УДК 004.92+004.94

И.С. Анурeев

НА ПУТИ К ТЕХНОЛОГИИ РАЗРАБОТКИ ОПЕРАЦИОННОЙ СЕМАНТИКИ КОМПЬЮТЕРНЫХ ЯЗЫКОВ: УНИФИЦИРОВАННЫЙ ФОРМАТ ПОМЕЧЕННЫХ СИСТЕМ ПЕРЕХОДОВ

Анурeев И.С. **На пути к технологии разработки операционной семантики компьютерных языков: унифицированный формат помеченных систем переходов.**

Аннотация. Предлагается формализм для описания помеченных систем переходов, который унифицируют формат состояний системы переходов, формат инструкций компьютерных языков, представляемых метками системы переходов, и формат и семантику правил перехода и, тем самым, делает процесс разработки операционной семантики компьютерных языков более технологичным.

Ключевые слова: операционная семантика, помеченные системы переходов.

Anureev I.S. **Towards technology of development of operational semantics of computer languages: unified format of labelled transition systems.**

Abstract. A formalism for description of labelled transition systems, which unifies the format of states of the systems, the format of computer language instructions represented by labels of the systems and the format and semantics of transition rules, and thus makes the development of operational semantics of computer languages more technological, is proposed.

Keywords: operational semantics, labelled transition systems.

1. Введение. Формальная спецификация компьютерного языка (КЯ) — важный шаг на пути работы с ним как с математическим объектом. Самым распространенным видом формальной спецификации КЯ является операционная семантика КЯ, представляющая собой описание некоторого абстрактного вычислителя, обрабатывающего инструкции КЯ. Это описание, как правило, более близко к неформальной спецификации КЯ, задаваемой стандартом КЯ, по сравнению с описаниями, основанными на других видах семантики (например, аксиоматической или денотационной).

Операционная семантика обычно использует известный универсальный формализм — помеченные системы переходов. Общий метод применения систем переходов к разработке формальной семантики КЯ — метод структурной операционной семантики — был предложен Гордоном Плоткиным в его работе [21].

Анализ современных формальных спецификаций КЯ, основанных на операционной семантике [6, 15, 17–20, 23], показывает, что они объединены только общей концепцией использования систем переходов, в то время как вид систем переходов, система обозначений, понятийный аппарат, методология и техники разработки операционной семантики на основе систем переходов, специфичны для каждой отдельной спецификации.

Понятийно-терминологический аппарат и методология разработки операционной семантики частично унифицируются на базе логико-алгебраического подхода, предложенного Юрием Гуревичем. Он основывается на машинах абстрактных состояний [13, 14] — обобщении систем переходов за счет использования в качестве состояний алгебраических структур. Частичная унификация при этом достигается за счет выбора при спецификации КЯ подходящей алгебраической структуры. Примеры применения этого формализма к разработке операционной семантики КЯ могут быть найдены в [16]. Этот подход реализован в языках ASML [12] и XASM [22].

Для унификации понятийного аппарата КЯ на базе его онтологии в работе [1] разработан операционно-онтологический подход к разработке операционной семантики КЯ, основанный на специальном виде систем переходов — онтологических системах переходов, которые комбинируют системы переходов с онтологическими моделями. На базе этих систем предложен новый вид операционной семантики – операционно-онтологическая семантика.

В этой статье предлагается формализм для описания помеченных систем переходов, который унифицируют формат состояний системы переходов, формат инструкций КЯ, представляемых метками системы переходов, и формат и семантику правил перехода и, тем самым, делает процесс разработки операционной семантики КЯ более технологичным.

Фактически этот формализм представляет собой предметно-ориентированный язык (domain-specific language) с заданными синтаксисом и семантикой, который описывает помеченные системы переходов и областью применения которого является разработка операционной семантики КЯ. Поэтому, описания систем переходов на этом языке называются ориентированными на операционную семантику системами переходов (operational semantics specific transition systems).

