Институт прикладной математики им. М.В. Келдыша РАН

На правах рукописи

Климов Юрий Андреевич

Специализация программ на объектно-ориентированных языках методом частичных вычислений

Специальность 05.13.11 — Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени кандидата физико-математических наук

Научный руководитель:
кандидат физико-математических наук
Романенко Сергей Анатольевич

Москва – 2009

Содержание

Введение ... 4
Глава 1. Специализация программ и метод частичных вычислений ... 9
1.1. Введение ... 9
1.2. Специализация программ методом частичных вычислений ... 11
1.3. Корректность специализатора ... 14
1.4. Анализ времен связывания ... 14
1.5. Генерация остаточной программы ... 17
1.6. Особенности объектно-ориентированных языков ... 18
1.7. Обзор специализаторов для объектно-ориентированных языков ... 18
1.8. Пример ... 25
1.9. Выводы ... 29
Глава 2. Стековый объектно-ориентированный язык SOOL ... 30
2.1. Введение ... 30
2.2. Описание программ на языке SOOL ... 30
2.3. Интерпретация программ на языке SOOL ... 39
2.4. Типизация языка SOOL ... 52
2.5. Выводы ... 67
Глава 3. Анализ времен связывания ... 69
3.1. Введение ... 69
3.2. BT-разметка ... 71
3.3. Правила BT-разметки ... 79
3.4. Построение BT-разметки ... 94
3.5. Выводы ... 96
Глава 4. Генерация остаточной программы ... 97
4.1. Введение ... 97
4.2. Описание генератора остаточной программы ... 97
4.3. Обработка программы ... 102
4.4. Обработка метода ... 104

4.5. Обработка последовательности инструкций 106
4.6. Обработка инструкций 112
4.7. Выводы.. 117
Глава 5. Доказательство корректности 118
5.1. Введение 118
5.2. Структура доказательства корректности.. 118
5.3. База индукции 123
5.4. Шаг индукции: программа.. 123
5.5. Шаг индукции: метод и инструкции CallMethod и Leave........................ 124
5.6. Шаг индукции: последовательность инструкций.................................. 130
5.7. Шаг индукции: инструкции 134
5.8. Завершение доказательства 149
5.9. Выводы.. 150
Заключение 151
Литература 152
Приложение 1 157

Введение

Объект исследования и актуальность работы

Сложность решаемых с помощью компьютера задач, а также затрачиваемое время на их реализацию постоянно возрастают. Для облегчения работы программиста используются различные методы: автоматическое управление памятью, наборы стандартных библиотек, проблемно-ориентированные языки программирования и т.п. Данные методы, с одной стороны, значительно облегчают труд программиста. С другой стороны, эффективность программ при их использовании может значительно снижаться по сравнению с полностью ручной оптимальной и эффективной реализацией, создание которой, однако, может занимать слишком много времени.

В данной работе исследуется проблема оптимизации и преобразования программ на объектно-ориентированных языках программирования на основе априорной информации об аргументах программ или контексте использования объектов и методов (подпрограмм).

Подход к оптимизации программ, основанный на учете дополнительной информации об условиях, в которых будет эксплуатироваться программа, называется *специализацией* программ. В частности, в качестве таких ограничений могут использоваться зафиксированные значения некоторой части исходных данных, не изменяющиеся от одного запуска программы к другому. Или, в общем случае, ограничения на исходные данные могут быть заданы в виде условий, выраженных на некотором языке спецификаций. В таких случаях принято говорить, что выполняется специализация программы по отношению *к ограничениям на исходные данные*.

Методы специализации изначально развивались для функциональных языков, и в этом направлении был достигнут существенный прогресс, причем наиболее широко распространенным подходом является метод *частичных вычислений* (Partial Evaluation, PE).

В настоящее время получили очень широкое распространение объектно-ориентированные языки программирования, такие как C# и Java. Однако методы специализации для объектно-ориентированных языков развиты в гораздо меньшей степени, чем для функциональных языков.

Цель и задачи работы

Целью работы является исследование и разработка основанных на частичных вычислениях методов и алгоритмов для специализации программ на объектно-ориентированных языках программирования, таких как C# и Java. А также проверка работоспособности этих алгоритмов путем создания экспериментального специализатора.

Основные задачи работы:

- Исследование существующих методов частичных вычислений для специализации объектно-ориентированных программ.
- Разработка метода частичных вычислений, обладающего поливариантностью и обеспечивающего повышение эффективности программ за счет выполнения части операций во время специализации, удаления части объектов и изменения представления данных, для полного объектно-ориентированного языка.
- Реализация разработанных методов и алгоритмов в экспериментальном специализаторе и их апробация на модельных задачах.

Научная новизна работы

В работе формально описан новый метод частичных вычислений для объектно-ориентированных языков. В отличие от предшествующих работ, в которых на входной язык специализатора накладывались значительные ограничения, разработанный метод поддерживает все основные конструкции объектно-ориентированных языков, таких как C# и Java.

Существенной особенностью описываемого метода является использование нового понятия: *BT-кучи*. BT-куча является абстрактным описанием со-

стояния реальной кучи (объектов и массивов, созданных в динамической памяти во время исполнения программы) и позволяет описать разделение данных на вычисляемые во время специализации и не вычисляемые. Это, в частности, дает возможность успешно специализировать программы, в которых библиотеки классов используются для определения понятий из некоторой предметной области и служат средством реализации проблемно-ориентированных языков. В результате специализации в остаточной программе вспомогательные объекты уже не создаются, а при необходимости вместо их полей используются локальные переменные.

Практическая значимость работы

Предложенный метод позволяет значительно повысить эффективность программ, в которых используются универсальные (но, возможно, неэффективные) библиотеки, вспомогательные проблемно-ориентированные языки и т.п. Это позволяет программисту не заниматься ручной оптимизацией таких программ, которая обычно требует много времени, приводит к трудно обнаруживаемым ошибкам, затрудняет последующее развитие и сопровождение программ. В результате повышается производительность труда программиста, повышается надежность программы, облегчается ее сопровождение.

Созданный метод может использоваться в компиляторах языков программирования и системах исполнения программ, позволяя без потери эффективности использовать методы, ускоряющие написание программ. Возможно использование разработанного метода в интегрированных средах разработки для анализа программ, поиска зависимостей, понимания чужого кода и т.д.

На основе предложенного метода разработан и реализован экспериментальный специализатор CILPE для промежуточного объектно-ориентированного языка CIL платформы Microsoft .NET.

Апробация работы и публикации

Результаты работы докладывались и обсуждались на:

- пятой международной конференции «Перспективы систем информатики» PSI'03, Россия, Новосибирск, Академгородок, 2003;
- конференции «Технологии Microsoft в научных исследованиях и высшем образовании», Россия, Москва, 2003;
- международной конференции «Microsoft Research Academic Conference: Compiler Architecture and Programming Languages», Венгрия, Будапешт, 2003;
- конференции «Microsoft Research Academic Days», Россия, Санкт-Петербург, 2004;
- Всероссийской конференции студентов, аспирантов и молодых ученых «Технологии Microsoft в теории и практике программирования», Россия, Москва, 2005;
- первом международном семинаре по метавычислениям Meta'08 «First International Workshop on Metacomputation in Russia» Meta'08, Россия, Переславль-Залесский, 2008;
- объединенном научном семинаре по робототехническим системам ИПМ им. М.В. Келдыша РАН, МГУ им. М.В. Ломоносова, МГТУ им. Н.Э. Баумана, ИНОТиИ РГГУ и семинаре отделения «Программирование» ИПМ им. М.В. Келдыша РАН, Россия, Москва, 2009;
- научном семинаре ИСП РАН, Россия, Москва, 2009;
- научном семинаре ЗАО «Авикомп Сервисез», Россия, Москва, 2009;
- научном семинаре ИПС им. А.К. Айламазяна РАН, Россия, Переславль-Залесский, 2009;
- одиннадцатой Всероссийской научной конференции «Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность», Россия, Новороссийск, 2009.

По результатам работы имеются 7 публикаций, включая 1 статью в рецензируемом научном журнале из списка ВАК [9], 1 статью в международном периодическом издании [28], 1 статью в сборнике трудов международного на-

учно-практического семинара [34], 4 статьи в сборниках трудов всероссийских конференций [4,5,6,8].

Глава 1. Специализация программ и метод частичных вычислений

1.1. Введение

С момента своего появления компьютеры применялись для решения задач все большей сложности. При этом увеличивалась и вычислительная мощность компьютеров. В 1965 году Гордон Мур обнаружил закономерность, впоследствии названную «Законом Мура»: количество транзисторов в микросхемах удваивалось каждые полтора-два года. Вместе с удвоением количества транзисторов в микросхеме, удваивалась и производительность компьютеров.

В то время как вычислительная мощность компьютеров росла экспоненциально, сложность задач, решаемых компьютерами, возрастала еще более быстрыми темпами, вследствие чего стало практически невозможно делать глобальную оптимизацию программ вручную. Таким образом, несмотря на рост производительности компьютеров, автоматическая оптимизация программ не только не потеряла свою актуальность, но и даже стало еще более важной.

Языки программирования также непрерывно развивались. В настоящее время для написания программ любого уровня — от простых программ до сложных систем — наиболее широко используются объектно-ориентированные языки типа C++, Java или C#.

Многие современные компиляторы обладают встроенными оптимизаторами. Они преобразуют программу в машинный код, оптимизируя один или несколько показателей, таких как: размер скомпилированной программы, среднее количество исполняемых инструкций, среднее время выполнения программы, объем используемой оперативной памяти и так далее. В результате таких оптимизаций получаем программу, эквивалентную исходной программе, но в некоторых случаях более оптимальную по этим показателям.

Для более эффективной оптимизации необходимо использовать допол-

нительную информацию о программе или об исполняемой системе. Например, для эффективной компиляции используется информация об инструкциях, поддерживаемых данным процессором. В результате такой компиляции получается программа, которая может быть исполнена только на конкретном процессоре, но более эффективно, чем программа, которая может исполняться на целом семействе процессоров.

Подход к оптимизации программ, основанный на учете дополнительной информации об условиях, в которых будет эксплуатироваться программа, называется *специализацией* программ. В частности, ограничения на условия эксплуатации могут заключаться в том, что часть исходных данных известна заранее и не меняется от одного запуска программы к другому. Или, в общем случае, ограничения на исходные данные могут быть заданы в виде каких-то условий. В таких случаях принято говорить, что выполняется специализация программы по отношению *к ограничениям на исходные данные*.

Методы специализации изначально развивались для функциональных языков, и в этом направлении был достигнут существенный прогресс, причем наиболее широко распространенным подходом является метод *частичных вычислений* (Partial Evaluation, PE) [31]. Но для объектно-ориентированных языков методы специализации в настоящее время развиты в гораздо меньшей степени.

В силу широкого применения объектно-ориентированных языков актуально и перспективно разработать методы частичных вычислений для объектно-ориентированных языков.

Основной особенностью объектно-ориентированных языков, отличающей их от функциональных языков, является наличие сложно устроенного состояния машины (стека, переменных, кучи). То есть в каждый момент (непосредственно или косвенно) для чтения или изменения доступны очень многие данные, тогда как в функциональных языках функция может читать значения только своих аргументов и создавать только новые данные, но не может изме-

нять уже созданные данные.

Поэтому многие методы частичных вычислений, применяющиеся для функциональных языков, не могут быть напрямую применены для объектно-ориентированных языков. Необходимо создавать новые идеи и решения, развивать и адаптировать метод частичных вычислений для объектно-ориентированных языков.

1.2. Специализация программ методом частичных вычислений

Оптимизация программ на основе использования априорной информации о значении части переменных называется *специализацией*.

Рассмотрим программу $f(x,y)$ от двух аргументов x и y и значение одного из ее аргументов $x=a$. Результатом специализации программы $f(x,y)$ по известному аргументу $x=a$ называется новая программа одного аргумента $g(y)$, обладающая следующим свойством: $f(a,y)=g(y)$ для любого y.

Частичные вычисления (Partial Evaluation, PE) — это такой метод специализации, который заключается в получении более эффективного кода на основе использования априорной информации о части аргументов и однократного выполнения той части кода, которая зависит только от известной части аргументов (и не зависит от неизвестной части).

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 и будет исполняться, только когда значения этих аргументов станут известны. Цель частичных вычислений — генерация остаточной программы.

Операции и элементы данных, которые выполняются и используются на этапе генерации остаточной программы, называются *статическими* и будут в дальнейшем обозначаться S, а остальные, оставленные в остаточной программе, — *динамическими* — D.

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, C# иJava.

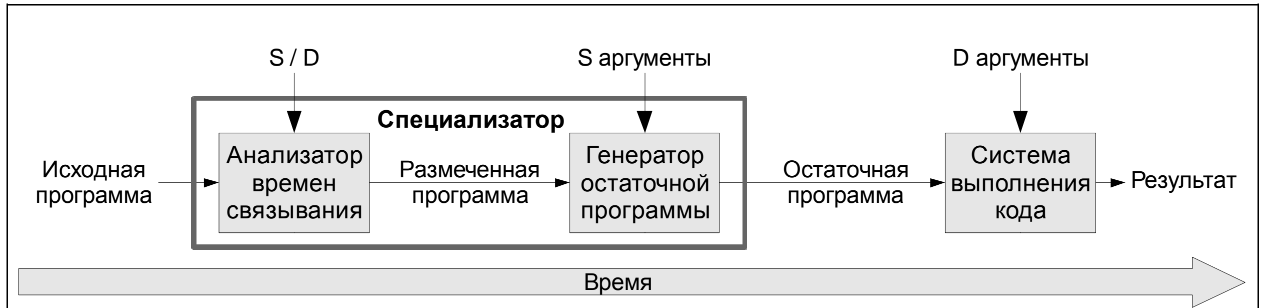


Рис. 1. Общепринятая структура специализатора, основанного на методе частичных вычислений.

Часть метода специализации, отвечающая за разделение операций и данных на статические и динамические, 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.

Для примера рассмотрим функцию возведения в степень, написанную на функциональном языке Haskell (рис. 2). И поставим задачу проспециализировать метод toPower по второму аргументу, равному 5.

Тогда мы считаем, что первый аргумент — динамический, а второй — статический, и строим разметку всей программы (рис. 2: статические данные и операции подчеркнуты и помечены S, динамические — D).

toPower :: Double -> Int -> Double	toPower :: Double^D -> $\underline{\text{Int}}^S$ -> Double^D
toPower x 0 = 1.0	toPower x^D $\underline{0}^S$ = 1.0^D
toPower x n = x*(toPower x (n-1))	toPower x^D $\underline{n}^S$ = x^{D}*^{D}($\underline{\text{toPower}}^S$ x^D ($\underline{n}^S\underline{-}^S\underline{1}^S$))

Рис. 2. Пример: исходная и размеченная программы на функциональном языке.

Проведя генерацию остаточной программы, выполнив все S-действия и оставив D-действия, получаем остаточную программу (рис. 3):

```
toPower5 :: Double -> Double
toPower5 x = x*x*x*x*x*1.0
```

Рис. 3. Пример: результат специализации программы на функциональном языке.

В результате остаточная программа не содержит проверок показателя степени на равенство нулю и операции с этим показателем.

Аналогично можно провести специализацию и императивной программы возведения в степень. Разметка программы строится аналогично (рис. 4).

double toPower (**double** x, **int** n) {
  if (n == 0)
    return 1.0;
  else
    return x*toPower(x, n-1);
}
	double^D toPower (**double** x^D, **int** $\underline{n}^S$) {
	  $\underline{\textbf{if}}^S$ ($\underline{n}^S$ $\underline{==}^S$ $\underline{0}^S$)
	    return^D 1.0^D;
	  $\underline{\textbf{else}}^S$
	    return^D x^D*^{D}$\underline{\text{toPower}}^S$(x^D, $\underline{n}^S$ $\underline{-}^S$ $\underline{1}^S$);
	}

Рис. 4. Пример: исходная и размеченная программы на императивном языке.

В результате специализации останутся только операции с *x* (рис. 5).

```
double toPower5 (double x) {
   return x*x*x*x*x*1.0;
}
```

Рис. 5. Пример: результат специализации программы на императивном языке.

Следует отметить, что BT-анализ и генератор остаточной программы могут работать как одновременно, так и по очереди: сначала анализ времен связывания, затем генератор остаточной программы. В первом случае мы получим так называемый online PE, во втором — offline PE. В дальнейшем мы будем говорить только об offline PE.

В offline частичном вычислителе сначала конструкции программы размечаются на статические (S) и динамические (D) с помощью анализа времен связывания. В процессе анализа могут быть выполнены некоторые преобразования программы для улучшения качества разметки (например, добавлены

операции поднятия Lifting).

После этого генератор остаточной программы, используя частично заданные аргументы и размеченную программу, выполняет все статические операции, а все динамические операции переносит в остаточную программу. В результате получаем остаточную программу, зависящую только от аргументов, неизвестных во время специализации.

Далее подробно рассмотрим оба этапа offline частичных вычислений: анализ времен связывания и генерация остаточной программы.

1.3. Корректность специализатора

Естественным и необходимым свойством любого специализатора программ является его *корректность*: при заданных ограничениях на условия работы программы, результат специализированной версии программы не должен отличаться от результата исходной программы.

Рассмотрим программу f от переменных $x,\dots,y,u,\dots,v$ и множество значений переменных D. Пусть значения аргументов $x,\dots,y$ известны и равны $x_0,\dots,y_0 \in D$. Пусть в результате специализации программы f при известных значениях $x_0,\dots,y_0$ переменных $x,\dots,y$, получилась программа g, зависящая от переменных $u,\dots,v$, неизвестных во время генерации остаточной программы. Тогда для любых значений $u_1,\dots,v_1 \in D$ переменных $u,\dots,v$ результат вычисления программы f от значений $x_0,\dots,y_0,u_1,\dots,v_1$ и программы g от значений $u_1,\dots,v_1$ должны совпадать (рис. 6).

Если $spec(f,x_0,\dots,y_o) = g$, где $x_0,\dots,y_o \in D$
то $\forall u_1,\dots,v_1 \in D : f(x_0,\dots,y_o,u_1,\dots,v_1) = g(u_1,\dots,v_1)$

Рис. 6. Определение корректности специализации.

1.4. Анализ времен связывания

Цель анализа времен связывания — разделить программу на статическую и динамическую части. Такое разделение может быть произведено несколькими способами. Однако необходимо, чтобы статические операции в та-

ком разделении всегда получали только статические данные, а операции, которые в каких-то случаях могут получить динамические данные, должны быть размечены как динамические.

Такое разделение, как правило, может быть выполнено различными способами. Можно даже все данные отнести к динамическим. И для любого способа остаточная программа будет получаться правильной (если разделение построено согласно описанным ниже правилам). Но чем больше операций будет отнесено к статическим, тем более эффективной будет остаточная программа.

Для разделения на статическую и динамическую части анализ времен связывания использует BT-разметку: разметку инструкций, элементов стека и переменных. Построенная разметка должна быть внутренне согласованной, удовлетворять правилам, которые будут описаны ниже: например, всякая статическая операция должна иметь статические аргументы, а динамическая — динамические. Статическое данное может стать аргументом динамической операции, если оно будет предварительно преобразовано специальной *операцией поднятия* Lifting. Обратный переход — из динамики в статику — невозможен.

Для достижения согласованности, помимо добавления операций поднятия, исходная программа может быть подвергнута и другим преобразованиям.

Возможно несколько вариантов реализации BT-анализа [2,26,41]:

1. *Моновариантный по переменным* — каждую переменную разрешается разметить одним способом независимо от точки ее использования в программе.
2. *Поливариантный по переменным* — разрешается в разных частях программы размечать одну и ту же переменную по-разному. Это позволяет более эффективно производить специализацию кода, в котором одна и та же переменная используется в разных местах программы с разными целями.
3. *Моновариантный по операциям* — запрещается преобразовывать и раз-

множать код внутри метода. Следовательно, каждая операция размечается только одним способом независимо от пути вычисления, по которому можно прийти к данной операции.

4. *Поливариантный по операциям* — можно преобразовывать и размножать код внутри метода, что позволяет одному участку кода исходной программы сопоставить несколько участков выходной размеченной программы с разными ВТ-разметками. Это позволяет более качественно разметить программу, отнеся к статическим заметно большую часть программы. В то же время существует опасность многократного роста размера программы.
5. *Моновариантный по методам* (подпрограммам) — каждый метод размечается только одним способом. В этом случае каждый метод будет размечен одним способом независимо от контекста его использования.
6. *Поливариантный по методам* (подпрограммам) — каждый метод может быть размечен в зависимости от его использования, в зависимости от разметки аргументов и результатов.
7. *Моновариантный по классам* — каждый класс размечается только одним способом. В таком варианте все объекты данного класса будут одинаково проспециализированы, независимо от особенностей их использования.
8. *Поливариантный по классам* — каждый класс может иметь несколько разметок. Это позволяет по-разному разметить различные использования объектов данного класса.

В итоге, анализ времен связывания по программе и начальной разметке, которая содержит разметку входных данных и, быть может, разметку других элементов, заданных пользователем, должен построить ВТ-разметку всех операций и используемых данных, удовлетворяющую правилам согласованной разметки в программе.

1.5. Генерация остаточной программы

После построения корректно размеченной программы, генератор остаточной программы, с помощью частичных вычислений, строит окончательный результат специализации: остаточную программу.

На вход генератору остаточной программы поступают размеченная программа и значения S-переменных. По этим данным он строит остаточную программу, зависящую только от части переменных — только от D-переменных, динамических исходных данных, неизвестных во время специализации программы.

Если разметка программы была построена согласованно, то генератор остаточной программы построит программу, которая будет эквивалентна исходной программе при указанных значениях S-переменных и всех значениях D-переменных. Это основное свойство генератора остаточной программы.

Генератор остаточной программы производит обобщенное выполнение программы: S-инструкции выполняются, а D-инструкции переходят в остаточную программу. Собственно на этом этапе выполняется часть операций, откуда и происходит термин «частичные вычисления».

В процессе порождения остаточной программы генератор сравнивает состояния для возможного построения циклов или рекурсии в остаточной программе. Но при некоторых начальных данных, т.е. для некоторых программ и их разметок, генератор остаточной программы может работать бесконечно долго, пытаясь построить остаточную программу.

Если работа генератора остаточной программы завершается, то получается остаточная программа эквивалентная исходной. Если в задании на частичные вычисления часть аргументов метода была статической, то в результате порождается новый метод, зависящий только от оставшихся динамических аргументов. И вызов метода необходимо вставить в остаточную программу.

Если же в задании на частичные вычисления все аргументы были динамическими, а частичные вычисления производились только по внутренним

статическим данным, то в результате получаем метод, полностью эквивалентный исходному методу. Поэтому в остаточную программу вставляется эквивалентный метод (хотя, может быть, и оптимизированный).

1.6. Особенности объектно-ориентированных языков

Функциональные языки обладают множеством свойств, облегчающих анализ программ. К числу таких свойств можно отнести отсутствие побочных эффектов, отсутствие глобального состояния, передача аргументов и результатов по значению, и т.п.

Объектно-ориентированные языки не обладают этими удобными свойствами, поэтому при анализе и преобразовании программ приходится учитывать:

1. Наличие глобального *изменяемого* состояния (в функциональных языках такое состояние отсутствует).
2. Наличие ссылок на элементы (объекты) глобального состояния (которые в некоторых языках могут идентифицироваться по именам, известным во время компиляции программы).
3. Объекты могут состоять из полей (обращение к которым происходит по имени, известному во время компиляции), либо являться массивами (обращение к элементам которых происходит по номерам, неизвестным во время компиляции программы).
4. Передача аргументов и результатов по ссылке (в функциональных языках аргументы и результаты передаются по значению).
5. Представление программы в виде последовательности инструкций (а не в виде графа передачи данных).
6. Наличие конструкций передачи управления (цикл, goto).
7. Наличие конструкции наследования классов.
8. Наличие виртуальных методов.

1.7. Обзор специализаторов для объектно-ориентированных

языков

Существует много работ, посвященных частичным вычислениям для функциональных языков. Одной из фундаментальных работ, посвященных методам частичных вычислений, является работа N.D. Jones, C.K. Gomard, P. Sestoft «Partial Evaluation and Automatic Program Generation» 1993г. [31]. В ней в основном рассматриваются методы частичных вычислений для функциональных языков.

Методы частичных вычислений развивались для императивных языков Алгол [27,30] и С [19]. Специализатору для языка С посвящена одна глава «Частичные вычисления для языка С» [18] в [31].

Существует немного работ по специализации и частичным вычислениям для языков с объектно-ориентированными особенностями, к рассмотрению которых мы и переходим.

1.7.1. Лямбда-исчисления с императивными конструкциями

Одним из направлений исследований частичных вычислителей для языков с императивными конструкциями является разработка частичного вычислителя для лямбда-исчисления с побочными эффектами, выполненная Кэнъити Асай (Kenichi Asai) из университета города Токио [23]. В представленном лямбда-исчислении присутствуют операции для изменения значения выражения. В работе предложен алгоритм разделения данных на изменяемые и неизменяемые и построен частичный вычислитель, использующий такое разделение. В нем обращения к потенциально изменяемым данным переходят в остаточную программу.

Вторым направлением работ К. Асай — разработка частичного вычислителя для лямбда-исчисления, способного одновременно генерировать код для вычисления выражения и вычислять это выражение [20]. Такая возможность используется для преобразования данных из статических в динамические: в момент преобразования выражение строится не сразу, а постепенно, и его части могут использоваться для преобразования других данных из стати-

ческих в динамические.

Третьим направлением работ группы под руководством К. Асай является создание частичного вычислителя для лямбда-исчисления с продолжениями (continuations) [22].

Разработанные К. Асай методы расширяют возможности классического метода частичных вычислений, делая его применимым к лямбда-исчислению с побочными эффектами. Этого все же недостаточно для применения этого метода к объектно-ориентированным языкам.

1.7.2. Специализаторы для языка C

Первыми работами по созданию специализаторов для императивных языков являются работы по созданию специализатора для языка C [16, 17,18,19,24,29,36]. Однако язык C сложен для автоматического анализа: его семантика не всегда достаточно прозрачна. Также трудности связаны со структурой данных, с которыми он работает: фактически вся память представляет собой один большой массив, из-за чего затруднительно реализовать достаточное для практических задач разделение данных на статические и динамические.

Также и сами программы на языке C обычно пишутся в сложном для анализа стиле: сам язык провоцирует программиста писать программы в таком стиле, который затрудняет автоматический анализ программ.

C-Mix

Еще в начале 90-х годов Ларс Оле Андерсен (Lars Ole Andersen) в университете Копенгагена в Дании разработал [16,17,18,19] специализатор C-MIX для языка C.

Специализация программы на языке C проводится в несколько этапов:

1. Первый этап — анализ ссылок. Для каждой переменной-ссылки строится множество переменных и областей памяти, на которые она может ссылаться. Если несколько ссылок-переменных в результате анализа мо-

гут ссылаться на одно и то же, то они должны быть размечены одинаково.

2. Второй этап — построение разметки времен связывания. Ссылки на одно и то же должны быть размечены одинаково.
3. Третий этап — генерация остаточной программы.

Tempo

В результате развития идей, заложенных в C-Mix, группой Шарля Консель (Charles Consel) (Университет города Копенгагена, Дания), Джулии Лаваль (Julia L. Lawall), Анн-Франсуаза Лё Мёр (Anne-Françoise Le Meur) в лаборатории исследований информатики и вычислительной техники в Бордо во Франции был создан специализатор Tempo [29,36]. Это специализатор для языка C, во многом аналогичный C-Mix.

1.7.3. Специализация для байт-кода языка Java

Работы в университете города Токио

В университете города Токио под руководством Акинори Ёнэдзава (Akinori Yonezawa) и Хидэхико Масухара (Hidehiko Masuhara) разрабатывается специализатор для байт-кода языка Java, который можно было бы встроить в JIT-компилятор байт-кода.

Описанный в работе [38] специализатор обрабатывает только примитивные данные и статические методы. Допускается различная разметка переменных в различных точках программы. В работе показано, как для метода построить систему ограничений на разметку локальных переменных и инструкций.

В поздней работе [15] группой Рейналд Аффелдт (Reynald Affeldt), Хидэхико Масухара (Hidehiko Masuhara), Эйдзиро Сумми (Eijiro Summi), Акинори Ёнэдзава (Akinori Yonezawa) предложено расширение специализатора, позволяющее обрабатывать и объекты. Разметка переменных описывается деревьями: S или D в вершине дерева определяет статический или динамический

объект будет находиться в этой переменной, а поддеревья определяют разметку полей объекта.

Анализ метода основан на разделении объектов на локальные объекты (которые создаются внутри этого метода) и нелокальные (которые передаются методу извне).

Локальные объекты делятся на те, которые могут быть выданы в качестве результата работы метода, и те, которые используются только внутри метода. Нелокальные объекты разделяются на те, в поля которых может осуществляться присваивание, и те, в поля которых присваивание не происходит. Инструкции создания локальных объектов, которые могут быть выданы, и присваивания в поля нелокальных объектов обязаны размечаться динамически и переходить в остаточную программу.

После разделения строится разметка всех инструкций в программе.

В обеих работах [15,38] описаны специализаторы времени исполнения (run-time specialization). По построенной разметке строится метапрограмма: программа, генерирующая результат специализации по известным значениям статических аргументов. Во время выполнения остаточной программы, сначала по статической части аргументов строится специализированная версия метода, а затем эта специализированная версия метода выполняется.

Работы П. Бертелсена

В королевском университете (Royal Veterinary and Agriculture University) в городе Копенгагене в Дании Питер Бертелсен (Peter Bertelsen) занимался семантикой и анализом времен связывания программ на языке Java.

В своей работе [25] П. Бертелсен рассматривает подмножество байт-кода языка Java. Это подмножество содержит инструкции для работы с локальными переменными, операции на стеке с примитивными данными и массивами и инструкции перехода Goto и If. Инструкций вызовов методов и возможность создания объектов нет.

В работе описаны допустимые разметки и ограничения на корректную

разметку программы и предложен метод нахождения решения для системы ограничений.

Описан генератор остаточной программы, который по размеченной программе и значениям статических аргументов строит остаточную программу.

Работы У. Шульца

В университете города Реннес (Rennes) во Франции Ульрик Пагх Шульц (Ulrik Pagh Schultz) в рамках своей диссертации разработал частичный вычислитель для подмножества байт-кода языка Java [42,43].

Описанное подмножество относительно широко: можно описывать классы, конструкторы и методы. Разрешены операции над примитивными данным и виртуальные вызовы.

Но отсутствуют операции работы с массивами, возможность изменять значения полей объектов, тело метода представляется не в виде последовательности инструкций, а в виде одного выражения.

Как и в предыдущих работах, программе сопоставляется система ограничений на разметку. Но в отличие от них, разметка строится для описаний классов, а не для переменных в программе. Разметка программы строится путем решения системы ограничений.

На основе размеченной программы генератором остаточной программы по известным значениям статических переменных строится остаточная программа.

В следующих главах У. Шульц описывает усовершенствования, которые необходимы для работы с реальными программами:

1. class polyvariance — класс размечается несколькими способами и каждому new соответствует одна из разметок;
2. method polyvariance — для каждого вызова метода метод размечается отдельно;
3. alias polyvariance — анализ, помечающий где могут быть одинаковые данные, а где — разные. Желательно, чтобы и alias analysis был полива-

риантным (размножал классы и методы).

Однако математического аппарата для реализации такого рода усовершенствований не предлагается.

1.7.4. Классификация специализаторов для объектно-ориентированного языка Java

Можно сделать вывод, что в рассмотренных выше работах предложены различные методы частичных вычислений для подмножеств объектно-ориентированного языка Java. При этом на каждое из этих подмножеств наложены сильные ограничения, которые обычно не выполняются для реальных программ на языке Java (таб. 1).

Для практического применения к объектно-ориентированным языкам метод частичных вычислений должен:

1. Обрабатывать инструкции передачи управления Goto и If.
2. Поддерживать работу с объектами и массивами, допускать возможность изменения полей объектов и элементов массива.
3. Обладать единым анализом времен связывания (не проводить отдельный alias analysis).
4. Обладать поливариантностью по переменным (допускать несколько различных разметок одной и той же переменной в различных точках программы) (таб. 2).
5. Обладать поливариантностью по инструкциям (допускать несколько различных разметок частей метода в зависимости от разметки переменных и стека) (таб. 2).
6. Обладать поливариантностью по методам (допускать несколько различных разметок метода в зависимости от разметки аргументов и результатов метода) (таб. 2).
7. Обладать поливариантностью по классам и массивам (допускать несколько различных разметок однотипных переменных) (таб. 2).
8. Обладать расширенной разметкой объектов и массивов: одни поля ста-

тических объектов могут иметь статическую разметку, в то время как другие поля — динамическую.

	Инструкции передачи управления Goto и If	Работа с	
		массивами	объектами
Х. Масухара	да	нет	да
П. Бертелсен	да	да	нет
У. Шульц	нет	нет	да

Таб. 1. Классификация специализаторов на базе метода частичных вычислений по полноте реализации объектно-ориентированного языка.

	Поливариантность по			
	Переменным	инструкциям	методам	классам / массивам
Х. Масухара	да	нет	нет	да
П. Бертелсен	да	нет	нет	да
У. Шульц	нет	нет	нет	нет

Таб. 2. Классификация специализаторов на базе метода частичных вычислений по поливариантности.

1.8. Пример

Следующий пример взят из книги «Refactoring to Patterns» [33] глава «Replace Implicit Language with Interpreter».

Допустим, что имеется набор объектов таких, что каждый объект обладает каким-нибудь набором свойств. И нужно отбирать объекты по этим свойствам с помощью каких-нибудь критериев.

Например, свойством объекта может быть значение целочисленного поля объекта, и нужно отобрать те объекты, у которых значение этого поля лежит в заданном интервале.

Обычное решение такой проблемы заключается в написании метода, в цикле проверяющего свойства каждого объекта. Но если нужно отбирать объекты по разным логическим условиям, то в программе возникает много методов, тела которых очень похожи: в каждом из методов требуется «прокручивать» цикл по всем объектам.

Структуру программы можно улучшить, вынеся общие части многих методов в один метод, который принимает на вход предикат, представленный в виде объекта и описывающий логическое условие, по которому следует провести поиск. Таким образом, цикл, проходящий по всем объектам и применяющий предикат к каждому объекту, появляется в теле только одного метода.

Часто бывает нужно не только построить несколько предикатов, но и уметь их комбинировать с помощью логический операций И, ИЛИ, НЕ. Тогда можно реализовать не только объекты-предикаты, но и объекты-операции.

```
class Product {
   public int color;
   public double price;
   public Product (int color, double price) {
      this.color = color; this.price = price;
   }
}
abstract class Spec {
   [Inline] public Spec () { }
   [Inline] public abstract bool IsSatisfiedBy (Product product);
}
class AndSpec : Spec {
   Spec x, y;
   [Inline] public AndSpec (Spec x, Spec y) { this.x = x; this.y = y; }
   [Inline] public override bool IsSatisfiedBy (Product product) {
      return this.x.IsSatisfiedBy(product) && this.y.IsSatisfiedBy(product);
   }
}
class NotSpec : Spec {
   Spec x;
   [Inline] public NotSpec (Spec x) { this.x = x; }
   [Inline] public override bool IsSatisfiedBy (Product product) {
      return ! this.x.IsSatisfiedBy(product);
   }
}
class ColorSpec : Spec {
   int color;
   [Inline] public ColorSpec (int color) { this.color = color; }
   [Inline] public override bool IsSatisfiedBy (Product product) {
```

```
      return product.color == this.color;
   }
}
class BelowPriceSpec : Spec {
   double price;
   [Inline] public BelowPriceSpec (double price) { this.price = price; }
   [Inline] public override bool IsSatisfiedBy (Product product) {
      return product.price < this.price;
   }
}
class ProductFinder {
   IEnumerable repository;
   public ProductFinder (IEnumerable repository) {
      this.repository = repository;
   }
   [Inline] public IList SelectBy (Spec spec) {
      IList foundProducts = new ArrayList();
      IEnumerator products = this.repository.GetEnumerator();
      while (products.MoveNext()) {
         Product product = (Product) products.Current;
         if (spec.IsSatisfiedBy(product))
            foundProducts.Add(product);
      }
      return foundProducts;
   }
   [Specialize]
   public IList BelowPriceAvoidingAColor (double price, int color) {
      Spec spec = new AndSpec(new BelowPriceSpec(price), new NotSpec(
                                                new ColorSpec(color)));
      return SelectBy(spec);
   }
}
```

Рис. 7. Исходная программа

В [33] рассмотрен следующий пример (рис. 7). Имеется класс Product для описания продуктов, имеющий два поля-критерия: цвет и цена. И есть абстрактный класс Spec с виртуальным методом-предикатом IsSatisfiedBy(Product) для описания предикатов. Описаны два объекта-предиката ColorSpec и BelowPriceSpec — для поиска товара по цвету и по цене. А также два объекта-операции AndSpec и NotSpec (для взятия логического

И и НЕ для предикатов). Тогда, чтобы найти все товары с цветом, отличным от заданного, и ценой, менее заданной, достаточно создать следующий объект: new AndSpec(new BelowPriceSpec(price), new NotSpec(new ColorSpec(color))).

Такой способ более удобен для чтения, анализа и преобразования программы. Но такая программа может работать менее эффективно, чем вариант с явными проверками полей объектов, поскольку при каждой проверке истинности предиката происходит обход дерева объектов, описывающих предикат. То есть вычисление предиката происходит «в режиме интерпретации».

```
class Product {
   public int color;
   public double price;
   public Product (int color, double price) {
      this.color = color; this.price = price;
   }
}
class ProductFinder {
   IEnumerable repository;
   public ProductFinder (IEnumerable repository) {
      this.repository = repository;
   }
   public IList BelowPriceAvoidingAColor (double price, int color) {
      IList foundProducts = new ArrayList();
      IEnumerator products = this.repository.GetEnumerator();
      while (products.MoveNext()) {
         Product product = (Product) products.Current;
         if (product.price<price && !(product.color==color))
            foundProducts.Add(product);
      }
      return foundProducts;
   }
}
```

Рис. 8. Результат специализации.

Для получения эффективной программы в данном случае уместно использовать методы специализации. Частичный вычислитель для объектно-ориентированных языков должен уметь специализировать такого рода программы, убирая создание и использование временных объектов, создание и

использование которых полностью прослеживается в период специализации.

В данном примере, частичный вычислитель, удовлетворяющий требованиям, сформулированным выше, уберет создание «лишних» объектов AndSpec, BelowPriceSpec, NotSpec и ColorSpec, а в том месте, где был вызов метода IsSatisfiedBy, поставит явную проверку полей объекта (рис. 8).

Данный пример показывает, что методы специализации позволяют использовать высокоуровневые методы программирования без потери эффективности.

1.9. Выводы

В главе рассмотрены существующие в данный момент специализаторы, основанные на методе частичных вычислений и применимые к программам, написанным на императивных и объектно-ориентированных языках. По этим работам видно, что полноценного специализатора для объектно-ориентированных языков типа C# или Java, применимого к реалистичным программам, пока не создано. Однако большая практическая важность таких языков требует создания подобных специализаторов.

В работах автора [4,6,8,10,11,12,13,34] описаны основные принципы, на которых основан специализатор для объектно-ориентированного языка, близкого к C# и Java, который обладает всеми существенными возможностями, присущими языкам C# и Java, что позволяет непосредственно применить разработанные методы для специализации внутреннего языка платформы Microsoft.NET [3,5,7,9,28].

Глава 2. Стековый объектно-ориентированный язык SOOL

2.1. Введение

В настоящее время существует множество объектно-ориентированных языков: C++, C#, Java, Python и другие. Эти языки используются для написания программ человеком, поэтому они достаточно сложны для анализа и описания методов преобразования программ. Для формального описания методов анализа и преобразования программ удобно использовать модельные языки программирования.

В главе представлен модельный стековый объектно-ориентированный язык SOOL (Stack-based Object-Oriented Language). За основу языка SOOL взято подмножество языка CIL платформы Microsoft .NET [46]. Он также очень близок к языку виртуальной машины Java [47]. Язык SOOL используется в специализаторе CILPE [3,5,9,28] языка CIL.

Язык SOOL содержит операции над стеком, операции ветвления, операции обращения к переменным, операции над данными примитивного и ссылочного типа. Операции над примитивными типами включают операции сравнения, арифметические и побитовые операции. Операции над ссылочными типами включают операции создания объектов и массивов, обращение к полям объектов и элементам массивов, вызовы методов объектов, приведение объектов к заданному типу, сравнение объектов и массивов по адресу. Язык SOOL не содержит операции для работы с исключениями, структурами и указателям.

2.2. Описание программ на языке SOOL

2.2.1. Программа

Программа на языке SOOL состоит из определений классов (рис. 9). Определение класса содержит:

- Имя класса. Имя класса должно быть уникальным в программе.

- Имена классов, *непосредственным наследником* которых является данный класс. Запрещается циклическое наследование: например, когда класс A объявлен наследником класса B, класс B — наследником класса C, а класс C — наследником класса A.
- Объявления полей этого класса. Объявление поля состоит из имени поля и его типа. Имена полей должны быть уникальными в программе.
- Определения методов этого класса. Определение метода состоит из имени метода, *сигнатуры* метода (типов аргументов и результатов метода), а также тела метода. Тип первого аргумента метода обязан совпадать с именем класса, в котором данный метод определен. Имена методов также должны быть уникальными в программе, за исключением *переопределенных методов*.

```
Program = [ClassDef]
ClassDef = (ClassName, [ClassName], [FieldDec], [MethodDef])
FieldDec = (FieldName, Type)
MethodDef = (MethodName, [Type], [Type], MethodBody)
ClassName = String       FieldName = String       MethodName = String
```

Рис. 9. Абстрактный синтаксис программ на языке SOOL.

При определении классов в объектно-ориентированных языках обычно допускается повторение имен полей и методов:

- **Экранирование полей.** Поля с одинаковыми именами могут встречаться в разных классах (но не могут в одном классе). Например, пусть в одном классе C_1 определено поле F. При определении класса C_2, наследующего класс C_1, также было определено поле F. Тогда у объекта класса C_2 будут два разных поля F: одно от класса C_1, другое от класса C_2. В зависимости от языка при обращении необходимо явно или неявно указывать к какому полю F происходит обращение.
- **Перегрузка методов.** Методы с одинаковыми именами, но с разной сигнатурой могут быть определены в одном классе. В точке вызова выбор нужного метода производится до выполнения программы по типам

аргументов.

- **Переопределение методов (виртуальные методы).** Методы с одинаковыми именами и с одинаковой сигнатурой (за исключением типа первого аргумента) могут быть определены в разных классах. В этом случае выбор нужного метода производится во время выполнения программы по типу значения первого аргумента. Часто, чтобы отличать перегрузку и переопределение методов, в языке вводят соответствующие ключевые слова.

В языке SOOL запрещены *экранирование полей* и *перегрузка методов*, но разрешено *переопределение методов*: все имена полей и методов должны быть различны, за исключением имен переопределенных методов. Уникальности имен полей и методов можно достигнуть переименованием.

Переопределение метода — это ситуация, когда в разных классах определяются методы с одинаковыми именами. Если в разных классах определены методы с одинаковыми именами, то требуется, чтобы типы аргументов, за исключением первого аргумента, и типы результатов совпадали. Также требуется, чтобы методы с одинаковыми именами образовывали иерархию: существовал *основной* класс с определением метода с данным именем, чтобы все остальные классы, содержащие определение метода с этим именем, являлись *наследниками* (но не обязательно непосредственными наследниками) этого класса. Такой класс будем называть основным классом для данного метода.

В каждой программе должен быть определен класс MAIN, в котором определен метод Main. Типы аргументов и результатов метода Main должны быть примитивными типами, за исключением первого аргумента (тип которого должен быть MAIN). Такое ограничение связано с тем, что значения других типов невозможно задать без кучи.

2.2.2. Типы

В языке SOOL данные могут быть либо *примитивного*, либо *ссылочного* типа (рис. 10).

К примитивным типам относятся тип целых чисел INT и тип чисел с плавающей запятой FLOAT. Если про переменную сказано, что она имеет примитивный тип INT или FLOAT, то во время выполнения программы в этой переменной может находиться только целое число или число с плавающей запятой соответственно.

```
Type = PrimType | RefType
PrimType = INT | FLOAT
RefType = ClassType | ArrayType | NULLTYPE | OBJECT
ClassType = ClassName
ArrayType = Type[]
```

Рис. 10. Типы.

Ссылочными типами являются массивы и классы. Тип массивов описывается типом элементов массива, тип классов описывается именем класса. Имеются два специальных ссылочных типа: NULLTYPE и OBJECT, обозначающих тип значения NULL и общий тип всех ссылочных типов соответственно.

Будем говорить, что тип *определен программой* prog, если:

- это встроенный тип INT, FLOAT, NULLTYPE, OBJECT;
- это имя класса, определенного в программе prog;
- это тип массива type[], где type определен программой prog.

По программе prog на типах, определенных программой, определим рефлексивное и транзитивное отношение *наследования* $<_{prog}$ по следующим правилам:

- **Наследование.** Если в программе prog в определении класса $class_1$ указано, что он является непосредственным наследником класса $class_2$, то говорим, что $class_1$ наследует $class_2$:

  $(class_1, [\ldots, class_2, \ldots], fields, methods) \in prog \Rightarrow class_1 <_{prog} class_2$

- **Рефлективность.** Тип наследует сам себя: $type <_{prog} type$
- **Транзитивность.** Если тип $type_1$ наследует тип $type_2$, а тип $type_2$ — тип

$type_3$, то $type_1$ наследует $type_3$:

$type_1 <_{prog} type_2, type_2 <_{prog} type_3 \Rightarrow type_1 <_{prog} type_3$

- **Супертип.** Любой ссылочный тип refType (будь то тип массива или класса) наследуется от специального ссылочного типа OBJECT, и специальный тип NULLTYPE наследует тип refType:
  $NULLTYPE <_{prog} refType <_{prog} OBJECT$
- **Массивы.** Если тип $type_1$ наследует тип $type_2$, то тип массива $type_1[]$ наследует тип массива $type_2[]$: $type_1 <_{prog} type_2 \Rightarrow type_1[] <_{prog} type_2[]$
- **Примитивные типы.** Примитивные типы являются наследниками только самих себя: $INT <_{prog} INT$, $FLOAT <_{prog} FLOAT$

Определив отношение для типов, продолжим его на список типов. Если списки типов имеют одинаковую длину, и каждый тип первого списка является наследником соответствующего типа второго списка, то будем говорить, что первый список наследует второй список:

$x_1 <_{prog} y_1, \ldots, x_n <_{prog} y_n \Rightarrow [x_1, \ldots, x_n] <_{prog} [y_1, \ldots, y_n]$

2.2.3. Метод

Тело определения метода состоит из списка объявлений переменных и списка инструкций (рис. 11).

```
MethodBody = ([VarDec], [Instruction])
VarDec = (VarName, Type)
VarName = String
Instruction = Leave | Goto N | Branch N | DuplicateStackTop | RemoveStackTop |
        LoadConst Const | UnaryOp Uop | BinaryOp Bop | LoadVar VarName |
        StoreVar VarName | NewObject ClassName | LoadField FieldName |
        StoreField FieldName | CallMethod MethodName | CastObject RefType |
        NewArray Type | LoadLength | LoadElement | StoreElement
N = Integer
Const = Integer | Float | NULL
Uop = NEG | NOT | INT2FLOAT | FLOAT2INT
Bop = ADD | AND | CEQ | CGT | CLT | DIV | MUL | OR | REM | SHL | SHR |
        SUB | XOR
```

Рис. 11. Тело метода.

Объявление переменной состоит из имени переменной и ее типа. Имена переменных должны быть уникальными во всей программе. Этого можно добиться переименованием имен переменных.

Инструкции в методе выполняются последовательно согласно расположению в списке инструкций в теле метода. Инструкции могут быть следующими (рис. 11):

- Leave — конец метода. При достижении этой инструкции выполнение метода заканчивается.
- Goto n — безусловный переход на n-ную инструкцию в списке инструкций. В методе должно быть не менее n инструкций.
- Branch n — условный переход (в зависимости от значения на вершине стека) на n-ную инструкцию в списке инструкций. В методе должно быть не менее n инструкций.
- DuplicateStackTop — удвоение вершины стека.
- RemoveStackTop — удаление вершины стека.
- LoadConst const — загрузка на стек константы const. Const — это либо целое число, либо число с плавающей точкой, либо значение NULL.
- UnaryOp op — выполнение над вершиной стека унарной операции op.
- BinaryOp op — выполнение над двумя верхними элементами стека бинарной операции op.
- LoadVar var — загрузка на стек значения переменной var. Переменная var должна быть объявлена в данном методе.
- StoreVar var — запись в переменную var значения, находящегося на вершине стека. Переменная var должна быть объявлена в данном методе.
- NewObject class — создание объекта класса class. Класс class должен быть описан в программе.
- LoadField fld — загрузка на стек поля fld объекта, адрес которого находится на стеке. Должен существовать класс, в описании которого опре-

делено поле fld.

- StoreField fld — запись значения в поле fld объекта. Значение и адрес объекта находятся на вершине стека. Должен существовать класс, в описании которого определено поле fld.
- CallMethod mthd — виртуальный вызов метода mthd. Должен существовать класс, в описании которого определен метод mthd.
- CastObject type — проверка является ли тип значения на вершине стека наследником типа type. Если тип значения является наследником, то значение оставляется на вершине стека, иначе загружается на стек значение NULL вместо исходного значения. Тип type должен быть определен программой.
- NewArray type — создание массива из элементов type, длина которого находится на вершине стека. Тип type должен быть определен программой.
- LoadLength — загрузка на стек длины массива, адрес которого находится на вершине стека.
- LoadElement — загрузка на стек значения элемента массива. Номер элемента и адрес массива находятся на вершине стека.
- StoreElement — запись значения в элемент массива. Значение, номер элемента и адрес массива находятся на вершине стека.

В описании языка SOOL приведен конкретный список операций, допустимых в инструкциях UnaryOp op и BinaryOp op. Однако этот набор может быть расширен дополнительными операциями.

2.2.4. Связь SOOL с CIL.NET

Язык SOOL основан на ядре языка CIL платформы Microsoft.NET. Сопоставление инструкций языков CIL.NET и SOOL дано в таб. 3 и таб. 4. Многим инструкциям языка CIL соответствуют инструкции языка SOOL, отличающиеся только названием. Другие отличия заключаются в следующем:

Инструкция CIL.NET	**Инструкции SOOL**
add	BinaryOp ADD
and	BinaryOp AND
beq n	BinaryOp CEQ, Branch n'
bge n	BinaryOp CLT, UnaryOp NOT, Goto n'
bgt n	BinaryOp CGT, Branch n'
ble n	BinaryOp CGT, UnaryOp NOT, Goto n'
blt n	BinaryOp CLT, Branch n'
bne n	BinaryOp CEQ, UnaryOp NOT, Goto n'
br n	Goto n'
brfalse n	UnaryOp NOT, Goto n'
brtrue n	Branch n'
call	-
ceq	BinaryOp CEQ
cgt	BinaryOp CGT
clt	BinaryOp CLT
conv t	UnaryOp t'
div	BinaryOp DIV
dup	DuplicateStackTop
ldarg	-
ldc const	LoadConst const
ldloc var	LoadVar var'
ldnull	LoadConst null
mul	BinaryOp MUL
neg	UnaryOp NEG
nop	-
not	UnaryOp NOT
or	BinaryOp OR
pop	RemoveStackTop
rem	BinaryOp REM
ret	Leave
shl	BinaryOp SHL
shr	BinaryOp SHR
starg	-
stloc var	StoreVar var'
sub	BinaryOp SUB

switch [n_0,n_1,n_2,…,n_{k-1}]	DuplicateStackTop, Branch n'_0, DuplicateStackTop, LoadConst 1, BinaryOp CEQ, Branch n'_1, …, DuplicateStackTop, LoadConst (k-1), BinaryOp CEQ, Branch n'_{k-1}, RemoveStackTop n'_0: RemoveStackTop, … n'_1: RemoveStackTop, … n'_{k-1}: RemoveStackTop, …
xor	BinaryOp XOR

Таб. 3. Преобразование основных инструкций.

Инструкция CIL.NET	**Инструкции SOOL**
callvirt mthd	CallMethod mthd
castclass	-
isinst	CastObject
ldelem	LoadElement
ldfld fld	LoadField fld
ldlen	LoadLength
newarr type	NewArray type
newobj ctor	NewObject class, CallMethod ctor
stelem	StoreElement
stfld fld	StoreField fld

Таб. 4. Преобразование объектных инструкций.

1. Для более компактного описания правил унарные инструкции conv, neg, not языка CIL.NET сгруппированы в одну инструкцию UnaryOp языка SOOL. Аналогично бинарные инструкции add, and, ceq, cgt, clt, div, neg, or, rem, shl, shr, sub, xor сгруппированы в одну инструкцию BinaryOp. Инструкции загрузки константы на стек ldc и ldnull сгруппированы в одну инструкцию LoadConst.
2. В языке SOOL есть только одна операция условного ветвления Branch, поэтому все инструкции ветвления beq, bge, bet, ble, blt, bne, brtrue, brfalse и инструкция switch языка CIL.NET представляются одной или несколькими инструкциями языка SOOL: инструкциями сравнения, отрицания и условного перехода.
3. Так как передача аргументов в языке SOOL происходит через стек, то

инструкции чтения/записи аргументов языка CIL.NET не требуются.

4. В языке SOOL нет аналога инструкции отсутствия операции nop в языке CIL.NET.
5. Все методы в языке SOOL являются виртуальными, поэтому нет аналога инструкции call языка CIL.NET, выполняющей вызов статического метода.
6. Если в языке CIL.NET инструкция создания объекта newobj, кроме собственно создания объекта, вызывает конструктор, то в языке SOOL инструкция NewObject только создает объект, а конструктор должен быть вызван с помощью инструкции CallMethod.

Инструкции проверки и приведения объекта isinst языка CIL.NET соответствует инструкция CastObject языка SOOL. Инструкция приведения объекта castclass в CIL.NET выбрасывает исключение, если объект не является наследником указанного типа, поэтому этой инструкции не соответствует ни одна инструкция в языке SOOL.

2.3. Интерпретация программ на языке SOOL

2.3.1. Состояние

В процессе выполнения каждая инструкция изменяет состояние вычислительной машины (State) (рис. 12). Состояние языка SOOL — это тройка: стек, среда и куча.

Стек (Stack) — это список значений, обращения к которому происходит только с «головы» стека.

Среда (Env) — это отображение переменных в значения.

Куча (Heap) — это отображение адресов в пару тип и объект или массив.

Значение (Value) — это либо целое число (Integer), либо число с плавающей точкой (Float), либо адрес (Address), либо специальный адрес NULL, которому в куче ничего не соответствует.

Объект (Object) — это отображение полей в значения. Массив (Array) —

это отображение целых чисел от 0 до длины массива минус 1 в значения.

Для описания результата вычислений отдельного метода и программы в целом используются состояния метода (MethodState) и программы (Program-State) соответственно. Состояние метода состоит из стека и кучи. А начальное состояние программы — только из стека.

```
State = (Stack, Env, Heap)
MethodState = ([Value], Heap)
ProgramState = [Value]

Stack = [Value]
Env = Var → Value
Heap = Address → (Type, ObjectOrArray)

Value = Integer | Float | AddressOrNull
ObjectOrArray = Object | Array
Object = Field → Value
Array = Integer → Value
AddressOrNull = NULL | Address
Address = Integer
```

Рис. 12. Состояние интерпретатора.

2.3.2. Вспомогательные функции

Для описания правил интерпретации, определим вспомогательные функции.

Каждое значение в языке SOOL имеет определенный тип: целое число имеет тип целых чисел INT, число с плавающей — тип чисел с плавающей точкой FLOAT, адрес NULL — тип NULLTYPE, другие адреса addr — тип объекта или массива, расположенный в первом элементе пары heap(addr). Если задана куча, то можно построить отображение значений в типы TypeOf_{heap}, указанным способом (рис. 13). Будем говорить, что данное val *удовлетворяет* типу type, если тип данного наследуется от этого типа: $\mathrm{TypeOf}_{heap}(val) <_{prog} type$.

Для случая, когда куча не известна, определим функцию TypeOf только для целых чисел, чисел с плавающей точкой и NULL аналогичным образом

(рис. 13).

TypeOf : Value → Type
TypeOf (val:Integer) = INT
TypeOf (val:Float) = FLOAT
TypeOf (NULL) = NULLTYPE

$TypeOf_{heap}$: Value → Type
$TypeOf_{heap}$ (val:Integer) = INT
$TypeOf_{heap}$ (val:Float) = FLOAT
$TypeOf_{heap}$ (NULL) = NULLTYPE
$TypeOf_{heap}$ (addr:Address) = type where (type, _) = heap(addr)

$TypeOf_{heap}$: [Value] → [Type]
$TypeOf_{heap}$(vals) = [$TypeOf_{heap}$(val) | val ← vals]

DefaultValue : Type → Value
DefaultValue (INT) = 0
DefaultValue (FLOAT) = 0.0
DefaultValue (RefType) = NULL

Рис. 13. Вспомогательные функции.

$VarType_{prog}$: VarName → Type
$ClassFields_{prog}$: ClassName → [FieldDec]
$FieldClass_{prog}$: FieldName → ClassName
$FieldType_{prog}$: FieldName → Type
$MethodSignature_{prog}$: MethodName → ([Type], [Type])
$MethodDefinition_{prog}$: MethodName × ClassName → MethodDef

Рис. 14. Вспомогательные функции.

Определим вспомогательную функцию DefaultValue (рис. 13), которая по типу возвращает значение по умолчанию. Если тип — целые числа, то значение по умолчанию — 0. Если тип — числа с плавающей точкой, то значение по умолчанию — 0.0. Если тип — ссылочный тип, то значение по умолчанию — NULL.

Будем считать, что для каждой программы prog определены функции $VarType_{prog}$, $ClassFields_{prog}$, $FieldClass_{prog}$, $FieldType_{prog}$, $MethodSignature_{prog}$ и $MethodDefinition_{prog}$ (рис. 14).

Функция $VarType_{prog}$ по имени переменной возвращает ее тип.

Функция $ClassFields_{prog}$ по имени класса возвращает все поля, которые

должен содержать объект данного класса (с учетом полей как в определении указанного класса, так и в определениях всех надклассов).

Функция $FieldClass_{prog}$ по имени поля возвращает имя класса, в котором указанное поле определено

Функция $FieldType_{prog}$ по имени поля возвращает его тип.

Функция $MethodSignature_{prog}$ по имени метода возвращает сигнатуру этого метода. Если метод с данным именем определен в нескольких классах, то есть метод переопределен, то возвращается сигнатура метода, определенного в основном классе.

Функция $GetMethodDefinition_{prog}$ по имени метода и имени класса возвращает определение указанного метода в этом классе, если он определен в нем или в ближайшем надклассе.

2.3.3. Правила выполнения программ на языке SOOL

Семантика языка SOOL описана в стиле операционной семантики [32,40]. То есть используются правила, описывающие изменение состояния. Эти правила близки к тройкам Хоара: $\{state_1\}$ instruction $\{state_2\}$, только записанные в виде правил.

Для описания правил преобразования состояния при выполнении одной инструкции будем использовать правила в стиле small step semantics (рис. 15).

$$\boxed{\text{context} \vdash_i \text{instruction} : \text{state}_1 \rightarrow \text{state}_2}$$

Рис. 15. Правило выполнения инструкции

Эти правила показывают, каким будет состояние $state_2$ после выполнения инструкции instruction, если состояние до выполнения инструкции было $state_1$ (в контексте context).

$$\boxed{\text{context} \vdash_i \text{method} : \text{state}_1 \Rightarrow \text{state}_2}$$

Рис. 16. Правило выполнения метода

Для описания правил выполнения метода, части метода или программы используются правила в стиле big step semantics, описывающие результат $state_2$

выполнения метода при начальных условиях $state_1$ (рис. 16).

Если ни одно из правил не может быть применено, то выполнение программы завершается с ошибкой.

2.3.4. Выполнение программы

Перед началом выполнения задается программа и список аргументов. Затем создается объект класса MAIN. После создания объекта, вызывается метод Main у этого объекта с заданными аргументами. Результат выполнения программы — это результат выполнения метода Main.

Правило выполнения программы описывает эти действия с помощью выполнения последовательности следующих инструкций: создание объекта класса MAIN (NewObject MAIN), вызов метода Main (CallMethod Main) и завершение выполнения метода (Leave) (рис. 17). Начальное состояние состоит из списка аргументов, пустого окружения и пустой кучи.

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

Рис. 17. Выполнение программы.

Состояние после выполнения последовательности инструкций состоит из стека, пустого окружения и кучи. Результатом выполнения программы являются данные, находящиеся на стеке.

2.3.5. Выполнение метода

Определим правило выполнения метода. Состояние в правиле — это набор значений и куча. Правило описывает результат выполнения одного метода.

Чтобы выполнить метод, необходимо (рис. 18):

1. Проверить, что типы значений аргументов наследуются от типов аргументов метода.
2. Из определения метода выделить список переменных и список инструк-

ций. Создать новое окружение env_1.

3. Начиная с нулевой инструкции, произвести выполнение списка инструкций из начального состояния, состоящего из списка аргументов, нового окружения и кучи.
4. По завершению выполнения тела метода список результатов и кучу считать результатом выполнения метода.

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

Рис. 18. Выполнение метода.

2.3.6. Выполнение последовательности инструкций

Последовательность инструкций instrs, начиная с n-ой инструкции, вычисляется следующим образом:

1. Если n-ная инструкция в списке instrs — это Leave (конец метода), то выполнение останавливается (рис. 19).

$$\frac{Leave = instrs[n]}{prog \vdash_i (instrs, n) : (stack, env, heap) \Rightarrow (stack, env, heap)}$$

Рис. 19. Выполнение последовательности инструкций, инструкция Leave.

2. Если n-ная инструкция в списке instrs — это инструкция Goto m (безусловный переход на m-ную инструкцию), то выполнение продолжается для m-ной инструкции в списке instrs (рис. 20).

$$\frac{\begin{array}{c}Goto\ m = instrs[n] \\ prog \vdash_i (instrs, m) : (stack_1, env_1, heap_1) \Rightarrow (stack_2, env_2, heap_2)\end{array}}{prog \vdash_i (instrs, n) : (stack_1, env_1, heap_1) \Rightarrow (stack_2, env_2, heap_2)}$$

Рис. 20. Выполнение последовательности инструкций, инструкция Goto m.

3. Если n-ная инструкция в списке instrs — это инструкция Branch m (условный переход на m-ную инструкцию), то проверяется значение на

вершине стека. Это значение должно быть целым числом. Если оно равно 0, то выполнение продолжается для (n+1)-ой инструкции, иначе — для m-ной (рис. 21).

$$\frac{\text{Branch } m = \text{instrs}[n] \qquad \text{val::stack}'_1 = \text{stack}_1 \\ \text{INT} = \text{TypeOf}_{\text{heap1}}(\text{val}) \qquad k = \text{if val==0 then } n+1 \text{ else } m \\ \text{prog} \vdash_i (\text{instrs}, k) : (\text{stack}'_1, \text{env}_1, \text{heap}_1) \Rightarrow (\text{stack}_2, \text{env}_2, \text{heap}_2)}{\text{prog} \vdash_i (\text{instrs}, n) : (\text{stack}_1, \text{env}_1, \text{heap}_1) \Rightarrow (\text{stack}_2, \text{env}_2, \text{heap}_2)}$$

Рис. 21. Выполнение последовательности инструкций, инструкция Branch m.

4. В остальных случаях применяется правило для выполнения одной n-ой инструкции и, начиная с нового состояния, продолжается выполнение последовательности инструкций для (n+1)-ой инструкции (рис. 22).

$$\frac{\text{prog} \vdash_i \text{instrs}[n] : (\text{stack}_1, \text{env}_1, \text{heap}_1) \rightarrow (\text{stack}_2, \text{env}_2, \text{heap}_2) \\ \text{prog} \vdash_i (\text{instrs}, n+1) : (\text{stack}_2, \text{env}_2, \text{heap}_2) \Rightarrow (\text{stack}_3, \text{env}_3, \text{heap}_3)}{\text{prog} \vdash_i (\text{instrs}, n) : (\text{stack}_1, \text{env}_1, \text{heap}_1) \Rightarrow (\text{stack}_3, \text{env}_3, \text{heap}_3)}$$

Рис. 22. Выполнение последовательности инструкций.

2.3.7. Выполнение инструкций

DuplicateStackTop

Инструкция DuplicateStackTop читает значение, находящееся на вершине стека, и добавляет в стек значение, равное исходному значению (рис. 23).

$$\text{prog} \vdash_i \text{DuplicateStackTop} : (\text{val::stack}, \text{env}, \text{heap}) \rightarrow (\text{val::val::stack}, \text{env}, \text{heap})$$

Рис. 23. Правило выполнения инструкции DuplicateStackTop.

Перед выполнением интерпретатор проверяет, что на стеке есть хотя бы один элемент. Если стек пуст, то выполнение программы прерывается.

RemoveStackTop

Инструкция RemoveStackTop удаляет значение, находящееся на вершине стека (рис. 24).

$$\text{prog} \vdash_i \text{RemoveStackTop} : (\text{val::stack}, \text{env}, \text{heap}) \rightarrow (\text{stack}, \text{env}, \text{heap})$$

Рис. 24. Правило выполнения инструкции RemoveStackTop.

Перед выполнением интерпретатор проверяет, что на стеке есть хотя бы один элемент. Если стек пуст, то выполнение программы прерывается.

LoadConst const

Инструкция LoadConst const добавляет в стек константу const, заданную параметром инструкции (рис. 25). Константа const может быть либо NULL, либо значением примитивного типа INT или FLOAT.

$$\text{prog} \vdash_i \text{LoadConst const} : (\text{stack}, \text{env}, \text{heap}) \rightarrow (\text{const::stack}, \text{env}, \text{heap})$$

Рис. 25. Правило выполнения инструкции LoadConst const.

UnaryOp op

Инструкция UnaryOp op вынимает значение val, находящееся на вершине стека, и добавляют в стек результат op(val) операции op над исходным значением value (рис. 26). Операция op — это:

- либо взятие отрицания NEG, эта операция применима к целому числу или числу с плавающей точкой;
- либо побитовое отрицание NOT, эта операция применима только к целому числу;
- либо преобразование из целых чисел в числа с плавающей точкой INT2FLOAT, применимо только к целому числу;
- либо преобразование из чисел с плавающей точкой в целые числа FLOAT2INT, применимо только к числу с плавающей точкой.

$$\frac{\text{op} = \text{INT2FLOAT, NOT, NEG} \qquad \text{TypeOf}_{\text{heap}}(\text{val}) = \text{INT}}{\text{prog} \vdash_i \text{UnaryOp op} : (\text{val::stack}, \text{env}, \text{heap}) \rightarrow (\text{op(val)::stack}, \text{env}, \text{heap})}$$

$$\frac{\text{op} = \text{FLOAT2INT, NEG} \qquad \text{TypeOf}_{\text{heap}}(\text{val}) = \text{FLOAT}}{\text{prog} \vdash_i \text{UnaryOp op} : (\text{val::stack}, \text{env}, \text{heap}) \rightarrow (\text{op(val)::stack}, \text{env}, \text{heap})}$$

Рис. 26. Правило выполнения инструкции UnaryOp op.

Перед выполнением этой операции интерпретатор проверяет, что стек не пуст и операция op применима к элементу, лежащему на вершине стека.

BinaryOp op

Инструкция BinaryOp op вынимает два значения val_1 и val_2, находящиеся на вершине стека, и добавляют в стек результат op(val_1, val_2) операции op над исходными значениями val_1 и val_2 (рис. 27). Операция op — это одна из следующих операций:

- ADD — сложение, применимо к двум целым числам или к двум числам с плавающей точкой;
- AND — побитовое И, применимо только к двум целым числам;
- CEQ — сравнение на равенство, применимо к двум целым числам, к двум числам с плавающей точкой, к двум данным ссылочного типа;
- CGT — сравнение на больше, применимо к двум целым числам или к двум числам с плавающей точкой;
- CLT — сравнение на меньше, применимо к двум целым числам или к двум числам с плавающей точкой;
- DIV — частное от целочисленного деления или деление чисел с плавающей точкой, применимо к двум целым числам или к двум числам с плавающей точкой;
- MUL — умножение, применимо к двум целым числам или к двум числам с плавающей точкой;
- OR — побитовое ИЛИ, применимо только к двум целым числам;
- REM — остаток от деления, применимо к двум целым числам или к двум числам с плавающей точкой;
- SHL — побитовый сдвиг влево, применимо только к двум целым числам;
- SHR — побитовый сдвиг вправо, применимо только к двум целым числам;
- SUB — вычитание, применимо к двум целым числам или к двум числам с плавающей точкой;

- XOR — побитовое исключающее ИЛИ, применимо только к двум целым числам.

$$\frac{\begin{array}{c}op = ADD, AND, CEQ, CGT, CLT, DIV, MUL,\\ OR, REM, SHL, SHR, SUB, XOR\\ TypeOf_{heap}(val_1) = INT \qquad TypeOf_{heap}(val_2) = INT \qquad val_3 = op(val_1,val_2)\end{array}}{prog \vdash_i BinaryOp\ op : (val_1::val_2::stack, env, heap) \to (val_3::stack, env, heap)}$$

$$\frac{\begin{array}{c}op = ADD, CEQ, CGT, CLT, DIV, MUL, REM, SUB\\ TypeOf_{heap}(val_1) = FLOAT \qquad TypeOf_{heap}(val_2) = FLOAT\\ val_3 = op(val_1,val_2)\end{array}}{prog \vdash_i BinaryOp\ op : (val_1::val_2::stack, env, heap) \to (val_3::stack, env, heap)}$$

$$\frac{\begin{array}{c}op = CEQ \qquad TypeOf_{heap}(val_1) = OBJECT\\ TypeOf_{heap}(val_2) = OBJECT \qquad val_3 = op(val_1,val_2)\end{array}}{prog \vdash_i BinaryOp\ op : (val_1::val_2::stack, env, heap) \to (val_3::stack, env, heap)}$$

Рис. 27. Правило выполнения инструкции BinaryOp op.

Перед выполнением этой операции интерпретатор проверяет, что в стеке лежат по крайней мере два элемента и операция op применима к двум элементам, лежащим на вершине стека.

LoadVar var

Инструкция LoadVar var читает значение переменной var среды env и добавляет в стек это значение (рис. 28).

$$prog \vdash_i LoadVar\ var : (stack, env, heap) \to (env(var)::stack, env, heap)$$

Рис. 28. Правило выполнения инструкции LoadVar var.

StoreVar var

Инструкция StoreVar var достает значение val, находящееся на вершине стека, и записывает его в переменную var в среду env (рис. 29).

$$\frac{TypeOf_{heap}(val) <_{prog} VarType_{prog}(var)}{prog \vdash_i StoreVar\ var : (val::stack, env, heap) \to (stack, env[var \mapsto val], heap)}$$

Рис. 29. Правило выполнения инструкции StoreVar var.

Перед выполнением инструкции интерпретатор проверяет, что на стеке лежит по крайней мере один элемент и его тип наследует тип переменной var.

NewObject class

Инструкция NewObject class создает объект obj класса class, кладет его в кучу heap и добавляет на вершину стека адрес созданного объекта (рис. 30).

$$\frac{\begin{array}{c} obj = [fld \mapsto DefaultValue(type) \mid (fld, type) \leftarrow ClassFields_{prog}(class)] \\ oref \text{ — новый адрес} \qquad heap' = heap[oref \mapsto (class, obj)] \end{array}}{prog \vdash_i NewObject\ class : (stack, env, heap) \rightarrow (oref::stack, env, heap')}$$

Рис. 30. Правило выполнения инструкции NewObject class.

LoadField fld

Инструкция LoadField fld достает адрес объекта oref, находящийся на стеке, по адресу oref находит в куче heap объект obj, читает поле fld объекта obj и добавляет в стек значение этого поля (рис. 31).

$$\frac{\begin{array}{c} TypeOf_{heap}(oref) <_{prog} FieldClass_{prog}(fld) \qquad oref \neq NULL \\ (_, obj) = heap(oref) \end{array}}{prog \vdash_i LoadField\ fld : (oref::stack, env, heap) \rightarrow (obj(fld)::stack, env, heap)}$$

Рис. 31. Правило выполнения инструкции LoadField fld.

Перед выполнением инструкции интерпретатор проверяет, что стек не пуст, на вершине стека лежит адрес oref объекта класса class и у этого объекта есть поле fld.

StoreField fld

Инструкция StoreField fld достает значение val и адрес объекта oref, находящиеся на вершине стека, по адресу oref находит в куче heap объект obj, заменяет значение поля fld на val, создавая новый объект obj', и записывает новый объект obj' по старому адресу oref в кучу heap (рис. 32).

Перед выполнением инструкции интерпретатор проверяет, что стек не пуст, на вершине лежат значение val и адрес oref объекта класса class, у этого объекта есть поле fld и тип значения val наследует тип этого поля.

$$\frac{\begin{array}{c}\text{TypeOf}_{\text{heap}}(\text{oref}) <_{\text{prog}} \text{FieldClass}_{\text{prog}}(\text{fld}) \qquad \text{oref} \neq \text{NULL} \\ \text{TypeOf}_{\text{heap}}(\text{val}) <_{\text{prog}} \text{FieldType}_{\text{prog}}(\text{fld}) \\ (\text{class, obj}) = \text{heap}(\text{oref}) \qquad \text{heap}' = \text{heap}[\text{oref} \mapsto (\text{class, obj}[\text{fld} \mapsto \text{val}])]\end{array}}{\text{prog} \vdash_i \text{StoreField fld} : (\text{val::oref::stack, env, heap}) \rightarrow (\text{stack, env, heap}')}$$

Рис. 32. Правило выполнения инструкции StoreField fld.

CallMethod mthd

Инструкция CallMethod mthd вынимает из стека аргументы метода, по методу и типу первого аргумента находит определение вычисляемого метода, вычисляет найденный метод, результаты переносятся на вершину исходного стека (рис. 33). Количество аргументов и результатов определяется по определению метода mthd, найденного с помощью функции $\text{MethodSignature}_{\text{prog}}$.

$$\frac{\begin{array}{c}(\text{tArg}, _) = \text{MethodSignature}_{\text{prog}}(\text{mthd}) \qquad \text{Length}(\text{arg}) = \text{Length}(\text{tArg}) \\ \text{type} = \text{TypeOf}_{\text{heap}}(\text{Head}(\text{arg})) \qquad \text{type} <_{\text{prog}} \text{Head}(\text{tArg}) \qquad \text{type} \neq \text{NULLTYPE} \\ \text{prog} \vdash_I \text{MethodDefinition}_{\text{prog}}(\text{mthd, type}) : (\text{arg}, \text{heap}_1) \Rightarrow (\text{res}, \text{heap}_2)\end{array}}{\text{prog} \vdash_i \text{CallMethod mthd} : (\text{arg++stack,env,heap}_1) \rightarrow (\text{res++stack,env,heap}_2)}$$

Рис. 33. Правило выполнения инструкции CallMethod mthd.

Перед выполнением инструкции интерпретатор проверяет, что количество элементов на стеке достаточно, чтобы вызвать метод: количество элементов больше или равно количеству аргументов метода mthd. Также проверяет, что на вершине стека лежит адрес val_1 объекта класса class и у этого класса есть метод mthd.

CastObject type

Инструкция CastObject type достает адрес ref объекта или массива, находящийся на вершине стека, проверяет, является ли объект наследником типа type. Если объект является наследником типа type, то на вершину стека выкладывается этот адрес ref, иначе выкладывается NULL (рис. 34).

Перед выполнением инструкции интерпретатор проверяет, что ref — это адрес или NULL.

$$\frac{TypeOf_{heap}(ref) <_{prog} type}{prog \vdash_i CastObject\ type : (ref::stack, env, heap) \rightarrow (ref::stack, env, heap)}$$

$$\frac{TypeOf_{heap}(ref) <_{prog} OBJECT \qquad TypeOf_{heap}(ref) \nless_{prog} type}{prog \vdash_i CastObject\ type : (ref::stack, env, heap) \rightarrow (NULL::stack, env, heap)}$$

Рис. 34. Правило выполнения инструкции CastObject type.

NewArray type

Инструкция NewArray type достает целое число n, находящееся на вершине стека, создает массив arr длины n, создает новый адрес aref, добавляет в кучу heap по адресу aref созданный массив arr и добавляет в стек адрес aref этого массива (рис. 35).

$$\frac{\begin{array}{c}TypeOf_{heap}(length) = INT \qquad arr = [i \mapsto DefaultValue(type) \mid i \leftarrow [0..n-1]] \\ aref \text{ — новый адрес} \qquad heap' = heap[aref \mapsto (type[], (length, arr))]\end{array}}{prog \vdash_i NewArray\ type : (length::stack, env, heap) \rightarrow (aref::stack, env, heap')}$$

Рис. 35. Правило выполнения инструкции NewArray type.

Перед выполнением инструкции интерпретатор проверяет, что стек не пуст и на вершине стека лежит целое число.

LoadLength

Инструкция LoadLength достает адрес aref массива, находящийся на вершине стека, из кучи heap по адресу aref достает массив arr и добавляет в стек длину NumberOfElements(arr) этого массива arr (рис. 36).

$$\frac{\begin{array}{c}TypeOf_{heap}(aref) <_{prog} someType[] \qquad aref \neq NULL \\ (_, (length, arr)) = heap(aref)\end{array}}{prog \vdash_i LoadLength : (aref::stack, env, heap) \rightarrow (length::stack, env, heap)}$$

Рис. 36. Правило выполнения инструкции LoadLength.

Перед выполнением инструкции проверяется, что стек не пуст и на вершине стека лежит адрес массива.

LoadElement

Инструкция LoadElement достает целое число n и адрес массива aref, находящиеся на стеке, по адресу aref достает из кучи heap массив arr, читает значение n-ого элемента массива arr(n) и добавляет его в стек (рис. 37).

$$\frac{TypeOf_{heap}(n) = INT \qquad TypeOf_{heap}(aref) <_{prog} someType[] \qquad aref \neq NULL \qquad (_, (length, arr)) = heap(aref) \qquad 0 \leq n < length}{prog \vdash_i LoadElement : (n::aref::stack, env, heap) \rightarrow (arr(n)::stack, env, heap)}$$

Рис. 37. Правило выполнения инструкции LoadElement.

Перед выполнением инструкции проверяется, что на стеке лежат по крайней мере два элемента: целое число n и адрес массива aref. Также проверяется, что в массиве есть элемент с номером n.

StoreElement

Инструкция StoreElement достает значение val, целое число n и адрес массива aref, находящиеся на стеке, по адресу aref достает из кучи heap массив arr, заменяет значение элемента с номером n на val, создавая новый массив arr', и записывает новый массив arr' в кучу heap по старому адресу aref (рис. 38).

$$\frac{TypeOf_{heap}(n) = INT \qquad TypeOf_{heap}(aref) <_{prog} someType[] \qquad aref \neq NULL \qquad (type[], (length, arr)) = heap(aref) \qquad TypeOf_{heap}(val) <_{prog} type \qquad 0 \leq n < length \qquad heap' = heap[aref \mapsto (type[], (length, arr[n \mapsto val]))]}{prog \vdash_i StoreElement: (val::n::aref::stack, env, heap) \rightarrow (stack, env, heap')}$$

Рис. 38. Правило выполнения инструкции StoreElement.

Перед выполнением проверяется, что на стеке лежат по крайней мере три элемента: значение val, целое число n и адрес массива aref. А также проверяется, что тип значения val наследует тип элементов массива и в массиве есть элемент с номером n.

2.4. Типизация языка SOOL

Во время выполнения постоянно необходимо проверять, что инструкция может выполнить операцию над данными. Например, чтобы прочитать поле,

нужно проверить, что на вершине стека лежит адрес объекта и что у этого объекта есть указанное поле.

Чтобы избежать многих проверок во время выполнения, можно произвести анализ программы до ее выполнения. Одним из таких анализов является *типизация*: вывод типов всех элементов и проверка выполнимости операции по типам ее аргументов. Типизация гарантирует, что во время выполнения программы на стеке, в переменных методов и в полях объектов будут находиться только значения, удовлетворяющие указанным типам. Поэтому многих проверок на возможность выполнения той или иной инструкции во время выполнения можно не производить. Типизация также помогает обнаружить на этапе компиляции программы ошибки, например, связанные с передачей методу данных, на которые он не рассчитан.

Для контролирования типов в языке SOOL пользователь задает типы аргументов и результатов метода, типы переменных методов и полей объектов. Анализ на основе этой информации строит типы элементов на стеке в каждой точке метода перед выполнением каждой инструкции.

Чтобы проверить, что программа на языке SOOL является *типизируемой*, необходимо для каждой инструкции каждого метода (то есть для всех чисел от нуля до количества инструкций в методе минус один) построить стек типов, — этот стек показывает, какого типа элементы будут находиться на стеке во время выполнения программы, — и проверить, что инструкции могут выполнить действие с данными указанных типов (рис. 39).

$$\frac{\forall\, \text{class} \in \text{prog},\ \text{class} = (_, _, _, \text{mthdDefs}) \qquad \forall\, \text{mthdDef} \in \text{mthdDefs}, \\ \exists\, \lambda_{\text{mthdDef}}\text{: Integer} \to [\text{Type}] : \qquad \text{prog}, \lambda_{\text{mthdDef}} \vdash_t \text{mthdDef}}{\text{prog — типизируема}}$$

Рис. 39. Типизация программы на языке SOOL.

Будем говорить, что программа prog на языке SOOL *типизируема*, если для каждого класса class из программы prog, для каждого определения метода mthdDef из класса class существует функция λ_{mthdDef}, отображающая целые

числа Integer в стек типов TStack, такая что она типизирует метод mthdDef. Функцию $\lambda_{mthdDef}$ будем называть *типизирующей* функцией метода mthdDef.

2.4.1. Типизация программы

Теорема (О типизации программы). Рассмотрим типизированную программу, то есть программу и типизирующие функции для всех методов. Путь даны аргументы программы arg — данные, удовлетворяющие типам аргументов tArg метода Main класса MAIN за исключением первого аргумента этого метода, который не задается в качестве аргумента программы. Тогда на каждом шаге:

1. данные, находящиеся на стеке, удовлетворяют типам, заданным типизирующей функцией для данного метода в данной точке;
2. данные, находящиеся в переменных методов, удовлетворяют типам этих переменных;
3. данные, находящиеся в полях объектов, удовлетворяют типам этих полей.

Из этой теоремы следует, что при выполнении типизируемой программы можно не выполнять многих проверок.

Ниже в работе одновременно приведены и правила, которым должны удовлетворять типизирующие функции, и доказательство теоремы методом математической индукцией по числу шагов при выполнении программы.

Правила типизации метода задают ограничения на типизирующую функцию. Они должны гарантировать, что во время выполнения программы в каждой точке программы элементы, находящиеся на стеке, удовлетворяют типам, указанным для этой точки типизирующей функцией.

Типизирующая функция должна удовлетворять этим правилам. То есть в отличие от правил выполнения программы, задающих преобразования состояния по шагам, правила типизации задают ограничения на типизирующую функцию.

Отметим, что типы можно обобщать: если данное удовлетворяет типу

type, то оно удовлетворяет любому типу, являющемуся предком данного типа:

$$TypeOf_{heap}(val) \lessdot_{prog} type,\ type \lessdot_{prog} type' \Rightarrow TypeOf_{heap}(val) \lessdot_{prog} type'$$

Обобщения необходимы для построения стека типов во всех точках программы: в некоторые точки программы возможен переход из разных точек программы, поэтому необходимо в этой точке построить такой стек типов, который был бы согласован и со стеками типов тех точек, из которых возможен переход в данную точку.

2.4.2. Типизация вызова метода Main

Начало выполнение программы — это построение маленького метода и начало его выполнения (рис. 40). Для проверки базы индукции необходимо проверить, что аргументы программы удовлетворяют типам tArg. Что выполняется по условию теоремы.

Далее доказательство идет по шагам выполнения программы. На каждом шаге проверяется, что если до выполнения шага данные на стеке удовлетворяют типам, заданным типизирующей функцией, а данные в переменных методов и полях объектов удовлетворяют заданным в программе типам, то после выполнения шага все данные также убудут удовлетворять соответствующим типам.

$(MAIN::tArg, tRes) = MethodSignature_{prog}(Main)$
$\lambda_0 : Integer \to [Type]$
$\lambda_0(0) = tArg \qquad \lambda_0(1) = MAIN::tArg \qquad \lambda_0(2) = tRes$
$prog, \lambda_0, tRes \vdash_t ([NewObject\ MAIN, CallMethod\ Main, Leave], 0)$

Рис. 40. Типизация программы на языке SOOL.

2.4.3. Типизация метода

Типизирующая функция $\lambda_{mthdDef}$ определяет типы в каждой точке программы, если (рис. 41):

1. Типы аргументов метода tArg из определения метода mthdDef являются наследниками соответствующих типов из стека $\lambda_{mthdDef}(0)$.
2. В контексте программы prog, типизирующей функции $\lambda_{mthdDec}$ и списка

типов результатов метода tRes каждая инструкция с номером n из списка instrs типизируется.

Отметим, что если при выполнении метода аргументы метода удовлетворяли типам tArg, то при переходе к выполнению первой инструкции (с номером 0) эти аргументы переносятся на стек и они будут удовлетворять типам $\lambda_{mthdDec}(0)$ ($tArg <_{prog} \lambda_{mthdDec}(0)$).

$$\frac{(_, tArg, tRes, (varDecs, instrs)) = mthdDef \qquad tArg <_{prog} \lambda_{mthdDec}(0) \quad \forall\, n, 0 \le n < Length(instrs):\ \ prog, \lambda_{mthdDec}, tRes \vdash_t (instrs, n)}{prog, \lambda_{mthdDec} \vdash mthdDef}$$

Рис. 41. Правило типизации метода.

2.4.4. Типизация последовательности инструкций

Типизируемость n-ой инструкции в списке инструкций instrs проверяется в зависимости от этой инструкции:

1. Если n-ая инструкция — это инструкция Leave, но нужно проверить, что типы из стека $\lambda_{mthdDec}(n)$ являются наследниками соответствующих типов из списка результатов tRes (рис. 42).

$$\frac{Leave = instrs[n] \qquad \lambda_{mthdDec}(n) <_{prog} tRes}{prog, \lambda_{mthdDec}, tRes \vdash_t (instrs, n)}$$

Рис. 42. Правило типизации инструкции Leave.

Если значения на стеке удовлетворяли типам $\lambda_{mthdDec}(n)$, то значения-результаты метода будут удовлетворять типам tRes ($\lambda_{mthdDec}(n) <_{prog} tRes$).

2. Если n-ая инструкция — это инструкция Goto m, то нужно проверить, что типы из стека $\lambda_{mthdDec}(n)$ являются наследниками соответствующих типов из стека $\lambda_{mthdDec}(m)$ (рис. 43).

$$\frac{Goto\ m = instrs[n] \qquad \lambda_{mthdDec}(n) <_{prog} \lambda_{mthdDec}(m)}{prog, \lambda_{mthdDec}, tRes \vdash_t (instrs, n)}$$

Рис. 43. Правило типизации инструкции Goto m.

Если значения на стеке до перехода удовлетворяли типам $\lambda_{mthdDec}(n)$, то

после перехода эти значения будут удовлетворять типам $\lambda_{mthdDec}(m)$ ($\lambda_{mthdDec}(n) <_{prog} \lambda_{mthdDec}(m)$).

3. Если n-ая инструкция — это инструкция Branch m, то нужно проверить, что тип на вершине стека $\lambda_{mthdDec}(n)$ является INT, а остальные типы этого стека $\lambda_{mthdDec}(n)$ являются наследниками соответствующих типов как из стека $\lambda_{mthdDec}(n+1)$, так и из стека $\lambda_{mthdDec}(m)$ (рис. 44).

$$\frac{\text{Branch m} = \text{instrs}[n] \qquad \lambda_{mthdDec}(n) <_{prog} \text{INT}::\lambda_{mthdDec}(n+1) \qquad \lambda_{mthdDec}(n) <_{prog} \text{INT}::\lambda_{mthdDec}(m)}{\text{prog}, \lambda_{mthdDec}, \text{tRes} \vdash_t (\text{instrs}, n)}$$

Рис. 44. Правило типизации инструкции Branch m.

Если значения на стеке до выполнения инструкции условного перехода удовлетворяли типам $\lambda_{mthdDec}(n)$ и на вершине было целое число, то после выполнения перехода на следующую инструкцию или на инструкцию с номером m значения на стеке будут удовлетворять типам $\lambda_{mthdDec}(n+1)$ или $\lambda_{mthdDec}(m)$ соответственно ($\lambda_{mthdDec}(n) <_{prog} \text{INT}::\lambda_{mthdDec}(n+1)$ и $\lambda_{mthdDec}(n) <_{prog} \text{INT}::\lambda_{mthdDec}(m)$).

4. Для остальных инструкций нужно построить такой стек типов tStack, что, во-первых, в контексте программы prog и типов переменных tEnv n-ая инструкция instrs[n] типизируется стеком до ее выполнения $\lambda_{mthdDec}(n)$ и стеком после ее выполнения tStack, и, во-вторых, типы из стека tStack являются наследниками соответствующих типов стека $\lambda_{mthdDec}(n+1)$ (рис. 45).

$$\frac{\text{prog}, \text{tEnv} \vdash_t \text{instrs}[n] : \lambda_{mthdDec}(n) \rightarrow \text{tStack} \qquad \text{tStack} <_{prog} \lambda_{mthdDec}(n+1)}{\text{prog}, \lambda_{mthdDec}, \text{tRes} \vdash_t (\text{instrs}, n)}$$

Рис. 45. Правило типизации инструкций.

Если значения на стеке удовлетворяют типам $\lambda_{mthdDec}(n)$, то после выполнения инструкции значения на стеке будут удовлетворять типам tStack, а поэтому при переходе на следующую инструкцию они будут удовлетво-

рять типам $\lambda_{mthdDec}(n+1)$ ($tStack <_{prog} \lambda_{mthdDec}(n+1)$).

2.4.5. Типизация инструкций

Правило типизации инструкции показывает, как связаны типы значений на стеке до выполнения инструкции и после ее выполнения (рис. 46). Если данные на стеке до выполнения инструкции удовлетворяют типам $tStack_1$, то инструкция instr может быть выполнена, и в результате данные на стеке будут удовлетворять типам $tStack_2$.

$$\boxed{prog \vdash_t instr : tStack_1 \rightarrow tStack_2}$$

Рис. 46. Правило типизации инструкций.

DuplicateStackTop

Правило типизации инструкции DuplicateStackTop требует, чтобы два типа на вершине стека после выполнения инструкции совпадали с типом на вершине стека до выполнения инструкции (рис. 47). Остальные элементы стека до и после выполнения инструкции должны совпадать.

$$\boxed{prog \vdash_t DuplicateStackTop : type::tStack \rightarrow type::type::tStack}$$

Рис. 47. Правило типизации инструкции DuplicateStackTop.

Если значение на вершине стека удовлетворяет типу type, то после выполнения операции DuplicateStackTop два значения на вершине стека будут удовлетворять типу type. Остальные значения не изменяются, поэтому они будут удовлетворять типам tStack.

RemoveStackTop

Правило типизации инструкции RemoveStackTop требует, чтобы стек до выполнения инструкции без верхнего элемента совпадал со стеком после выполнения инструкции (рис. 48).

$$\boxed{prog \vdash_t RemoveStackTop : type::tStack \rightarrow tStack}$$

Рис. 48. Правило типизации инструкции RemoveStackTop.

Инструкция RemoveStackTop удаляет верхний элемент стека, остальные

не изменяются, поэтому они будут удовлетворять типам tStack.

LoadConst const

Правило типизации инструкции LoadConst const требует, чтобы стек после выполнения инструкции получался из стека до выполнения инструкции добавлением на вершину стека типа константы (рис. 49).

$$\boxed{\text{prog} \vdash_t \text{LoadConst const : tStack} \rightarrow \text{TypeOf(const)::tStack}}$$

Рис. 49. Правило типизации инструкции LoadConst const.

Значение на вершине стека после выполнения инструкции const — удовлетворяет типу type, остальные значения не изменились, поэтому удовлетворяют типам tStack.

UnaryOp op

Правило типизации инструкции UnaryOp op требует, чтобы совпадали стеки до и после выполнения за исключением вершин стеков. Типы, расположенные на вершинах стека, могут быть следующими (рис. 50):

$$\frac{\text{op = NOT, NEG}}{\text{prog} \vdash_t \text{UnaryOp op : INT::tStack} \rightarrow \text{INT::tStack}}$$

$$\frac{\text{op = NEG}}{\text{prog} \vdash_t \text{UnaryOp op : FLOAT::tStack} \rightarrow \text{FLOAT::tStack}}$$

$$\frac{\text{op = INT2FLOAT}}{\text{prog} \vdash_t \text{UnaryOp op : INT::tStack} \rightarrow \text{FLOAT::tStack}}$$

$$\frac{\text{op = FLOAT2INT}}{\text{prog} \vdash_t \text{UnaryOp op : FLOAT::tStack} \rightarrow \text{INT::tStack}}$$

Рис. 50. Правило типизации инструкции UnaryOp op.

- оба типа INT, если операция op — это NOT или NEG;
- оба типа FLOAT, если операция op — это NEG;
- тип до выполнения — INT, а после выполнения — FLOAT, если опера-

ция op — это INT2FLOAT;

- тип до выполнения — FLOAT, а после выполнения — INT, если операция op — это FLOAT2INT.

Так как правило согласовано с типами аргумента и результата операции op, то тип элемента на вершине стека до выполнения и после выполнения этой инструкции согласованы согласно правилу типизации. Остальные элементы не изменились, поэтому они удовлетворяют типам tStack.

BinaryOp op

Правило типизации инструкции BinaryOp op требует, чтобы два типа, находящиеся на вершине до выполнения инструкции, были одинаковыми. Эти типы и тип, находящийся на вершине стека после выполнения инструкции, могут быть следующими (рис. 51):

$$\frac{op = ADD, AND, CEQ, CGT, CLT, DIV, MUL, OR, REM, SHL, SHR, SUB, XOR}{prog \vdash_t BinaryOp\ op : INT::INT::tStack \rightarrow INT::tStack}$$

$$\frac{op = ADD, DIV, MUL, REM, SUB}{prog \vdash_t BinaryOp\ op : FLOAT::FLOAT::tStack \rightarrow FLOAT::tStack}$$

$$\frac{op = CEQ, CGT, CLT}{prog \vdash_t BinaryOp\ op : FLOAT::FLOAT::tStack \rightarrow INT::tStack}$$

$$\frac{op = CEQ}{prog \vdash_t BinaryOp\ op : OBJECT::OBJECT::tStack \rightarrow INT::tStack}$$

Рис. 51. Правила типизации инструкции BinaryOp op.

- все типы INT, если op — любая операция: арифметическая (ADD, DIV, MUL, REM, SUB), побитовая (AND, OR, XOR, SHL, SHR) или операция сравнения (CEQ, CGT, CLT);
- все типы FLOAT, если op — арифметическая операция (ADD, DIV, MUL, REM, SUB);

- типы на вершине стека до выполнения инструкции — FLOAT, тип на вершине стека после выполнения инструкции — INT, если op — операция сравнения (CEQ, CGT, CLT);
- типы на вершине стека до выполнения инструкции — OBJECT, тип на вершине стека после выполнения инструкции — INT, если op — операция сравнения на равенство (CEQ).

Остальные элементы стека до и после выполнения инструкции должны совпадать.

Так как правила типизации согласованы с типами аргументов и результата операции op, то если два элемента на вершине стека до выполнения инструкции удовлетворяли типам одного из правил, то результат операции op также будет удовлетворять типу этого правила. Остальные элементы не изменились, поэтому они удовлетворяют типам tStack.

LoadVar var

Правило типизации инструкции LoadVar var требует, чтобы стек после выполнения инструкции состоял из типа переменной и стека до выполнения инструкции (рис. 52).

$$\boxed{\text{prog} \vdash_t \text{LoadVar var : tStack} \rightarrow \text{VarType}_{\text{prog}}(\text{var})\text{::tStack}}$$

Рис. 52. Правило типизации инструкции LoadVar var.

На вершине стека после выполнения операции находится значение переменной var, которое удовлетворяет типу переменной tEnv (var), и остальные элементы стека не изменились, поэтому элементы на стеке после выполнения операции удовлетворяют типам tEnv (var)::tStack.

StoreVar var

Правило типизации инструкции StoreVar var требует, чтобы стек до выполнения инструкции состоял из типа переменной и стека после выполнения инструкции (рис. 53).

На вершине стека до выполнения операции находится значение, удовле-

творяющее типу tEnv (var), которое записывается в переменную с типом tEnv(var). Остальные значения не изменяются, поэтому значения на стеке после выполнения инструкций удовлетворяют типам tStack.

$$\boxed{\text{prog} \vdash_t \text{StoreVar var} : \text{VarType}_{\text{prog}} (\text{var})\text{::tStack} \rightarrow \text{tStack}}$$

Рис. 53. Правило типизации инструкции StoreVar var.

NewObject class

Правило типизации инструкции NewObject class требует, чтобы стек после выполнения инструкции состоял из класса и стека до выполнения инструкции (рис. 54).

$$\boxed{\text{prog} \vdash_t \text{NewObject class} : \text{tStack} \rightarrow \text{class::tStack}}$$

Рис. 54. Правило типизации инструкции NewObject class.

На вершине стека после выполнения инструкции находится адрес объекта — значение типа class, остальные значения не изменились, поэтому они удовлетворяют типам tStack.

LoadField fld

Правило типизации инструкции LoadField fld требует, чтобы тип на вершине стека до выполнения инструкции совпадал с классом, в котором определено это поле, тип на вершине стека после выполнения инструкции совпадал с типом поля (рис. 55). Остальные элементы стека до и после выполнения инструкции должны совпадать.

$$\boxed{\text{prog} \vdash_t \text{LoadField fld} : \text{FieldClass}_{\text{prog}}(\text{fld})\text{::tStack} \rightarrow \text{FieldType}_{\text{prog}}(\text{fld})\text{::tStack}}$$

Рис. 55. Правило типизации инструкции LoadField fld.

Значение на вершине стека до выполнения инструкции удовлетворяет типу class, имеющему поле fld. После выполнения инструкции на вершине стека находится значение поля fld, удовлетворяющее типу поля fld. Остальные элементы стека не изменились, поэтому удовлетворяют типу tStack.

StoreField fld

Правило типизации инструкции StoreField fld требует, чтобы два типа на вершине стека до выполнения инструкции совпадали с типом этого поля и классом, в котором это поле определено (рис. 56). Остальные элементы стека до выполнения инструкции должны совпадать с элементами стека после выполнения этой инструкции.

$$\boxed{prog \vdash_t \text{LoadField fld} : \text{FieldType}_{prog}(\text{fld})::\text{FieldClass}_{prog}(\text{fld})::\text{tStack} \rightarrow \text{tStack}}$$

Рис. 56. Правило типизации инструкции StoreField fld.

Два значения на вершине стека до выполнения инструкции удовлетворяют типам type и class соответственно, значение, удовлетворяющее типу type, записывается в поле объекта с типом type. Элементы стека при выполнении инструкции не меняются, поэтому удовлетворяют типу tStack.

CallMethod mthd

Правило типизации инструкции CallMethod mthd требует, чтобы типы верхних n элементов стека до выполнения инструкции совпадали с типами аргументов метода, а типы верхних m элементов стека после выполнения инструкции — с типами результатов метода (рис. 57). Количество и типы аргументов и результатов находятся с помощью вызова функции $\text{MethodSignature}_{prog}(\text{mthd})$. Остальные элементы стека до и после выполнения инструкции должны совпадать.

$$\frac{(\text{tArg}, \text{tRes}) = \text{MethodSignature}_{prog}(\text{mthd})}{prog \vdash_t \text{CallMethod mthd} : \text{tArg}{+}{+}\text{tStack} \rightarrow \text{tRes}{+}{+}\text{tStack}}$$

Рис. 57. Правило типизации инструкции CallMethod.

Значения на вершине стека до выполнения инструкции удовлетворяют типам аргументов метода tArg. В результате выполнения инструкции аргументы на стеке заменяются результатами метода, удовлетворяющими типам результатов метода tRes. Остальные элементы стека не изменяются, поэтому они удовлетворяют типам tStack.

CastObject type

Правило типизации инструкции CastObject type требует, чтобы на вершине стека до выполнения инструкции был тип OBJECT, а после выполнения — тип type (рис. 58). Остальные элементы стека до и после выполнения инструкции должны совпадать.

$$\text{prog} \vdash_t \text{CastObject type : OBJECT::tStack} \rightarrow \text{type::tStack}$$

Рис. 58. Правило типизации инструкции CastObject type.

На вершине стека до выполнения инструкции находится адрес объекта, а после выполнения — либо NULL, либо тот же адрес объекта, если это значение удовлетворяет типу type. Так как NULL удовлетворяет любому ссылочному типу, то результат всегда будет удовлетворять типу type. Остальные элементы не изменились, поэтому они удовлетворяют типам tStack.

NewArray type

Правило типизации инструкции NewArray type требует, чтобы на вершине стека до выполнения инструкции был тип INT, а после выполнения инструкции — тип массива type[] (рис. 59). Остальные элементы стека до и после выполнения инструкции должны совпадать.

$$\text{prog} \vdash_t \text{NewArray type : INT::tStack} \rightarrow \text{type[]::tStack}$$

Рис. 59. Правило типизации инструкции NewArray type.

На вершине стека до выполнения инструкции находится целое число — длина массива, а после выполнения — адрес массива из элементов типа type — значение типа type[]. Остальные элементы стека не изменились, поэтому удовлетворяют типам tStack.

LoadLength

Правило типизации инструкции LoadLength требует, чтобы на вершине стека до выполнения инструкции был тип массива type[] для какого-то типа type, а после выполнения — тип INT (рис. 60). Остальные элементы стека до и после выполнения инструкции должны совпадать.

На вершине стека до выполнения инструкции находится адрес — значение типа массива type[], а после выполнения — целое число — длина массива. Остальные элементы стека не изменились, поэтому удовлетворяют типам tStack.

$$\boxed{\text{prog} \vdash_t \text{LoadLength} : \text{type[]::tStack} \rightarrow \text{INT::tStack}}$$

Рис. 60. Правило типизации инструкции LoadLength.

LoadElement

Правило типизации инструкции LoadElement требует, чтобы до выполнения инструкции на вершине стека лежали тип INT и тип массива type[], а после выполнения инструкции — тип элементов массива type (рис. 61). Остальные элементы стека до и после выполнения инструкции должны совпадать.

$$\boxed{\text{prog} \vdash_t \text{LoadElement} : \text{INT::type[]::tStack} \rightarrow \text{type::tStack}}$$

Рис. 61. Правило типизации инструкции LoadElement.

Перед выполнением инструкции на вершине стека находятся целое число (номер элемента) и адрес массива — значение, удовлетворяющее типу type[]. Элементы массива удовлетворяют типу type. Поэтому после выполнения инструкции на вершине стека будет находиться значение, удовлетворяющее типу type. Остальные элементы стека не изменились.

StoreElement

Правило типизации инструкции StoreElement требует, чтобы на вершине стека до выполнения инструкции располагались тип элементов массива type, тип INT и тип массива type[] (рис. 62). Остальные элементы стека до выполнения инструкции должны совпадать с элементами стека после выполнения инструкции.

$$\boxed{\text{prog} \vdash_t \text{StoreElement} : \text{type::INT::type[]::tStack} \rightarrow \text{tStack}}$$

Рис. 62. Правило типизации инструкции StoreElement.

Инструкция StoreElement, в отличие от StoreField, практически не накладывает ограничения на типы. Согласно правилам типизации последовательности инструкций, типы на стеке могут быть в любой момент обобщены. Поэтому можно считать, что типы максимально обобщены перед проверкой этого правила. То есть типы на вершине стека — это либо INT, либо FLOAT, либо OBJECT, а третий тип в стеке — это либо INT[], либо FLOAT[], либо OBJECT[]. Поэтому это правило только проверяет, что эти типы согласованы:

- либо тип записываемого значения INT и тип массива INT[];
- либо тип записываемого значения FLOAT и тип массива FLOAT[];
- либо тип записываемого значения OBJECT и тип массива OBJECT[].

Тем самым типизация не позволяет избежать проверки во время выполнения: при выполнении инструкции необходимо проверить, что записываемое значение удовлетворяет типу элементов массива.

На стеке до выполнения инструкции находятся значение, которое необходимо записать (удовлетворяющее типу type), целое число (номер элемента) и адрес массива (значение, удовлетворяющее типу type[]). Во время выполнения инструкции происходит проверка, что значение удовлетворяет реальному типу элементов массива.

2.4.6. Система уравнений на типы

Выше доказано, что если программа типизируема, то в каждый момент времени на стеке, в переменных методов и в полях объектов находятся значения, удовлетворяющие соответствующим типам.

Одним из способов нахождения типизирующей функции метода является составление и решение системы уравнений.

Вначале по описанным правилам составляются и разрешаются уравнения на размер стека в каждой точке метода.

Затем по описанным правилам составляются уравнения на типы элементов стека в каждой точке метода. Количество элементов стека определяется согласно решению предыдущей системы.

Переменные принимают только конечное число значений: они могут быть либо INT, либо FLOAT, либо OBJECT, либо именем описанного класса, либо массивом. Тип массива определяется типом его элементов, который явно прописан в инструкции создания массива, поэтому вложенность массивов (когда значения элементов одного массива являются адресами других массивов) друг в друга ограничена и определяется по исходной программе.

В результате получаем конечную систему конечных уравнений, переменные которых пробегают конечное множество. Для решения этой системы можно использовать один из хорошо известных (и эффективных) методов.

Если система разрешима, то по решению строятся типизирующие функции методов. Это означает, что вся программа типизируемая.

2.5. Выводы

В работах [15,25,38,42] по специализации программ [31] используются различные модельные объектно-ориентированные языки [7] (таб. 3). В работах Х. Масухара [15,38] используется стековый язык без возможности работы с массивами, все вызовы статические. В работе П. Бертелсена [25] используется стековый язык без возможности работы с объектами и без вызовов методов. В работе У. Шульца [42] используется нестековый язык: каждый метод — это одно выражение.

В книге «A Theory of Objects» [14] представлены различные теоретические объектно-ориентированные языки. Однако они заметно отличаются от широко используемых языков платформ Java [47] и Microsoft .NET [46].

Язык	Инструкции передачи управления Goto и If	Работа с		Вызовы методов
		массивами	объектами	
Х. Масухара	да	нет	да	статические
П. Бертелсен	да	да	нет	отсутствуют
У. Шульц	нет	нет	да	виртуальные
SOOL	да	да	да	виртуальные

Таб. 5. Классификация используемых объектно-ориентированных языков.

Стековый объектно-ориентированный язык SOOL является аналогом таких языков, как CIL платформы Microsoft.NET или Java Byte Code платформы Java. Но в отличие от этих языков, SOOL обладает простым синтаксисом и формально заданной семантикой, что позволяет более компактно и просто описывать правила анализа и преобразования программ.

В то же время, язык SOOL содержит все необходимые операции над объектами и массивами (таб. 5), что позволяет переносить все разработанные преобразования для SOOL на другие объектно-ориентированные языки.

В главе описаны вычисление и типизация программы на языке SOOL. Доказана теорема о типизации программы: если программа типизируема, то во время выполнения значения будут удовлетворять указанным типам. Это позволяет не производить проверок на соответствие типов во время исполнения программ, а также устраняет необходимость некоторых проверок в процессе преобразования или генерации программ.

Глава 3. Анализ времен связывания

3.1. *Введение*

В данной главе формально описывается поливариантный [2,41,26] анализ времен связывания [6,8,10] для стекового объектно-ориентированного языка SOOL (Stack-based Object-oriented Language) [13]. Описана структура BT-разметки и правила, которым должна удовлетворять корректно построенная разметка.

Данный анализ времен связывания используется в специализаторе CILPE [3,5,7,9,28], созданном на основе метода частичных вычислений, для платформы Microsoft .NET [46,48].

Цель анализа времен связывания (Binding Time Analysis, BT-анализ) — построить BT-разметку: приписать каждой инструкции ее BT-вид, а аргументам и результатам методов, элементам стека и переменным — BT-значение.

Для разметки инструкций используются BT-виды S, D и X. Разметка S (Static) показывает, что эта инструкция может быть выполнена генератором остаточной программы. Такие инструкции будем называть S-инструкциями. Разметка D (Dynamic) показывает, что инструкция не может быть обработана генератором остаточной программы и должна перейти в остаточную программу без изменения. Такие инструкции будем называть D-инструкциями. Разметка X (eXclusive) используется для разметки особых инструкций, которые переходят в остаточную программу в преобразованном виде. Такие инструкции будем называть X-инструкциями.

Аргументы и результаты методов, элементы стека и переменные размечаются BT-значениями с BT-видами S или D и в зависимости от типа элемента дополнительной разметкой, о которой будет сказано ниже. Аргументы, результаты, элементы стека и переменные, размеченные S, будем называть S-аргументами, S-результатами, S-элементами стека и S-переменными соответственно. Данные, хранящиеся в них, будут вычислены генератором остаточ-

ной программы. Такие данные будем называть S-данными. Сущности, размеченные как D, будем называть D-аргументами, D-результатами, D-элементами стека и D-переменными. Данные, записанные в таких местах, будем называть D-данными.

Разметка должна быть корректной, внутренне согласованной: S-инструкции должны потреблять и вырабатывать только S-данные, а D-инструкции — D-данные. Только в этом случае генератор остаточной программы сможет выполнить S-инструкции и вычислить все S-данные. Точные правила BT-разметки даны ниже.

Примитивное S-данное может стать аргументом D-инструкции, если оно будет предварительно «поднято» в D, путем вставки операции «поднятия» Lift. Обратный переход из D в S невозможен.

Итак, задача BT-анализа — построить корректную BT-разметку программы и программных элементов, основываясь на разметке аргументов и, может быть, каких-нибудь других элементов программы.

Программу можно корректным образом разметить многими способами. Но для более эффективной работы остаточной программы нужно, чтобы как можно больше инструкций были классифицированы как S и выполнены во время генерации остаточной программы. Поэтому при построении разметки необходимо разметить S как можно больше инструкций.

При построении разметки каждому исходному методу может соответствовать ровно один размеченный метод. В этом случае BT-анализ называется моновариантным по методам. Это влечет одинаковость разметки аргументов метода во всех точках вызова этого метода. Поэтому удобней и эффективней рассматривать поливариантный по методам BT-анализ [6,8,10]. В этом случае BT-анализ может построить несколько разметок метода в зависимости от разметок аргументов метода.

Также во время построения разметки метода BT-анализ может выполнять эквивалентные преобразования программы (например, разворачивать

циклы или дублировать код после условия) для достижения более качественной разметки. Такой BT-анализ называют поливариантным по инструкциям [6,8,10]. В противоположность этому, моновариантный по инструкциям BT-анализ размечает методы без преобразования (за исключением добавления инструкции Lift, о которой сказано ниже).

Опишем допустимую разметку и требования, предъявляемые к разметке. Данные требования подходят как для моновариантного по методам и инструкциям BT-анализа, так и для поливариантного BT-анализа.

3.2. BT-разметка

Размеченная программа состоит из двух частей: BT-программы и BT-кучи. В BT-программе к каждой инструкции приписана разметка S, D или X, а к каждому типу аргумента и результата приписана разметка, описываемая объектами в BT-куче.

3.2.1. BT-программа

Разметка определяется для всей программы, поэтому дадим определение размеченной BT-программе (рис. 63).

```
BTProgram = [BTClassDef]

BTClassDef = (ClassName, [ClassName], [FieldDec], [BTMethodDef])
BTMethodDef = (MethodName, IsInline, [BTType], [BTType], BTMethodBody)
BTMethodSig = (MethodName, IsInline, [BTType], [BTType])

IsInline = INLINE | NOINLINE
BTType = (Type, BTValue)
```

Рис. 63. BT-программа.

BT-программа состоит из определений BT-классов. Каждый BT-класс, как и обычный класс, описывается именем класса, списком имен классов (от которых данный класс наследуется), объявлением полей и определением BT-методов.

Каждое определение BT-метода состоит из имени метода, флага IsInline,

о котором сказано ниже, двух списков BT-типов, обозначающих разметку аргументов и результатов, и BT-тело метода. BT-тип — это тип и BT-значение, о котором сказано ниже.

В BT-программе к каждому имени метода может быть приписан флаг INLINE или NOINLINE. INLINE означает, что вызов этого метода будет раскрыт при генерации остаточной программы: вместо вызова метода будет подставлено тело метода.

BT-тело метода состоит из объявлений переменных и BT-инструкций, состоящих из инструкции и разметки S, D или X (рис. 64). По сравнению с описанием обычного языка SOOL добавлена новая инструкция Lift, которая описана ниже.

```
BTMethodBody = ([VarDec], [BTInstruction])
BTInstruction = (Instruction', BTInstructionAnnotation) = Instruction^BTInstructionKind
BTInstructionAnnotation = BTInstructionKind | BTNewObjectAnnotation
BTNewObjectAnnotation = (BTInstructionKind, BTRefValue)
BTInstructionKind = S | D | X
Instruction' = Instruction | Lift
```

Рис. 64. BT-метод.

BT-виды, записанные в BT-инструкциях, будем называть BT-разметкой или разметкой BT-инструкций.

Для компактного описания правил разметки BT-инструкции будем записывать не в виде пары инструкция и BT-вид, а в виде инструкции с верхним индексом BT-вида: например, $\text{DuplicateStackTop}^{S}$.

Для инструкции NewObject class используется расширенная разметка BTNewObjectAnnotation, описывающая как разметку инструкции, так и разметку создаваемого объекта. В таком случае разметку будем записывать в виде $\text{NewObject}^{\text{BTInstructionKind}}\ \text{ClassName}^{\text{BTRefValue}}$.

Если сравнивать программу и размеченную BT-программу, то разница заключается в следующих аспектах: во-первых, к каждому методу приписан флаг, показывающий нужно ли раскрывать метод (INLINE) во время генера-

ции остаточной программы или нет (NOINLINE); во-вторых, к каждому аргументу и результату метода в BT-программе приписана разметка BT-Value; в-третьих, к каждой инструкции метода в BT-программе приписана разметка BTInstructionKind.

3.2.2. Выполнение размеченной программы

Размеченную программу можно выполнять как любую другую программу на языке SOOL.

Отличие размеченной программы от программы на языке SOOL заключается, во-первых, в разметках, приписанных к инструкциям и типам аргументов и результатов метода, во-вторых, в дополнительной информации, заданной BT-кучей, и, в-третьих, в новой инструкции Lift, которая может встречаться в теле размеченной программы.

При выполнении размеченной программы разметка и BT-куча игнорируются, а инструкция Lift выполняется, как ничего не делающая инструкция (рис. 65).

$$\text{prog} \vdash_i \text{Lift} : (\text{val::stack, env, heap}) \rightarrow (\text{val::stack, env, heap})$$

Рис. 65. Правило выполнения инструкции Lift.

Аналогично правило типизации инструкции Lift показывает, что типы до и после выполнения инструкции должны совпадать (рис. 66).

$$\text{prog} \vdash_t \text{Lift} : \text{primType::tStack} \rightarrow \text{primType::tStack}$$

Рис. 66. Правило типизации инструкции Lift.

Приведенные правила показывают, что размеченную программу можно рассматривать как обычную программу на языке SOOL.

Приведенные выше правила корректности BT-программы показывают только внутреннюю согласованность разметки. Но при построении размеченной программы, также необходимо гарантировать, что исходная и размеченная программы эквивалентны: при любых входных данных результаты выполнения программ должны совпадать.

Одним из способов достижения эквивалентности является минимальное изменение программы при разметке: только добавление разметки и в необходимых местах добавление инструкции Lift. Но такой способ, как было отмечено выше, не всегда позволяет достаточно качественно разметить программу: почти все инструкции могут оказаться размечены D.

3.2.3. Корректность BT-программы

Как и для корректных программ на языке SOOL, для корректных размеченных программ требуется, чтобы все имена локальных переменных, классов, полей и методов (за исключением переопределенных методов) были уникальными. Этого можно добиться переименованием имен. Также требуется, чтобы в теле метода в инструкциях использовались только определенные в программе локальные переменные, классы, поля и методы.

Все вызовы в языке SOOL виртуальные — т.е. тело метода будет выбрано по типу первого аргумента. Поэтому при определении BT-программы требуется, чтобы все методы с именем mthd имели одинаковую BT-сигнатуру: чтобы значения флага и BT-значения соответствующих аргументов и результатов совпадали.

Будем считать, что для каждой размеченной программы определена функция $\text{MethodBTSignature}_{\text{btProg}}$, которая по имени метода возвращает разметку аргументов и результатов (рис. 67).

$$\boxed{\text{MethodBTSignature}_{\text{btProg}} : \text{MethodName} \rightarrow ([\text{BTValue}], [\text{BTValue}])}$$

Рис. 67. Вспомогательная функция.

Как и для обычных программ, метод Main класса MAIN должен существовать и его аргументы должны иметь примитивные типы. Этот метод должен иметь флаг NOINLINE.

Программа должна быть типизируемой. О выполнении и типизации новой инструкции Lift сказано ниже.

В зависимости от значения флага метода IsInline на разметку аргументов

и результатов метода накладываются различные условия.

Если метод с флагом INLINE, то BT-вид первого аргумента обязан быть S. В этом случае генератор остаточной программы сможет произвести раскрытие виртуального вызова: по типу первого аргумента выбрать нужный метод.

Метод может возвращать значение как через результаты, так и изменяя значения полей объектов или элементов массивов, переданных как аргументы метода. Если указан флаг NOINLINE, то генератор остаточной программы не раскрывает метод, а генерирует вызов специализированной версии метода. И если бы возвращаемые значения (как результаты, так и поля объектов или элементы массивов в аргументах) имели разметку S, то возможна ситуация, что в зависимости от неизвестных данных в одном случае метод должен вернуть одно S-значение, а в другом — другое. И если метод не раскрывается, то нельзя выбрать подходящее S-значение для передачи в вызывающий метод. Поэтому если метод с флагом NOINLINE, то BT-виды результатов и BT-виды полей объектов или элементов массивов в аргументах должны быть D.

Если вышеприведенные требования выполнены, то такую BT-программу будем называть корректной (рис. 68). Ниже рассматриваются только корректные BT-программы.

- Все локальные переменные, классы, поля имеют различные уникальные названия.
- Все методы имеют различные уникальные названия за исключением переопределенных методов.
- btProg — типизируема
- Все методы с одним именем имеют одинаковую BT-сигнатуру.
- Метод Main класса MAIN определен и имеет только примитивные аргументы и флаг NOINLINE.
- Если метод INLINE, то BT-вид первого аргумента S.
- Если метод NOINLINE, то BT-вид всех результатов, полей объектов и элементов массивов D.

---

btProg — корректна

Рис. 68. Корректность BT-программы.

3.2.4. BT-куча

Для определения разметки аргументов и результатов будем использовать *BT-значения* и *BT-кучу*. BT-куча описывает все возможные состояния динамической памяти (кучи) во время исполнения программы. Она показывает, какая часть объектов статическая, т.е. над какими объектами операции могут быть выполнены во время генерации остаточной программы, а какая часть динамическая — операции над этими объектами перейдут в остаточную программу. Такие объекты будут созданы и операции над ними будут выполнены во время исполнения остаточной программы.

BT-значение — это либо примитивное BT-значение, состоящее из примитивного типа и BT-вида S или D, либо ссылочное BT-значение, являющееся адресом (рис. 69).

```
BTHeap = Address → (BTKind, [Type], BTObject)

BTValue = BTPrimValue | BTRefValue
BTPrimValue = (PrimType, BTKind) = PrimType^BTKind
BTKind = S | D

BTRefValue = Address
BTObject = (FieldName | ELEMENT) → BTValue

BTKindOf_btHeap : BTValue→ BTKind
BTKindOf_btHeap (primType, btKind) = btKind
BTKindOf_btHeap ref = btKind where (btKind, _, _) = btHeap(ref)
```

Рис. 69. BT-куча.

BT-куча — это отображение адреса в тройку: BT-вид S или D, список типов и BT-объект. В отличие от обычной кучи, в BT-куче каждый BT-объект определяется списком типов. BT-объект — это отображение имен полей всех классов, определенных в этом списке типов, или особого имени ELEMENT в BT-значение. Это отображение используется для определения разметки полей BT-объекта и разметки элементов массива (по ключу ELEMENT).

Для удобства описания примитивные типы будем записывать не в виде

пары (тип, BT-вид), а приписывать BT-вид в виде верхнего индекса к типу, например, INT^S, $FLOAT^D$.

Для определения BT-вида ссылочных и примитивных BT-значений будем использовать функцию $BTKindOf_{btHeap}$, которая по BT-значению возвращает BT-вид. Этот BT-вид будем называть BT-разметкой или просто разметкой BT-значений.

Таким образом, BT-куча — это абстракция обычной кучи времени исполнения программы: вместо обычных значений в полях хранятся BT-значения.

Следует обратить внимание на ключевое отличие BT-объектов от обычных объектов и разметок, которые описываются в предшествующих работах: BT-объект может определяться несколькими классами. В BT-объекте присутствуют все поля этих классов. Это позволяет размечать статически поля объектов, точный тип которых не может быть определен во время анализа.

Отметим, что в BT-куче может быть несколько BT-объектов, с одним и тем же классом. Это поливариантность анализа по классам.

При выполнении исходной программы в каждый момент времени можно построить отображение текущего состояния кучи в BT-кучу (рис. 70). Диагональной штриховкой отмечены те объекты в куче, которым соответствуют BT-объекты с BT-видом S. А серым — с BT-видом D.

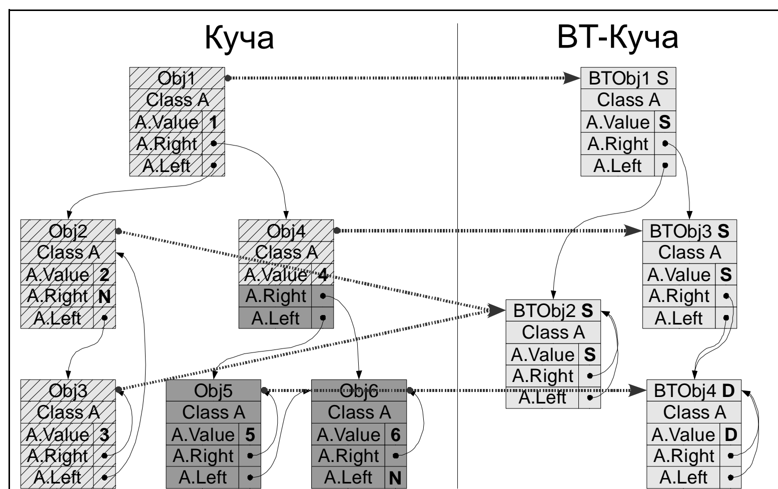


Рис. 70. BT-куча и связь с кучей времени исполнения.

Причем это отображение согласовано со ссылками между объектами и массивами, т.е. является гомоморфизмом. Пусть *F* — отображение, *obj* — объект с полем *fld* ссылочного типа, тогда *F(obj.fld) = F(obj).fld*.

Описанная ВТ-куча обладает следующими характеристиками, из-за которых разработанный метод частичных вычислений отличается от работ предшественников:

- она представляет собой граф;
- каждый ВТ-объект может описываться несколькими классами;
- каждый класс может встречаться в нескольких ВТ-объектах.

Именно эти особенности позволяют эффективно специализировать объектно-ориентированные программы.

Вторая часть разметки — это разметка методов. Она состоит из

- ВТ-объектов, приписанных каждому аргументу или результату метода, инструкции создания объектов или массивов ВТ-объекта. Они задают разметку аргумента или результата, локальной переменной, создаваемого объекта или массива соответственно;
- ВТ-вид S, D или X, приписанных каждой инструкции (ВТ-вид X используется только для инструкций). Они обозначают, что данная инструкция статическая, динамическая или особая соответственно. Этими ВТ-видами руководствуется генератор остаточной программы для построения остаточной программы.

В процессе анализа строятся разметки локальных переменных и элементов на стеке. Отображение между кучей и ВТ-кучей продолжается на отображение состояния в ВТ-состояние, которое включает в себя разметку элементов стека, локальных переменных и ВТ-кучу.

3.2.5. Корректность ВТ-кучи

ВТ-куча корректна, если у объектов ВТ-вида D все поля имеют ВТ-вид D (рис. 71).

$$\frac{\forall btRefValue \in Domain(btHead),\quad (_, _, btObject) = btHead(btRefValue),\quad \forall fldName \in Domain(btObject),\quad btVal = btObject(fldName) :\quad BTKindOf_{btHeap}(btRefValue) = D \Rightarrow BTKindOf_{btHeap}(btVal) = D}{btHeap \text{ — корректна}}$$

Рис. 71. Корректность BT-кучи.

3.2.6. Корректность BT-разметки

Для проверки, что BT-разметка построена правильно, как и в случае с типизацией, необходимо построить размеченное состояние в каждой точке метода, которое определяет разметку элементов стека и переменных.

BT-разметка зависит от BT-разметки аргументов программы. Напомним, что аргументами и результатами программы могут быть только данные примитивных типов. Поэтому для задания BT-разметки аргументов достаточно задать BT-виды аргументов.

3.3. Правила BT-разметки

Правила BT-разметки описывают ограничения на разметку до и после выполнения инструкции. Если построены BT-программа, BT-куча и разметка в каждой точке программы и они согласованы по ниже приведенным правилам, то такая разметка называется корректной.

По корректной разметке и начальным данным генератор остаточной программы может построить специализированную версию программы, выполнив все S-инструкции.

Правила BT-разметки похожи на правила, описываемые в стиле операционной семантики [32,40]. Однако эти правила следует рассматривать, не как правила вывода BT-состояний, а как ограничение на состояние до и после выполнения инструкции (аналогично правилам для типизации программы). И для построения корректной разметки необходимо решить систему уравнений, рассматривая каждое правила как уравнение.

3.3.1. BT-состояние

BT-состояние похоже на обычное состояния, только в нем нет кучи. BT-

состояние состоит из BT-стека и BT-окружения (рис. 72). BT-стек — это список BT-значений, а BT-окружение — это отображение переменных в BT-значения.

BTState = (BTStack, BTEnv)

BTStack = [BTValue]
BTEnv = Var → BTValue

Рис. 72. BT-состояние.

3.3.2. BT-разметка программы

Чтобы проверить, что BT-программа btProg и BT-куча btHeap построены правильно, нужно проверить следующее (рис. 73):

1. btHeap — корректна
2. btProg — корректна и типизируема
3. $\exists\lambda_0$: Integer → BTState :
   instrs = [NewObject^D MAIN, CallMethod^X Main, Leave^X],
   btArg = fst $\lambda_0(0)$,   btRes = fst $\lambda_0(2)$,
   $\forall$btVal $\in$ btArg :   (primType, _) = btVal,
   $\forall$btVal $\in$ btRes :   BTKind(btVal) = D,
   $\forall$n $0 \le n < 3$:   btProg, btHeap, λ_0, btRes $\vdash_{bt}$ (instrs, n)
4. $\forall$btClass $\in$ btProg,   (_, _, _, btMthdDefs) = btClasss,
   $\forall$btMthdDef $\in$ btMthdDefs, $\exists\lambda_{btMthdDec}$: Integer → BTState :
   btProg, btHeap, $\lambda_{btMthdDec} \vdash_{bt}$ btMthdDef

---

btHeap $\vdash_{bt}$ btProg

Рис. 73. Правило BT-разметки программы.

1. Правила BT-разметки корректны только для корректной BT-кучи btHeap.
2. Правила BT-разметки корректны только для корректной и типизируемой программы, поэтому необходимо проверить, что программа btProg корректна и типизируема.
3. Для начального метода [NewObject^D MAIN, CallMethod^X Main, Leave^X] необходимо проверить существование отображения λ_0 : Integer → BTState, проверяющего BT-разметку этого метода. Причем

a. BT-аргументы программы $\lambda_0(0)$ должны быть примитивного типа, т.к. аргументы программы и метода Main должны быть примитивного типа.
b. BT-виды разметки результатов $\lambda_0(2)$ должны быть D.

4. Для каждого определения BT-метода btMthdDef необходимо проверить существование отображения $\lambda_{btMthdDec}$: Integer → BTState, проверяющего BT-разметку этого метода.

3.3.3. BT-разметка метода

Размеченная BT-программа btProg, BT-куча btHeap и отображение $\lambda_{btMthdDec}$ корректно указывают BT-типы для BT-метода btMthdDef, если (рис. 74):

(_, btTypeArg, btTypeRes, (_, btInstrs)) = btMthdDef
$\forall$btType $\in$ btTypeArg :

- либо (btType — ссылочный BT-тип)
  (type, btOref) = btType, (_, types, _) = btHeap(btOref), type $\in$ types
- либо (btType — примитивный BT-тип)
  ($type_1$, ($type_2$, btKind)) = btType, $type_1 = type_2$

$\forall$btType $\in$ btTypeRes :

- либо (btType — ссылочный BT-тип)
  (type, btOref) = btType, (_, types, _) = btHeap(btOref), type $\in$ types
- либо (btType — примитивный BT-тип)
  ($type_1$, ($type_2$, btKind)) = btType, $type_1 = type_2$

fst $\lambda_{btMthdDec}(0)$ = btArg = map snd btTypeArg   btRes = map snd btTypeRes
$\forall$n $0 \le n < length(instrs)$: btProg, btHeap, $\lambda_{btMthdDec}$, btRes $\vdash_{bt}$ (btInstrs, n)

---

btProg, btHeap, $\lambda_{btMthdDec} \vdash_{bt}$ btMthdDef

Рис. 74. Правило BT-разметки метода.

1. Для всех BT-типов аргументов и результатов метода необходимо, чтобы их тип и тип BT-значения совпадали в случае примитивного типа или их тип был элементом списка типов BT-значения в случае ссылочного типа.
2. BT-значения, находящиеся на стеке перед выполнением первой инструкции (fst $\lambda_{btMthdDec}(0)$), совпадают с BT-значениями, приписанными к типам в BT-типах.

3. Для каждого номера инструкции n в контексте BT-программы btProg, BT-кучи btHeap, отображения $\lambda_{btMthdDec}$ и BT-типов результата btRes разметка n-ой инструкции из списка instrs корректна.

3.3.4. BT-разметка последовательности инструкций

Корректность разметки n-ой инструкции в списке инструкций instrs проверяется в зависимости от этой инструкции:

1. Если n-ая инструкция — это инструкция $Leave^X$, то BT-значения на стеке (fst $\lambda_{btMthdDec}(n)$) должны совпадать с BT-значениями результата (рис. 75).

$$\frac{Leave^X = btInstrs[n] \qquad fst\ \lambda_{btMthdDec}(n) = btRes}{btProg, btHeap, \lambda_{btMthdDec}, btRes \vdash_{bt} (btInstrs, n)}$$

Рис. 75. Правило BT-разметки последовательности инструкций, инструкция $Leave^X$.

2. Если n-ая инструкция — это инструкция $Goto^S$ m, то BT-значения на стеке и в переменных перед выполнением этой инструкции $\lambda_{btMthdDec}(n)$ должны совпадать с BT-значениями перед m-ой инструкцией $\lambda_{btMthdDec}(m)$ (рис. 76).

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

Рис. 76. Правило BT-разметки последовательности инструкций, инструкция $Goto^S$ m.

3. Если n-ая инструкция — это инструкция $Branch^x$ m, то разметка x инструкции — либо S, либо D. BT-окружения до выполнения этой n-ой инструкции, до выполнения (n+1)-ой инструкции и до выполнения m-ой инструкции должны совпадать. BT-стек до выполнения этой n-ой инструкции без головного элемента и BT-стеки до выполнения (n+1)-ой инструкции и до выполнения m-ой инструкции должны совпадать. BT-значение, находящееся на вершине стека до выполнения этой n-ой инст-

рукции, должно быть INT^x, где x — BT-вид инструкции Branchx m (рис. 77).

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

Рис. 77. Правило BT-разметки последовательности инструкций, инструкция $Branch^x$ m.

4. Для остальных инструкций необходимо проверить правила, записанные в виде правил связи состояния до и после выполнения инструкции (рис. 78):

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

Рис. 78. Правило BT-разметки последовательности инструкций.

3.3.5. BT-разметка инструкций

DuplicateStackTop

Инструкция DuplicateStackTop копирует элемент на вершине стека.

Правило BT-разметки требует, чтобы два BT-значения на вершине стека после выполнения инструкции совпадали с BT-значением на вершине стека до выполнения инструкции (рис. 79). Остальные элементы стека и окружения до и после выполнения инструкции также должны совпадать.

$$\frac{x = BTKindOf_{btHeap}(btVal) = S \mid D \qquad btStack' = btVal::btVal::btStack}{prog,btHeap \vdash_{bt} DuplicateStackTop^x:(btVal::btStack,btEnv) \rightarrow (btStack',btEnv)}$$

Рис. 79. Правило BT-разметки инструкции $DuplicateStackTop^x$.

Также правило требует, чтобы разметка BT-значения на вершине стека до выполнения инструкции и разметка инструкции совпадали и равнялись либо S, либо D.

RemoveStackTop

Инструкция RemoveStackTop удаляет элемент на вершине стека.

Правило BT-разметки требует, чтобы стек до выполнения инструкции без верхнего элемента совпадал со стеком после выполнения инструкции (рис. 80). Окружение не изменялось.

$$\frac{x = \mathrm{BTKindOf}_{btHeap}(btVal) = S \mid D}{prog,btHeap \vdash_{bt} \mathrm{RemoveStackTop}^{x} : (btVal::btStack,btEnv) \rightarrow (btStack,btEnv)}$$

Рис. 80. Правило BT-разметки инструкции RemoveStackTop^x.

Разметка инструкции и разметка BT-значения на вершине стека до выполнения инструкции должны совпадать и равняться либо S, либо D.

LoadConst const

Инструкция LoadConst const загружает константу const.

Правило BT-разметки требует, чтобы стек после выполнения инструкции без верхнего элемента совпадал со стеком до выполнения инструкции (рис. 81). Окружение не изменялось. BT-значение, находящееся на вершине стека после выполнения инструкции может быть следующим:

$x = \mathrm{BTKindOf}_{btHeap}(btVal) = S \mid D$

- либо $btVal = INT^x$, если const — целое число
- либо $btVal = FLOAT^x$, если const — число с плавающей точкой
- либо btVal : BTRefValue, если const = NULL

$$\overline{prog,btHeap \vdash_{bt} \mathrm{LoadConst}^{x}\ const : (btStack,btEnv) \rightarrow (btVal::btStack,btEnv)}$$

Рис. 81. Правило BT-разметки инструкции LoadConst^x const.

- если const — целое число, то BT-значение — это INT^x;
- если const — число с плавающей точкой, то BT-значение — это FLOAT^x;
- если const — NULL, то BT-значение — адрес какого-либо BT-объекта в BT-куче btHeap.

Разметка инструкции и разметка этого BT-значения должны совпадать и могут быть либо S, либо D.

UnaryOp op

Инструкция UnaryOp op выполняет унарную операцию op.

Правило BT-разметки требует, чтобы совпадали окружения и стеки до и после выполнения за исключением вершин стеков (рис. 82). BT-значения, расположенные на вершинах стеков могут быть следующими:

- либо оба типа INT^x, если операция op — это NOT или NEG;
- либо оба типа $FLOAT^x$, если операция op — это NEG;
- либо BT-значение до выполнения — INT^x, а после выполнения — $FLOAT^x$, если операция op — это INT2FLOAT;
- либо BT-значение до выполнения — $FLOAT^x$, а после выполнения — INT^x, если операция op — это FLOAT2INT.

$$\frac{\begin{array}{l} x = BTKindOf_{btHeap}(btVal_1) = BTKindOf_{btHeap}(btVal_2) = S \mid D \\ \bullet\ \text{либо } btVal_1 = btVal_2 = INT^x\text{, если } op = NOT \text{ или } op = NEG \\ \bullet\ \text{либо } btVal_1 = btVal_2 = FLOAT^x\text{, если } op = NEG \\ \bullet\ \text{либо } btVal_1 = INT^x, btVal_2 = FLOAT^x\text{, если } op = INT2FLOAT \\ \bullet\ \text{либо } btVal_1 = FLOAT^x, btVal_2 = INT^x\text{, если } op = FLOAT2INT \end{array}}{prog,btHeap \vdash_{bt} UnaryOp^x\ op{:}(btVal_1{::}btStack,btEnv) \rightarrow (btVal_2{::}btStack,btEnv)}$$

Рис. 82. Правило BT-разметки инструкции $UnaryOp^x$ op

Разметка инструкции и разметка этих BT-значений должны совпадать и могут быть либо S, либо D.

BinaryOp op

Инструкция BinaryOp op выполняет бинарную операцию op.

Правило BT-разметки требует, чтобы совпадали окружения и стеки до и после выполнения за исключением двух BT-значений, расположенных на вершине стека до выполнения инструкции, и одного BT-значения, расположенного на вершине стека после выполнения инструкции (рис. 83). Эти BT-значения могут быть следующими:

- все три BT-значения — INT^x, если операция op — любая операция: арифметическая (ADD, DIV, MUL, REM, SUB), побитовая (AND, OR, XOR, SHL, SHR) или операция сравнения (CEQ, CGT, CLT);
- все три BT-значения — $FLOAT^x$, если операция op — арифметическая

операция (ADD, DIV, MUL, REM, SUB);

- два BT-значения на вершине стека до выполнения инструкции — $FLOAT^x$, а BT-значение на вершине стека после выполнения инструкции — INT^x, если операция op — операция сравнения (CEQ, CGT, CLT);
- два BT-значения на вершине стека — это адреса BT-объектов в BT-куче btHeap, а BT-значение на вершине стека после выполнения инструкции — INT^x, если операция op — операция сравнения на равенство (CEQ).

$$\frac{\begin{array}{l} x = BTKindOf_{btHeap}(btVal_1) = BTKindOf_{btHeap}(btVal_2) = \\ \qquad = BTKindOf_{btHeap}(btVal_3) = S \mid D \\ \bullet \text{ либо } btVal_1 = btVal_2 = btVal_3 = INT^x, \\ \quad \text{если } op = ADD, DIV, MUL, REM, SUB, ADD, DIV, MUL, \\ \quad REM, SUB, AND, OR, XOR, SHL, SHR, CEQ, CGT, CLT \\ \bullet \text{ либо } btVal_1 = btVal_2 = btVal_3 = FLOAT^x, \\ \quad \text{если } op = ADD, DIV, MUL, REM, SUB \\ \bullet \text{ либо } btVal_1 = btVal_2 = FLOAT^x, btVal_3 = INT^x, \\ \quad \text{если } op = CEQ, CGT, CLT \\ \bullet \text{ либо } btVal_1 : BTRefValue, btVal_2 : BTRefValue, btVal_3 = INT^x, \\ \quad \text{если } op = CEQ \end{array}}{\begin{array}{l} prog, btHeap \vdash_{bt} UnaryOp^x\ op : (btVal_1::btVal_2::btStack, btEnv) \rightarrow \\ \qquad (btVal_3::btStack, btEnv) \end{array}}$$

Рис. 83. Правило BT-разметки инструкции $BinaryOp^x$ op.

Разметка инструкции и разметка этих BT-значений должны совпадать и могут быть либо S, либо D.

LoadVar var

Инструкция LoadVar var читает значение переменной var.

$$\frac{\begin{array}{c} btVal = btEnv(var) \\ \bullet \text{ либо } x = S, BTKindOf_{btHeap}(btVal) = S \\ \bullet \text{ либо } x = X, BTKindOf_{btHeap}(btVal) = D \end{array}}{prog, btHeap \vdash_{bt} LoadVar^x\ var : (btStack, btEnv) \rightarrow (btVal::btStack, btEnv)}$$

Рис. 84. Правило BT-разметки инструкции $LoadVar^x$ var.

Правило BT-разметки требует, чтобы стек после выполнения инструкции состоял из BT-значения переменной var, записанного в окружении, и сте-

ка до выполнения инструкции (рис. 84). Окружения до и после выполнения инструкции должны совпадать.

В зависимости от разметки BT-значения переменной var S или D инструкция должна быть размечена либо S, либо X соответственно.

StoreVar var

Инструкция StoreVar var записывает значение переменной var.

Правило BT-разметки требует, чтобы стек до выполнения инструкции состоял из BT-значения и стека после выполнения инструкции (рис. 85). Окружения после выполнения инструкции должны совпадать с окружением до выполнения инструкции за исключением значения на переменной var, которое должно быть равным BT-значению со стека.

$$\frac{\begin{array}{c}\bullet\ \text{либо } x = S,\ \mathrm{BTKindOf}_{btHeap}(btVal) = S \\ \bullet\ \text{либо } x = X,\ \mathrm{BTKindOf}_{btHeap}(btVal) = D \\ btEnv' = btEnv[var \mapsto btVal]\end{array}}{prog,\ btHeap \vdash_{bt} \mathrm{StoreVar}^{x}\ var : (btVal{::}btStack,\ btEnv) \rightarrow (btStack,\ btEnv')}$$

Рис. 85. Правило BT-разметки инструкции StoreVar^{x} var.

В зависимости от разметки BT-значения на вершине стека S или D инструкция должна быть размечена либо S, либо X соответственно.

NewObject class

Инструкция NewObject class создает новый объект класса class.

$$\frac{\begin{array}{c}(_,\ types,\ btObj) = btHeap(btOref) \qquad class \in types \\ \bullet\ \text{либо } x = X,\ \mathrm{BTKindOf}_{btHeap}(btOref) = S \\ \bullet\ \text{либо } x = D,\ \mathrm{BTKindOf}_{btHeap}(btOref) = D\end{array}}{\begin{array}{r}prog,\ btHeap \vdash_{bt} \mathrm{NewObject}^{x}\ class^{btOref} : (btStack, btEnv) \rightarrow \\ (btOref{::}btStack,\ btEnv)\end{array}}$$

Рис. 86. Правило BT-разметки инструкции NewObject^{x} class.

Правило BT-разметки требует, чтобы стек после выполнения инструкции состоял из BT-значения, адреса BT-объекта в BT-куче btHeap и стека до выполнения инструкции (рис. 86). Окружения до и после выполнения инст-

рукции должны совпадать. Также необходимо, чтобы в списке типов этого BT-объекта содержался класс class.

В зависимости от разметки S или D BT-объекта на вершине стека после выполнения инструкции инструкция должна быть размечена либо X, либо D соответственно.

LoadField fld

Инструкция LoadField fld читает у объекта поле fld.

Правило BT-разметки инструкции LoadField fld требует, чтобы BT-значение на вершине стека до выполнения инструкции было адресом BT-объекта в BT-куче btHeap (рис. 87). BT-значение на вершине стека после выполнения инструкции должно совпадать с BT-значением, соответствующим полю fld в этом BT-объекте. Остальные элементы стеков и окружения до и после выполнения инструкции должны совпадать.

$$\frac{\begin{array}{c}(_,\ _,\ btObj) = btHeap(btOref) \qquad btVal = btObj(fld)\\ \bullet\ \text{либо } x = S,\ BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = S\\ \bullet\ \text{либо } x = X,\ BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = D\\ \bullet\ \text{либо } x = D,\ BTKindOf_{btHeap}(btOref) = D,\ BTKindOf_{btHeap}(btVal) = D\end{array}}{prog,btHeap \vdash_{bt} LoadField^{x}\ fld:(btOref::btStack,btEnv) \rightarrow (btVal::btStack,btEnv)}$$

Рис. 87. Правило BT-разметки инструкции $LoadField^{x}$ fld.

В зависимости от разметки BT-объекта на вершине стека до выполнения инструкции и разметки BT-значения на вершине стека после выполнения инструкции инструкция должна быть размечена:

- либо S, если BT-объект и BT-значение размечены S;
- либо X, если BT-объект размечен S, а BT-значение размечено D;
- либо D, если BT-объект и BT-значение размечены D.

StoreField fld

Инструкция StoreField fld записывает у объекта поле fld.

Правило BT-разметки инструкции LoadField fld требует, чтобы на стеке до выполнения инструкции находилось BT-значение и адрес BT-объекта в BT-

куче btHeap, причем BT-значение должно совпадать с BT-значением, соответствующим полю fld в этом BT-объекте (рис. 88). Остальные элементы стеков и окружения до и после выполнения инструкции должны совпадать.

$$\frac{\begin{array}{c}(_,\ _,\ btObj) = btHeap(btOref) \qquad btVal = btObj(fld)\\ \bullet\ \text{либо } x = S,\ BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = S\\ \bullet\ \text{либо } x = X,\ BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = D\\ \bullet\ \text{либо } x = D,\ BTKindOf_{btHeap}(btOref) = D,\ BTKindOf_{btHeap}(btVal) = D\end{array}}{prog, btHeap \vdash_{bt} StoreField^{x}\ fld : (btVal::btOref::btStack, btEnv) \rightarrow (btStack, btEnv)}$$

Рис. 88. Правило BT-разметки инструкции $StoreField^{x}$ fld.

В зависимости от разметки BT-значения и BT-объекта на вершине стека до выполнения инструкции инструкция должна быть размечена:

- либо S, если BT-объект и BT-значение размечены S;
- либо X, если BT-объект размечен S, а BT-значение размечено D;
- либо D, если BT-объект и BT-значение размечены D.

CallMethod mthd

Инструкция CallMethod mthd вызывает у объекта виртуальный метод mthd.

Правило BT-разметки инструкции CallMethod mthd требует, чтобы на вершине стека до выполнения инструкции находились BT-значения, совпадающие с BT-значениями аргументов метода mthd, заданные в сигнатуре этого метода (рис. 89). BT-значения на вершине стека после выполнения инструкции должны совпадать с BT-значениями результатов метода mthd, заданных в сигнатуре этого метода. Остальные элементы стеков и окружения до и после выполнения инструкции должны совпадать. Разметка инструкции всегда X.

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

Рис. 89. Правило BT-разметки инструкции $CallMethod^{x}$ mthd.

CastObject type

Инструкция CastObject type приводит объект или массив к типу type.

Правило BT-разметки инструкции CastObject type требует, чтобы стеки и окружения до и после выполнения инструкции совпадали (рис. 90). Но также требует, чтобы на вершине стека находился адрес BT-объекта. Причем в списке типов этого BT-объекта должен быть хотя бы один тип, являющийся наследником типа type, заданного в параметре инструкции.

$$\frac{\begin{array}{c} btOref::_ = btStack \qquad (_, types, _) = btHeap(btOref) \\ \exists type' \in types : \quad type' \prec_{prog} type \qquad x = BTKindOf_{btHeap}(btOref) = S \mid D \end{array}}{prog, btHeap \vdash_{bt} CastObject^{x}\ type : (btStack, btEnv) \rightarrow (btStack, btEnv)}$$

Рис. 90. Правило BT-разметки инструкции CastObject^{x} type.

Разметка инструкции и разметка BT-объекта на вершине стека должны совпадать и могут быть либо S, либо D.

NewArray type

Инструкция NewArray type создает новый массив из элементов типа type.

$$\frac{\begin{array}{c} (_,types,btObj)=btHeap(btOref) \qquad type[] \in types \\ btVal=btObj(ELEMENT) \\ \bullet\ \text{либо } x = S,\ x_1 = BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = S \\ \bullet\ \text{либо } x = X,\ x_1 = BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = D \\ \bullet\ \text{либо } x = D,\ x_1 = BTKindOf_{btHeap}(btOref) = D,\ BTKindOf_{btHeap}(btVal) = D \end{array}}{\begin{array}{l} prog, btHeap \vdash_{bt} NewArray^{x}\ type : (INT^{x1}::btStack, btEnv) \rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad (btOref::btStack, btEnv) \end{array}}$$

Рис. 91. Правило BT-разметки инструкции NewArray^{x} type.

Правило BT-разметки инструкции NewArray type требует, чтобы на вершине стека до выполнения инструкции находилось BT-значение INT^{x1}, а после выполнения инструкции — ссылочное BT-значение btOref (рис. 91). Остальные элементы стека и окружения до и после выполнения инструкции должны совпадать. Ссылочное BT-значение btOref должно указывать в BT-куче на BT-объект, содержащий особое поле ELEMENT. Также требуется,

чтобы список типов BT-объекта содержал тип массива type[], где тип type задается в параметре инструкции.

Разметка инструкции, BT-значение INT^{x1}, ссылочное BT-значения btOref и BT-значение btVal, соответствующее элементам массива, связаны следующими правилами. Во-первых, разметки BT-значения INT^{x1} и ссылочного BT-значения btOref должны совпадать. Во-вторых, в зависимости от разметки BT-значения INT^{x1} и BT-значения btVal инструкция должна быть размечена:

- либо S, если BT-значения размечены S;
- либо X, если BT-значение INT^{x1} размечено S, а BT-значение btVal — D;
- либо D, если BT-значения размечены D.

LoadLength

Инструкция LoadLength загружает длину массива.

Правило BT-разметки инструкции LoadLength требует, чтобы на вершине стека до выполнения инструкции находилось ссылочное BT-значение btOref, которое указывает в BT-куче btHeap на BT-объект btObj (рис. 92). Это отображение btObj должно быть определено на особом значении ELEMENT. На вершине стека после выполнения инструкции должно находиться примитивное BT-значение INT^x. Все остальные элементы стека и окружения до и после выполнения инструкции должны совпадать.

$$\frac{(_, _, \text{btObj}) = \text{btHeap}(\text{btOref}) \qquad \text{btObj}(\text{ELEMENT}) \text{ — определено} \quad x = \text{BTKindOf}_{\text{btHeap}}(\text{btOref}) = S \mid D}{\text{prog},\text{btHeap}\vdash_{\text{bt}} \text{LoadLength}^x : (\text{btOref}::\text{btStack},\text{btEnv}) \rightarrow (\text{INT}^x::\text{btStack},\text{btEnv})}$$

Рис. 92. Правило BT-разметки инструкции LoadLength^x.

Разметки инструкции, ссылочного BT-значения на вершине стека до выполнения и примитивного BT-значения на вершине стека после выполнения инструкции должны совпадать и могут быть либо S, либо D.

LoadElement

Инструкция LoadElement читает элемент массива.

Правило BT-разметки инструкции LoadElement требует, чтобы на вершине стека до выполнения инструкции находились примитивное BT-значение INT^{x1} и ссылочное BT-значение btOref, которое указывает в BT-куче btHeap на BT-объект btObj (рис. 93). Значение этого отображения btObj на особом значении ELEMENT (соответствующего элементам массива) должно совпадать с BT-значением btVal. На вершине стека после выполнения инструкции должно находиться BT-значение btVal, соответствующее элементам массива. Остальные элементы стека и окружения до и после выполнения инструкции должны совпадать.

$$\frac{\begin{array}{c}(_, _, btObj) = btHeap(btOref) \qquad btVal = btObj(ELEMENT) \\ \bullet\ \text{либо } x = S,\ x_1 = BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = S \\ \bullet\ \text{либо } x = X,\ x_1 = BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = D \\ \bullet\ \text{либо } x = D,\ x_1 = BTKindOf_{btHeap}(btOref) = D,\ BTKindOf_{btHeap}(btVal) = D\end{array}}{\begin{array}{l}prog, btHeap \vdash_{bt} LoadElement^{x} : (INT^{x1}::btOref::btStack, btEnv) \to \\ \qquad\qquad\qquad\qquad\qquad\qquad (btVal::btStack, btEnv)\end{array}}$$

Рис. 93. Правило BT-разметки инструкции $LoadElement^{x}$.

Разметки инструкции, BT-значения INT^{x1}, ссылочного BT-значения btOref и BT-значения btVal, соответствующего элементам массива, связаны следующими правилами. Во-первых, разметки BT-значения INT^{x1} и ссылочного BT-значения btOref должны совпадать. Во-вторых, в зависимости от разметок BT-значения INT^{x1} и BT-значения btVal инструкция должна быть размечена:

- либо S, если BT-значения размечены S;
- либо X, если BT-значение INT^{x1} размечено S, а BT-значение btVal — D;
- либо D, если BT-значения размечены D.

StoreElement

Инструкция StoreElement записывает элемент массива.

Правило BT-разметки инструкции LoadElement требует, чтобы на вершине стека до выполнения инструкции находились BT-значение btVal, примитивное BT-значение INT^{x1} и ссылочное BT-значение btOref, которое указывает

в BT-куче btHeap на BT-объект btObj (рис. 94). Значение этого отображения btObj на особом значении ELEMENT (соответствующего элементам массива) должно совпадать с BT-значением btVal. Остальные элементы стека и окружения до и после выполнения инструкции должны совпадать.

$$\frac{\begin{array}{c}(_, _, btObj) = btHeap(btOref) \qquad btVal = btObj(ELEMENT) \\ \bullet \text{ либо } x = S, x_1 = BTKindOf_{btHeap}(btOref) = S, BTKindOf_{btHeap}(btVal) = S \\ \bullet \text{ либо } x = X, x_1 = BTKindOf_{btHeap}(btOref) = S, BTKindOf_{btHeap}(btVal) = D \\ \bullet \text{ либо } x = D, x_1 = BTKindOf_{btHeap}(btOref) = D, BTKindOf_{btHeap}(btVal) = D\end{array}}{prog, btHeap \vdash_{bt} StoreElement^{x} : (btVal::INT^{x1}::btOref::btStack, btEnv) \rightarrow (btStack, btEnv)}$$

Рис. 94. Правило BT-разметки инструкции $StoreElement^{x}$.

Разметки инструкции, BT-значения INT^{x1}, ссылочного BT-значения btOref и BT-значения btVal, соответствующего элементам массива, связаны следующими правилами. Во-первых, разметки BT-значения INT^{x1} и ссылочного BT-значения btOref должны совпадать. Во-вторых, в зависимости от разметок BT-значения INT^{x1} и BT-значения btVal, соответствующего элементам массива, инструкция должна быть размечена:

- либо S, если BT-значения размечены S;
- либо X, если BT-значение INT^{x1} размечено S, а BT-значение btVal — D;
- либо D, если BT-значения размечены D.

Lift

Инструкция Lift вводится в язык на этапе BT-анализа. В размеченной программе она «преобразует» S-данные в D-данные, но разрешается преобразовывать только примитивные данные — данные типа INT или типа FLOAT.

Правило BT-разметки инструкции Lift требует, чтобы на вершине стека до выполнения инструкции должно находиться примитивное BT-значение с разметкой S, а на вершине стека после выполнения инструкции — примитивное BT-значение с разметкой D с тем же типом (рис. 95). Инструкция Lift всегда имеет разметку X.

$prog,btHeap \vdash_{bt} Lift^X : (primType^S::btStack,btEnv) \rightarrow (primType^D::btStack,btEnv)$

Рис. 95. Правило BT-разметки инструкции $Lift^X$.

3.4. Построение BT-разметки

Приведенные выше правила описывают требования к BT-разметке программы. Для нахождения BT-разметки, удовлетворяющей указанным требованиям, можно использовать различные подходы.

Следует отметить, что построение разметки может быть связано с изменением исходной программы.

3.4.1. Моновариантная BT-разметка

Одним из основных подходов к нахождению разметки является моновариантный подход, когда BT-анализ не изменяет исходную программу. В этом случае необходимо только построить BT-кучу, разметку аргументов и результатов методов и разметку инструкций (и при необходимости добавить инструкции Lift).

В этом случае по программе и описанным правилам строится конечная система уравнений на BT-разметку. Неизвестные в системе описывают

- разметку инструкций, область значений — BT-виды S, D или X;
- разметку аргументов и результатов методов, локальных переменных и элементов на стеке примитивного типа, область значений — BT-виды S или D;
- разметку аргументов и результатов методов, локальных переменных и элементов на стеке ссылочного типа, область значений — BT-объекты.

В программе конечно число инструкций создания объектов или массивов, поэтому конечно число BT-объектов в BT-куче. И следовательно, конечно число возможных BT-куч.

В результате имеем конечную систему уравнений и конечную область значений переменных. Для такой ситуации можно легко адаптировать известные методы, применяемые, например, для типизации или анализа времен свя-

зывания. Например, метод constraint solving.

3.4.2. Поливариантная BT-разметка

Но наибольшую эффективность метод частичных вычислений показывает, если он обладает поливариантностью по методам и коду методов. Т.е. обладает возможностью изменять программы в процессе анализа времен связывания.

Для нахождения разметки с одновременным изменением программы можно использовать модифицированый метод абстрактной интерпретации. Такой метод используется в реализованном экспериментальном специализаторе CILPE.

Метод основан на выполнении правил анализа времен связывания аналогично выполнению программы. Но в случае разветвления тела метода или вызова виртуального метода анализ продолжается по всем возможным путям выполнения программы.

Если построенные по разным путям BT-состояния совпадают, то данные ветви анализа объединяются в данной точке. Например, если встретилась инструкция условного перехода, то независимо анализируются оба пути вычисления. А в точке схождения путей проверяется равенство BT-состояний. Если они совпадают, то в дальнейшем строится одна трасса анализа, общая для обоих путей вычислений.

Данный метод позволяет найти наилучшую локальную BT-разметку. Но возможно бесконечное построение разметки для всей программы.

Чтобы нахождение разметки не происходило бесконечно долго, необходимы механизмы для завершения процесса абстрактной интерпретации и построения конечной размеченной программы. Для этого можно использовать различные эвристики.

Самой простой является счетчик: не допускать размножения данного метода или кода метода более чем N раз (где N фиксировано). Как показано на примерах применения специализатора CILPE в приложении 1, даже использо-

вание такой эвристики позволяет получить желаемый эффект от специализации программ.

Следует отметить, что существует много, а за счет поливариантности иногда и бесконечно много, корректных разметок программы. В некоторых случаях из них нельзя выбрать наилучшую разметку. Поэтому для нахождения разметок можно использовать различные методы. Например, в одних ситуациях лучше использовать более быстрый метод (например, при специализации во время исполнения программы), а в других — более долгий (например, при специализации в интегрированной среде разработки, где время анализа менее критично и завершаемость процесса может контролировать программист). Однако подробный анализ достоинств и недостатков различных методов поиска разметки заслуживает отдельного исследования и находится за рамками данной работы.

3.5. Выводы

В главе описаны BT-разметка программы и правила, которым должна удовлетворять эта разметка. Поливариантный анализ времен связывания [6,8,10] должен строить BT-разметку, удовлетворяющую описанным правилам. По этой разметке генератор остаточной программы [4,11] может построить остаточную программу.

На основе этих правил построен поливариантный анализ времен связывания [6,8,10], использующийся в специализаторе CILPE [3,5,7,9,28] для языка CIL платформы Microsoft .NET [46,48].

Глава 4. Генерация остаточной программы

4.1. Введение

В данной главе формально описывается генератор остаточной программы [4,11] для языка SOOL [13], который используется в специализаторе CILPE. Генератор принимает на вход размеченную программу [6,8,10] и значения статических аргументов и строит остаточную программу.

4.2. Описание генератора 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 [6,8,10] 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 [6,8,10], правила которого, как и правила типизации, задают условия корректной разметки, правила генератора остаточной программы, как и правила интерпретации, однозначно предписывают необходимые вычисления.

4.2.1. Преобразование размеченной программы

На вход генератора остаточной программы поступают размеченная про-

грамма 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. 96). 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)

   В дальнейшем будем использовать только измененную BT-программу.

```
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. 96. Преобразование BT-программы.

4.2.2. Состояние

Генератор остаточной программы производит частичные вычисления, поэтому его состояние близко к состоянию интерпретатора: оно состоит из состояния интерпретатора и отображения Pointer2Var (рис. 97).

```
RPG-State = (State, Pointer2Var)
Value' = Value | DYNVALUE
Pointer2Var = Pointer → TypedVar
Pointer = Var | (Address, Field) | (Address, Integer)
TypedVar = (Type, Var)
```

Рис. 97. Состояние генератора остаточной программы.

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.

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 var^{type}.

Для создания и использования Pointer2Var будем использовать вспомогательные функции (рис. 98).

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.

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.

```
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. 98. Вспомогательные функции для работы с Pointer2Var.

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

4.2.3. Остаточная программа

При генерации остаточной программы последовательно строятся различные части программы. Для независимого построения частей программы тело метода будет немного изменено (рис. 99).

```
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. 99. Тело генерируемого метода.

Во-первых, добавлена инструкция LABEL str (n, rpgState), описывающая метку и состояние генератора остаточной программы. Метка будет использоваться в инструкциях условного и безусловного переходов 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 — целым числом 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}.

При генерации тела метода необходимо проверять, встречалось ли данное состояние ранее. Это производится с помощью функции FindState (рис. 100). Если состояние ранее не встречалось, то возвращается NOTHING, а если встречалось — возвращается тело метода и инструкция безусловного перехода.

При добавлении новых инструкций также необходимо запомнить состояние. Для этого используется функция AddInstrs.

После завершения генерации, по [RPG-Instrution] восстанавливается те-

ло 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. 100. Вспомогательные функции для генерации тела метода.

4.3. *Обработка программы*

Обработка размеченной программы, как и в случае интерпретатора, начинается с обработки начальной последовательности инструкций btInstrs (рис. 101). 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.

$btInstrs = [NewObject^D\ MAIN, CallMethod^X\ Main, Leave^X]$
$rgpState = ((sArg, [], []), [])$
$btProg \vdash_{rpg} (btInstrs, 0) : (rpgState, []) \Rightarrow (_, mthds_2)$
$btProg' = btProg$
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. 101. Правило обработки программы.

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.

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. 102), 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. 102. Вспомогательная функция для обработки методов.

4.4. *Обработка метода*

Обработка метода генератором остаточной программы начинается так же, как и интерпретатором: созданием начального состояния (рис. 103). 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 перемен-

ные, построенные функцией $MkNewVars_{btProg}$.

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

$$\frac{\begin{array}{l}(mthd, _, _, _, (varDecs, btInstrs)) = mthdDef \\ \\ ptr2var_1 = MkNewVars_{btProg}(arg_1, heap_1) \\ rpgState_1 = ((arg_1, [], heap_1), ptr2var_1) \\ rpgInstrs_1 = Stack2Vars(arg_1, heap_1, ptr2var_1) \\ \\ rpgState_2 = AddVars(varDecs, rpgState_1) \\ \\ prog \vdash_{rpg} (btInstrs, 0) : (rpgState_2, rpgInstrs_1) \Rightarrow (rpgInstrs_3, mthds) \\ \\ rpgInstrs_4 = AddBeforeLeave(rpgInstrs_3, Vars2Stack(arg_1, heap_1, ptr2var_1)) \\ \\ mthd' = MakeName(mthd, (arg_1, heap_1)) \\ tArg = [type \mid (type, D) \leftarrow btArg]{+}{+}GetTypes_{btProg}(arg_1, heap_1) \\ tRes = [type \mid (type, D) \leftarrow btRes]{+}{+}GetTypes_{btProg}(arg_1, heap_1) \\ mthdBody' = MkMethodBody(rpgInstrs_4) \\ mthdDef' = (mthd', tArg, tRes, mthdBody')\end{array}}{btProg \vdash_{rpg} mthdDef : (arg_1, heap_1) \Rightarrow (mthdDef', mthds)}$$

Рис. 103. Правило обработки метода.

По завершению обработки тело метода модифицируется (вызов функции AddBeforeLeave, рис. 104) — 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. 104. 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.

4.5. 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. 105).

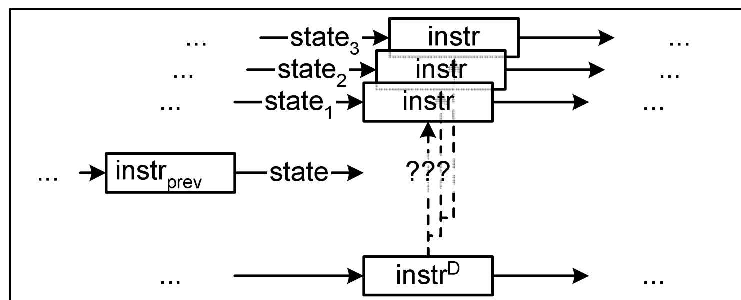


Рис. 105. Процесс генерации остаточной программы: D-инструкция.

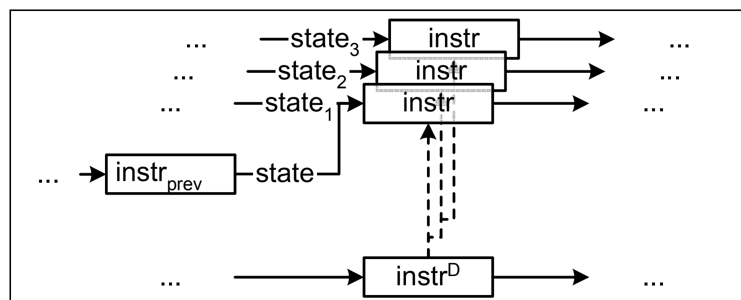


Рис. 106. Процесс генерации остаточной программы: состояние повторилось.

Если такая ситуация встречалась раньше, то генерируется переход на

инструкцию, которая была сгенерирована в аналогичной предыдущей ситуации (рис. 106).

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. 107).

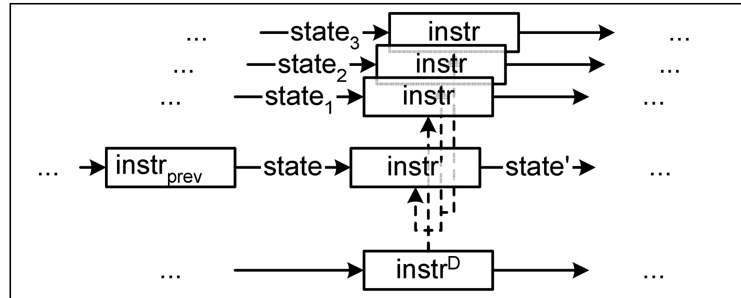


Рис. 107. Процесс генерации остаточной программы: новое состояние.

4.5.1. 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. 108).

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

Рис. 108. 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.

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

Рис. 109. Обработка последовательности инструкций, инструкция Leave^X.

1. Если n-ная инструкция в списке instrs — это инструкция Leave^X, то генерируется инструкция Leave (рис. 109).
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. 110).

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

Рис. 110. 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. 111). Это значение должно быть целым числом. Если оно равно 0, то выполнение продолжается для (n+1)-ой инструкции, иначе — для m-ной.

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

Рис. 111. 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. 112).

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

Рис. 112. 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 инструкции, и далее, начиная с нового состояния, продолжается обработка последовательности инструкций для (n+1)-ой инструкции (рис. 113).

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

Рис. 113. Обработка последовательности инструкций.

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. 114. Обработка последовательности инструкций, инструкция CallMethod^X mthd, INLINE.

4.5.2. 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. 114, рис. 115):

```
RemoveVars :: ([VarDec], RPG-State) → RPG-State
RemoveVars (varDecs, (stack1, env1, heap1), ptr2var1)) =
                                        ((stack1, env2, heap1), ptr2var2)
   where  env2 = filter p env1
          ptr2var2 = filter p env1 ptr2var1
          p (var↦_) = var `notelem` (map fst varDecs))

ReplaceLeave :: [RPG-Instruction] → ([RPG-Instruction], [(String, RPG-State)])
ReplaceLeave [] = ([], [])
ReplaceLeave (Label str rpgState)::(Leave)::rpgInstrs1 = (rpgInstrs3, rgpStates3)
   where  rpgInstrs3 = (Label str rpgState)::(Goto str')::rpgInstrs2
          rgpStates3 = (str', rpgState)::rgpStates2
          (rpgInstrs2, rgpStates2) = ReplaceLeave rpgInstrs1
          str' — новая метка
```

Рис. 115. Вспомогательные функции для обработки последовательности инструкций, инструкция CallMethod^X mthd, INLINE.

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).

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.

Поэтому для обработки NOINLINE инструкции CallMethod^{X} mthd необходимо (рис. 116):

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, соответствующих 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.

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

Рис. 116. Обработка последовательности инструкций, инструкция CallMethod^{X} mthd, NOINLINE.

4.6. Обработка инструкций

Для описания правил обработки отдельных инструкций используются сокращенные правила (рис. 117).

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

Рис. 117. Правило обработки инструкции.

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

4.6.1. S-инструкции

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

Такие инструкции зависят только от S-данных, поэтому эти данные бу-

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

Обработка S-инструкции заключается в интерпретации этой инструкции (рис. 118).

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

Рис. 118. Правило обработки S-инструкций.

4.6.2. 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. 119).

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

Рис. 119. 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.

4.6.3. 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 создаваемый 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. 120, рис. 121). А в последствии заменять инструкции доступа к этим полям или элементам на инструкции доступа к локальным переменным.

oref — новый адрес
obj = [fld↦value | (fld, type) ← $GetFieldDecs_{btProg}$(class),
value = if btFld(fld) == S then DefaultValue(type) else DYNVALUE]
— новый объект класса class
heap' = heap[oref↦(class, obj)]
ptr2var' = ptr2var[(oref, fld)↦ var_{fld}^{type} | (fld, type) ← $GetFieldDecs_{btProg}$(class),
btFld(fld) = D, var_{fld}^{type} — новая переменная типа type]

---

btProg $\vdash_{rpg}$ $NewObject^X$ $class^{btFld}$: ((stack, env, heap), ptr2var) →
(((oref::stack, env, heap'), ptr2var'), [])

Рис. 120. Правило специализации инструкции $NewObject^X$ $class^{btFld}$.

aref — новый адрес
arr = [i↦DYNVALUE | i ← [0..n-1]] — новый массив
heap' = heap[aref↦(type[], (length, arr))]
ptr2var' = ptr2var[(aref, i)↦ var_i^{type} | i ← [0..n-1],
var_i^{type} — новая переменная типа type]

---

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

Рис. 121. Правило специализации инструкции $NewArray^X$ type.

При специализации инструкции $NewObject^X$ генератор остаточной про-

граммы, во-первых, выполняет эту инструкцию: создает объект obj класса class. 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 программы, как и в случае инструкции $NewObject^X$, во-первых, выполняет эту инструкцию и создает массив array с элементами типа type. 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.

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

Рис. 122. Правило специализации инструкции $LoadVar^X$ var.

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

Рис. 123. Правило специализации инструкции $StoreVar^X$ var.

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. 122, рис. 123). Эта переменная определяется по переменной исходной

программы и отображению ptr2var.

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. 124, рис. 125).

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

Рис. 124. Правило специализации инструкции LoadField^X fld.

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

Рис. 125. 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. 126, рис. 127).

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

Рис. 126. Правило специализации инструкции LoadElement^X.

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

Рис. 127. Правило специализации инструкции StoreElement^X.

Инструкция Lifting^X

Инструкция Lifting переносит примитивное данное из состояния генера-

тора 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. 128).

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

Рис. 128. Правило специализации инструкции Lifting^{X}.

4.7. Выводы

В главе 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 [4,11] для языка SOOL [13], который используется в специализаторе CILPE [3,5,7,9,28] для платформы Microsoft .NET [46,48].

Глава 5. Доказательство корректности

5.1. Введение

В данной главе доказывается корректность генератора остаточных программ на языке SOOL [13]. В доказательстве используются формальные описания корректной разметки программы [6,8,10] и генератора остаточной программы [4,11].

5.2. Структура доказательства 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.

5.2.1. 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 состояния 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, совпадает 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 элемен-

та 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. 129):

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 значение 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 переменные. 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. 129. Соединение состояний генератора остаточной программы и интерпретатора остаточной программы.

5.2.2. 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 требование — 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. 130. Сравнение состояний.

Данное требование позволит сравнивать состояния (рис. 130):

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 счита-

ем одинаковыми, если они отличаются только в частях, относящихся к недостижимым объектам и массивам. В реальных системах недостижимые объекты и массивы удаляются сборщиком мусора.

5.3. База индукции

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

Доказательство будем проводить индукцией по количеству шагов интерпретатора 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.

5.4. 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.

Рассмотрим правило выполнения размеченной программы, правило обработки размеченной программы генератором остаточной программы и правило выполнения остаточной программы (рис. 131).

Во всех случаях создается состояние и применяется правило для одина-

ковых последовательностей инструкций.

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

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

$$\frac{\begin{array}{l}btInstrs = [NewObject^{D}\ MAIN,\ CallMethod^{X}\ Main,\ Leave^{X}] \\ rgpState = ((args_S, [], []), []) \\ btProg \vdash_{rpg} (btInstrs, 0) : (rpgState, []) \Rightarrow (_, mthds_2) \\ \dots\end{array}}{\vdash_{rpg} btProg : args_S \Rightarrow btProg'}$$

$$\frac{\begin{array}{c}rInstrs = [NewObject\ MAIN,\ CallMethod\ Main,\ Leave] \\ rProg \vdash_i (rInstrs, 0) : (arg_D, [], []) \Rightarrow (res_D, [], heap_D)\end{array}}{\vdash_i rProg : arg_D \Rightarrow res_D}$$

Рис. 131. Правила для программы.

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

Так как локальных переменных нет и кучи пусты, то согласно определению функции $JoinState_{btHeap}$ для построения соединенного состояния необходимо соединить $args_S$ и arg_D согласно btArg. Но согласно базе индукции в результате получим arg. Что и требовалось доказать.

5.5. *Шаг индукции: метод и инструкции CallMethod и Leave*

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

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

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

Рис. 132. Правила для инструкции CallMethod^X mthd.

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

$(_, btTypeArg, btTypeRes, (_, btInstrs)) = btMthdDef$

$\forall btType \in btTypeArg$:

- либо (btType — ссылочный BT-тип)
  $(type, btOref) = btType$, $(_, types, _) = btHeap(btOref)$, $type \in types$
- либо (btType — примитивный BT-тип)
  $(type_1, (type_2, btKind)) = btType$, $type_1 = type_2$

$\forall btType \in btTypeRes$:

- либо (btType — ссылочный BT-тип)
  $(type, btOref) = btType$, $(_, types, _) = btHeap(btOref)$, $type \in types$
- либо (btType — примитивный BT-тип)
  $(type_1, (type_2, btKind)) = btType$, $type_1 = type_2$

$fst\ \lambda_{btMthdDec}(0) = btArg = map\ snd\ btTypeArg \qquad btRes = map\ snd\ btTypeRes$

$\forall n\ 0 \leq n < length(instrs) : \quad btProg, btHeap, \lambda_{btMthdDec}, btRes \vdash_{bt} (btInstrs, n)$

$btProg, btHeap, \lambda_{btMthdDec} \vdash_{bt} btMthdDef$

Рис. 133. Правила для метода.

5.5.1. Метод INLINE

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

Так как элемент на вершине стека размечен S (рис. 134) и состояния до шага совпадают, то на вершине стека интерпретатора исходной программы и на вершине стека генератора остаточной программы лежит одно и то же зна-

чение — ссылка на объект. Поэтому в обоих случаях выберем один и тот же метод (рис. 132, рис. 135).

…
- 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. 134. Корректность BT-программы.

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)
$rpgState_2$ = (_, $ptr2var_2$) = AddVars(varDecs, $rpgState_1$)
$rpgInstrs_2$ = concat [[LoadConst DefaultValue(type),
StoreVar $ptr2var_2$(var)] | (var, type) ← varDecs]
prog $\vdash_{rpg}$ (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 $\vdash_{rpg}$ (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 $\vdash_{rpg}$ (btInstrs, n) : ($rpgState_1$, $rpgInstrs_1$) ⇒ ($rpgInstrs_7$, $mthds_7$)

Рис. 135. Обработка последовательности инструкций,
инструкция CallMethod^X mthd, INLINE.

Состояния для обработки вызываемого метода строятся по-разному. Интерпретатор берет часть состояния вызывающего метода — вершину стека, соответствующую аргументам. А генератор — все состояние (рис. 135). 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 (рис. 133). А генератором остаточной программы добавляются инструкции присваивания значений по умолчанию в переменные остаточной программы, которые соответствуют переменным вызываемого метода размеченной программы (рис. 135). А также добавляются переменные в состояние генератора остаточной программы. В результате состояние интерпретатора совпадает с соединенным состоянием (рис. 136).

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.

$$(arg{+}{+}stack, env, heap) = JoinState_{btHeap}\ (btArg{+}{+}btStack, btEnv))\ ((stack_S, env_S, heap_S), ptr2var)\ state_D$$

$$\Downarrow$$

$$(arg, [], heap) = JoinState_{btHeap}\ (btArg, [])\ ((stack_S, env_S, heap_S), ptr2var)\ state_D$$

$$\Downarrow\ (_, btEnv_1) = \lambda_{btMthdDec}(0)$$

$$(arg, env_1, heap) = JoinState_{btHeap}\ (btArg, btEnv_1)\ ((stack_S, env_{S1}, heap_S), ptr2var_1)\ state_{D1}$$

Рис. 136. Проверка совпадения состояний,

инструкция $CallMethod^X$ mthd, INLINE.

5.5.2. Метод NOINLINE

В случае NOINLINE генератор остаточной программы добавляет в остаточную программу вызов специализированного метода (рис. 137).

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

жены следующие ограничения (рис. 134):

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.

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

Рис. 137. Обработка последовательности инструкций, инструкция CallMethod^{X} mthd, NOINLINE.

При обработке метода генератор добавляет в остаточную программу инструкции для передачи дополнительных параметров (рис. 137). Эти параметры соответствуют D-полям S-объектов и D-элементам 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. 138).

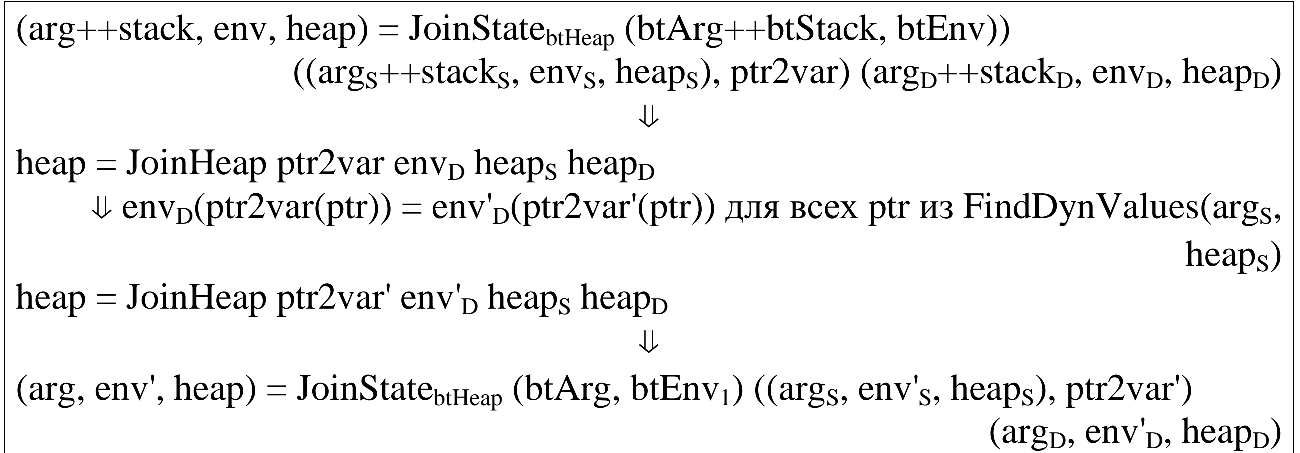
$$
\frac{\begin{array}{l}
(\text{mthd}, _, _, _, (\text{varDecs}, \text{btInstrs})) = \text{mthdDef} \\
\\
\text{ptr2var}' = \text{MkNewVars}_{\text{btProg}}(\text{args}_S, \text{heap}_S) \\
\text{rpgState}_1 = ((\text{args}_S, [], \text{heap}_S), \text{ptr2var}') \\
\text{rpgState}_2 = \text{AddVars}(\text{varDecs}, \text{rpgState}_1) \\
\\
\text{rpgInstrs}_1 = \text{Stack2Vars}(\text{args}_S, \text{heap}_S, \text{ptr2var}') \\
\text{prog} \vdash_{\text{rpg}} (\text{btInstrs}, 0) : (\text{rpgState}_2, \text{rpgInstrs}_1) \Rightarrow (\text{rpgInstrs}_3, \text{mthds}) \\
\text{rpgInstrs}_4 = \text{AddBeforeLeave}(\text{rpgInstrs}_3, \text{Vars2Stack}(\text{args}_S, \text{heap}_S, \text{ptr2var}')) \\
\\
\text{mthd}' = \text{MakeName}(\text{mthd}, (\text{args}_S, \text{heap}_S)) \\
\text{tArg} = [\text{type} \mid (\text{type}, \text{D}) \leftarrow \text{btArg}]{+}{+}\text{GetTypes}_{\text{btProg}}(\text{args}_S, \text{heap}_S) \\
\text{tRes} = [\text{type} \mid (\text{type}, \text{D}) \leftarrow \text{btRes}]{+}{+}\text{GetTypes}_{\text{btProg}}(\text{args}_S, \text{heap}_S) \\
\text{mthdBody}' = \text{MkMethodBody}(\text{rpgInstrs}_4) \\
\text{mthdDef}' = (\text{mthd}', \text{tArg}, \text{tRes}, \text{mthdBody}')
\end{array}}{\text{btProg} \vdash_{\text{rpg}} \text{mthdDef} : (\text{args}_S, \text{heap}_S) \Rightarrow (\text{mthdDef}', \text{mthds})}
$$

Рис. 138. Правило обработки метода.

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. 139).



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

Рис. 139. Проверка совпадения состояний,
инструкция $CallMethod^X$ mthd, NOINLINE.

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

5.6. Шаг индукции: последовательность инструкций

5.6.1. 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. 140) и прекращает обработку данной ветви. Так как состояние полностью совпало, то дальнейшая обработка полностью повторяла бы ту, которая уже была произведена.

$$\frac{JUST\ rpgInstrs_2 = FindState(rpgInstrs_1, n, rpgState)}{btProg \vdash_{rpg} (btInstrs, n) : (rpgState, rpgInstrs_1) \Rightarrow (rpgInstrs_2\ [])}$$

Рис. 140. 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 мес-

та.

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.

5.6.2. 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. 141).

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

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

$$\frac{\begin{array}{c}Goto^S\ m = btInstrs[n] \qquad NOTHING = FindState(rpgInstrs_1, n, rpgState) \\ btProg \vdash_{rpg} (btInstrs, m) : (rpgState, instrs_1) \Rightarrow (instrs_2, mthds_2)\end{array}}{btProg \vdash_{rpg} (btInstrs, n) : (rpgState, instrs_1) \Rightarrow (instrs_2, mthds_2)}$$

Рис. 141. 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.

5.6.3. 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. 142).

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

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

$$\frac{\begin{array}{c} Branch^S\ m = btInstrs[n] \qquad NOTHING = FindState(rpgInstrs_1, n, rpgState) \\ ((val::stack_S, env_S, heap_S), ptr2var) = rpgState, \text{ где } val \text{ — целое число} \\ rpgState' = ((stack_S, env_S, heap_S), ptr2var) \qquad k = \text{if } val{=}{=}0 \text{ then } n{+}1 \text{ else } m \\ btProg \vdash_{rpg} (btInstrs, k) : (rpgState', rpgInstrs_1) \Rightarrow (rpgInstrs_2, mthds_2) \end{array}}{btProg \vdash_{rpg} (btInstrs, n) : (rpgState, rpgInstrs_1) \Rightarrow (rpgInstrs_2, mthds_2)}$$

Рис. 142. 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 вершинах $(val::stack'_1)$ и $(val::stack_S)$ лежит одно и то же 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.

5.6.4. 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. 143).

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.

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

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

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

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

Рис. 143. Правила для последовательности инструкций, инструкция $Branch^D$ m.

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

5.6.5. Остальные инструкции

Правила обработки остальных инструкций в последовательности инструкций указывают, что необходимо применить правила для отдельных инструкций. Состояния передаются и возвращаются без преобразований (рис. 144).

Поэтому очевидно, что если перед выполнением правила для последова-

тельности инструкций состояния совпадали, то и перед выполнением правил для отдельных инструкций они будут совпадать.

$$\frac{\begin{array}{c}\text{prog} \vdash_{i} \text{instrs[n]} : (\text{stack}_1, \text{env}_1, \text{heap}_1) \rightarrow (\text{stack}_2, \text{env}_2, \text{heap}_2) \\ \text{prog} \vdash_{i} (\text{instrs, n+1}) : (\text{stack}_2, \text{env}_2, \text{heap}_2) \Rightarrow (\text{stack}_3, \text{env}_3, \text{heap}_3)\end{array}}{\text{prog} \vdash_{i} (\text{instrs, n}) : (\text{stack}_1, \text{env}_1, \text{heap}_1) \Rightarrow (\text{stack}_3, \text{env}_3, \text{heap}_3)}$$

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

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

Рис. 144. Правила для последовательности инструкций.

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.

5.7. Шаг индукции: инструкции

5.7.1. 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.

$$\text{btProg} \vdash_{i} \text{btInstr}^S : \text{state} \rightarrow \text{state'}$$

$$\text{btProg, btHeap} \vdash_{bt} \text{btInstr}^S : \text{btState} \rightarrow \text{btState'}$$

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

Рис. 145. Правила для S-инструкций.

При обработке S-инструкций генератор остаточной программы выполняет указанную инструкцию, не генерируя инструкций в остаточную программу (рис. 145).

По предположению индукции известно, что state = $\text{JoinState}_{\text{btHeap}}$ 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' = $\text{JoinState}_{\text{btHeap}}$ 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.

5.7.2. 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.

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

Рис. 146. Правила для D-инструкций.

При обработке D-инструкций генератор остаточной программы добавляет указанную инструкцию в остаточную программу, не меняя своего со-

стояния (рис. 146).

По предположению индукции известно, что state = $\text{JoinState}_{\text{btHeap}}$ 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' = $\text{JoinState}_{\text{btHeap}}$ btState' rpgState state'_D.

5.7.3. X-инструкции

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

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.

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

$$\frac{\begin{array}{c}(_, \text{types, btObj}) = \text{btHeap(btOref)} \qquad \text{class} \in \text{types} \\ \text{BTKindOfbtHeap(btOref)} = S\end{array}}{\begin{array}{l}\text{btProg, btHeap} \vdash_{bt} \text{NewObject}^X \text{ class}^{\text{btOref}} : \text{(btStack,btEnv)} \rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{(btOref::btStack, btEnv)}\end{array}}$$

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

Рис. 147. Правила для инструкции NewObject^X class$^{\text{btFld}}$.

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. 147).

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. 148).

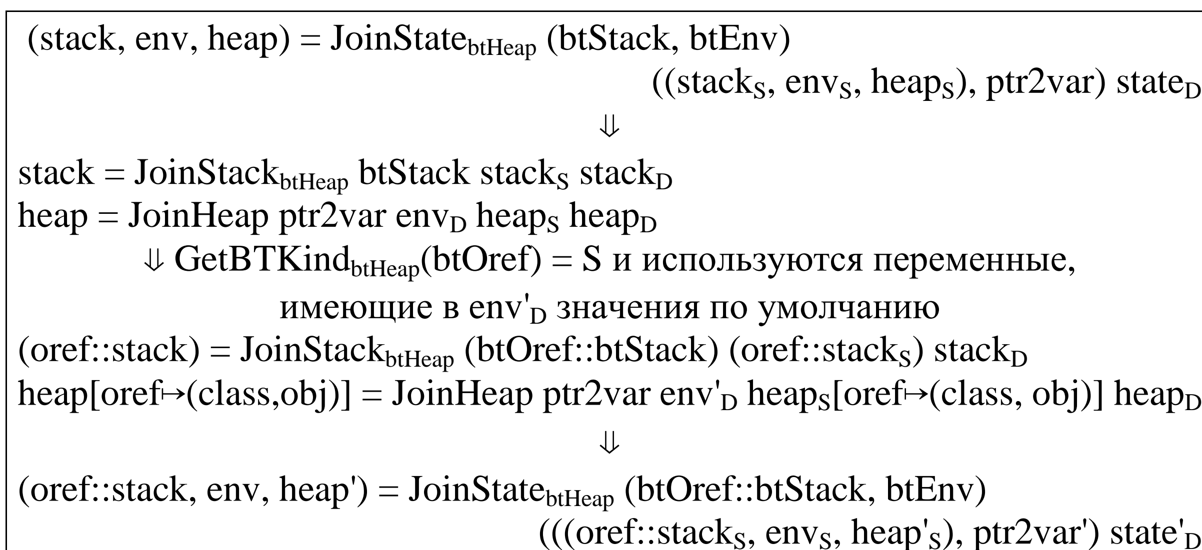


Рис. 148. Проверка совпадения состояний.

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.

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.

В остаточную программу добавляются инструкции, «обнуляющие» значения переменных, которые соответствуют элементам массива (рис. 149).

$\mathrm{TypeOf}_{heap}(\mathrm{len}) = \mathrm{INT}$ arr = [i↦DefaultValue(type) | i ⟵ [0.. len-1]] aref — новый адрес   heap' = heap[aref ↦ (type[], (len, arr))]
btProg $\vdash_i$ NewArray^X type : (len::stack, env, heap) → (aref::stack, env, heap')
(_,types,btObj) = btHeap(btOref)   type[] ∈ types   btVal = btObj(ELEMENT) $\mathrm{BTKindOf}_{btHeap}(\mathrm{btOref}) = \mathrm{S}$ $\mathrm{BTKindOf}_{btHeap}(\mathrm{btVal}) = \mathrm{D}$
btProg, btHeap $\vdash_{bt}$ NewArray^X type : (INT^S::btStack, btEnv) → (btOref::btStack, btEnv)
arr = [i↦DYNVALUE | i ⟵ [0..len-1]] aref — новый адрес $\mathrm{heap'}_S = \mathrm{heap}_S$[aref↦(type[], (len, arr))] ptr2var' = ptr2var[(aref, i)↦ var_i^{type} | i ⟵ [len..n-1], var_i^{type} — новая переменная типа type] instrs = concat [[LoadConst DefaultValue(type), StoreVar ptr2var'(aref, i)] | i ⟵ [len..n-1]]
btProg $\vdash_{rpg}$ NewArray^X type : ((len::stack_S, env_S, heap_S), ptr2var) → (((aref::stack_S, env_S, $\mathrm{heap'}_S$), ptr2var'), instrs)

Рис. 149. Правила для NewArray^X type.

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. 150).

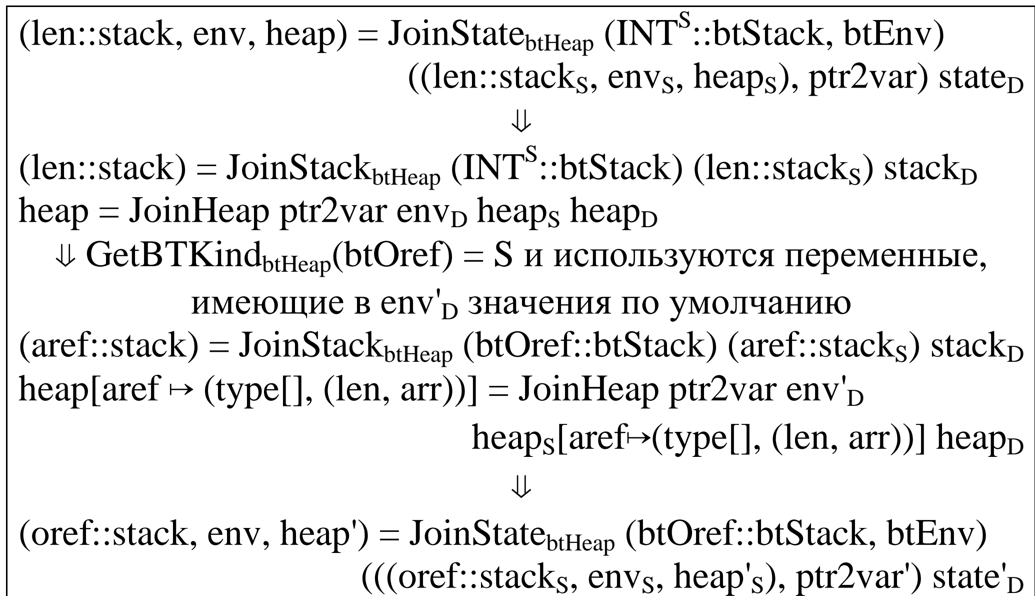


Рис. 150. Проверка совпадения состояний.

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

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. 151).

Правило разметки требует, чтобы значение переменной было D.

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

Рис. 151. Правила для инструкции $LoadVar^X$ var.

$$(stack, env, heap) = JoinState_{btHeap}\ (btStack, btEnv)\ (state_S, ptr2var)\ (stack_D, env_D, heap_D)$$
$$\Downarrow$$
$$env = JoinEnv_{btHeap}\ btEnv\ env_S\ env_D$$
$$stack = JoinStack_{btHeap}\ btStack\ stack_S\ stack_D$$
$$\Downarrow var' = ptr2var(var),\ BTKindOf_{btHeap}(btEnv(var)) = D$$
$$env(var) = env_D(var')$$
$$\Downarrow BTKindOf_{btHeap}(btVal) = D$$
$$(env(var)::stack) = JoinStack_{btHeap}\ (btVal::btStack)\ stack_S\ (env_D(var')::stack_D)$$
$$\Downarrow$$
$$(env(var)::stack, env, heap) = JoinState_{btHeap}\ (btVal::btStack, btEnv)\ (state_S, ptr2var)\ (env_D(var')::stack_D, env_D, heap_D)$$

Рис. 152. Проверка совпадения состояний.

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. 152).

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

Инструкция StoreVar^X var

При обработке инструкции $LoadVar^X$ var генератор остаточной программы генерирует инструкцию записи в переменную StoreVar ptr2var(var) (рис. 153).

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.

$$\frac{\mathrm{TypeOf}_{heap}(\mathrm{val}) <_{btProg} \mathrm{VarType}_{btProg}(\mathrm{var})}{\mathrm{btProg} \vdash_{i} \mathrm{StoreVar}^{X}\ \mathrm{var} : (\mathrm{val::stack}, \mathrm{env}, \mathrm{heap}) \rightarrow (\mathrm{stack}, \mathrm{env}[\mathrm{var} \mapsto \mathrm{val}], \mathrm{heap})}$$

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

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

$$\frac{\mathrm{TypeOf}_{heapD}(\mathrm{val}) <_{rProg} \mathrm{VarType}_{rProg}(\mathrm{var'})}{\mathrm{rProg} \vdash_{i} \mathrm{StoreVar\ var'} : (\mathrm{val::stack}_{D}, \mathrm{env}_{D}, \mathrm{heap}_{D}) \rightarrow (\mathrm{stack}_{D}, \mathrm{env}_{D}[\mathrm{var'} \mapsto \mathrm{val}], \mathrm{heap}_{D})}$$

Рис. 153. Правила для инструкции StoreVar^{X} var.

$$(\mathrm{val::stack}, \mathrm{env}, \mathrm{heap}) = \mathrm{JoinState}_{btHeap}\ (\mathrm{btVal::btStack}, \mathrm{btEnv})\ (\mathrm{state}_{S}, \mathrm{ptr2var})\ (\mathrm{val'::stack}_{D}, \mathrm{env}_{D}, \mathrm{heap}_{D})$$

$$\Downarrow$$

$$(\mathrm{val::stack}) = \mathrm{JoinStack}_{btHeap}\ (\mathrm{btVal::btStack})\ \mathrm{stack}_{S}\ (\mathrm{val'::stack}_{D})$$

$$\mathrm{env} = \mathrm{JoinEnv}_{btHeap}\ \mathrm{btEnv}\ \mathrm{env}_{S}\ \mathrm{env}_{D}$$

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

$$\mathrm{val} = \mathrm{val'}$$

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

$$\mathrm{env}[\mathrm{var} \mapsto \mathrm{val}] = \mathrm{JoinEnv}_{btHeap}\ \mathrm{btEnv'}\ \mathrm{env}_{S}\ \mathrm{env}_{D}[\mathrm{var} \mapsto \mathrm{val}]$$

$$\Downarrow$$

$$(\mathrm{stack}, \mathrm{env}[\mathrm{var} \mapsto \mathrm{val}], \mathrm{heap}) = \mathrm{JoinState}_{btHeap}\ (\mathrm{btStack}, \mathrm{btEnv'})\ (\mathrm{state}_{S}, \mathrm{ptr2var})\ (\mathrm{stack}_{D}, \mathrm{env}_{D}[\mathrm{var} \mapsto \mathrm{val}], \mathrm{heap}_{D})$$

Рис. 154. Проверка совпадения состояний.

Так как var' = ptr2var(var), то новое значение переменной var в env[var↦val] равно новому значению var' в env_{D}[var'↦val]. Следовательно, после шага соединенное окружение будет совпадать с окружением интерпретатора размеченной программы (рис. 154).

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

Инструкция LoadField^{X} fld

Инструкция LoadField^{X} fld обрабатывается аналогично LoadVar^{X} var — в

остаточную программу добавляется инструкция LoadVar var' (рис. 155).

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

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

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

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

Рис. 155. Правила для инструкции LoadField^X fld.

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

$\Downarrow$ согласно правилу создания объекта

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

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

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

$oref = oref'$

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

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

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

$(_, obj) = heap(oref), obj(fld) = env_D(var')$

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

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

$\Downarrow$

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

Рис. 156. Проверка совпадения состояний.

Доказательство проводится в четыре этапа (рис. 156).

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 у 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.

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

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

Инструкция StoreField^X fld

Инструкция StoreField^X fld обрабатывается аналогично StoreVar^X var — в остаточную программу добавляется инструкция StoreVar var' (рис. 157).

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

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

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

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

Рис. 157. Правила для инструкции StoreField^X fld.

Доказательство проводится в четыре этапа (рис. 158).

Во-первых, так как состояния до шага совпадают, то на вершине стеков

лежат одинаковые значения.

$$\begin{array}{l}
\text{BTKindOf}_{\text{btHeap}}(\text{btOref}) = \text{S},\ \text{BTKindOf}_{\text{btHeap}}(\text{btObj}(\text{fld})) = \text{D} \\
\qquad\qquad \Downarrow \text{ согласно правилу создания объекта} \\
(_,\ \text{obj}_\text{S}) = \text{heap}_\text{S}(\text{oref}),\ \text{obj}_\text{S}(\text{fld}) = \text{DYNVALUE} \\
\hline
(\text{val::oref::stack, env, heap}) = \text{JoinState}_{\text{btHeap}}\ (\text{btVal::btOref::btStack, btEnv}) \\
\qquad ((\text{oref'::stack}_\text{S},\ \text{env}_\text{S},\ \text{heap}_\text{S}),\ \text{ptr2var})\ (\text{val'::stack}_\text{D},\ \text{env}_\text{D},\ \text{heap}_\text{D}) \\
\qquad \Downarrow \text{BTKindOf}_{\text{btHeap}}(\text{btVal}) = \text{D},\ \text{BTKindOf}_{\text{btHeap}}(\text{btOref}) = \text{S} \\
\text{val=val'},\ \ \text{oref = oref'} \\
\text{stack} = \text{JoinStack}_{\text{btHeap}}\ \text{btStack stack}_\text{S}\ \text{stack}_\text{D} \\
\text{heap} = \text{JoinHeap ptr2var env}_\text{D}\ \text{heap}_\text{S}\ \text{heap}_\text{D} \\
\quad \Downarrow \text{var'} = \text{ptr2var(oref, fld)},\ (_,\ \text{obj}_\text{S}) = \text{heap}_\text{S}(\text{oref}),\ \text{obj}_\text{S}(\text{fld}) = \text{DYNVALUE} \\
\text{heap}[\text{oref}\mapsto(\text{class, obj}[\text{fld}\mapsto\text{val}])] = \text{JoinHeap ptr2var env}_\text{D}[\text{var'}\mapsto\text{val}]\ \text{heap}_\text{S}\ \text{heap}_\text{D} \\
\qquad\qquad \Downarrow \\
(\text{stack, env, heap'}) = \text{JoinState}_{\text{btHeap}}\ (\text{btStack, btEnv}) \\
\qquad (((\text{stack}_\text{S},\ \text{env}_\text{S},\ \text{heap}_\text{S}),\ \text{ptr2var})\ (\text{stack}_\text{D},\ \text{env}_\text{D}[\text{var'}\mapsto\text{val}],\ \text{heap}_\text{D})
\end{array}$$

Рис. 158. Проверка совпадения состояний.

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 локальной 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 все равно будет совпадать с соединенной кучей.

В-четвертых, так как до шага стеки совпадали, то после выполнения шага (удаления одинаковых значений из стеков), стеки все равно будут совпадать. Как показано выше, кучи тоже будут совпадать. Следовательно, состояние интерпретатора размеченной программы совпадает с соединенным со-

стоянием. Что и требовалось доказать.

Инструкция LoadElement^X

Инструкция LoadElement^X обрабатывается аналогично LoadField^X fld — в остаточную программу добавляется инструкция LoadVar var' (рис. 159).

Доказательство проводится аналогично LoadField^X fld (рис. 160).

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

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

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

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

Рис. 159. Правила для инструкции LoadElement^X.

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

⇓ согласно правилу создания массива

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

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

⇓

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

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

$i = i'$, $aref = aref'$

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

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

⇓ $var' = ptr2var(aref, i)$, $(_, (_, arr_S)) = heap_S(aref)$, $arr_S(i) = \mathrm{DYNVALUE}$

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

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

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

⇓

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

Рис. 160. 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 массива равно значению локальной переменной.

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

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

Инструкция StoreElement^X

$$\frac{\begin{array}{l}\mathrm{TypeOf}_{heap}(i) = \mathrm{INT} \qquad \mathrm{TypeOf}_{heap}(aref) <_{btProg} \mathrm{someType}[] \qquad aref \neq \mathrm{NULL} \\ (\mathrm{type}[], (\mathrm{length}, arr)) = heap(aref) \qquad \mathrm{TypeOf}_{heap}(val) <_{btProg} \mathrm{type} \\ 0 \leq n < \mathrm{length} \qquad heap' = heap[aref \mapsto (\mathrm{type}[], (\mathrm{length}, arr[i \mapsto val]))]\end{array}}{btProg \vdash_i \mathrm{StoreElement}^X : (val::i::aref::stack, env, heap) \rightarrow (stack, env, heap')}$$

$$\frac{\begin{array}{c}(_, _, btObj) = btHeap(btOref) \qquad btVal = btObj(\mathrm{ELEMENT}) \\ \mathrm{BTKindOf}_{btHeap}(btOref) = S \qquad \mathrm{BTKindOf}_{btHeap}(btVal) = D\end{array}}{\begin{array}{l}btProg, btHeap \vdash_{bt} \mathrm{StoreElement}^X : (btVal::\mathrm{INT}^S::btOref::btStack, btEnv) \rightarrow \\ \hfill (btStack, btEnv)\end{array}}$$

$$\begin{array}{l}btProg \vdash_{rpg} \mathrm{StoreElement}^X : ((i::aref::stack_S, env_S, heap_S), ptr2var) \rightarrow \\ \hfill (((stack_S, env_S, heap_S), ptr2var), [\mathrm{StoreVar}\ ptr2var(aref, i)])\end{array}$$

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

Рис. 161. Правила для инструкции StoreElement^X.

Инструкция StoreElement^X обрабатывается аналогично StoreField^X fld — в остаточную программу добавляется инструкция StoreVar var' (рис. 161).

Доказательство проводится аналогично StoreField^X fld (рис. 162).

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.

$$BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btObj(ELEMENT)) = D$$
$$\Downarrow \text{ согласно правилу создания массива}$$
$$(_, (_, arr_S)) = heap_S(aref),\ arr_S(i) = DYNVALUE$$

$$(val::i::aref::stack, env, heap) = JoinState_{btHeap}$$
$$(btVal::INT^S::btOref::btStack, btEnv)\ ((i'::aref'::stack_S, env_S, heap_S), ptr2var)$$
$$(val'::stack_D, env_D, heap_D)$$
$$\Downarrow BTKindOf_{btHeap}(btVal) = D,\ BTKindOf_{btHeap}(btOref) = S$$
$$val=val',\quad i=i',\quad aref = aref'$$
$$stack = JoinStack_{btHeap}\ btStack\ stack_S\ stack_D$$
$$heap = JoinHeap\ ptr2var\ env_D\ heap_S\ heap_D$$
$$\Downarrow var' = ptr2var(oref, fld)\ ,\ (_, (_, arr_S)) = heap_S(aref),\ arr_S(i) = DYNVALUE$$
$$heap[aref \mapsto (type[], (length, arr[i \mapsto val]))] = JoinHeap\ ptr2var\ env_D[var' \mapsto val]$$
$$heap_S\ heap_D$$
$$\Downarrow BTKindOf_{btHeap}(btVal) = D$$
$$(stack, env, heap[aref \mapsto (type[], (length, arr[i \mapsto val]))]) = JoinState_{btHeap}$$
$$(btStack, btEnv)\ (((stack_S, env_S, heap_S), ptr2var)\ (stack_D, env_D[var' \mapsto val], heap_D)$$

Рис. 162. Проверка совпадения состояний.

В-третьих, до шага куча интерпретатора совпадает с соединенной кучей, поэтому значения элементов массивов совпадают. Но так как элементы массива размечены D, то значение элемента в соединенном состоянии копируются 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 про-

граммы, то куча интерпретатора все равно будет совпадать с соединенной кучей.

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

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. 163).

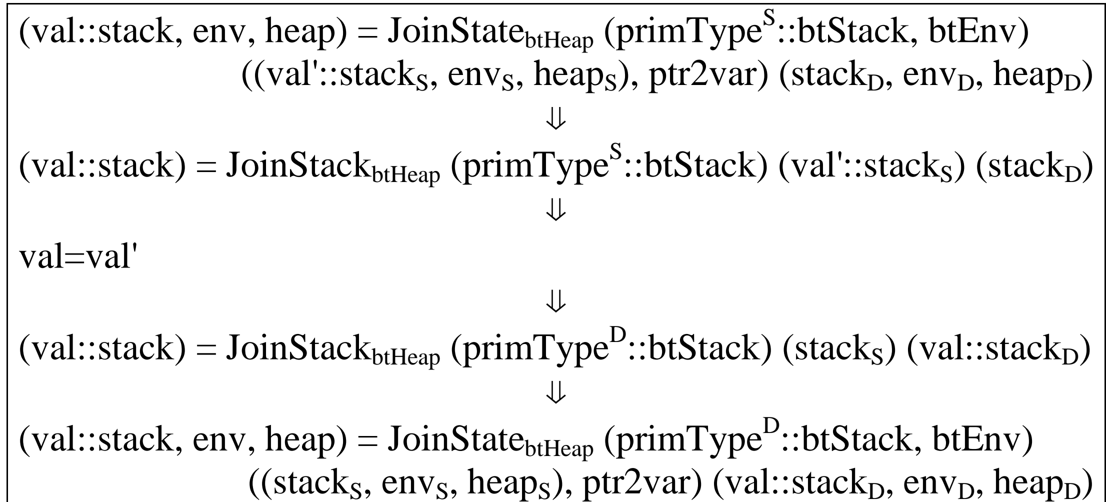
btProg ⊢i Lift^X : (val::stack, env, heap) → (val::stack, env, heap)
btProg,btHeap ⊢bt Lift^X : (primType^S::btStack,btEnv) → (primType^D::btStack,btEnv)
btProg ⊢rpg Lift^X : ((val::stack$_S$, env$_S$, heap$_S$), ptr2var) → (((stack$_S$, env$_S$, heap^S), ptr2var), [LoadConst val])
rProg ⊢i LoadConst const : (stack$_D$,env$_D$,heap$_D$) → (const::stack$_D$,env$_D$,heap$_D$)

Рис. 163. Правила для инструкции Lift^X.

Заметим, что состояния до шага совпадают. Поэтому значение на вершине стека интерпретатора размеченной программы совпадает со значением на вершине стека генератора остаточной программы (рис. 164).

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 вер-

шину D-стека, то соединенный стек не изменится.



(val::stack, env, heap) = $\text{JoinState}_{\text{btHeap}}$ (primType^{S}::btStack, btEnv)
((val'::stack_S, env_S, heap_S), ptr2var) (stack_D, env_D, heap_D)

⇓

(val::stack) = $\text{JoinStack}_{\text{btHeap}}$ (primType^{S}::btStack) (val'::stack_S) (stack_D)

⇓

val=val'

⇓

(val::stack) = $\text{JoinStack}_{\text{btHeap}}$ (primType^{D}::btStack) (stack_S) (val::stack_D)

⇓

(val::stack, env, heap) = $\text{JoinState}_{\text{btHeap}}$ (primType^{D}::btStack, btEnv)
((stack_S, env_S, heap_S), ptr2var) (val::stack_D, env_D, heap_D)

Рис. 164. Проверка для инструкции Lift^{X}.

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

5.8. 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 про-

граммы. 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.

5.9. Выводы

В данной главе 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.

Заключение

В настоящей работе изложен метод частичных вычислений для объектно-ориентированных языков, таких как C# и Java. Для формального описания метода используется модельный объектно-ориентированный язык SOOL. Описаны оба этапа метода частичных вычислений: анализ времен связывания и генерация остаточной программы. Доказана корректность предложенного метода.

Основные результаты работы:

- На основе проведенного исследования существующих методов частичных вычислений для функциональных и объектно-ориентированных программ разработан новый метод специализации программ на объектно-ориентированных языках, впервые обеспечивающий специализацию всех основных конструкций таких языков, как C# и Java, и обладающий поливариантностью. Повышение эффективности программ достигается за счет выполнения части операций во время специализации, удаления части объектов и изменения представления данных. Доказана корректность предложенного метода.
- Разработанный метод частичных вычислений реализован в экспериментальном специализаторе CILPE для промежуточного объектно-ориентированного языка CIL платформы Microsoft .NET и апробирован на модельных задачах. Показана возможность ускорения программ до 10 и более раз.

Литература

1. Ершов А.П. О сущности трансляции // Программирование. — 1977. — № 5. — С. 21-39.
2. Иткин В.Э. О частичном и смешанном выполнении программ // Материалы Всесоюзного семинара «Оптимизация и преобразования программ». — Новосибирск, 1983. — Ч.1. — С. 17-30.
3. 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.
4. 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.
5. Климов Ю.А. Метод частичных вычислений, позволяющий преобразовывать объектно-ориентированные программы в императивные // Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность: Труды Всероссийской научной конференции (21-26 сентября 2009 г., г. Новороссийск). — М.: Изд-во МГУ, 2009. — С. 241-246.
6. 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.
7. 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.
8. 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. Цен-

тральный регион. Москва, 17-18 февраля 2005 г. — М.: Изд-во МГТУ им. Н.Э. Баумана, 2005. — С. 128.

9. Климов Ю.А. Преобразование объектно-ориентированных программ в императивные методом частичных вычислений // Программные продукты и системы. — 2009. — № 2 (86). — С. 71-74.
10. Климов Ю.А. Специализатор CILPE: анализ времен связывания // Препринты ИПМ им.М.В.Келдыша. — 2009. — № 7. — 28 с.
11. Климов Ю.А. Специализатор CILPE: генерация остаточной программы // Препринты ИПМ им.М.В.Келдыша. — 2009. — № 8. — 26 с.
12. Климов Ю.А. Специализатор CILPE: доказательство корректности // Препринты ИПМ им.М.В.Келдыша. — 2009. — № 33. — 32 с.
13. 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.
14. Abadi M., Cardelli L. A Theory of Objects // Springer-Verlag, 1996.
15. Affeldt R., Masuhara H., Sumii E., Yonezawa A. Supporting objects in run-time bytecode specialization // ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation, , PEPM 02. ACM Press, 2002. — P. 50-60.
16. Andersen L.O. Binding-Time Analysis and the Taming of C Pointers // ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 93. — ACM Press, 1993. — P. 47-58.
17. Andersen L.O. C program specialization // Master’s thesis, DIKU, University of Copenhagen. — December 1991. — DIKU Student Project 91-12-17.
18. Andersen L.O. Partial Evaluation of C // Chapter 11 in «Partial Evaluation and Automatic Compiler Generation» by N.D. Jones, C.K. Gomard, P. Sestoft, pp.229-259, C.A.R. Hoare Series, Prentice-Hall, 1993.
19. Andersen L.O. Program Analysis and Specialization for the C Programming Language // PhD thesis, Computer Science Department, University of Copenhagen. — May 1994. — DIKU Technical Report 94/19.
20. Asai K. Binding-time analysis for both static and dynamic expressions // A. Cor-

tesi and G. File (eds.): Static Analysis Symposium. — Lecture Notes in Computer Science, volume 1694/1999. — Springer-Verlag Berlin Heidelberg, 1999. — P. 117-133.

21. Asai K. Can partial evaluation improve the performance of ray tracing // Natural Science Report, Ochanomizu University. — 2002. — Vol. 53. — No. 1.
22. Asai K. Offline Partial Evaluation for Shift and Reset // ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 04. . — ACM Press, 2004. — P. 3-14.
23. Asai K., Masuhara H., Yonezawa A. Partial evaluation of call-by-value lambda-calculus with side-effects // C.Consel (ed.): ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 97. — ACM Press, 1997. — P. 12-21,
24. Beckmann O. Partial Evaluation, Imperative Languages and C // 1996. — URL: http://www.doc.ic.ac.uk/~ob3/Publications/SurveyPaper.ps.gz (дата обращения: 15.09.2009).
25. Bertelsen P. Binding-time analysis for a JVM core language // 1999. — URL: ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/bta-core-jvm.ps.gz (дата обращения: 15.09.2009).
26. Bulyonkov M.A. Polyvariant mixed computation for analyzer programs // Acta Informatica, volume 21/1984, number 5. — Springer-Verlag Berlin Heidelberg, 1984. — P. 473-484.
27. Bulyonkov M.A., Kochetov D.V. Practical Aspects of Specialization of Algol-like Programs // Internaltional Seminar on Partial Evaluation table of contents. — Lecture Notes in Computer Science, volume 1110/1996. — Springer-Verlag Berlin Heidelberg, 1996. — P. 17-32.
28. Chepovskiy A.M., Klimov And.V., Klimov Ark.V., Klimov Yu.A., Mishchenko A.S., Romanenko S.A., Skorobogatov S.Yu. 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. — Springer-Verlag

Berlin Heidelberg, 2003. — P. 171-177.

29. Consel C., Lawall J.L., Le Meur A.-F. A Tour of Tempo A Program Specializer for the C Language // Science of Computer Programming, 2003.
30. Ershov A.P., Itkin V.E. Correctness of Mixed Computation in Algol-Like Programs // Lecture Notes in Computer Science, volume 53/1977. —Springer-Verlag Berlin Heidelberg, 1977. — P. 59-77.
31. Jones N.D., Gomard C.K., Sestoft P. Partial Evaluation and Automatic Compiler Generation // C.A.R. Hoare Series, Prentice-Hall, 1993.
32. Kahn G. Natural semantics // Proceedings of the Symposium on Theoretical Aspects of Computer Sciences. — Lecture Notes in Computer Science, volume 247/1987. — Springer-Verlag Berlin Heidelberg, 1987. — P. 22-39.
33. Kerievsky J. Refactoring to Patterns // Addison Wesley, 2004.
34. Klimov Yu.A. 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. — P. 78-84.
35. Knuth D.E., Morris J.H., Pratt V.R. Fast Pattern Matching in Strings // SIAM Journal on Computing. — 1977. — № 6(2). — P. 323-350.
36. Le Meur A.-F., Lawall J.L., Consel C. Towards Bridging the Gap Berween Programming Languages and Partial Evaluation // Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 02. — ACM Press, 2002. — P. 9-18.
37. Marquard M., Steensgaard B. Partial Evaluation of an Object-Oriented Imperative Language // DIKU, University of Copenhagen. — 1992.
38. Masuhara H., Yonezawa A. Run-time Program Specialization in Java Bytecode // Proceedings of the JSSST SIGOOC 1999 Workshop on Systems for Programming and Applications SPA'99. — 1999.
39. Mogensen T. The application of partial evaluation to ray tracing // Master’s thesis, DIKU, University of Copenhagen. — 1986.
40. Plotkin G.D. An Operational Semantics for CSP // D.Bjorner (ed.): Formal De-

scription of Programming Concepts II. — North-Holland, Amsterdam, 1983. — P.199-223.
41. Rytz B., Gengler M. A Polyvariant Binding Time Analysis // Proceedings of the ACM SIGPLAN Worksop on Partial Evaluation and Semantics-Based Program Manipulation PEPM 92. — ACM Press, 1992
42. Schultz U.P. Object-Oriented Software Engineering Using Partial Evaluation // PhD thesis, University of Rennes I, Rennes, France. — 2000.
43. Schultz U.P., Lawall J.L., Consel C. Automatic program specialization for Java // ACM Transactions on Programming Languages and Systems. — 2003. — № 25 (4). — P. 452-499.
44. Wadler P. Deforestation: Transforming Programs to Eliminate Trees // Proceedings of European Symposium on Programming, ESOP 99. — Lecture Notes in Computer Science, volume 300/1988. — Springer-Verlag Berlin Heidelberg, 1988. — P. 344-358.
45. C# programming language // URL: http://msdn.microsoft.com/vcsharp/ (дата обращения: 15.09.2009).
46. Common Language Infrastructure (CLI) // URL: http://www.ecma-international.org/publications/standards/Ecma-335.htm (дата обращения: 15.09.2009).
47. Java Virtual Machine (JVM) // URL: http://java.sun.com/docs/books/jvms/ (дата обращения: 15.09.2009).
48. Microsoft .NET Framework // URL: http://www.microsoft.com/net/ (дата обращения: 15.09.2009).
49. Ray Tracer Benchmark // URL: http://www.ffconsultancy.com/languages/ray_tracer/benchmark.html (дата обращения: 15.09.2009).
50. Refal programming language // URL: http://www.refal.ru/ (дата обращения: 15.09.2009).
51. Standard ML programming language // URL: http://www.itu.dk/~sestoft/mosml.html (дата обращения: 15.09.2009), URL: http://www.smlnj.org/ (дата обращения: 15.09.2009).

Приложение 1

Введение

На основе разработанного метода был реализован специализатор CILPE для промежуточного объектно-оринетированного языка CIL платформы Microsoft .NET [46,48].

Общая схема специализатора CILPE приведена на рис. 165.

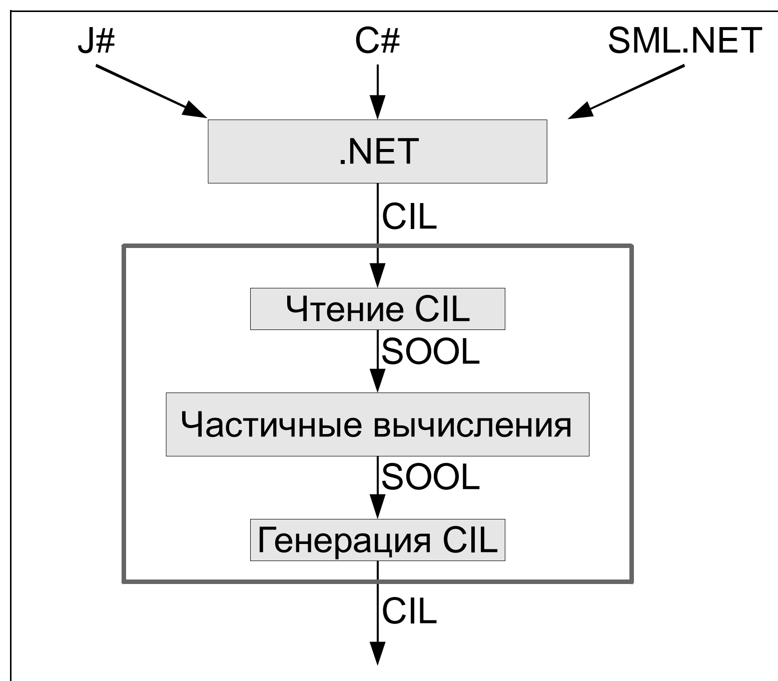


Рис. 165. Схема специализатора CILPE

Исходная программа на языке CIL преобразовывалась во внутреннее представление на языке SOOL [13]. Далее программа на языке SOOL специализировалась согласно разработанному методу. Результат специализации на языке SOOL преобразовывался в программу на языке CIL.

Использование промежуточного языка CIL в качестве входного и выходного языков специализатора CILPE позволяет специализировать программы на всех языках платформы Microsoft .NET, таких как C# [45], J#, SML.NET [51].

В данном приложении приведены примеры специализации на объектно-ориентированном языке C# [45] платформы Microsoft .NET [46,48]. Эти при-

меры компилировались в CIL и пропускались через специализатор CILPE. Для удобства анализа результата специализации и сравнения с исходной программой в данной работе результат специализации приводится на языке C#.

Описание специализируемых примеров

Задание на специализацию формулируется с помощью атрибутов методов, которые вставляются в программу.

Атрибут *[Specialize]* указывает специализируемый метод, причем считается, что все аргументы данного метода динамические, то есть неизвестны. Специализатор строит новую версию заданного метода и заменяет исходный метод на специализированный. Также специализатор может добавить новые методы, которые используются в специализированной версии метода.

Атрибут *[Inline]* указывает специализатору, что при специализации вызовы данного метода могут быть заменены текстами этого метода.

Все методы, не помеченные атрибутом *[Specialize],* переходят в остаточную программу без изменения. Для уменьшения размера примеров не будем приводить такие методы остаточной программы.

Специализация программ, работающих с примитивными данными

Вычисление степени

Рассмотрим классический пример А.П. Ершова [1] — программу возведения в степень (рис. 166). Метод *ToPower* возводит *x* в степень *n* с помощью цикла. Метод *Sample* вычисляет 2.0 в степени 5.

В программе проставлены следующие атрибуты: атрибут *[Specialize]* у метода *Sample* указывает специализируемый метод; атрибут *[Inline]* у метода *ToPower* указывает специализатору, что вызовы данного метода нужно заменить на тело метода.

В результате специализации получаем новую программу, которая отличается от старой программы заменой метода *Sample* (рис. 167). В специализи-

рованном методе *Sampl*e не осталось никаких вычислений: присутствует только оператор возврата значения.

```
class TestClass {
   [Inline] public static double ToPower (double x, int n) {
      double r = 1.0;
      while (n != 0)
         if (n%2 == 0) {
            x = x*x; n = n/2;
         } else {
            r = r*x; n = n-1;
         }
      return r;
   }
   [Specialize] public static double Sample () { return ToPower(2.0, 5); }
}
```

Рис. 166. Вычисление степени: исходная программа.

```
class TestClass {
   public static double Sample () { return 32.0; }
}
```

Рис. 167. Вычисление степени: результат специализации.

Специализация вычисления степени

Рассмотрим немного измененный предыдущий пример (рис. 168): специализируется метод *ToPower20*, вычисляющий двадцатую степень своего аргумента.

В результате специализации получаем программу, состоящую из линейной последовательности операций, вычисляющих двадцатую степень аргумента (рис. 169). Все проверки и операции с известной степенью *n* были выполнены специализатором. А все операции с неизвестным аргументом *x* перешли в остаточную программу.

Следует отметить, что специализатор CILPE является поливариантным по переменным и по коду: значение переменной *r* известно во время специализации до первого присваивания в эту переменную. Поэтому известное значение *1.0* не хранилось в переменной *r* остаточной программы, а непосредст-

венно использовалось при умножении на *x*.

```
class TestClass {
   [Inline] public static double ToPower (double x, int n) {
      double r = 1.0;
      while (n != 0)
         if (n%2 == 0) {
            x = x*x; n = n/2;
         } else {
            r = r*x; n = n-1;
         }
      return r;
   }
   [Specialize]
   public static double ToPower20 (double x) { return ToPower(x, 20); }
}
```

Рис. 168. Специализация вычисления степени: исходная программа.

```
class TestClass {
   public static double ToPower20 (double x) {
      double r;
      x = x*x; x = x*x; r = 1.0*x;
      x = x*x; x = x*x; r = r*x;
      return r;
   }
}
```

Рис. 169. Специализация вычисления степени: результат специализации.

Поиск подстроки в строке

Рассмотрим следующий пример — поиск подстроки в строке. Ожидаем, что специализатор по известной подстроке построит эффективную программу, работающую по алгоритму Крута-Морриса-Пратта [35] за линейное время от длины строки.

В «стандартной» программе сравниваются элементы строки и элементы подстроки, т.е. известные и неизвестные значения. Ожидать эффективной специализации такой программы нельзя. Изменим немного программу, чтобы в основном сравнивались элементы из известной подстроки (рис. 170).

Программа состоит из двух основных методов: *IsSubstring* и *BackTrack*.

Первый метод по очереди сравнивает элементы строки и подстроки. Если на каком-то шаге элементы стали неравными, то вызывается метод *BackTrack*, который для заданной строки ищет максимальный нетривиальный префикс, который являлся бы и суффиксом этой строки. В этом методе строка задается неявно: она является началом длины *j* строки *s*. Метод *IsSubstring* возвращает длину такого префикса.

В результате специализации получаем программу (рис. 171) — конечный автомат, который один раз проходит по строке, проверяя, содержит ли строка-аргумент заданную подстроку.

В результате получаем снижение сложности вычисления примерно в 3.5 раза для примера, приведенного выше.

```
class TestClass {
   public static int D (int x) { return x; }
   [Inline] public static int BackTrack (string s, int j) {
      int k;
      for (k = j-1; k > 0; k--) {
         int i;
         for (i = 0; i < k; i++) if (s[i] != s[j-k+i]) break;
         if (i == k && s[j] != s[k]) break;
      }
      return k;
   }
   [Inline] public static bool IsSubstring (string s, string x) {
      int j = 0;
      for (int i = D(0); i < x.Length; i++) {
         while (x[i] != s[j]) { if (j == 0) continue; j = BackTrack(s, j); }
         j++;
         if (j == s.Length) return true;
      }
      return false;
   }
   [Specialize] public static bool Test (string d) {
      return IsSubstring("aaaab", d);
   }
}
```

Рис. 170. Поиск подстроки в строке: исходная программа.

```
class TestClass {
   public static int D_0 () { return 0; }
   public static bool Test (string x) {
         int i = D_0();
   L0:  if (i >= x.Length) return false;
         if (x[i++] != 'a') goto L0;
         if (i >= x.Length) return false;
         if (x[i++] != 'a') goto L0;
         if (i >= x.Length) return false;
         if (x[i++] != 'a') goto L0;
         if (i >= x.Length) return false;
   L1:  if (x[i++] != 'a') goto L0;
         if (i >= x.Length) return false;
         if (x[i] != 'b') goto L1;
         return true;
} }
```

Рис. 171. Поиск подстроки в строке: результат специализации.

Специализация функции Аккермана

В предыдущих примерах все функции были помечены *[Inline]*, что позволяло разворачивать функции во время специализации. Но если глубина рекурсии неизвестна во время специализации, то такое разворачивание будет продолжаться бесконечно долго.

Рассмотрим следующий пример — программу, вычисляющую функцию Аккермана (рис. 172). В специализированном методе вызовем функцию Аккермана с известным первым аргументом, равным *3*.

```
class TestClass {
   public static ulong A (ulong x, ulong y) {
      if (x == 0) return y+1;
      else if (y == 0) return A(x-1, 1);
      else return A(x-1, A(x, y-1));
   }
   [Specialize] public static ulong Test (ulong y) { return A(3, y); }
}
```

Рис. 172. Функция Аккермана: исходная программа.

Метод *A* не помечен *[Inline]*, поэтому специализатор не может подставить тело этого метода в точке вызова. Поэтому в результате специализации

получаем (рис. 173) несколько проспециализированных версий данного метода. Раскрытие методов и отсутствие операций с первым, а иногда и со вторым аргументами позволяет вычислять функцию более быстро. Например, ускорение вычисления функции Аккермана *A(3,10)* составило около 5 раз.

```
class TestClass {
  public static ulong A_3 (ulong y) { // x = 3
    return y == 0 ? A_2_1() : A_2(A_3(y-1));
  }
  public static ulong A_2 (ulong y) { // x = 2
    return y == 0 ? A_1_1() : A_1(A_2(y-1));
  }
  public static ulong A_1 (ulong y) { // x = 1
    return y == 0 ? A_0_1() : A_0(A_1(y-1));
  }
  public static ulong A_0 (ulong y) { return y+1; } // x = 0
  public static ulong A_2_1 () { return A_1(A_2_0()); } // x = 2, y = 1
  public static ulong A_2_0 () { return A_1_1(); } // x = 2, y = 0
  public static ulong A_1_1 () { return A_0(A_1_0()); } // x = 1, y = 1
  public static ulong A_1_0 () { return A_0_1(); } // x = 1, y = 0
  public static ulong A_0_1 () { return 2; } // x = 0, y = 1
  public static ulong Test (ulong y) { return A_3(y); }
}
```

Рис. 173. Функция Аккермана: результат специализации.

Отметим, что проспециализированные версии отличаются не только значением известного аргумента (для которого они были сгенерированы), но и набором известных аргументов. Методы *A_1*, *A_2*, *A_3* были получены для известного значения первого аргумента x и неизвестного значения второго аргумента y. А методы *A_0_1*, *A_1_0*, *A_1_1*, *A_2_0*, *A_2_1* — для известных значений и первого, и второго аргументов.

Этот пример показывает, что CILPE — поливариантный по методам специализатор: в зависимости от использования, от известности или неизвестности аргументов получаются специализированные версии методов с разным набором параметров.

Специализация программ, работающих с объектами

Вычисление объектов

```
public class Container {
   public Container prev;
   public Container next;
   public int val;
   [Inline] public Container () { }
}
public class FibTest {
   [Inline] static public Container Create (int n) {
      Container first = new Container();
      Container last = first;
      for (int i = 1; i < n; i++) {
         last.next = new Container();
         last.next.prev = last;
         last = last.next;
      }
      return first;
   }
   [Inline] static public void Calculate (Container con) {
      if (con == null) return;
      con.val = 0;
      con = con.next;
      if (con == null) return;
      con.val = 1;
      con = con.next;
      for ( ; con != null; con = con.next)
         con.val = con.prev.val + con.prev.prev.val;
   }
   [Specialize] static public string Test () {
      Container con = Create(11);
      Calculate(con);
      string res = null;
      for ( ; con != null; con = con.next)
         res = (res == null ? "" : res + " ") + con.val;
      return res;
   }
}
```

Рис. 174. Вычисление объектов: исходная программа.

В предыдущих примерах присутствовали только операции над прими-

тивными данными и строками. Этот пример показывает работу специализатора с полностью известными данными, в том числе и объектами, операции над которыми полностью выполняются во время специализации. В частности, если у метода с атрибутом *[Specialize]* нет аргументов и побочных эффектов, а также возвращаемое значение имеет примитивный тип (*int*, *double*, …) или тип строки (*string*), то этот метод будет заменен методом, просто возвращающим значение.

Часть библиотечных методов обладают побочным эффектом: скажем, печать на экран или, наоборот, чтение данных из файла. Мы ожидаем, что эти операции не будут выполнены во время специализации и перейдут в остаточную программу. Другие же методы, наоборот, не обладают побочным эффектом и могут быть выполнены во время специализации программы.

Если специализируемый метод и программа в целом удовлетворяют следующим условиям, то специализатор полностью его вычислит, оставив в остаточной программе только оператор возврата значения:

1. Метод не имеет аргументов, возвращаемое значение имеет примитивный тип (*int*, *double*, …) или тип строки (*string*).
2. Про все используемые в программе библиотечные методы указано, что они не имеют побочных эффектов.
3. Про все методы указано, что специализатор имеет право их развернуть (указан атрибут *[inline]*).

В качестве примера рассмотрим программу, вычисляющую последовательность чисел Фибоначчи до десятого элемента (рис. 174). Вначале создается двусвязный список из объектов класса Container, потом в эти объекты записываются числа Фибоначчи.

Вычисление чисел Фибоначчи запрограммировано таким неэффективным образом, чтобы показать, что специализатор вычислит все объекты, и в результате получается метод, возвращающий строку из чисел Фибоначчи (рис. 175).

```
public class FibTest {
   static public string Test () { return "0 1 1 2 3 5 8 13 21 34 55"; }
}
```

Рис. 175. Вычисление объектов: результат специализации.

Специализация массива

Массивы широко используются в объектно-ориентированных языках, поэтому должны эффективно поддерживаться специализатором.

Рассмотрим пример вычисления значения многочлена по схеме Горнера (рис. 176). Класс *Polynomial* представляет собой многочлен. Коэффициенты многочлена хранятся в массиве *coef*. Метод *Calc(double x)* вычисляет значение многочлена в точке *x*.

```
class Polynomial {
   double[] coef;
   [Inline] public Polynomial (double[] coef) { this.coef = coef; }
   [Inline] public double Calc (double x) {
      double res = 0;
      for (int i = coef.Length-1; i >= 0; i--) res = res*x+coef[i];
      return res;
   }
}
class TestClass {
   [Specialize] public static double TestS (double x) {
      double[] coef = {1, 2, 3}; // p(x) = 1+2x+3x^2
      Polynomial p = new Polynomial(coef);
      return p.Calc(x);
   }
   [Specialize]
   public static double TestD (double a0, double a1, double a2, double x) {
      double[] coef = {a0, a1, a2}; // p(x) = a0+a1*x+a2*x^2
      Polynomial p = new Polynomial(coef);
      return p.Calc(x);
   }
}
```

Рис. 176. Вычисление значения многочлена: исходная программа.

Атрибутом *[Specialize]* помечены два метода: *TestS* и *TestD*. В методе *TestS* коэффициенты массива заданы явно, а переменная *x* — параметр метода.

В методе *TestD* коэффициенты массива, как и переменная *x*, передаются через параметры.

В результате специализации обоих методов получаем арифметические выражения, вычисляющие значение многочлена (рис. 177). В методе *TestS* остается выражение только от аргумента *x*, а в примере *TestD* — выражение, вычисляющее значения многочлена второй степени через коэффициенты *a0*, *a1* и *a2* и переменную *x*.

```
class TestClass {
   public static double TestS (double x) {
      return ((0*x+3)*x+2)*x+1;
   }
   public static double TestD (double a0, double a1, double a2, double x) {
      return ((0*x+a2)*x+a1)*x+a0;
   }
}
```

Рис. 177. Вычисление значения многочлена: результат специализации.

Объект класса *Polynomial* и массив коэффициентов *coef* известны во время специализации программы и полностью вычисляются специализатором. Так как длина массива в обоих случаях известна, цикл в методе *Calc* полностью разворачивается, и операции над переменной *i* выполняются.

В остаточную программу переходят только операции над параметром *x* в первом случае и операции над параметром x и значениями коэффициентов массива *a0*, *a1* и *a2* во втором случае.

Удаление избыточных объектов

В некоторых случаях объекты создаются для временного хранения или передачи данных. В предыдущих двух примерах создавался один такой «временный» объект.

В следующем примере (рис. 178) в специализируемом методе *Test* в цикле создаются несколько (в данном случае 3) временных объектов, которые используются ниже в теле этого метода.

В результате специализации получаем программу, не содержащую инст-

рукций создания объектов класса *Container* (рис. 179).

Каждый объект класса *Container* обладает тремя полями — *n*, *x* и *y*. Значение поля n будет известно во время специализации, а значения полей *x* и *y* — нет.

```
class Container {
   int n;
   double x, y;
   [Inline] public Container (int n, double x) {
      this.n = n; this.x = x; this.y = this.n*this.x;
   }
   [Inline] public double GetX () { return x; }
   [Inline] public double GetY () { return y; }
}
class TestClass {
   [Specialize] public static double Test (double x) {
      Container[] cs = new Container[3];
      for (int i = 0; i < cs.Length; i++) cs[i] = new Container(i+1, x);
      double sx = 0.0;
      for (int i = 0; i < cs.Length; i++) sx += cs[i].GetX();
      double sy = 0.0;
      for (int i = 0; i < cs.Length; i++) sy += cs[i].GetY();
      return sx/sy;
   }
}
```

Рис. 178. Удаление избыточных объектов: исходная программа.

```
class TestClass {
   public static double Test (double x) {
      container1_x = x; // cs[0] = new Container(1, x);
      container1_y = 1*x;
      container2_x = x; // cs[1] = new Container(2, x);
      container2_y = 2*x;
      container3_x = x; // cs[2] = new Container(3, x);
      container3_y = 3*x;
      double sx = 0.0+container1_x+container2_x+container3_x;
      double sy = 0.0+container1_y+container2_y+container3_y;
      return sx/sy;
   }
}
```

Рис. 179. Удаление избыточных объектов: результат специализации.

Поэтому в остаточной программе вместо заведения объектов класса *Container* заведены локальные переменные для каждого поля для каждого создания объекта: переменные *container1_x* и *container1_y* заведены для объекта, который создается при первом проходе цикла, переменные *container2_x* и *container2_y* — при втором, *container3_x* и *container3_y* — при третьем проходе цикла. А для поля *n* переменные не заводятся, т.к. значения этого поля известны во время специализации и подставлены непосредственно при вычислении переменных container1_y, container2_y и container3_y.

Отметим, что переменные заводятся для каждого *создания* объекта, а не для каждого оператора *new* в исходной программе.

Таким образом, все неизвестные поля известных объектов переходят в локальные переменные. А известные поля обрабатываются аналогично известным локальным переменным — их значения вычисляются и подставляются при вычислении значений неизвестных переменных.

Этот и предыдущие примеры показывают, что когда создание и использование объекта или массива полностью прослеживается, специализатор CILPE может создать и использовать объект или массив во время специализации. В остаточную программу не переходят операции создания и использования объекта. Если поля объекта или элементы массива неизвестны, то они заменяются локальными переменными. А если известны — то аналогично самому объекту вычисляются во время специализации.

Тем самым, специализатор удаляет избыточные объекты и промежуточные данные, что близко к идеям и результатам deforestation [44].

Специализация возведения в степень комплексного числа

Рассмотрим пример возведение в степень (рис. 180). В отличие от рассмотренного ранее примера, в степень возводится комплексное число, а не действительное.

Комплексное число представлено классом Complex с двумя полями: re и im для действительной и мнимой части комплексного числа соответственно.

Метод toPower возводит комплексное число в указанную степень. Отметим, что на каждом проходе цикла в методе toPower создается новый объект в методе Times.

Проспециализируем метод toPower10, возводящий комплексное число в 10 степень. В результате специализации получаем программу, в которой не заводятся промежуточные объекты (рис. 181)!

```
class Complex {
   public readonly double re, im;
   [Inline] public Complex (double re, double im) {
      this.re = re; this.im = im;
   }
   [Inline] public Complex Times (Complex c) {
      double r = re * c.re - im * c.im;
      double i = re * c.im + im * c.re;
      return new Complex(r, i);
   }
}
class TestClass {
   [Inline] public static Complex toPower (Complex x, int n) {
      Complex res = new Complex(1.0, 0.0);
      while (n != 0)
         if (n % 2 == 1) {
            n -= 1;
            res = res.Times(x);
         } else {
            n /= 2;
            x = x.Times(x);
         }
      return new Complex(res.re, res.im);
   }
   [Specialize] public static Complex toPower10 (Complex x) {
      return toPower(x, 10));
   }
}
```

Рис. 180. Возведение комплексного числа в 10 степень: исходная программа.

```
class TestClass {
   public static Complex toPower10 (Complex x) {
      double d1 = x.re*x.re - x.im*x.im;
      double d2 = x.re*x.im + x.im*x.re;
      double d3 = 1.0*d1 - 0.0*d2;
      double d4 = 1.0*d2 + 0.0*d1;
      double d5 = d1*d1 - d2*d2;
      double d6 = d1*d2 + d2*d1;
      double d7 = d5*d5 - d6*d6
      double d8 = d5*d6 + d6*d5;
      double d9 = d3*d7 - d4*d8;
      double d10 = d3*d8 + d4*d7;
      return new Complex(d9, d10);
   }
}
```

Рис. 181. Возведение комплексного числа в 10 степень:

результат специализации.

Специализация возведения в степень комплексного числа при неизвестной степень

Рассмотрим измененный предыдущий пример — специализировать будем метод toPower (рис. 182). То есть все аргументы метода toPower специализатору не известны.

```
…
class TestClass {
   [Specialize] public static Complex toPower (Complex x, int n) {
      Complex res = new Complex(1.0, 0.0);
      while (n != 0)
         if (n % 2 == 1) {
            n -= 1;
            res = res.Times(x);
         } else {
            n /= 2;
            x = x.Times(x);
         }
      return new Complex(res.re, res.im);
   }
}
```

Рис. 182. Возведение комплексного числа в степень: исходная программа.

В результате специализации получаем программу, в которой и в этом случае не заводятся промежуточные объекты (рис. 183)!

```
class TestClass {
   public static Complex ToPower (Complex x, int n){
      double x_re = x.re;
      double x_im = x.im;
      double r_re = 1.0;
      double r_im = 0.0;
      while (n != 0)
         if (n % 2 == 1) {
            n -= 1;
            r_re = r_re*x_re - r_im*r_im;
            r_im = r_re*x_im + r_im*x_re;
         } else {
            n /= 2;
            x_re = x_re *x_re - x_im*x_im;
            x_im = x_re *x_im + x_im*x_re;
         }
      return new Complex(r_re,r_im);
   }
}
```

Рис. 183. Возведение комплексного числа в степень: результат специализации.

Виртуальные методы и арифметические выражения

В рассмотренных выше случаях методы были не виртуальными, а статическими: всегда известно тело вызываемого метода. Специализатор CILPE объектно-ориентированного языка поддерживает виртуальные методы, тела которых определяются по точному типу объекта.

Отметим, что на втором этапе специализации, на этапе генерации остаточной программы, известны типы всех известных объектов, поэтому всегда можно точно определить вызываемый метод. Но во время анализа времен связывания, точный тип неизвестен. Следовательно, неизвестно и тело метода при виртуальном вызове. В этом случае анализ времен связывания должен разобрать все возможные типы значений переменных.

Рассмотрим программу, в которой из объектов конструируется выражение, которое затем вычисляется (рис. 184).

```
abstract class Expr {
   [Inline] public Expr () { }
   [Inline] public abstract double GetValue ();
}
class Plus : Expr {
   Expr x, y;
   [Inline] public Plus (Expr x, Expr y) { this.x = x; this.y = y; }
   [Inline] public override double GetValue () {
      return x.GetValue()+y.GetValue();
   }
}
class Value : Expr {
   double x;
   [Inline] public Value () { this.x = 0; }
   [Inline] public Value (double x) { this.x = x; }
   [Inline] public void SetValue (double x) { this.x = x; }
   [Inline] public override double GetValue () { return x; }
}
class TestClass {
   [Specialize] public static double Test (double x) {
      Value var = new Value();
      Expr expr = var;
      for (int i = 1; i < 3; i++) expr = new Plus(expr, new Value(i));
      var.SetValue(x);
      return expr.GetValue();
}  }
```

Рис. 184. Арифметическое выражение: исходная программа.

```
class TestClass {
   public static double Test (double x) { return (x+1)+2; }
}
```

Рис. 185. Арифметическое выражение: результат специализации.

В программе описан абстрактный класс *Expr*, у которого определен виртуальный метод *GetValue* для вычисления значения выражения. Также определены два подкласса класса *Expr*: для операции сложения и для константы и переменной. Можно определить подклассы и для других арифметических операций, но для примера достаточно этих двух подклассов.

В методе *Test* в цикле создается конфигурация из объектов, соответст-

вующая выражению *(x+1)+2*, потом в объект-переменную *x* записывается значение переменной и выражение вычисляется.

В результате специализации получаем остаточную программу, в которой отсутствуют операции создания объектов, а присутствует формула для вычисления выражения *(x+1)+2* (рис. 185).

Специализация шаблона программирования «посетитель»

Одним из часто используемых подходов к программированию является шаблон «посетитель» (visitor pattern). «Посетители» используются, чтобы избежать разбора случаев в зависимости от типа аргумента.

Пример взят со страницы http://en.wikipedia.org/wiki/Visitors_pattern (рис. 186). В этом примере описывается абстрактный класс *Visitor* — класс, описывающий методы для каждого потомка класса *Visitable*, набор наследников класса *Visitable*, представляющих собой машину, двигатель, кузов и колеса. И два наследника класса *Visitor* — один для печати списка деталей, другой для их подсчета. Тест состоит в вызове для созданного объекта одного из визитеров.

```
abstract class Visitor {
   [Inline] public Visitor () { }
   [Inline] public abstract void visit (Car car);
   [Inline] public abstract void visit (Wheel wheel); }
abstract class Visitable {
   [Inline] public Visitable () { }
   [Inline] public abstract void accept (Visitor visitor); }
class Car : Visitable {
   Visitable[] parts;
   [Inline] public Car () {
      parts = new Visitable[] { new Wheel("front left"), new Wheel("front right"),
                                new Wheel("back left"), new Wheel("back right") }; }
   [Inline] public override void accept (Visitor visitor) {
      visitor.visit(this);
      for (int i = 0; i <parts.Length; i++) parts[i].accept(visitor); } }
class Wheel : Visitable {
   string name;
   [Inline] public Wheel (string name) { this.name = name; }
```

```
  [Inline] public string getName () { return.name; }
  [Inline] public override void accept (Visitor visitor) { visitor.visit(this); } }
class PrintVisitor : Visitor {
  [Inline] public PrintVisitor () {}
  [Inline] public override void visit (Car car) {
    Console.WriteLine("Visiting car"); }
  [Inline] public override void visit (Wheel wheel) {
    Console.WriteLine("Visiting {0} wheel", wheel.getName()); } }
class CountVisitor : Visitor {
  int carCount, wheelCount;
  [Inline] public CountVisitor () { carCount = 0; wheelCount = 0; }
  [Inline] public override void visit (Car car) {
    Console.WriteLine("Visiting {0} car", carCount++); }
  [Inline] public override void visit (Wheel wheel) {
    Console.WriteLine("Visiting {0} wheel", wheelCount++); } }
public class VisitorPatternTest {
  [Inline] static Visitor CreateVisitor (bool p) {
    return p ? (Visitor) new CountVisitor() : new PrintVisitor(); }
  [Inline] static Visitable CreateObject (bool q) {
    return q ? (Visitable) new Car() : new Wheel("one"); }
  [Specialize] static public void Test (bool p, bool q) {
    Visitor vis = CreateVisitor(p);
    Visitable obj = CreateObject(q);
    obj.accept(vis); } }
```

Рис. 186. Шаблон «посетитель»: исходная программа.

В этом примере специализируемый метод ничего не возвращает, но вызывает функцию с побочным эффектом Console.WriteLine, которая выдает строку на экран. Специализатор CILPE все функции с побочными эффектами переносит в остаточную программу. Иначе, в данном примере печать происходила бы не во время исполнения остаточной программы, как это ожидается, а во время специализации.

В результате специализации получаем программу, проводящую проверки аргументов и выводящую результат на печать (рис. 187). В остаточной программе не содержатся создания объектов и вызовы методов.

```
class VisitorPatternTest {
   static public void Test (bool p, bool q) {
      if (p) {
         if (q) {
            Console.WriteLine("Visiting {0} car", 0);
            Console.WriteLine("Visiting {0} wheel", 0);
            Console.WriteLine("Visiting {0} wheel", 1);
            Console.WriteLine("Visiting {0} wheel", 2);
            Console.WriteLine("Visiting {0} wheel", 3);
         } else
            Console.WriteLine("Visiting {0} wheel", 0);
      } else {
         if (q) {
            Console.WriteLine("Visiting car");
            Console.WriteLine("Visiting {0} wheel", "front left");
            Console.WriteLine("Visiting {0} wheel", "front right");
            Console.WriteLine("Visiting {0} wheel", "back left");
            Console.WriteLine("Visiting {0} wheel", "back right");
         } else
            Console.WriteLine("Visiting {0} wheel", "one");
      }
   }
}
```

Рис. 187. Шаблон «посетитель»: результат специализации.

Специализация нераскрываемых методов

В предыдущих примерах методы раскрывались (атрибут *[Inline]*): их тела подставлялись в вызывающий метод. Но во многих случаях это нецелесообразно или невозможно: например, в случае рекурсии такое раскрытие может продолжаться бесконечно долго.

Раскрывать или не раскрывать метод — это алгоритмически неразрешимая задача: невозможно понять, будет ли раскрытие метода продолжаться бесконечно долго. Поэтому программист должен указать те методы, которые могут быть раскрыты атрибутом *[Inline]*, а остальные методы считаются нераскрываемыми.

Если не указан атрибут *[Inline]*, то по исходному методу специализатор сгенерирует один или несколько методов:

1. Если в нераскрываемый метод в качестве аргумента передается известный объект, то вместо передачи объекта в метод в остаточной программе должны передаваться *по ссылке* локальные переменные, созданные для полей этого объекта.
2. Специализатор по значениям известных аргументов метода должен сгенерировать специализированную версию исходного метода. Поэтому в остаточной программе может получиться несколько методов из одного.

В следующем примере (рис. 188) описан класс *Container* с полем *n*. Статический метод *Increase* увеличивает поле *n* первого аргумента *c* на второй аргумент *i* и печатает новое значение этого поля.

В специализируемом методе *Test* создаются два объекта класса *Container*, для каждого них вызывается метод *Increase*, и второй объект возвращается.

```
class Container {
   public int n;
   [Inline] public Container (int n) { this.n = n; }
}
class TestClass {
   public static void Increase (Container c, int i) {
      c.n += i; Console.WriteLine("{0}", c.n);
   }
   [Specialize] public static Container Test (int n) {
      Container cs0 = new Container(n);
      Container cs1 = new Container(0);
      Increase(cs0, 5);
      Increase(cs0, n);
      Increase(cs1, 5);
      return cs1;
   }
}
```

Рис. 188. Нераскрываемые методы: исходная программа.

В результате специализации (рис. 189) получаем программу, содержащую создание только одного объекта класса *Container*. И вместо универсального метода *Inscrease* вызываются его специализированные версии.

Объект, записанный в переменную *cs0*, известен во время специализации, поэтому он полностью вычисляется и не переходит в остаточную программу. Но поле *n* этого объекта неизвестно, поэтому вместо поля для объекта заводится локальная переменная *cs0_n*, которая передается по ссылке в методы *Increase*.

```
class Container {
   public int n;
   public Container () { this.n = 0; } // n = 0
}
class TestClass {
   public static void Increase_n_5 (ref int c_n) { // i = 5
      c_n += 5; Console.WriteLine("{0}", c_n);
   }
   public static void Increase_n (ref int c_n, int i) {
      c_n += i; Console.WriteLine("{0}", c_n);
   }
   public static void Increase_5 (Container c) { // i = 5
      c.n += 5; Console.WriteLine("{0}", c.n);
   }
   public static Container Test (int n) {
      int cs0_n = n;
      Container cs1 = new Container();
      Increase_n_5(ref cs0_n);
      Increase_n(ref cs0_n, n);
      Increase_5(cs1);
      return cs1;
   }
}
```

Рис. 189. Нераскрываемые методы: результат специализации.

Отметим, что объекты класса *Container cs0* и *cs1* в исходной программе хоть и описаны одинаково, но используются по-разному: первый известен и вычисляется специализатором, второй переходит в остаточную программу. Это показывает, что специализатор CILPE обладает поливариантностью по классам.

В исходной программе одна и та же часть кода — метод Increase — используется в различных ситуациях: в одном случае объект некоторого клас-

са используется для временного хранения данных, в другом — с помощью него данные возвращаются из специализированного метода.

В таких ситуациях специализатор генерирует остаточные методы для каждого случая. Это означает, что он обладает поливариантностью по методам.

В приведенном примере создаются три специализируемых версии метода *Increase*:

1. Метод *Increase_n_5* — специализированная версия для известного объекта и известного второго аргумента, равного 5.
2. Метод *Increase_n* — специализированная версия для известного объекта и неизвестного второго аргумента.
3. Метод *Increase_5* — специализированная версия для неизвестного объекта и известного второго аргумента, равного 5.

Специализация программ на функциональных языках

Объектно-ориентированные языки являются основой современных многоязычных платформ MS.NET и Java. Для этих платформ существуют компиляторы различных языков программирования. Поэтому специализатор CILPE может быть применен к программам на языках программирования, реализованных для MS.NET. Примером таких языков являются функциональные языки SML и Refal и объектно-ориентированный язык J#.

Специализация функции map в SML.NET

Рассмотрим пример на языке SML [51], на котором описана функция высшего порядка map — применение функции (первый аргумент) к каждому элементу списка (второй аргумент) (рис. 190). Специализируемая функция *test* вызывает функцию map с известным первым аргументом — функцией увеличения аргумента на *1*.

Перед специализацией исходная программа на SML компилируется в объектно-ориентированный язык CIL — внутренний язык платформы MS.NET

[46,48]. Специализатор обрабатывает этот язык и выдает результат на том же языке CIL. Однако для сопоставления исходной программы и результата специализации результат специализации представлен на языке SML.

В результате специализации получаем специализированную версию функции *map* — функцию *map_1*, которая увеличивает на 1 все элементы списка. Ее эквивалентное представление на SML показано на рис. 191.

```
fun map f []        = []
|   map f (x::xs) = (f x)::(map f xs)

fun test xs = map (fn x => x+1) xs
```

Рис. 190. Функция map на SML: исходная программа.

```
fun map_1 []        = []
|   map_1 (x::xs) = (x+1)::(map_1 xs)

fun test xs = map_1 xs
```

Рис. 191. Функция map на SML: результат специализации.

В рассмотренном примере функция ***(fn** x => x+1)* при компиляции в CIL представляется объектом. Специализатор прослеживает использование этого объекта и полностью вычисляет его во время специализации. В остаточной программе не остается этого объекта, а прибавление единицы явно производится в теле метода *map_1*.

Специализация Рефал-программы

Рассмотрим пример программы на функциональном языке Рефал (рис. 192) [50]. В данном примере функция *ExprInt* разбирает и вычисляет арифметическое выражение. В этом выражении могут присутствовать: символ *'x'*, обозначающий переменную, числа, скобки и знаки сложения и умножения. В функции *ExprInt* разбираются все эти пять случаев, и в зависимости от ситуации выполняются необходимые действия.

В этом примере специализируется функция *Test* с одним аргументом. Она вызывает функцию *ExprInt* с двумя аргументами: результатом работы

функции *Expr* и результатом работы функции *FILTER_INT*. Функция *Expr* возвращает выражение в виде последовательности символов, чисел и правильно расставленных скобок (последовательность символов на Рефале записывается в одинарных кавычках). Функция *FILTER_INT* проверяет, что аргумент является числом.

```
ExprInt eExpr sX = eExpr : {
   'x' = sX;
   sN = sN;
   (eExpr1) = <ExprInt eExpr1 sX>;
   eA'+'eB = <ADD <ExprInt eA sX> <ExprInt eB sX>>;
   eA'*'eB = <MUL <ExprInt eA sX> <ExprInt eB sX>>;
};
Expr = 1 '+x*' ( 2 '+x*' ( 3 '+x*' ( 4 '+x*' ( 5 '+x' ))));
Test sX = <ExprInt <Expr> <FILTER_INT sX>>;
```

Рис. 192. Вычисление выражения на Рефале: исходная программа.

Программа на Рефале сначала компилируется в язык Java. Для работы программы кроме результата компиляции еще требуются библиотеки, которые также реализованы на языке Java. В дальнейшем результат компиляции примера и библиотеки компилируются в язык CIL. С этим результатом и работает специализатор CILPE.

В результате компиляции получаем линейную программу на языке CIL, которая может быть представлена на языке C# следующим образом (рис. 193).

```
class Test {
   public static int Test (int x) {
      return 1+x*(2+x*(3+x*(4+x*(5+x))));
   }
}
```

Рис. 193. Вычисление выражения на Рефале: результат специализации.

Экспериментальные данные

Для измерения ускорения, получаемого в результате специализации, вышеописанные примеры, а также некоторые другие, запускались до и после специализации, и сравнивалось время выполнения программы.

В таб. 6 приведены результаты экспериментов: ускорение, получаемое в результате специализации, время, затрачиваемое на специализацию, и объем памяти, используемой специализатором. Показано, что возможно ускорение программ до 10 и более раз.

Пример	**Ускорение (раз)**	**Специализация**	
		Время (сек)	**Память (МБ)**
Вычисление степени	2.5	2.1	30
Поиск подстроки в строке	6	0.5	22
Функция Аккермана	5-6	0.4	17-21
Вычисление значения многочлена	2.5	1	23
Задержанные вычисления	3-4	0.3	18
Трассировка лучей	1.5	8	56
Быстрое преобразование Фурье	3-4	50-720	104-137
Map на SML	1.15	0.3	19
Возведение в степень на SML	5-6	0.6	23
Выражение на Рефале	15	16	65
Интерпретатор на Рефале	100		

Таб. 6. Экспериментальные данные.

Кроме описанных выше примеров запускались примеры «Трассировка лучей» и «Быстрое преобразование Фурье», исходные тексты которых и результат специализации достаточно велики.

«Трассировка лучей»— программа на C#, строящая картинку методом трассировки лучей [21,39]. Пример был взят из [49] и переписан на C#. Программа специализировалась с известным параметром, задающим сцену.

«Быстрое преобразование Фурье» — стандартная реализация быстрого преобразования Фурье на C#. Специализация проводилась по известному количеству отсчетов, длине массива.

Выводы

В данном приложении приведены примеры специализации программ на объектно-ориентированном языке C# и примеры на функциональных языках SML [51] и Refal [50]. Эти примеры показывают, что специализатор CILPE об-

ладает широкими возможностями по специализации реальных программ:

1. Специализатор CILPE обладает поддержкой необходимого набора конструкций языка.
2. Специализатор CILPE является поливариантным по коду, по переменным, по методам и по классам.