\ 0 \le n < length(instrs) :\ btProg, btHeap, \lambda_{btMthdDec}, btRes \vdash_{bt} (btInstrs, n)\end{array}}{btProg, btHeap, \lambda_{btMthdDec} \vdash_{bt} btMthdDef}$$

Рис. 6. Правила для метода.

Во всех случаях создается состояние и применяется правило для одинаковых последовательностей инструкций.

Необходимо показать, что состояние интерпретатора размеченной программы (arg, [], []) совпадает с соединенным состоянием $JoinState_{btHeap}$ btState $((arg_S, [], []), [])$ $(arg_D, [], [])$, где btState – это состояние при выполнении первой инструкции, т.е. $\lambda_0(0) = (btArg, \_)$.

Т.к. локальных переменных нет и кучи пусты, то согласно определе-

нию функции $\text{JoinState}_{\text{btHeap}}$ для построения соединенного состояния необходимо соединить arg_S и arg_D согласно btArg. Но, согласно базе индукции, в результате получим arg. Что и требовалось доказать.

5. Шаг индукции: метод и инструкции CallMethod и Leave

Вызов метода обрабатывается генератором остаточной программы в зависимости от значения флага IsInline.

5.1. Метод INLINE

Если метод INLINE, то генератор остаточной программы подставляет тело метода вместо вызова.

Так как элемент на вершине стека размечен S (рис. 7) и состояния до шага совпадают, то на вершине стека интерпретатора исходной программы и на вершине стека генератора остаточной программы лежит одно и то же значение – ссылка на объект. Поэтому в обоих случаях выберем один и тот же метод (рис. 5, 8).

…
- Если метод INLINE, то BT-вид первого аргумента S.
- Если метод NOINLINE, то BT-вид всех результатов и полей объектов и элементов массивов аргументов D.

btProg — корректна

Рис. 7. Корректность BT-программы.

Состояния для обработки вызываемого метода строятся по-разному. Интерпретатор берет часть состояния вызываемого метода - вершину стека, соответствующую аргументам. А генератор – всё состояние. Но так как состояния совпадали до шага, то после перехода к обработке метода состояние интерпретатора будет совпадать с соединенным состоянием. При переходе от обработки метода к обработке последовательности инструкций интерпретатором заводятся новые локальные переменные. А генератором остаточной программы добавляются инструкции присваивания значений по умолчанию в переменные остаточной программы, которые соответствуют переменным вызываемого метода размеченной программы. А также добавляются переменные в состояние генератора остаточной программы. В результате состояние интерпретатора совпадает с соединенным состоянием (рис. 9).

Переменные добавляются и в остаточную программу и в состояние генератора, так как в разных точках метода переменная может иметь разную разметку: в одном месте S, в другом – D. Поэтому необходимо завести переменные для обоих случаев.

После завершения обработки тела метода состояния совпадают, поэтому, что показывается аналогично, состояния после обработки вызова метода

тоже совпадают.

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

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

(_, INLINE, _, _, (varDecs, btInstrs')) = $\text{MethodBTDefinition}_{\text{btProg}}$(mthd, class)
rpgState_2 = (_, ptr2var_2) = AddVars(varDecs, rpgState_1)
rpgInstrs_2 = concat [[LoadConst DefaultValue(type),
StoreVar ptr2var_2(var)] | (var, type) ← varDecs]
prog $\vdash_{\text{rpg}}$ (btInstrs', 0) : (rpgState_2, []) ⇒ (rpgInstrs_3, mthds_3)

(rpgInstrs_4, rpgStates_4) = ReplaceLeave(rpgInstrs_3)
rpgInstrs_5 = AddInstrs(rpgInstrs_1++rpgInstrs_2, 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 $\vdash_{\text{rpg}}$ (btInstrs, n+1) : (rpgState_5, rpgInstrs_6++[Label str_4 _]) ⇒
(rpgInstrs_7, mthds_7)
  rpgInstrs_6 = rpgInstrs_7
  mthds_6 = mthds_6++mthds_7

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

Рис. 8. Обработка последовательности инструкций,
инструкция CallMethod^X mthd, INLINE.

(arg++stack, env, heap) = $\text{JoinState}_{\text{btHeap}}$ (btArg++btStack, btEnv))
((stack_S, env_S, heap_S), ptr2var) state_D
⇓
(arg, [], heap) = $\text{JoinState}_{\text{btHeap}}$ (btArg, []) ((stack_S, env_S, heap_S), ptr2var) state_D
⇓ (_, btEnv_1) = $\lambda_{\text{btMthdDec}}$(0)
(arg, env_1, heap) = $\text{JoinState}_{\text{btHeap}}$ (btArg, btEnv_1) ((stack_S, env_{S1}, heap_S), ptr2var_1)
state_{D1}

Рис. 9. Проверка совпадения состояний,
инструкция CallMethod^X mthd, INLINE.

5.2. Метод NOINLINE

В случае NOINLINE, генератор остаточной программы добавляет в остаточную программу вызов специализированного метода.

Если метод NOINLINE, то на разметку аргументов и результатов на-

ложены следующие ограничения (рис. 7):

1. результаты должны быть размечены D;
2. если аргумент имеет объектный тип и размечен S, то все поля (в том числе и специальное поле ELEMENT для разметки элементов массива, если оно есть) должны быть размечены D.

Из данных ограничений следует, что во время специализации метод ничего не возвращает, не изменяет состояние генератора остаточной программы. Все возможные «выходные» данные (результаты и поля аргументов) размечены D. Поэтому генерация остаточной программы для вызываемого и вызывающего методов может идти независимо (рис. 10).

При обработке метода генератор добавляет в остаточную программу инструкции для передачи дополнительных параметров. Эти параметры соответствуют D-полям S-объектов и D-элементов S-массивов, которые передаются через аргументы. В остаточной программе данным полям и элементам соответствуют локальные переменные. И при вызове необходимо передать значения этих переменных в вызываемый метод. А после вызова – записать значения обратно в переменные. Такая передача делается с помощью вспомогательных функций Vars2Stack и Stack2Vars (рис. 10, 11).

$$\frac{\begin{array}{l}\text{CallMethod}^{X}\ \text{mthd} = \text{btInstrs}[n] \\ \text{NOTHING} = \text{FindState}(\text{rpgInstrs}_1, n, \text{rpgState}) \\ \\ (\text{NOINLINE}, \text{btArg}, \_) = \text{MethodBTSignature}_{\text{btProg}}(\text{mthd}) \\ ((\text{arg}_S{+}{+}\text{stack}_S, \text{env}_S, \text{heap}_S), \text{ptr2var}) = \text{rpgState}_1 \\ \text{Length arg}_1 = \text{Length (filter (S ==) (map snd btArg))} \\ \text{rpgState}_2 = ((\text{stack}_S, \text{env}_S, \text{heap}_S), \text{ptr2var}) \\ \\ \text{mthdArg} = (\text{arg}_S, \text{heap}_S) \\ \text{mthd'} = (\text{mthd}, \text{mthdArg}) \\ \\ \text{instrs} = \text{Vars2Stack}(\text{arg}_S, \text{heap}_S, \text{ptr2var}){+}{+}[\text{CallMethod MakeName(mthd')}]{+}{+} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Stack2Vars}(\text{arg}_S, \text{heap}_S, \text{ptr2var}) \\ \text{rpgInstrs}_2 = \text{AddInstrs}(\text{rpgInstrs}_1, n, \text{rpgState}_1, \text{instrs}) \\ \text{btProg} \vdash_{\text{rpg}} (\text{btInstrs}, n{+}1) : (\text{rpgState}_2, \text{rpgInstrs}_2) \Rightarrow (\text{rpgInstrs}_3, \text{mthds}_3)\end{array}}{\text{btProg} \vdash_{\text{rpg}} (\text{btInstrs}, n) : (\text{rpgState}_1, \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_3, \text{mthd'}::\text{mthds}_3)}$$

Рис. 10. Обработка последовательности инструкций,
инструкция CallMethod^{X} mthd, NOINLINE.

Таким образом, при обработке вызова метода и начала метода происходит изменение переменных, соответствующих D-полям и D-элементам. А в остаточной программе генерируются инструкции для перекладывания значений между этими переменными (до вызова – выкладывание значений на стек,

в начале метода – запись в переменные значений со стека).

После завершения обработки метода происходит обратное действие: перед каждой инструкцией Leave в вызываемом методе вставляются инструкции загрузки значений на стек. А в вызывающем методе после инструкции вызова вставляются инструкции записи значений в переменные.

Значит, если состояние интерпретатора совпадало с соединенным состоянием до обработки вызова метода, то они будут совпадать и перед обработкой тела метода (после обработки интерпретатором остаточной программы инструкций записи переменных) (рис. 12).

После завершения обработки вызываемого метода, аналогично, если состояния будут совпадать перед обработкой инструкции Leave, то будут совпадать и после полной обработки инструкции вызова.

$$
\begin{array}{l}
(mthd, \_, \_, \_, (varDecs, btInstrs)) = mthdDef \\
\\
ptr2var' = MkNewVars_{btProg}(arg_S, heap_S) \\
rpgState_1 = ((arg_S, [], heap_S), ptr2var') \\
rpgState_2 = AddVars(varDecs, rpgState_1) \\
\\
rpgInstrs_1 = Stack2Vars(arg_S, heap_S, ptr2var') \\
prog \vdash_{rpg} (btInstrs, 0) : (rpgState_2, rpgInstrs_1) \Rightarrow (rpgInstrs_3, mthds) \\
rpgInstrs_4 = AddBeforeLeave(rpgInstrs_3, Vars2Stack(arg_S, heap_S, ptr2var')) \\
\\
mthd' = MakeName(mthd, (arg_S, heap_S)) \\
tArg = [type \mid (type, D) \leftarrow btArg] ++ GetTypes_{btProg}(arg_S, heap_S) \\
tRes = [type \mid (type, D) \leftarrow btRes] ++ GetTypes_{btProg}(arg_S, heap_S) \\
mthdBody' = MkMethodBody(rpgInstrs_4) \\
mthdDef' = (mthd', tArg, tRes, mthdBody') \\
\hline
btProg \vdash_{rpg} mthdDef : (arg_S, heap_S) \Rightarrow (mthdDef', mthds)
\end{array}
$$

Рис. 11. Правило обработки метода.

$$
\begin{array}{l}
(arg{+}{+}stack, env, heap) = JoinState_{btHeap}\ (btArg{+}{+}btStack, btEnv)) \\
\qquad ((arg_S{+}{+}stack_S, env_S, heap_S), ptr2var)\ (arg_D{+}{+}stack_D, env_D, heap_D) \\
\qquad\qquad \Downarrow \\
heap = JoinHeap\ ptr2var\ env_D\ heap_S\ heap_D \\
\quad \Downarrow env_D(ptr2var(ptr)) = env'_D(ptr2var'(ptr)) \text{ для всех ptr из } FindDynValues(arg_S, heap_S) \\
heap = JoinHeap\ ptr2var'\ env'_D\ heap_S\ heap_D \\
\qquad\qquad \Downarrow \\
(arg, env', heap) = JoinState_{btHeap}\ (btArg, btEnv_1)\ ((arg_S, env'_S, heap_S), ptr2var') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (arg_D, env'_D, heap_D)
\end{array}
$$

Рис. 12. Проверка совпадения состояний,
инструкция $CallMethod^X$ mthd, NOINLINE.

6. Шаг индукции: последовательность инструкций

6.1. Обнаружение повтора генератором остаточной программы

Для построения конечной программы генератор остаточной программы постоянно сравнивает свое текущее состояние с предыдущими своими состояниями. И в случае обнаружения совпадения, генерирует инструкцию перехода Goto (рис. 13) и прекращает обработку данной ветви, так как состояние полностью совпало, то дальнейшая обработка полностью повторяла бы ту, которая уже была произведена.

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

Рис. 13. Обработка последовательности инструкций, обнаружен повтор состояния.

Но для одновременной генерации и интерпретации остаточной программы удобнее считать, что генератор остаточной программы не останавливается. Поэтому продолжим генерацию остаточной программы с этого же места.

Интерпретатор остаточной программы выполняет инструкцию безусловного перехода Goto, поэтому его состояние до и после выполнения этой инструкции совпадают.

Так как состояния генератора остаточной программы полностью совпадают, а интерпретатор размеченной программы в данном случае не делает шага, то соединенное состояние до и после выполнения инструкции Goto интерпретатором остаточной программы совпадают.

Ниже разбираются случаи, когда генератор остаточной программы на n-ом шаге не обнаружил повторения состояния.

6.2. Инструкция Goto^S

Если n-ая инструкция – это инструкция Goto^S m, то генератор остаточной программы ничего не генерирует в остаточной программе. Поэтому состояние state_D не меняется (рис. 14).

$$\frac{\text{Goto}^S\ \text{m} = \text{instrs}[\text{n}] \qquad \text{btProg} \vdash_{\text{i}} (\text{instrs}, \text{m}) : \text{state} \Rightarrow \text{state}'}{\text{btProg} \vdash_{\text{i}} (\text{instrs}, \text{n}) : \text{state} \Rightarrow \text{state}'}$$

$$\frac{\text{Goto}^S\ \text{m} = \text{btInstrs}[\text{n}] \qquad \lambda_{\text{btMthdDec}}(\text{n}) = \lambda_{\text{btMthdDec}}(\text{m})}{\text{btProg}, \text{btHeap}, \lambda_{\text{btMthdDec}}, \text{btRes} \vdash_{\text{bt}} (\text{btInstrs}, \text{n})}$$

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

Рис. 14. Правила для последовательности инструкций, инструкция Goto^S m.

Согласно правилам, интерпретатор размеченной программы и генератор остаточной программы, не меняя состояния, переходят на m-ную инструкцию.

Поэтому необходимо проверить, что соединенные в разных точках программы состояния совпадают: ($JoinState_{btHeap}$ $\lambda_{btMthdDec}$(n) rpgState $state_D$) = ($JoinState_{btHeap}$ $\lambda_{btMthdDec}$(m) rpgState $state_D$).

Но это очевидно из правила разметки программы: из него следует, что $\lambda_{btMthdDec}(n) = \lambda_{btMthdDec}(m)$. Что и требовалось доказать.

6.3. Инструкция $Branch^S$ m

Если n-ая инструкция $Branch^S$ m, то генератор остаточной программы выполняет проверку и переходит на k-ую инструкцию. В остаточную программу инструкции не добавляются (рис. 15).

$Branch^S$ m = instrs[n] val::$stack'_1$ = $stack_1$ INT = $TypeOf_{heap1}$(val) k = if val==0 then n+1 else m btProg $\vdash_i$ (btInstrs, k) : ($stack'_1$, env_1, $heap_1$) $\Rightarrow$ ($stack_2$, env_2, $heap_2$) --- btProg $\vdash_i$ (btInstrs, n) : ($stack_1$, env_1, $heap_1$) $\Rightarrow$ ($stack_2$, env_2, $heap_2$)
$Branch^S$ m = btInstrs[n] (INT^S::btStack, btEnv) = $\lambda_{btMthdDec}$(n) (btStack, btEnv) = $\lambda_{btMthdDec}(n+1) = \lambda_{btMthdDec}(m)$ --- btProg, btHeap, $\lambda_{btMthdDec}$, btRes $\vdash_{bt}$ (btInstrs, n)
$Branch^S$ m = btInstrs[n] NOTHING = FindState($rpgInstrs_1$, n, rpgState) ((val::$stack_S$, env_S, $heap_S$), ptr2var) = rpgState, где val — целое число rpgState' = (($stack_S$, env_S, $heap_S$), ptr2var) k = if val==0 then n+1 else m btProg $\vdash_{rpg}$ (btInstrs, k) : (rpgState', $rpgInstrs_1$) $\Rightarrow$ ($rpgInstrs_2$, $mthds_2$) --- btProg $\vdash_{rpg}$ (btInstrs, n) : (rpgState, $rpgInstrs_1$) $\Rightarrow$ ($rpgInstrs_2$, $mthds_2$)

Рис. 15. Правила для последовательности инструкций, инструкция $Branch^S$ m.

По предположению индукции известно, что ($stack_1$, env_1, $heap_1$) = $JoinState_{btHeap}$ $\lambda_{btMthdDec}$(n) rpgState $state_D$. Значит (val::$stack'_1$) = JoinStack (INT^S::btStack) (val::$stack_S$) $stack_D$. И на вершине (val::$stack'_1$) и (val::$stack_S$) лежит одно и тоже значение val. Поэтому значения k в правилах на рис. 15 будут найдены одинаково.

Осталось показать, что ($stack'_1$, env_1, $heap_1$) = $JoinState_{btHeap}$ $\lambda_{btMthdDec}$(k) rpgState' $state_D$. Но это очевидно, так как $stack'_1$ = JoinStack btStack $stack_S$ $stack_D$, что очевидно следует из равенства val::$stack'_1$ = JoinStack (INT^S::btStack) (val::$stack_S$) $stack_D$. Что и требовалось доказать.

6.4. Инструкция $Branch^D$ m

Если n-ая инструкция $Branch^D$ m, то генератор остаточной программы обрабатывает обе ветви и в остаточную программу помещает условный переход Branch (рис. 16).

По предположению индукции известно, что $(stack_1, env_1, heap_1) = JoinState_{btHeap}\ \lambda_{btMthdDec}(n)\ rpgState\ (stack_{D1}, env_{D1}, heap_{D1})$. Откуда следует, что $(val::stack'_1) = JoinStack\ (INT^D::btStack)\ stack_S\ (val::stack'_{D1})$. Поэтому на вершинах $(val::stack'_1)$ и $(val::stack'_{D1})$ лежит одно и то же значение val. Поэтому k и k’ в правилах на рис. 16 будут найдены одинаковым образом.

$$\frac{Branch^D\ m = instrs[n] \qquad val::stack'_1 = stack_1 \qquad INT = TypeOf_{heap1}(val) \qquad k = \text{if } val==0 \text{ then } n+1 \text{ else } m \qquad btProg \vdash_i (btInstrs, k) : (stack'_1, env_1, heap_1) \Rightarrow (stack_2, env_2, heap_2)}{btProg \vdash_i (btInstrs, n) : (stack_1, env_1, heap_1) \Rightarrow (stack_2, env_2, heap_2)}$$

$$\frac{Branch^D\ m = btInstrs[n] \qquad (INT^D::btStack, btEnv) = \lambda_{btMthdDec}(n) \qquad (btStack, btEnv) = \lambda_{btMthdDec}(n+1) = \lambda_{btMthdDec}(m)}{btProg, btHeap, \lambda_{btMthdDec}, btRes \vdash_{bt} (btInstrs, n)}$$

$$\frac{Branch^D\ m = btInstrs[n] \qquad NOTHING = FindState(rpgInstrs_1, n, rpgState) \qquad rpgInstrs_2 = AddInstrs(rpgInstrs_1, n, rpgState, [Branch\ str]) \qquad str \text{ — имя новой метки} \qquad btProg \vdash_{rpg} (btInstrs, n+1) : (rpgState, rpgInstrs_2) \Rightarrow (rpgInstrs_3, mthds_3) \qquad rpgInstrs'_3 = rpgInstrs_3 ++ [Label\ str\ \_] \qquad btProg \vdash_{rpg} (btInstrs, m) : (rpgState, rpgInstrs'_3) \Rightarrow (rpgInstrs_4, mthds_4)}{btProg \vdash_{rpg} (btInstrs, n) : (rpgState, rpgInstrs_1) \Rightarrow (rpgInstrs_4, mthds_2 ++ mthds_4)}$$

$$\frac{Branch\ m = rInstrs[n'] \qquad val::stack'_{D1} = stack_{D1} \qquad INT = TypeOf_{heap1}(val) \qquad k' = \text{if } val==0 \text{ then } n'+1 \text{ else } m' \qquad rProg \vdash_i (rInstrs, k') : (stack'_{D1}, env_{D1}, heap_{D1}) \Rightarrow (stack_{D2}, env_{D2}, heap_{D2})}{rProg \vdash_i (rInstrs, n') : (stack_{D1}, env_{D1}, heap_{D1}) \Rightarrow (stack_{D2}, env_{D2}, heap_{D2})}$$

Рис. 16. Правила для последовательности инструкций, инструкция $Branch^D$ m.

$$\frac{prog \vdash_i instrs[n] : (stack_1, env_1, heap_1) \rightarrow (stack_2, env_2, heap_2) \qquad prog \vdash_i (instrs, n+1) : (stack_2, env_2, heap_2) \Rightarrow (stack_3, env_3, heap_3)}{prog \vdash_i (instrs, n) : (stack_1, env_1, heap_1) \Rightarrow (stack_3, env_3, heap_3)}$$

$$\frac{prog, btHeap \vdash_{bt} btInstrs[n] : \lambda_{btMthdDec}(n) \rightarrow \lambda_{btMthdDec}(n+1)}{btProg, btHeap, \lambda_{btMthdDec}, btRes \vdash_{bt} (btInstrs, n)}$$

$$\frac{NOTHING = FindState(rpgInstrs_1, n, rpgState) \qquad btProg \vdash_{rpg} btInstrs[n] : rpgState_1 \rightarrow (rpgState_2, instrs) \qquad rpgInstrs_2 = AddInstrs(rpgInstrs_1, n, rpgState_1, instrs) \qquad btProg \vdash_{rpg} (btInstrs, n+1) : (rpgState_2, rpgInstrs_2) \Rightarrow (rpgInstrs_3, mthds_3)}{btProg \vdash_{rpg} (btInstrs, n) : (rpgState_1, rpgInstrs_1) \Rightarrow (rpgInstrs_3, mthds_3)}$$

Рис. 17. Правила для последовательности инструкций.

Затем генератор остаточной программы переходит на обработку k-ой инструкции. Необходимо проверить, что $(stack'_1, env_1, heap_1) = JoinState_{btHeap}\ \lambda_{btMthdDec}(k)\ rpgState\ (stack'_{D1}, env_{D1}, heap_{D1})$. Что, очевидно, следует из равенст-

ва $stack'_1$ = JoinStack btStack $stack_S$ $stack'_{D1}$. Что и требовалось доказать.

6.5. Остальные инструкции

Правила обработки остальных инструкций в последовательности инструкций указывают, что необходимо применить правила для отдельных инструкций. Состояния передаются и возвращаются без преобразований (рис. 17).

Поэтому очевидно, что если перед выполнением правила для последовательности инструкций состояния совпадали, то и перед выполнением правил для отдельных инструкций они будут совпадать.

А если будет доказано, что состояния совпадают после выполнения правил для отдельных инструкций, то, очевидно, состояния будут совпадать и после этого шага. Что и требовалось доказать.

7. Шаг индукции: инструкции

7.1. S-инструкции

Правила BT-разметки для S-инструкций гарантируют, что все необходимые для ее выполнения данные размечены S. Также btState изменяется на btState' полностью аналогично тому, как изменяются state на state' и $state_S$ на $state'_S$. Причем в btState меняются только элементы, размеченные S. Для каждого отдельного правила такая проверка производится простым сопоставлением правил разметки программы и правил выполнения программы. Указанную проверку опустим.

При обработке S-инструкций генератор остаточной программы выполняет указанную инструкцию, не генерируя инструкций в остаточную программу (рис. 18).

$btProg \vdash_i btInstr^S : state \rightarrow state'$
$btProg, btHeap \vdash_{bt} btInstr^S : btState \rightarrow btState'$
$\dfrac{btProg \vdash_i instr^S : state_S \rightarrow state'_S}{btProg \vdash_{rpg} instr^S : (state_S, ptr2var) \rightarrow ((state'_S, ptr2var), [])}$

Рис. 18. Правила для S -инструкций.

По предположению индукции известно, что state = $JoinState_{btHeap}$ btState ($state_S$, ptr2var) $state_D$. Так как состояние интерпретатора остаточной программы не изменяется, а состояния state, btState и $state_S$ изменяются одинаково, то легко показать, что state' = $JoinState_{btHeap}$ btState' ($state'_S$, ptr2var) $state_D$.

Это означает, что операция выполнения инструкции $instr^S$ и соединение состояний коммутируют: если к результату соединения состояний применить инструкцию instr, то получится то же самое, если бы мы сначала применили к состоянию $state_S$ инструкцию $instr^S$, а потом соединили с состоянием $state_D$.

7.2. D-инструкции

Доказательство для D-инструкций аналогично доказательству для S-инструкций, за исключением того, что если в предыдущем случае не изменялось состояние интерпретатора остаточной программы, то теперь не изменяется состояние генератора остаточной программы.

Если инструкция размечена D, то все ее входные данные размечены D. Также btState изменяется на btState', что полностью аналогично изменению state на state' и $state_D$ на $state'_D$. Причем в btState меняются только элементы, размеченные D. Для каждого отдельного правила такая проверка производится простым сопоставлением правил разметки программы и правил выполнения программы. Указанную проверку опустим.

При обработке D-инструкций генератор остаточной программы добавляет указанную инструкцию в остаточную программу, не меняя своего состояния (рис. 19).

$btProg \vdash_i btInstr^D : state \rightarrow state'$
$btProg, btHeap \vdash_i btInstr^D : btState \rightarrow btState'$
$btProg \vdash_{rpg} instr^D : rpgState \rightarrow (rpgState, [instr])$
$rProg \vdash_i btInstr^D : state_D \rightarrow state'_D$

Рис. 19. Правила для D-инструкций.

По предположению индукции известно, что $state = JoinState_{btHeap}\ btState\ rpgState\ state_D$. Так как состояние генератора остаточной программы не изменяется, а состояния state, btState и $state_D$ изменяются одинаково, то легко показать, что $state' = JoinState_{btHeap}\ btState'\ rpgState\ state'_D$.

7.3. X-инструкции

Самый существенный случай – это инструкция X. Подробно рассмотрим все X-инструкции.

7.3.1. Инструкция NewObject^X class

Генератор остаточной программы при обработке инструкции NewObject^X class создает объект и заводит в остаточной программе новые переменные для каждого D поля. У созданного объекта значения полей следующие: если поле S, то его значение – значение по умолчанию для данного типа; если поле D, то его значение – DYNVALUE.

Будем считать, что адреса oref в интерпретаторе размеченной программы и генераторе остаточной программы одинаковые.

В остаточную программу добавляются инструкции, «обнуляющие» значения переменных, которые соответствуют D-полям созданного объекта

(рис. 20).

Правило разметки программы утверждает, что разметка создаваемого объекта – S. Поэтому при соединении стеков oref::$stack_S$ и $stack_D$ по разметке btOref::btStack получим на вершине стека адрес oref. Поэтому соединенный стек будет совпадать со стеком (oref::stack) интерпретатора размеченной программы (рис. 21).

$$\frac{obj = [fld \mapsto DefaultValue(type) \mid (fld, type) \leftarrow ClassFields_{prog}(class)] \quad oref \text{ — новый адрес} \quad heap' = heap[oref \mapsto (class, obj)]}{btProg \vdash_i NewObject^X\ class : (stack, env, heap) \rightarrow (oref::stack, env, heap')}$$

$$\frac{(\_, types, btObj) = btHeap(btOref) \quad class \in types \quad BTKindOfbtHeap(btOref) = S}{btProg, btHeap \vdash_{bt} NewObject^X\ class^{btOref} : (btStack, btEnv) \rightarrow (btOref::btStack, btEnv)}$$

$$\frac{\begin{array}{l} oref \text{ — новый адрес} \\ obj = [fld \mapsto value \mid (fld, type) \leftarrow GetFieldDecs_{btProg}(class), \\ \quad value = \text{if } btFld(fld) == S \text{ then } DefaultValue(type) \text{ else } DYNVALUE] \\ heap'_S = heap_S[oref \mapsto (class, obj)] \\ ptr2var' = ptr2var[(oref, fld) \mapsto var_{fld}^{type} \mid (fld, type) \leftarrow GetFieldDecs_{btProg}(class), \\ \quad btFld(fld) = D,\ var_{fld}^{type} \text{ — новая переменная типа type}] \\ instrs = concat\ [[LoadConst\ DefaultValue(type), StoreVar\ ptr2var'(oref, fld)] \mid \\ \quad (fld, type) \leftarrow GetFieldDecs_{btProg}(class), btFld(fld) = D] \end{array}}{btProg \vdash_{rpg} NewObject^X\ class^{btFld} : ((stack_S, env_S, heap_S), ptr2var) \rightarrow (((oref::stack_S, env_S, heap'_S), ptr2var'), instrs)}$$

Рис. 20. Правила для инструкции $NewObject^X\ class^{btFld}$.

$$\begin{array}{l} (stack, env, heap) = JoinState_{btHeap}\ (btStack, btEnv)\ ((stack_S, env_S, heap_S), ptr2var)\ state_D \\ \Downarrow \\ stack = JoinStack_{btHeap}\ btStack\ stack_S\ stack_D \\ heap = JoinHeap\ ptr2var\ env_D\ heap_S\ heap_D \\ \Downarrow\ GetBTKind_{btHeap}(btOref) = S \text{ и используются переменные, имеющие в } env'_D \text{ значения по умолчанию} \\ (oref::stack) = JoinStack_{btHeap}\ (btOref::btStack)\ (oref::stack_S)\ stack_D \\ heap[oref \mapsto (class, obj)] = JoinHeap\ ptr2var\ env'_D\ heap_S[oref \mapsto (class, obj)]\ heap_D \\ \Downarrow \\ (oref::stack, env, heap') = JoinState_{btHeap}\ (btOref::btStack, btEnv)\ (((oref::stack_S, env_S, heap'_S), ptr2var')\ state'_D \end{array}$$

Рис. 21. Проверка совпадения состояний.

Генератор остаточной программы и интерпретатор размеченной программы создают объекты одного класса class. Разница заключается в том, что

значения полей: у интерпретатора – значения по умолчанию, а у генератора – либо DYNVALUE, либо значение по умолчанию. Причем значение DYNVALUE имеют только те поля, которые имеют разметку D.

При соединении куч значения DYNVALUE заменяются значениями соответствующих локальных переменных. Но данные локальные переменные имеют значение по умолчанию. Поэтому соединенная куча совпадает с кучей интерпретатора размеченной программы.

Как показано, соединенные стеки и кучи совпадают со стеком и кучей интерпретатора размеченной программы, поэтому совпадают и состояния. Что и требовалось доказать.

7.3.2. Инструкция NewArray^X type

Доказательство для инструкции NewArray^X type аналогично доказательству для NewObject^X class.

Генератор остаточной программы при обработке инструкции NewArray^X type создает массив и заводит в остаточной программе новые переменные для всех элементов массива. У созданного массива значения полей DYNVALUE. Отметим, что все элементы массивов имеют разметку D.

Будем считать, что адреса aref в интерпретаторе размеченной программы и генераторе остаточной программы одинаковые.

$$\frac{\mathrm{TypeOf}_{heap}(len) = INT \qquad arr = [i \mapsto \mathrm{DefaultValue}(type) \mid i \leftarrow [0..\,len-1]] \qquad aref \text{ — новый адрес} \qquad heap' = heap[aref \mapsto (type[], (len, arr))]}{btProg \vdash_i \mathrm{NewArray}^X\ type : (len::stack, env, heap) \rightarrow (aref::stack, env, heap')}$$

$$\frac{(\_,types,btObj) = btHeap(btOref) \quad type[] \in types \quad btVal = btObj(ELEMENT) \quad \mathrm{BTKindOf}_{btHeap}(btOref) = S \quad \mathrm{BTKindOf}_{btHeap}(btVal) = D}{btProg, btHeap \vdash_{bt} \mathrm{NewArray}^X\ type : (INT^S::btStack, btEnv) \rightarrow (btOref::btStack, btEnv)}$$

$$\frac{\begin{array}{l} arr = [i \mapsto DYNVALUE \mid i \leftarrow [0..len-1]] \\ aref \text{ — новый адрес} \qquad heap'_S = heap_S[aref \mapsto (type[], (len, arr))] \\ ptr2var' = ptr2var[(aref, i) \mapsto var_i^{type} \mid i \leftarrow [len..n-1], \\ \qquad\qquad var_i^{type} \text{ — новая переменная типа type}] \\ instrs = concat\ [[LoadConst\ \mathrm{DefaultValue}(type), StoreVar\ ptr2var'(aref, i)] \mid \\ \qquad\qquad i \leftarrow [len..n-1]] \end{array}}{\begin{array}{l} btProg \vdash_{rpg} \mathrm{NewArray}^X\ type : ((len::stack_S, env_S, heap_S), ptr2var) \rightarrow \\ \qquad (((aref::stack_S, env_S, heap'_S), ptr2var'), instrs) \end{array}}$$

Рис. 22. Правила для NewArray^X type.

В остаточную программу добавляются инструкции, «обнуляющие» значения переменных, которые соответствуют элементам массива (рис. 22).

Правило разметки программы утверждает, что разметка создаваемого массива – S. Поэтому при соединении стеков aref::stack$_S$ и stack$_D$ по разметке

btOref::btStack получим на вершине стека адрес aref. Поэтому соединенный стек будет совпадать со стеком (aref::stack) интерпретатора размеченной программы.

Генератор остаточной программы и интерпретатор размеченной программы создают массивы одного типа type[]. Разница заключается в том, что значения элементов: у интерпретатора – значения по умолчанию, а у генератора – DYNVALUE.

При соединении куч значения DYNVALUE заменяются значениями соответствующих локальных переменных. Но эти локальные переменные имеют значения по умолчанию. Поэтому соединенная куча совпадает с кучей интерпретатора размеченной программы (рис. 23).

Как показано, соединенные стеки и кучи совпадают со стеком и кучей интерпретатора размеченной программы, поэтому совпадают и состояния. Что и требовалось доказать.

$(len::stack, env, heap) = JoinState_{btHeap}\ (INT^S::btStack, btEnv)$
$((len::stack_S, env_S, heap_S), ptr2var)\ state_D$
$\Downarrow$
$(len::stack) = JoinStack_{btHeap}\ (INT^S::btStack)\ (len::stack_S)\ stack_D$
$heap = JoinHeap\ ptr2var\ env_D\ heap_S\ heap_D$
$\Downarrow GetBTKind_{btHeap}(btOref) = S$ и используются переменные,
имеющие в env'_D значения по умолчанию
$(aref::stack) = JoinStack_{btHeap}\ (btOref::btStack)\ (aref::stack_S)\ stack_D$
$heap[aref \mapsto (type[], (len, arr))] = JoinHeap\ ptr2var\ env'_D$
$heap_S[aref \mapsto (type[], (len, arr))]\ heap_D$
$\Downarrow$
$(oref::stack, env, heap') = JoinState_{btHeap}\ (btOref::btStack, btEnv)$
$(((oref::stack_S, env_S, heap'_S), ptr2var')\ state'_D$

Рис. 23. Проверка совпадения состояний.

7.3.3. Инструкция LoadVar^X var

Генератор остаточной программы при обработке инструкции LoadVar^X var в остаточную программу добавляет инструкцию LoadVar var' для переменной var' = ptr2var(var) (рис. 24).

$btProg \vdash_i LoadVar^X\ var : (stack, env, heap) \rightarrow (env(var)::stack, env, heap)$
$\dfrac{btVal = btEnv(var) \qquad BTKindOf_{btHeap}(btVal) = D}{btProg, btHeap \vdash_{bt} LoadVar^X\ var : (btStack, btEnv) \rightarrow (btVal::btStack, btEnv)}$
$btProg \vdash_{rpg} LoadVar^X\ var : (state_S, ptr2var) \rightarrow ((state_S, ptr2var), [LoadVar\ ptr2var(var)])$
$rProg \vdash_i LoadVar\ var' : (stack_D, env_D, heap_D) \rightarrow (env_D(var')::stack_D, env_D, heap_D)$

Рис. 24. Правила для инструкции LoadVar^X var.

Правило разметки требует, чтобы значение переменной было D.

Так как var' = ptr2var(var) и то, что состояния до выполнения шага совпадают, то значение env(var) в интерпретаторе размеченной программы равно значению $env_D(var')$ в интерпретаторе остаточной программы (рис. 25).

Следовательно, после шага на вершине стека будут одинаковые значения. Откуда следует равенство состояний. Что и требовалось доказать.

$$(stack, env, heap) = JoinState_{btHeap}\ (btStack, btEnv)\ (state_S, ptr2var)\ (stack_D, env_D, heap_D)$$

$$\Downarrow$$

$$env = JoinEnv_{btHeap}\ btEnv\ env_S\ env_D$$

$$stack = JoinStack_{btHeap}\ btStack\ stack_S\ stack_D$$

$$\Downarrow var' = ptr2var(var),\ BTKindOf_{btHeap}(btEnv(var)) = D$$

$$env(var) = env_D(var')$$

$$\Downarrow BTKindOf_{btHeap}(btVal) = D$$

$$(env(var)::stack) = JoinStack_{btHeap}\ (btVal::btStack)\ stack_S\ (env_D(var')::stack_D)$$

$$\Downarrow$$

$$(env(var)::stack, env, heap) = JoinState_{btHeap}\ (btVal::btStack, btEnv)\ (state_S, ptr2var)\ (env_D(var')::stack_D, env_D, heap_D)$$

Рис. 25. Проверка совпадения состояний.

7.3.4. Инструкция $StoreVar^X$ var

При обработке инструкции $LoadVar^X$ var генератор остаточной программы генерирует инструкцию записи в переменную StoreVar ptr2var(var) (рис. 26).

$$\frac{TypeOf_{heap}(val) <_{btProg} VarType_{btProg}(var)}{btProg \vdash_i StoreVar^X\ var : (val::stack, env, heap) \rightarrow (stack, env[var \mapsto val], heap)}$$

$$\frac{BTKindOf_{btHeap}(btVal) = D \quad btEnv' = btEnv[var \mapsto btVal]}{btProg, btHeap \vdash_{bt} StoreVar^X\ var : (btVal::btStack, btEnv) \rightarrow (btStack, btEnv')}$$

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

$$\frac{TypeOf_{heapD}(val) <_{rProg} VarType_{rProg}(var')}{rProg \vdash_i StoreVar\ var' : (val::stack_D, env_D, heap_D) \rightarrow (stack_D, env_D[var' \mapsto val], heap_D)}$$

Рис. 26. Правила для инструкции $StoreVar^X$ var.

Правило разметки требует, чтобы значение на вершине стека было D.

Так как состояния совпадают и вершина стека размечена D, то значения на вершинах стеков совпадают.

Так как var' = ptr2var(var), то новое значение переменной var в env[var↦val] равно новому значению var' в env_D[var'↦val]. Следовательно, по-

сле шага соединенное окружение будет совпадать с окружением интерпретатора размеченной программы (рис. 27).

Что означает, что будут совпадать и состояния. Что и требовалось доказать.

$$(val::stack, env, heap) = JoinState_{btHeap}\ (btVal::btStack, btEnv)\ (state_S, ptr2var)\ (val'::stack_D, env_D, heap_D)$$

$$\Downarrow$$

$$(val::stack) = JoinStack_{btHeap}\ (btVal::btStack)\ stack_S\ (val'::stack_D)$$

$$env = JoinEnv_{btHeap}\ btEnv\ env_S\ env_D$$

$$\Downarrow BTKindOf_{btHeap}(btVal) = D$$

$$val = val'$$

$$\Downarrow var' = ptr2var(var),\ btEnv' = btEnv[var \mapsto btVal],\ BTKindOf_{btHeap}(btVal) = D$$

$$env[var \mapsto val] = JoinEnv_{btHeap}\ btEnv'\ env_S\ env_D[var \mapsto val]$$

$$\Downarrow$$

$$(stack, env[var \mapsto val], heap) = JoinState_{btHeap}\ (btStack, btEnv')\ (state_S, ptr2var)\ (stack_D, env_D[var \mapsto val], heap_D)$$

Рис. 27. Проверка совпадения состояний.

7.3.5. Инструкция LoadField^X fld

Инструкция LoadField^X fld обрабатывается аналогично LoadVar^X var – в остаточную программу добавляется инструкция LoadVar var' (рис. 28).

$$\frac{TypeOf_{heap}(oref) <_{btProg} FieldClass_{btProg}(fld)\quad oref \neq NULL\quad (\_, obj) = heap(oref)}{btProg \vdash_i LoadField^X\ fld : (oref::stack, env, heap) \rightarrow (obj(fld)::stack, env, heap)}$$

$$\frac{(\_, \_, btObj) = btHeap(btOref)\qquad btVal = btObj(fld)\qquad BTKindOf_{btHeap}(btOref) = S\qquad BTKindOf_{btHeap}(btVal) = D}{btProg, btHeap \vdash_{bt} LoadField^X\ fld : (btOref::btStack, btEnv) \rightarrow (btVal::btStack, btEnv)}$$

$$\frac{oref \neq NULL}{btProg \vdash_{rpg} LoadField^X\ fld : ((oref::stack_S, env_S, heap_S), ptr2var) \rightarrow (((stack_S, env_S, heap_S), ptr2var), [LoadVar\ ptr2var(oref, fld)])}$$

$$rProg \vdash_i LoadVar\ var' : (stack_D, env_D, heap_D) \rightarrow (env_D(var')::stack_D, env_D, heap_D)$$

Рис. 28. Правила для инструкции LoadField^X fld.

Доказательство проводится в четыре этапа (рис. 29).

Во-первых, так как состояния до шага совпадают, то на вершине стеков лежат одинаковые значения.

Во-вторых, так как поле размечено D, то значение поля объекта в куче генератора остаточной программы равно DYNVALUE. Это следует из правила создания объекта и из того, что значения D-полей S-объектов (DYNVALUE) никогда не изменяются.

В-третьих, до шага куча интерпретатора совпадает с соединенной кучей, поэтому значения поля объектов совпадают. Но так как поле размечено D, то значение поля в соединенном состоянии копируется из значения локальной переменной. Следовательно, значение поля объекта равно значению локальной переменной.

И, в-четвертых, так как до шага стеки совпадали, то после выполнения шага (добавления одинаковых значений в стеки), стеки всё равно будут совпадать.

Следовательно, состояние интерпретатора размеченной программы совпадает с соединенным состоянием. Что и требовалось доказать.

$BTKindOf_{btHeap}(btOref) = S$, $BTKindOf_{btHeap}(btObj(fld)) = D$ ⇓ согласно правилу создания объекта $(\_, obj_S) = heap_S(oref)$, $obj_S(fld) = DYNVALUE$
$(oref::stack, env, heap) = JoinState_{btHeap}\ (btOref::btStack, btEnv)$ $((oref'::stack_S, env_S, heap_S), ptr2var)\ (stack_D, env_D, heap_D)$ ⇓ $BTKindOf_{btHeap}(btOref) = S$ $oref = oref'$ $stack = JoinStack_{btHeap}\ btStack\ stack_S\ stack_D$ $heap = JoinHeap\ ptr2var\ env_D\ heap_S\ heap_D$ ⇓ $var' = ptr2var(oref, fld)$, $(\_, obj_S) = heap_S(oref)$, $obj_S(fld) = DYNVALUE$ $(\_, obj) = heap(oref)$, $obj(fld) = env_D(var')$ ⇓ $BTKindOf_{btHeap}(btVal) = D$ $(obj(fld)::stack) = JoinStack_{btHeap}\ (btVal::btStack)\ stack_S\ (env_D(var')::stack_D)$ ⇓ $(obj(fld)::stack, env, heap) = JoinState_{btHeap}\ (btVal::btStack, btEnv)$ $(state_S, ptr2var)\ (env_D(var')::stack_D, env_D, heap_D)$

Рис. 29. Проверка совпадения состояний.

7.3.6. Инструкция StoreField^X fld

Инструкция StoreField^X fld обрабатывается аналогично StoreVar^X var – в остаточную программу добавляется инструкция StoreVar var' (рис. 30).

Доказательство проводится в четыре этапа (рис. 31).

Во-первых, так как состояния до шага совпадают, то на вершине стеков лежат одинаковые значения.

Во-вторых, так как поле размечено D, то значение поля объекта в куче генератора остаточной программы равно DYNVALUE. Это следует из правила создания объекта и того, что значения D-полей S-объектов (DYNVALUE) никогда не изменяются.

В-третьих, до шага куча интерпретатора совпадает с соединенной кучей, поэтому значения поля у объектов совпадают. Но так как поле размечено D, то значение поля в соединенном состоянии копируется из значения локаль-

ной переменной. Следовательно, значение поля объекта равно значению локальной переменной. Поэтому, если изменить значение поля объекта в состоянии интерпретатора размеченной программы и значение локальной переменной в состоянии интерпретатора остаточной программы, то куча интерпретатора всё равно будет совпадать с соединенной кучей.

$$\frac{\mathrm{TypeOf}_{heap}(oref) <_{btProg} \mathrm{FieldClass}_{btProg}(fld) \quad oref \neq NULL \quad \mathrm{TypeOf}_{heap}(val) <_{btProg} \mathrm{FieldType}_{btProg}(fld) \quad (class, obj) = heap(oref) \quad heap' = heap[oref \mapsto (class, obj[fld \mapsto val])]}{btProg \vdash_i \mathrm{StoreField}^X\ fld : (val::oref::stack, env, heap) \rightarrow (stack, env, heap')}$$

$$\frac{(\_, \_, btObj) = btHeap(btOref) \quad btVal = btObj(fld) \quad \mathrm{BTKindOf}_{btHeap}(btOref) = S \quad \mathrm{BTKindOf}_{btHeap}(btVal) = D}{btProg, btHeap \vdash_{bt} \mathrm{StoreField}^X\ fld : (btVal::btOref::btStack, btEnv) \rightarrow (btStack, btEnv)}$$

$$btProg \vdash_{rpg} \mathrm{StoreField}^X\ fld : ((oref::stack_S, env_S, heap_S), ptr2var) \rightarrow (((stack_S, env_S, heap_S), ptr2var), [StoreVar\ ptr2var(oref, fld)])$$

$$\frac{\mathrm{TypeOf}_{heapD}(val) <_{rProg} \mathrm{VarType}_{rProg}(var')}{rProg \vdash_i \mathrm{StoreVar}\ var' : (val::stack_D, env_D, heap_D) \rightarrow (stack_D, env_D[var' \mapsto val], heap_D)}$$

Рис. 30. Правила для инструкции $\mathrm{StoreField}^X$ fld.

$$\mathrm{BTKindOf}_{btHeap}(btOref) = S,\ \mathrm{BTKindOf}_{btHeap}(btObj(fld)) = D$$

$$\Downarrow \text{согласно правилу создания объекта}$$

$$(\_, obj_S) = heap_S(oref),\ obj_S(fld) = DYNVALUE$$

$$(val::oref::stack, env, heap) = \mathrm{JoinState}_{btHeap}\ (btVal::btOref::btStack, btEnv)\ ((oref'::stack_S, env_S, heap_S), ptr2var)\ (val'::stack_D, env_D, heap_D)$$

$$\Downarrow \mathrm{BTKindOf}_{btHeap}(btVal) = D,\ \mathrm{BTKindOf}_{btHeap}(btOref) = S$$

$$val = val',\ oref = oref'$$

$$stack = \mathrm{JoinStack}_{btHeap}\ btStack\ stack_S\ stack_D$$

$$heap = \mathrm{JoinHeap}\ ptr2var\ env_D\ heap_S\ heap_D$$

$$\Downarrow var' = ptr2var(oref, fld),\ (\_, obj_S) = heap_S(oref),\ obj_S(fld) = DYNVALUE$$

$$heap[oref \mapsto (class, obj[fld \mapsto val])] = \mathrm{JoinHeap}\ ptr2var\ env_D[var' \mapsto val]\ heap_S\ heap_D$$

$$\Downarrow$$

$$(stack, env, heap') = \mathrm{JoinState}_{btHeap}\ (btStack, btEnv)\ (((stack_S, env_S, heap_S), ptr2var)\ (stack_D, env_D[var' \mapsto val], heap_D)$$

Рис. 31. Проверка совпадения состояний.

И, в-четвертых, так как до шага стеки совпадали, то после выполнения шага (удаления одинаковых значений из стеков), стеки всё равно будут совпадать. Как показано выше, кучи тоже будут совпадать. Следовательно, состояние интерпретатора размеченной программы совпадает с соединенным состоянием. Что и требовалось доказать.

7.3.7. Инструкция LoadElement^X

Инструкция LoadElement^X обрабатывается аналогично LoadField^X fld – в остаточную программу добавляется инструкция LoadVar var' (рис. 32).

Доказательство проводится аналогично LoadField^X fld (рис. 33).

$$\frac{\mathrm{TypeOf}_{heap}(i) = INT \qquad \mathrm{TypeOf}_{heap}(aref) <_{btProg} someType[] \qquad aref \neq NULL \qquad (\_, (len, arr)) = heap(aref) \qquad 0 \le i < len}{btProg \vdash_i \mathrm{LoadElement}^X : (i::aref::stack, env, heap) \rightarrow (arr(i)::stack, env, heap)}$$

$$\frac{(\_, \_, btObj) = btHeap(btOref) \qquad btVal = btObj(ELEMENT) \qquad \mathrm{BTKindOf}_{btHeap}(btOref) = S \qquad \mathrm{BTKindOf}_{btHeap}(btVal) = D}{btProg, btHeap \vdash_{bt} \mathrm{LoadElement}^X : (INT^S::btOref::btStack, btEnv) \rightarrow (btVal::btStack, btEnv)}$$

$$btProg \vdash_{rpg} \mathrm{LoadElement}^X : ((i::aref::stack_S, env_S, heap_S), ptr2var) \rightarrow (((stack_S, env_S, heap_S), ptr2var), [\mathrm{LoadVar}\ ptr2var(aref, i)])$$

$$rProg \vdash_i \mathrm{LoadVar}\ var' : (stack_D, env_D, heap_D) \rightarrow (env_D(var')::stack_D, env_D, heap_D)$$

Рис. 32. Правила для инструкции LoadElement^X.

$\mathrm{BTKindOf}_{btHeap}(btOref) = S,\ \mathrm{BTKindOf}_{btHeap}(btObj(ELEMENT)) = D$

$\Downarrow$ согласно правилу создания массива

$(\_, (\_, arr_S)) = heap_S(aref),\ arr_S(i) = DYNVALUE$

$(i::aref::stack, env, heap) = \mathrm{JoinState}_{btHeap}\ (INT^S::btOref::btStack, btEnv)\ ((i'::aref'::stack_S, env_S, heap_S), ptr2var)\ (stack_D, env_D, heap_D)$

$\Downarrow$

$(i::aref::stack) = \mathrm{JoinStack}_{btHeap}\ (INT^S::btOref::btStack)\ (i'::aref'::stack_S)\ stack_D$

$\Downarrow \mathrm{BTKindOf}_{btHeap}(btOref) = S$

$i = i',\quad aref = aref'$

$stack = \mathrm{JoinStack}_{btHeap}\ btStack\ stack_S\ stack_D$

$heap = \mathrm{JoinHeap}\ ptr2var\ env_D\ heap_S\ heap_D$

$\Downarrow var' = ptr2var(aref, i),\ (\_, (\_, arr_S)) = heap_S(aref),\ arr_S(i) = DYNVALUE$

$(\_, (\_, arr)) = heap(aref) \quad arr(i) = env_D(var')$

$\Downarrow \mathrm{BTKindOf}_{btHeap}(btVal) = D$

$(arr(i)::stack) = \mathrm{JoinStack}_{btHeap}\ (btVal::btStack)\ stack_S\ (env_D(var')::stack_D)$

$\Downarrow$

$(arr(i)::stack, env, heap) = \mathrm{JoinState}_{btHeap}\ (btVal::btStack, btEnv)\ (state_S, ptr2var)\ (env_D(var')::stack_D, env_D, heap_D)$

Рис. 33. Проверка совпадения состояний.

Во-первых, так как состояния до шага совпадают, то на вершине стеков лежат одинаковые значения.

Во-вторых, так как элементы размечены D, то значение элемента массива в куче генератора остаточной программы равно DYNVALUE. Это следует из правила создания массива и того, что значения D-элементов S-массивов

(DYNVALUE), никогда не изменяются.

В-третьих, до шага куча интерпретатора совпадает с соединенной кучей, поэтому значения элементов массивов совпадают. Но так как элементы размечены D, то значение элемента в соединенном состоянии копируется из значения локальной переменной. Следовательно, значение элемента объекта равно значению локальной переменной.

И, в-четвертых, так как до шага стеки совпадали, то после выполнения шага (добавления одинаковых значений в стеки), стеки всё равно будут совпадать.

Следовательно, состояние интерпретатора размеченной программы совпадает с соединенным состоянием. Что и требовалось доказать.

7.3.8. Инструкция StoreElement^X

Инструкция StoreElement^X обрабатывается аналогично StoreField^X fld – в остаточную программу добавляется инструкция StoreVar var' (рис. 34).

$TypeOf_{heap}(i) = INT \quad TypeOf_{heap}(aref) <_{btProg} someType[] \quad aref \neq NULL$ $(type[], (length, arr)) = heap(aref) \quad TypeOf_{heap}(val) <_{btProg} type$ $0 \le n < length \quad heap' = heap[aref \mapsto (type[], (length, arr[i \mapsto val]))]$ $btProg \vdash_i StoreElement^X : (val::i::aref::stack, env, heap) \rightarrow (stack, env, heap')$
$(\_, \_, btObj) = btHeap(btOref) \quad btVal = btObj(ELEMENT)$ $BTKindOf_{btHeap}(btOref) = S \quad BTKindOf_{btHeap}(btVal) = D$ $btProg, btHeap \vdash_{bt} StoreElement^X : (btVal::INT^S::btOref::btStack, btEnv) \rightarrow (btStack, btEnv)$
$btProg \vdash_{rpg} StoreElement^X : ((i::aref::stack_S, env_S, heap_S), ptr2var) \rightarrow (((stack_S, env_S, heap_S), ptr2var), [StoreVar\ ptr2var(aref, i)])$
$TypeOf_{heapD}(val) <_{rProg} VarType_{rProg}(var')$ $rProg \vdash_i StoreVar\ var' : (val::stack_D, env_D, heap_D) \rightarrow (stack_D, env_D[var' \mapsto val], heap_D)$

Рис. 34. Правила для инструкции StoreElement^X.

Доказательство проводится аналогично StoreField^X fld (рис. 35).

Во-первых, так как состояния до шага совпадают, то на вершине стеков лежат одинаковые значения.

Во-вторых, так как элементы размечены D, то значение элемента массива в куче генератора остаточной программы равно DYNVALUE. Это следует из правила создания массива и из того, что значения D-элементов S- массивов (DYNVALUE) никогда не изменяются.

В-третьих, до шага куча интерпретатора совпадает с соединенной кучей, поэтому значения поля объектов совпадают. Но так как элементы массива размечены D, то значения элементов в соединенном состоянии копируется из

значения локальной переменной. Следовательно, значение элемента массива равно значению локальной переменной. Поэтому, если изменить значение элемента массива в состоянии интерпретатора размеченной программы и значение локальной переменной в состоянии интерпретатора остаточной программы, то куча интерпретатора всё равно будет совпадать с соединенной кучей.

И, в-четвертых, так как до шага стеки совпадали, то после выполнения шага (удаления одинаковых значений из стеков), стеки всё равно будут совпадать. Как показано выше, кучи будут совпадать. Следовательно, состояние интерпретатора размеченной программы совпадает с соединенным состоянием. Что и требовалось доказать.

$BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btObj(ELEMENT)) = D$

$\Downarrow$ согласно правилу создания массива

$(\_, (\_, arr_S)) = heap_S(aref),\ arr_S(i) = DYNVALUE$

$(val::i::aref::stack, env, heap) = JoinState_{btHeap}$
$(btVal::INT^S::btOref::btStack, btEnv)\ ((i'::aref'::stack_S, env_S, heap_S), ptr2var)$
$(val'::stack_D, env_D, heap_D)$

$\Downarrow BTKindOf_{btHeap}(btVal) = D,\ BTKindOf_{btHeap}(btOref) = S$

$val=val',\quad i=i',\quad aref = aref'$
$stack = JoinStack_{btHeap}\ btStack\ stack_S\ stack_D$
$heap = JoinHeap\ ptr2var\ env_D\ heap_S\ heap_D$

$\Downarrow var' = ptr2var(oref, fld)\ ,\ (\_, (\_, arr_S)) = heap_S(aref),\ arr_S(i) = DYNVALUE$

$heap[aref \mapsto (type[], (length, arr[i \mapsto val]))] = JoinHeap\ ptr2var\ env_D[var' \mapsto val]$
$heap_S\ heap_D$

$\Downarrow BTKindOf_{btHeap}(btVal) = D$

$(stack, env, heap[aref \mapsto (type[], (length, arr[i \mapsto val]))]) = JoinState_{btHeap}$
$(btStack, btEnv)\ (((stack_S, env_S, heap_S), ptr2var)\ (stack_D, env_D[var' \mapsto val], heap_D)$

Рис. 35. Проверка совпадения состояний.

7.3.9. Инструкция Lifting^X

Инструкция Lifting^X добавляется в программу анализом времен связывания чтобы преобразовать S-данное в D-данное.

Генератор остаточной программы при обработке инструкции Lifting^X добавляет в остаточную программу инструкцию загрузки константы на стек LoadConst val. Значение val берется со стека генератора остаточной программы (рис. 36).

Заметим, что состояния до шага совпадают. Поэтому значение на вершине стека интерпретатора размеченной программы совпадает со значением на вершине стека генератора остаточной программы (рис. 37).

Согласно определению функции $JoinStack_{btHeap}$, если на вершине BT-

стека лежит статическое BT-значение – то на вершине соединенного стека будет лежать значение с вершины S-стека. А если динамическое BT-значение – то значение с вершины D-стека. Поэтому если изменить BT-разметку на вершине BT-стека с S на D и перенести значение с вершины S-стека на вершину D-стек, то соединенный стек не изменится.

$btProg \vdash i\ Lift^X : (val::stack, env, heap) \rightarrow (val::stack, env, heap)$
$btProg,btHeap \vdash bt\ Lift^X : (primType^S::btStack,btEnv) \rightarrow (primType^D::btStack,btEnv)$
$btProg \vdash rpg\ Lift^X : ((val::stack_S, env_S, heap_S), ptr2var) \rightarrow (((stack_S, env_S, heap^S), ptr2var), [LoadConst\ val])$
$rProg \vdash i\ LoadConst\ const : (stack_D,env_D,heap_D) \rightarrow (const::stack_D,env_D,heap_D)$

Рис. 36. Правила для инструкции $Lift^X$.

$$(val::stack, env, heap) = JoinState_{btHeap}\ (primType^S::btStack, btEnv)\ ((val'::stack_S, env_S, heap_S), ptr2var)\ (stack_D, env_D, heap_D)$$
$$\Downarrow$$
$$(val::stack) = JoinStack_{btHeap}\ (primType^S::btStack)\ (val'::stack_S)\ (stack_D)$$
$$\Downarrow$$
$$val=val'$$
$$\Downarrow$$
$$(val::stack) = JoinStack_{btHeap}\ (primType^D::btStack)\ (stack_S)\ (val::stack_D)$$
$$\Downarrow$$
$$(val::stack, env, heap) = JoinState_{btHeap}\ (primType^D::btStack, btEnv)\ ((stack_S, env_S, heap_S), ptr2var)\ (val::stack_D, env_D, heap_D)$$

Рис. 37. Проверка для инструкции $Lift^X$.

Следовательно, состояние интерпретатора размеченной программы будет совпадать с соединенным состоянием после шага доказательства. Что и требовалось доказать.

8. Завершение доказательства

В приведенных выше рассуждениях показано, что при совместной (1) интерпретации размеченной программы для аргументов arg_{S+D}, (2) генерации остаточной программы для аргументов arg_S и (3) интерпретации остаточной программы для аргументов arg_D, состояния интерпретатора исходной программы и соединенные состояния генератора и интерпретатора остаточной программы всегда совпадают:

1. они совпадают на первом шаге – вызове программы;
2. если они совпадают на некотором n-ом шаге, то на следующем шаге они снова будут совпадать.

Отсюда следует, что (1) интерпретация размеченной программы, (2)

генерация остаточной программы и (3) интерпретация остаточной программы будут завершены одновременно, и соответствующие состояния будут совпадать.

Результат программы имеет разметку D согласно правилу разметки программы. Поэтому результат в объединенном состоянии полностью определяется результатом интерпретатора остаточной программы.

Следовательно, если остаточная программа построена для аргументов arg_S, то для любых аргументов arg_D результат интерпретации остаточной программы для аргументов arg_D будет совпадать с результатом интерпретации размеченной программы для аргументов $arg_{S+D.}$

Тем самым доказано, что генератор остаточной программы по корректно построенной разметке порождает остаточную программу, эквивалентную исходной размеченной программе при заданных S-аргументах и произвольном значении D-аргументов.

9. Заключение

В данной работе приведено доказательство корректности генератора остаточной программы. В доказательстве используются правила разметки программы, правила генерации остаточной программы и правила интерпретации программы.

Корректность доказывается индукцией по количеству шагов, при помощи одновременной (1) интерпретации размеченной программы, (2) генерации остаточной программы и (3) интерпретации остаточной программы.

10. Литература

[Chep03] Andrei M. Chepovsky, Andrei V. Klimov, Arkady V. Klimov, Yuri A. Klimov, Andrei S. Mishchenko, Sergei A. Romanenko, and Sergei Yu. Skorobogatov. Partial Evaluation for Common Intermediate Language // M. Broy and A.V. Zamulin (Eds.): Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers. Lecture Notes in Computer Science, volume 2890/2003, pages 171-177. Springer-Verlag Berlin Heidelberg, 2003.

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

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

[Klim05b] Климов Ю.А. О поливариантном анализе времен связывания в

специализаторе объектно-ориентированного языка // Научный сервис в сети Интернет: технологии распределенных вычислений: Труды Всероссийской научной конференции (19-24 сентября 2005 г., г. Новороссийск). М.: Изд-во МГУ, 2005. С. 89-91.

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

[Klim08a] Климов Ю.А. Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных языках // Препринты ИПМ им.М.В.Келдыша. 2008. № 12. 27 с.

[Klim08b] Климов Ю.А. Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках // Препринты ИПМ им.М.В.Келдыша. 2008. № 30. 28 с.

[Klim08c] Климов Ю.А. SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ // Препринты ИПМ им.М.В.Келдыша. 2008. № 44. 32 с.

[Klim08d] Yuri A. Klimov: An Approach to Polyvariant Binding Time Analysis for a Stack-Based Language // A.P. Nemytykh (Ed.): Proceedings of the First International Workshop on Metacomputation in Russia. Pereslavl-Zalessky, Russia, July 2-5, 2008. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008. Pages 78-84.

[Klim09a] Климов Ю.А. Специализатор CILPE: анализ времен связывания // Препринты ИПМ им.М.В.Келдыша. 2009. № 7. 28 с.

[Klim09b] Климов Ю.А. Специализатор CILPE: генерация остаточной программы // Препринты ИПМ им.М.В.Келдыша. 2009. № 8. 26 с.

[Klim09c] Климов Ю.А. Преобразование объектно-ориентированных программ в императивные методом частичных вычислений // Программные продукты и системы. 2009. №2 (86). С. 71-74.