Наш подход является дальнейшим развитием логико-алгебраического и операционно-онтологического подходов в контексте решения задачи разработки операционной семантики КЯ (машины абстрактных состояний имеют более широкую область применения, также они используются для спецификации произвольных программных моделей и систем).

Также как и в логико-алгебраическом подходе, состояниями ориентированных на операционную семантику систем переходов являются алгебраические структуры. Однако, вследствие унификации формата состояний, символы сигнатуры алгебраических структур представляют собой не элементы некоторого неконкретизируемого множества, а шаблоны для вызовов функций с предопределенными местами для аргументов. Разбиение аргументов функций на два класса – вычисляемые и невычисляемые – позволяет не ограничиваться функциональными и предикатными символами, а также моделировать сущности более высокого порядка, например логические кванторы. Кроме того, наш подход фиксирует также формат правил переходов и их семантику.

Что касается операционно-онтологического подхода, то ориентированные на операционную семантику системы переходов позволяют реализовывать операционно-онтологическую семантику посредством введения специальных онтологических символов в сигнатуру, что делает этот подход применимым на практике.

2. Предварительные понятия и обозначения. Опишем нотацию основных структур данных, используемых в этой статье как на уровне объектного описания, так и на уровне метаописания, — списков, последовательностей и функций.

Списки имеют вид $(\mathtt{a}_1\ \ldots\ \mathtt{a}_n)$, где элементы $\mathtt{a}_i$ списка разделены пробелами. Пусть `(lists of x)` обозначает множество всех списков из элементов множества `x`, `(lists of x of length n)` — множество всех списков из элементов множества `x` длины `n`, `(length of a)` — число элементов в списке `a`.

Конечные последовательности элементов имеют вид $\mathtt{a}_1\ \ldots\ \mathtt{a}_n$, где $\mathtt{a}_i$ разделены пробелами. Пусть `(sequences of x)` обозначает множество всех конечных последовательностей элементов множества `x` и `(sequences of x of length n)` — множество всех последовательностей элементов множества `x` длины `n`.

Пусть `bool` обозначает множество `{true, false}` и `nat` — множество натуральных чисел с нулем. Пусть `(union of` $\mathtt{A}_n$ `where (n`

$\in$ x)) обозначает объединение всех множеств A_n для всех (n $\in$ x).

Пусть (f x) обозначает применение функции f к последовательности аргументов x (вызов функции f на аргументах x).

Пусть undef обозначает тот факт, что некоторая функция не имеет значения для некоторого аргумента и ((domain of f) = {x | ((f x) $\neq$ undef))} — область определения функции f. Пусть f и g — функции такие, что (((domain of f) $\cap$ (domain of g)) = $\emptyset$). Тогда объединением (f $\cup$ g) функций f и g называется функция h такая, что ((domain of h) = (domain of f) $\cup$ (domain of g)), ((h x) = (f x)) для (x $\in$ (domain of f)) и ((h x) = (g x)) для (x $\in$ (domain of g)).

Для функций (как правило, конечных) мы будем также использовать альтернативную атрибутную нотацию. Пусть f — функция. В этом случае, функция f называется атрибутной структурой, элементы области определения f — атрибутами, а объект (f a) (обозначаемый f.a) — значением атрибута a структуры f.

Определим операции доступа . и модификации upd на функциях следующим образом:

– (f.x = (f x));

– если (y $\neq$ x), то (((upd f x e) y) = (f y));

– (((upd f x e) x) = e).

Операции . и upd переносятся на последовательности и списки, если их рассматривать как функции, областью определения которых является целочисленный отрезок.

Будем говорить, что функция f может отличаться от функции g только на множестве x и обозначать это факт (f can differ from g only on x), если ((f a) = (g a)) для любого (a $\notin$ x).

Помеченная система переходов lts — это тройка (states labels tr), где states и labels — множества, элементы которых называются состояниями и метками, соответственно, логическая функция (tr $\in$ ((states $\times$ labels $\times$ states) $\to$ bool)) называется отношением перехода. Говорят, что из состояния s можно перейти в состояние ss по метке lab, если (tr s lab ss).

3. Системы переходов, ориентированные на программы. Прежде чем определять ориентированные на операционную семантику системы переходов, введем более общий вид систем

переходов – ориентированные на программы системы переходов (Program Specific Transition Systems, P-STS). Эти системы используются для формализации различных аспектов работы с программами (определение семантики, задание стратегий верификации программ и т.п.).

Состояниями в P-STS являются алгебраические системы специального вида.

Пусть `atoms` — некоторое множество объектов, называемых атомами. Выражение — это либо список выражений, либо атом. Пусть `expressions` обозначает множество всех выражений. В контексте последующего изложения будем называть элементы множеств `(sequences of expressions)` и `(lists of atoms)` программами и символами, соответственно, и использовать обозначения `programs` и `symbols` для этих множеств. Пусть `elements` — множество объектов, называемых элементами, такое, что `(expressions` $\subseteq$ `elements)`.

Состояние `s` относительно `(atoms elements)` определяется как тотальная функция из `(symbols` $\to$ `((union of (elements`^{n} $\to$ `elements) where (n` $\in$ `nat))` $\cup$ `{undef}))`. Множество `elements` называется носителем `s`, а его элементы — элементами `s`.

Множество `{(f` $\in$ `symbols)` | `((s f)` $\neq$ `undef)}` называется сигнатурой `s` и обозначается `(symbols of s)`, а элементы этого множества — символами `s`. Функция `(s f)` называется интерпретацией символа `f` в состоянии `s`. В отличие от стандартной алгебраической системы, в которой символы сигнатуры — атомы, символы состояния — списки атомов. Использование списков атомов в качестве символов состояния позволяет приблизить описание вызовов функций, обозначаемых этими символами, к описанию на естественном языке. Другой важной особенностью символов состояния является использование в них специальных атомов `_` и `__`, которые делают такие символы шаблонами для вызовов функций, обозначаемых этими символами. Эти атомы, называемые спецификаторами аргументов, обозначают места для аргументов функций. Пусть `(f` $\in$ `(symbols of s))`. Атом `_`, входящий в `f`, дополнительно указывает на то, что аргумент при вызове функции `(s f)` нужно сначала вычислить, а атом `__`, входящий в `f`, — что вычислять аргумент не нужно. Число вхождений спецификаторов аргументов в `f` называется местностью `f` и обозначается `(arity of f)`. Для `s` должно выполняться свойство: если `(n = (arity of f))`, то `((s`

f) ∈ (elements^n → elements)).

Пусть (u, w ∈ (sequences of expressions)). Пусть typed-args — список структур с атрибутами arg и type со значениями из expressions и {value, itself}, называемых типизированными аргументами. Выражение e — пример f относительно typed-args, тогда и только тогда, когда (применяется первое подходящее правило)

- если (e = ()) и (f = ()), то (typed-args = ());
- если (e = (typed-args.1.arg u)) и (f = (_ w)), то (typed-args.1.type = value) и (u) — пример (w) относительно (typed-args.2 ... typed-args.n);
- если (e = (typed-args.1.arg u)) и (f = (__ w)), то (typed-args.1.type = itself) и (u) — пример (w) относительно (typed-args.2 ... typed-args.n);
- если (e = (b u)) и (f = (b w)), то (u) — пример (w) относительно typed-args;
- false.

Выражение e называется примером f, если e — пример f относительно некоторого typed-args. Одно и то же выражение может быть примером для разных символов состояния, поэтому возможен конфликт при выборе символа состояния для этого выражения. Мы считаем, что определена функция (match ∈ ((expressions × states) → ((symbols × expressions) ∪ {undef}))), разрешающая этот конфликт, такая, что

- если ((match e s) = (f typed-args)), то (f ∈ (symbols of s)) и e — пример f относительно typed-args;
- если ((match e s) = undef), то не существуют (f ∈ (symbols of s)) и typed-args таких, что e — пример f относительно typed-args.

Значение (value of e in s) выражения e в состоянии s определяется следующим образом (применяется первое подходящее правило):

– если e — атом, то ((value of e in s) = e);

– если e = ((ee)) и (ee ∈ (sequences of expressions)), то ((value of e in s) = ((value of (ee) in s)));

– если (e ∈ (symbols of s)), то ((value of e in s) = (s e));

– если ((match e) = (f typed-args)), то ((value of e in s) = ((s f) arg-values));

– ((value of e in s) = undef).

Список (arg-values ∈ (lists of elements of length n)), где (n = (arity of f)), в вышеприведенном определении задается следующим образом:

– если (typed-args.i.type = value), то (arg-values.i = (value of typed-args.i.arg in s));

– если (typed-args.i.type = itself), то (arg-values.i = typed-args.i.arg).

Функция (σ ∈ (atoms → (sequences of expressions))) называется подстановкой. Если (domain of σ) = {x_1, ..., x_n}, то σ может записываться как ((x_1 (σ x_1)) ... (x_n (σ x_n))). Функция подстановки subst относительно σ определяется следующим образом (применяется первое подходящее правило):

– если (e ∈ (domain of σ)), то ((subst e σ) = (σ e));

– если (e ∈ atoms), то ((subst e σ) = e);

– ((subst (e_1 ... e_n) σ) = ((subst e_1 σ) ... (subst e_n σ))).

Опишем теперь компоненты, из которых строится P-STS (p-sts = (states labels tr)).

Состояние predefined-interpretation определяет интерпретацию предопределенных символов. Интерпретация этих символов не меняется, когда p-sts переходит из состояния в состояние. Пусть predefined-symbols обозначает множество (symbols of predefined-interpretation).

Множество `modifiable-symbols` включает модифицируемые символы. Их интерпретация может меняться, когда `p-sts` переходит из состояния в состояние. Множества `predefined-symbols` и `modifiable-symbols` могут пересекаться. Состояние `s` системы `p-sts` расширяется на `predefined-symbols` таким образом, что `(((s f) a) = ((predefined-interpretation f) a))` для любого `(a` $\notin$ `(domain of (s f)))`.

Множество `modifiable-symbols` включает специальные символы `(value _)`, `(history length)` и `(_ is new-element)`. Символ `(value _)` используется для хранения промежуточных значений, появляющихся при функционировании P-STS. Для каждого состояния `s` промежуточные значения суть значения выражений `(value 1)`, ..., `(value k)` в `s`, где `(k = (value of (history length) in s))`. Пусть `((new elements of s) = {(a` $\in$ `elements)` | `(((s (_ is new-element)) a) = true)})`. Это множество характеризует элементы из `elements`, которые «не используются» в состоянии `s` и состояниях, которые предшествовали `s` относительно `tr`. Элементы этого множества будем называть новыми элементами в `s`. Для любого `(s` $\in$ `states)` символ `(_ is new-element)` обладает следующими свойствами:

- `(((s (is new-element)) a)` $\in$ `bool)` для любого `(a` $\in$ `elements)`. Это свойство означает тотальность символа `(_ is new-element)`;
- если `(tr s lab ss)`, то `((new elements of ss)` $\subseteq$ `(new elements of s))`. Это свойство означает монотонное невозрастание множества `(new elements of s)` относительно `tr`;
- если `(tr s lab ss)`, то `((new elements of s)` $\setminus$ `(new elements of ss))` конечно. Это свойство означает, что при каждом переходе «используется» только конечное число новых элементов;
- `(new elements of s)` бесконечно. Это свойство означает, что число новых элементов всегда «достаточно» для «использования» в любом числе переходов;
- `((value of e in s)` $\in$ `(new elements of s)` для любого `(e` $\in$ `expressions)` и `(((new elements of s)` $\cap$ `expressions)` $= \emptyset$`)`. Эти свойства формализуют понятие «не используется».

Помеченная система переходов `p-sts = (states labels tr)` называется P-STS относительно `(atoms elements predefined-interpretation modifiable-symbols)`, если `lab = (p | pp)` для некоторых `(p, pp ∈ programs)` и для любого `(s ∈ states)` выполнены следующие условия:

- `s` — состояние относительно `(atoms elements)`;
- `((symbols of s) = (predefined-symbols ∪ modifiable-symbols))`. Это свойство означает, что сигнатура любого состояния содержит в точности символы из `predefined-symbols` и `modifiable-symbols`;
- `((s (history length)) ∈ nat)`;
- если `(tr s lab ss)`, то `((s (history length)) ≤ (ss (history length)))` для любого `(ss ∈ states)`. Это свойство означает, что множество промежуточных значений может только пополняться;
- `((value of (value i) in s) = undef)` для любого `(i ≥ (s (history length)))`. Это свойство означает, что множество промежуточных значений ограничено константой `(history length)`.

Если `(tr s (p | pp) ss)`, то говорят, что `p` преобразует `s` в `ss` относительно `pp`. Таким образом, программы можно рассматривать как преобразователи состояний. Список `(p s)` называется конфигурацией. Конфигурация должна удовлетворять следующему ограничению: если `p` содержит выражение вида `(value a)`, то `(a ∈ nat)` и `(a ≤ (s (history length)))`. Если `(tr s (p | pp) ss)`, то говорят, что из `(p s)` можно перейти в `(pp ss)`. Конфигурация `(p s)` называется заключительной, если не существует конфигурации `(pp ss)` такой, что `(tr s (p | pp) ss)`. Состояние `s` называется заключительным, если `(p s)` — заключительная конфигурация для любой программы `p`. Конфигурация называется точкой ветвления, если из нее можно перейти более чем в одну конфигурацию.

Трассой называется конечная или бесконечная последовательность конфигураций $(p_1\ s_1)\ (p_2\ s_2)\ \ldots$ такая, что из $(p_i\ s_i)$

можно перейти в (p$_{i+1}$ s$_{i+1}$). Специальный атом backtrack специфицирует фиктивную трассу исполнения программы. Конфигурация (p s) называется конфигурацией отката, если (p.1 = (backtrack)). Отношение tr должно удовлетворять следующему свойству: если (p s) — конфигурация отката, то (p s) — заключительная конфигурация. Конечная трасса, последний элемент которой — конфигурация отката, называется фиктивной. Откат заключается в том, что при достижении конфигурации отката (p s) система p-sts возвращается (откатывается) в ближайшую из точек ветвления, из которой p-sts перешла в (p s), и выбирает другую трассу для выполнения. Если таких трасс нет, то p-sts откатывается в предыдущую точку ветвления. Если таких точек нет, то все трассы фиктивны (возможно за исключением трассы, состоящей из единственного элемента — начальной конфигурации).

Отношение перехода tr определяется как объединение отношений перехода, каждое из которых характеризуется видом выражения p.1. В свою очередь, эти выражения делятся на регулярные выражения и нерегулярные выражения. Нерегулярные выражения изменяют состояние p-sts специфичным образом и для каждого вида таких выражений дается свое определение семантики. Регулярное выражение p.1 изменяет состояние некоторым унифицированным образом, задаваемым правилами перехода (или правилами операционной семантики) для p.1. Правило перехода r имеет вид (if sam var x hvar w then c), где (sam ∈ expressions), (x ∈ (sequences of expressions)),
(w ∈ symbols) и (c ∈ programs). В случае ((length of x) = 0) или ((length of w) = 0), соответствующие компоненты var x и hvar w могут опускаться.

Выражение sam называется образцом правила r. Образец sam определяет множество выражений p.1, к которым применимо правило r. Элементы x называются спецификаторами переменных образца. Если спецификатор u является атомом, то он специфицирует переменную образца u типа exp. Если u имеет вид (seq v), где (v ∈ atoms), то он специфицирует переменную образца v типа seq. В переменных образца сохраняются значения, полученные при сопоставлении с образцом выражения p.1. В переменных типа exp хранятся выражения, а в переменных типа seq — последовательности выражений. Пусть u.var обозначает переменную, которую специфицирует u, а u.type — тип u.var. Элементы w называются

переменные истории правила r. Они используются для сохранения промежуточных значений в символе (value _). Значение символа (history length) при переходе с помощью правила r увеличивается на (length of w). Программа c называется телом правила r. Результат cc модификации тела c в соответствии со значениями переменных образца и переменных истории подставляется в p вместо p.1. В результате получается pp. Тело c не должно включать примеры символов (value _) и (history length).

Для разных видов P-SPS множества модифицируемых символов, виды нерегулярных выражений, их семантика и семантика выполнения правил перехода может отличаться.

4. Системы переходов, ориентированные на операционную семантику. Ориентированные на операционную семантику системы переходов (Operational Semantics Specific Transition Systems, OS-STS) — это специальный случай P-STS, используемый для описания операционной семантики программ.

Множество modifiable-symbols включает символ (value), который используется чтобы моделировать возвращение значений при функционировании P-STS и хранит последнее возвращенное значение. Говорят, что p возвращает значение v относительно pp, если (tr s (p | pp) ss) для некоторого ss и ((ss (value)) = v). Говорят, что p возвращает значение v, если p возвращает значение v относительно некоторой pp.

Нерегулярные выражения в OS-STS бывают четырех видов.

Нерегулярное выражение (stop) называется остановом и имеет следующую семантику: (tr s ((stop) p | pp) ss) тогда и только тогда, когда (ss = s) и pp — пустая последовательность.

Нерегулярное выражение (assume a) называется условием продолжения и имеет следующую семантику: (tr s ((assume a) p | pp) ss) тогда и только тогда, когда (ss = s) и (применяется первое подходящее правило)

– если ((value of a in s) = true), то (pp = p);

– (pp = (backtrack) p).

Нерегулярное выражение (modify a) называется условием модификации и имеет следующую семантику: (tr s ((modify a) p | pp) ss) тогда и только тогда, когда (применяется первое подходящее правило)

– если ((value of a in s wrt ss) = true), то (pp = p) и (ss can differ from s only on x), где x — множество модифицируемых символов, для которых существует пример (e) такой, что (: e) входит в a;

– (ss = s) и (pp = (backtrack) p).

Значение (value of e in s wrt ss) выражения e в состоянии s относительно ss определяется следующим образом (применяется первое подходящее правило):

– если (e ∈ atoms), то ((value of e in s wrt ss) = e);

– если e = ((ee)) и (ee ∈ (sequences of expressions)), то ((value of e in s wrt ss) = ((value of (ee) in s wrt ss)));

– если (e = (: ee)) и ((ee) ∈ (symbols of s)), то ((value of e in s wrt ss) = (ss (ee)));

– если (e = (: ee)) и ((match (ee) ss) = (f typed-args)), то ((value of e in s wrt ss) = ((ss f) arg-values));

– если (e.1 ≠ :) и (e ∈ (symbols of s)), то ((value of e in s wrt ss) = (s e));

– если (e.1 ≠ :) и ((match e s) = (f typed-args)), то ((value of e in s wrt ss) = ((s f) arg-values));

– ((value of e in s wrt ss) = undef).

Список (arg-values ∈ (lists of elements of length n)) в вышеприведенном определении задается также, как в определении для (value of e in s). Специальный атом : в выражении (: ee) означает, что символ, примером которого является (ee), интерпретируется в модифицированном состоянии ss.

Нерегулярное выражение (a ::= b) называется модификацией символа и имеет следующую семантику: (tr s ((a ::= b) p | pp) ss) тогда и только тогда, когда (применяется первое подходящее правило)

– если ((match a s) = (f typed-args)) и (f ∈ modifiable-symbols), то ss = s и (pp = (modify ((: f) = (updv f args b))) p)

– (ss = s) и (pp = (fail) p).

Предопределенный символ (updv _ _ _) имеет следующую интерпретацию: ((value of (updv g (x_1 ... x_n) y) in s) = (upd (value of g in s) ((value of x_1 in s) ... (value of x_n in s)) (value of y in s))).

Список args ∈ (lists of expressions of length (arity of f)) определяется следующим образом:

– если (typed-args.i.type = itself), то (args.i = (quote typed-args.i.arg));

– если (typed-args.i.type = value), то (args.i = typed-args.i.arg).

Предопределенный символ (quote __) имеет следующую интерпретацию: ((value of (quote a) in s) = a).

Определим семантику правила r системы op-sts. Пусть σ — подстановка на множестве переменных образца правила r такая, что ((σ x.i.var) ∈ expressions), если (x.i.type = exp) и ((σ x.i.var) ∈ (sequences of expressions)), если (x.i.type = seq), δ — подстановка на переменных истории правила r такая, что (δ w.j) = (value ((value of (history length) in s) + j)) и (γ = (σ ∪ δ)). Пусть (cc ∈ programs) определяется следующим образом: ((length of cc) = (length of c)) и (cc.k = (del* (subst c.k γ) s)) для всех (1 ≤ k ≤ (length of c)).

Функция del* вычисляет выражения, помеченные специальным атомом * в теле правила r, и определяется следующим образом (применяется первое подходящее правило):

– ((del* (* a) s) = (value of (del* a) in s));

– ((del* (a_1 ... a_n) s) = ((del* a_1 s) ... (del* a_n s)));

– ((del* a s) = a).

Пусть ((length of w) = m) и ((length of p) = n). Отношение перехода tr для регулярного выражения p.1 определяется правилами перехода следующим образом: (tr s (p | pp) ss) тогда и только тогда, когда существует подстановка σ такая, что

– ((subst a σ) = p.1);

– (ss can differ from s only on {(history length)}) и ((ss (history length)) = ((ss (history length)) + m));

– (pp = cc p.2 ... p.n).

В качестве примера применения правил OP-STS определим операционную семантику выражений, часто используемых в OP-STS.

Выражение (new element) называется генератором нового элемента и определяется правилом

```
(if (new element)
 then (modify (((: value) is new-element) and
  ((: _ is new-element) =
   (updv (_ is new-element) ((: value)) false)))))
```

Выражение (fail) обозначает некорректное завершение программы и называется небезопасным завершением. Конфигурация (p s) называется небезопасной, если (p.1 = (fail)). В противном случае, (p s) называется безопасной. Отношение tr должно удовлетворять следующему свойству: если (p s) — небезопасная конфигурация, то (p s) — заключительная конфигурация. Конечная трасса, последний элемент которой — небезопасная конфигурация, называется небезопасной. Выражение (assert a) называется условием безопасности и определяется правилами

```
(if (assert a) var a then (assume a))

(if (assert a) var a then (assume (not a)) (fail) (stop))
```

5. Пример. Рассмотрим разработку операционной семантики на примере модельного компьютерного языка L. Пусть s — текущее состояние. Язык L допускает следующие виды выражений:

– (block p) последовательно вычисляет выражения из (p $\in$ programs);

– x возвращает значение переменной (x $\in$ atoms). В случае, если переменной с именем x нет, программа порождает

(fail). Это выражение называется выражением доступа к переменной. Для простоты будем считать, что все переменные целочисленные. Целые числа определяются предопределенным символом (_ is integer);

– (x := e) присваивает переменной x значение выражения e. В случае если (x := e) — первое присваивание атому x, это присваивание выполняет роль декларации переменной x с инициализатором e. Для простоты будем считать, что e — либо выражение доступа к переменной, либо целое число;

– c возвращает c, если ((value of (c is integer) in s) = true).

– (if x then y else z) — условный оператор с условием (x ∈ expressions), then-ветвью (y ∈ programs) и else-ветвью (z ∈ programs);

– (while x do y) — цикл с условием (x ∈ expressions) и телом (y ∈ programs);

– (random) возвращает произвольное целое число.

В первую очередь, рассмотрим применение OS-STS для разработки операционной семантики языка L. Опишем шаги этой разработки.

На первом шаге определяется множество modifiable-symbols для языка L. Оно включает символы (_ is variable) и (value of _) в дополнение к обязательным символам. Символ (_ is variable) определяет множество переменных программы на языке L. Символ (value of _) хранит значения переменных L-программы.

На втором шаге определяются правила перехода, задающие семантику L-выражений:

```
(if (block a) var (seq a) then a)

(if a var a
 then (assert (a is variable)) ((value) ::= (value of a)))

(if a var a then (assume (a is integer)) ((value) ::= a))
```

```
(if (a := b) var a b then b ((a is variable) ::= true)
  ((value of a) ::= (value)))

(if (if a then b else c) var a (seq b) (seq c)
 then a (assume ((value) = true)) b)

(if (if a then b else c) var a (seq b) (seq c)
 then a (assume ((value) = false)) c)

(if (while a do b) var a (seq b)
 then (assume a) b (while a do b))

(if (while a do b) var a (seq b) then (assume (not a)))

(if (random) then (modify ((: value) is integer)))
```

На третьем шаге определяется множество `predefined-symbols`. Оно состоит из символов, которые вводятся при описании операционной семантики выражений КЯ. Описание этих символов и их интерпретации завершает определение состояния программы на этом языке. В нашем случае, `predefined-symbols` включает символы `(_ = _)`, `(not _)` и `(_ is integer)` с обычной интерпретацией (равенство, отрицание и «быть целым числом»).

Заметим, что если КЯ имеет синтаксис, отличный от синтаксиса выражений, то требуется предварительный шаг. Этот шаг заключается в определении отношения эквивалентности между конструкциями этого языка и выражениями и трансляции его конструкций в эквивалентные им выражения. Поскольку такую трансляцию можно рассматривать как вид денотационной семантики, формальная семантика такого КЯ определяется как комбинированная денотационно-операционная семантика.

Рассмотрим на этом же примере применение OS-STS для разработки операционно-онтологической семантики.

На первом шаге строится онтология КЯ. Для этого множество символов пополняется следующими символами, специфицирующими элементы онтологии КЯ и связи элементов онтологии с экземплярами (Буквы `a`, `b`, `c` используются вместо спецификаторов аргументов, чтобы определить неформальный смысл этих символов.):

– `(a is concept)` означает, что `a` — понятие;

– `(a is attribute of b)` означает, что `a` — атрибут понятия `b`;

– `(a is b)` означает, что `a` — экземпляр понятия `b`;

– `(a of b)` означает значение атрибута `a` экземпляра `b` некоторого понятия.

Онтология КЯ строится как последовательность модификаций символов. Например, онтология оператора `if` строится следующим образом:

```
((if-statement is concept) ::= true)
((condition is attribute of if-statement)) ::= true)
((then is attribute of if-statement)) ::= true)
((else is attribute of if-statement)) ::= true)
```

На втором шаге описываются правила OS-STS, специфицирующей операционно-онтологическую семантику КЯ. Например, для оператора `if` правила имеют вид:

```
(if a var a then
 (assume (a is if-statement)) (* condition of a)
 (assume ((value) = true)) (* then of a))

(if a var a then
 (assume (a is if-statement)) (* condition of a)
 (assume ((value) = false)) (* else of a))
```

6. Заключение. В работе определены два предметно-ориентированных класса помеченных систем переходов – системы переходов, ориентированные на программы, и его подкласс – системы переходов, ориентированные на операционную семантику. Системы переходов, ориентированные на программы, используются для формализации различных аспектов работы с программами (определение семантики, задание стратегий верификации программ и т.п.). Системы переходов, ориентированные на операционную семантику, позволяют сделать процесс разработки операционной семантики компьютерных языков более технологичным.

На примере модельного языка программирования показано, как ориентированные на операционную семантику системы переходов применяются для разработки операционной семантики процедурных языков программирования. Этот же пример иллюстрирует их применение для разработки операционно-онтологической семантики. