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

Препринты ИПМ • Препринт № 44 за 2008 г.

Климов Ю. А.

SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ

Рекомендуемая форма библиографической ссылки: 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. URL: http://library.keldysh.ru/preprint.asp?id=2008-44

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

Ю.А. Климов

SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ

Москва
2008

Ю.А. Климов

SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ

Аннотация

В работе представлен модельный язык SOOL, который близок к внутренним языкам платформ Java и Microsoft .NET. Формально описаны его синтаксис, семантика и типизация. Язык SOOL разрабатывался для формального описания и реализации методов специализации для объектно-ориентированных языков. Он используется как внутренний язык специализатора CILPE.

Работа поддержана проектами РФФИ № 06-01-00574-а и № 08-07-00280-а и проектом Роснауки № 2007-4-1.4-18-02-64.

Yu.A. Klimov

SOOL: an object-oriented stacked-based language for specification and implementation of program specialization techniques

Abstract

The paper presents SOOL, an object-oriented stack-based language, which is similar to the internal languages of the platforms Java and Microsoft .NET. The language is meant for specification and implementation of program specialization techniques. In particular, SOOL has been used as the internal language of the program specializer CILPE. A formal description of the syntax, semantics and typing system of the language is given.

Содержание

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 работе представлен модельный стековый объектно-ориентированный язык SOOL (Stack-based Object-Oriented Language). За основу языка SOOL взято подмножество языка CIL платформы Microsoft .NET [CLI]. Он также очень близок к языку виртуальной машины Java [Java]. Язык SOOL используется в специализаторе [Jone93] CILPE [Chep03,Klim08b] 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. Описание языка SOOL

2.1. Программа

Программа на языке SOOL состоит из определений классов (рис. 1). Определение класса содержит:

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

Рис. 1. Абстрактный синтаксис программ на языке SOOL.

- 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 сов-

падать с именем класса, в котором данный метод определен. Имена методов также должны быть уникальными в программе, за исключением *переопределенных методов*.

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 непосредственным наследником) 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. Типы

В языке SOOL данные могут быть либо *примитивного*, либо *ссылочно-*

го типа (рис. 2).

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. 2. 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 яв-

ляется наследником соответствующего типа второго списка, то будем говорить, что первый список наследует второй список:

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

2.3. Метод

Тело определения метода состоит из списка объявлений переменных и списка инструкций (рис. 3).

```
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. 3. 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. 3):

- 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 определено 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.

2.4. Связь SOOL с CIL.NET

Язык SOOL основан на ядре языка CIL платформы Microsoft.NET. Сопоставление инструкций языков CIL.NET и SOOL дано в таблицах 1 и 2. Многим инструкциям языка CIL соответствуют инструкции языка SOOL, отличающиеся только названием. Другие отличия заключаются в следующем:

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.

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. 1. 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. 2. Преобразование объектных инструкций.

3. Интерпретация программ на языке SOOL

3.1. Состояние

В процессе выполнения каждая инструкция изменяет состояние вычислительной машины (State) (рис. 4). 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 мас-

сив.

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 (ProgramState) 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. 4. Состояние интерпретатора.

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, друге 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. 5). Будем говорить, что данное val *удовлетворяет* типу type, если тип данного наследуется от этого типа: $\text{TypeOf}_{heap}(val) <_{prog} type$.

Для случая, когда куча не известна, определим функцию TypeOf только для целых чисел, чисел с плавающей точкой и NULL аналогичным образом (рис. 5).

Определим вспомогательную функцию DefaultValue (рис. 5), 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. 6).

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 в нем определен, или в ближайшем надклассе.

```
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. 5. 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. 6. Вспомогательные функции.

3.3. Правила выполнения программ на языке SOOL

Семантика языка SOOL описана в стиле операционной семантики [Plot83,Kahn87]. Т.е. используются правила, описывающие изменение состояния.

Для описания правил преобразования состояния при выполнении одной инструкции будем использовать правила в стиле small step semantics (рис. 7). Эти правила близки к тройкам Хоара: $\{state_1\}$ instruction $\{state_2\}$, только записанные в виде правил.

$$\boxed{context \vdash_i instruction : state_1 \rightarrow state_2}$$

Рис. 7. 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).

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. 8).

$$\boxed{context \vdash_i method : state_1 \Rightarrow state_2}$$

Рис. 8. Правило выполнения метода

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

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. 9). Начальное состояние состоит из списка аргументов, пустого окружения и пустой кучи.

$$\boxed{\begin{array}{c} instrs = [\text{NewObject MAIN, CallMethod Main, Leave}] \\ prog \vdash_i (instrs, 0) : (arg, [], []) \Rightarrow (res, [], heap) \\ \hline \vdash_i prog : arg \Rightarrow res \end{array}}$$

Рис. 9. Выполнение программы.

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

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. 10):

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

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

Рис. 10. Выполнение метода.

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. 11).

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

Рис. 11. 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. 12).

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

Рис. 12. 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. 13).
4. В остальных случаях, применяется правило для выполнения одной n-ой инструкции, и далее, начиная с нового состояния, продолжается выполнение последовательности инструкций для (n+1)-ой инструкции (рис. 14).

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

Рис. 13. Выполнение последовательности инструкций, инструкция Branch m.

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

Рис. 14. Выполнение последовательности инструкций.

3.7. Выполнение инструкций

DuplicateStackTop

Инструкция DuplicateStackTop читает значение, находящееся на вершине стека, и добавляет в стек значение, равное исходному значению (рис. 15).

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

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

Рис. 15. Правило выполнения инструкции DuplicateStackTop.

RemoveStackTop

Инструкция RemoveStackTop удаляет значение, находящееся на вершине стека (рис. 16).

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

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

Рис. 16. Правило выполнения инструкции RemoveStackTop.

LoadConst const

Инструкция LoadConst const добавляет в стек константу const, заданную параметром инструкции (рис. 17). Константа const может быть либо NULL, либо значением примитивного типа INT или FLOAT.

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

Рис. 17. 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. 18). 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.

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

$$\frac{op = INT2FLOAT, NOT, NEG \quad TypeOf_{heap}(val) = INT}{prog \vdash_i UnaryOp\ op : (val::stack, env, heap) \rightarrow (op(val)::stack, env, heap)}$$

$$\frac{op = FLOAT2INT, NEG \quad TypeOf_{heap}(val) = FLOAT}{prog \vdash_i UnaryOp\ op : (val::stack, env, heap) \rightarrow (op(val)::stack, env, heap)}$$

Рис. 18. Правило выполнения инструкции UnaryOp op.

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. 19). 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.

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.

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

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

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

Рис. 19. Правило выполнения инструкции BinaryOp op.

LoadVar var

Инструкция LoadVar var читает значение переменной var среды env и добавляет в стек это значение (рис. 20).

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

Рис. 20. 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. 21).

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

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

Рис. 21. Правило выполнения инструкции StoreVar var.

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

Рис. 22. Правило выполнения инструкции NewObject class.

NewObject class

Инструкция NewObject class создает объект obj класса class, кладет его

в кучу heap и добавляет на вершину стека адрес созданного объекта (рис. 22).

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. 23).

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

$$\frac{\mathrm{TypeOf}_{heap}(oref) <_{prog} \mathrm{FieldClass}_{prog}(fld) \qquad oref \neq NULL \qquad (\_, obj) = heap(oref)}{prog \vdash_i \text{LoadField fld} : (oref::stack, env, heap) \rightarrow (obj(fld)::stack, env, heap)}$$

Рис. 23. Правило выполнения инструкции LoadField 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. 24).

Перед выполнением инструкции интерпретатор проверяет, что стек не пуст, и 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.

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

Рис. 24. 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. 25). Количество аргументов и результатов определяется по определению метода mthd, найденного с помощью функции $\mathrm{MethodSignature}_{prog}$.

$$\frac{(tArg, \_) = \mathrm{MethodSignature}_{prog}(mthd) \qquad Length(arg) = Length(tArg) \qquad type = \mathrm{TypeOf}_{heap}(Head(arg)) \qquad type <_{prog} Head(tArg) \qquad type \neq NULLTYPE \qquad prog \vdash_I \mathrm{MethodDefinition}_{prog}(mthd, type) : (arg, heap_1) \Rightarrow (res, heap_2)}{prog \vdash_i \text{CallMethod mthd} : (arg{+}{+}stack, env, heap_1) \rightarrow (res{+}{+}stack, env, heap_2)}$$

Рис. 25. 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. 26).

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

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

$$\frac{\mathrm{TypeOf}_{heap}(ref) <_{prog} \text{OBJECT} \qquad \mathrm{TypeOf}_{heap}(ref) \not<_{prog} type}{prog \vdash_i \text{CastObject type} : (ref::stack, env, heap) \rightarrow (NULL::stack, env, heap)}$$

Рис. 26. 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. 27).

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

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

Рис. 27. Правило выполнения инструкции NewArray type.

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. 28).

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

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

Рис. 28. Правило выполнения инструкции LoadLength.

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. 29).

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.

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

Рис. 29. Правило выполнения инструкции LoadElement.

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. 30).

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.

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

Рис. 30. Правило выполнения инструкции StoreElement.

4. Типизация

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. 31).

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

Рис. 31. 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 $\lambda_{mthdDef}$, отображающая целые числа Integer в стек типов TStack, такая что она типизирует метод mthdDef. Функцию $\lambda_{mthdDef}$ будем называть *типизирующей* функцией метода mthdDef.

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 ти-

пам, 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:

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

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 возможет переход в данную точку.

4.2. Типизация вызова метода Main

Начало выполнение программы — это построение маленького метода и начало его выполнения (рис. 32). 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.

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

Рис. 32. Типизация программы на языке SOOL.

4.3. Типизация метода

Типизирующая функция $\lambda_{mthdDef}$ определяет типы в каждой точке программы, если (рис. 33):

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

Рис. 33. Правило типизации метода.

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 результата 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 $\lambda_{mthdDec}(0)$ ($tArg <_{prog} \lambda_{mthdDec}(0)$).

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 $\lambda_{mthdDec}(n)$ является наследниками соответствующих типов из списка результатов tRes (рис. 34).

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

Рис. 34. 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. 35).

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

Рис. 35. 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. 36).

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

Рис. 36. 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 ($\lambda_{mthdDec}(n) <_{prog}$

$INT::\lambda_{mthdDec}(n+1)$ и $\lambda_{mthdDec}(n) <_{prog} INT::\lambda_{mthdDec}(m)$).

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. 37).

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

Рис. 37. Правило типизации инструкций.

Если значения на стеке удовлетворяют типам $\lambda_{mthdDec}(n)$, то после выполнения инструкции значения на сетке будут удовлетворять типам tStack, а поэтому при переходе на следующую инструкцию они будут удовлетворять типам $\lambda_{mthdDec}(n+1)$ ($tStack <_{prog} \lambda_{mthdDec}(n+1)$).

4.5. Типизация инструкций

$$prog \vdash_t instr : tStack_1 \rightarrow tStack_2$$

Рис. 38. Правило типизации инструкций.

Правило типизации инструкции показывает, как связаны типы значений на стеке до выполнения инструкции и после ее выполнения (рис. 38). 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 $tStack_2$.

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. 39). Остальные элементы стека до и после выполнения инструкции должны совпадать.

$$prog \vdash_t DuplicateStackTop : type::tStack \rightarrow type::type::tStack$$

Рис. 39. 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. 40).

$$prog \vdash_t RemoveStackTop : type::tStack \rightarrow tStack$$

Рис. 40. Правило типизации инструкции RemoveStackTop.

Инструкция RemoveStackTop удаляет верхний элемент стека, осталь-

ные 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. 41).

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

Рис. 41. 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. 42):

- оба типа INT, если операция op — это NOT или NEG;
- оба типа FLOAT, если операция — это NEG;
- тип до выполнения — INT, а после выполнения — FLOAT, если op — это INT2FLOAT;
- тип до выполнения — FLOAT, а после выполнения — INT, если 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.

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

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

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

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

Рис. 42. Правило типизации инструкции UnaryOp op.

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. 43):

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

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

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

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

Рис. 43. 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. 44).

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

Рис. 44. 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. 45).

$$\boxed{prog \vdash_t StoreVar\ var : VarType_{prog}(var)::tStack \rightarrow tStack}$$

Рис. 45. Правило типизации инструкции StoreVar var.

На вершине стека до выполнения операции находится значение, удовлетворяющее 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.

NewObject class

Правило типизации инструкции NewObject class требует, чтобы стек после выполнения инструкции состоял из класса и стека до выполнения инструкции (рис. 46).

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

Рис. 46. 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. 47). Остальные элементы стеков до и после выполнения инструкции должны совпадать.

$$\boxed{prog \vdash_t LoadField\ fld : FieldClass_{prog}(fld)::tStack \rightarrow FieldType_{prog}(fld)::tStack}$$

Рис. 47. 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 поле 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 соответственно, в котором это поле определено (рис. 48). Остальные элементы стека до выполнения инструкции должны совпадать с элементами стека после выполнения этой инструкции.

$$\boxed{prog \vdash_t LoadField\ fld : FieldType_{prog}(fld)::FieldClass_{prog}(fld)::tStack \rightarrow tStack}$$

Рис. 48. Правило типизации инструкции StoreField fld.

Два значения на вершине стека до выполнения инструкции удовлетворяют типу 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. 49). Количество и типы аргументов и результатов находятся с помощью вызова функции $\text{MethodSignature}_{prog}(mthd)$. Остальные элементы стеков до и после выполнения инструкции должны совпадать.

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

Рис. 49. 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. 50). Остальные элементы стеков до и после выполнения инструкции должны совпадать.

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

Рис. 50. 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. Т.к. 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. 51). Остальные элементы стеков до и после выполнения инструкции должны совпадать.

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

Рис. 51. 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. 52). Остальные элементы стеков до и после выполнения инструкции должны совпадать.

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

Рис. 52. Правило типизации инструкции LoadLength.

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.

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. 53). Остальные элементы стеков до и после выполнения инструкции должны совпадать.

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

Рис. 53. 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. 54). Остальные элементы стека до выполнения инструкции должны совпадать с элементами стека после выполнения инструкции.

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

Рис. 54. 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.

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.

5. Заключение

В работах [Affe02,Bert99,Masu99,Schu00] по специализации программ [Jone93] используются различные модельные объектно-ориентированные языки [Klim08a] (таб. 3). В работах Х. Масухара [Affe02,Masu99] используется стековый язык без возможности работы с массивами, все вызовы статические. В работе П. Бертелсена [Bert99] используется стековый язык без возможности работы с объектами и без вызовов методов. В работе У.П. Шульца [Schu00] используется нестековый язык: каждый метод — это одно выражение.

В книге A Theory of Objects [Abad96] представлены различные теоретические объектно-ориентированные языки. Однако они заметно отличаются от широко используемых языков платформ Java [Java] и Microsoft .NET [CLI].

Стековый объектно-ориентированный язык SOOL является аналогом таких языков, как CIL платформы Microsoft.NET или Java Byte Code платфор-

мы 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.

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.П. Шульц	нет	нет	да	виртуальные
SOOL	да	да	да	виртуальные

Таб. 3. Классификация используемых объектно-ориентированный языков.

В тоже время, язык SOOL содержит все необходимые операции над объектами, что позволяет переносить все разработанные преобразования для SOOL на другие объектно-ориентированные языки.

В работе 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.

6. Литература

[Abad96] M.Abadi, L.Cardelli "A Theory of Objects" // Springer-Verlag, 1996.

[Affe02] R.Affeldt, H.Masuhara, E.Sumii, A. Yonezawa "Supporting objects in run-time bytecode specialization" // In Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation, pp.50-60. ACM Press, 2002.

[Bert99] P.Bertelsen "Binding-time analysis for a JVM core language" // April 1999. Unpublished note; available from http://www.dina.kvl.dk/~pmb.

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

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

[Java] Java Virtual Machine (JVM) // http://java.sun.com/docs/books/jvms/

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

[Kahn87] G.Kahn "Natural semantics" // In Proceedings of the Symposium on

Theoretical Aspects of Computer Sciences. LNCS 247, Springer-Verlag, 1987, pp.22-39.

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

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

[Masu99] H.Masuhara, A.Yonezawa "Run-time Program Specialization in Java Bytecode" // In Proceedings of the JSSST SIGOOC 1999 Workshop on Systems for Programming and Applications (SPA'99), March 1999.

[Plot83] G.D.Plotkin "An Operational Semantics for CSP" // D.Bjorner (ed.), Formal Description of Programming Concepts II. North-Holland, Amsterdam, 1983, pp.199-223.

[Schu00] U.P.Schultz "Object-Oriented Software Engineering Using Partial Evaluation" // PhD thesis, University of Rennes I, Rennes, France, December 2000.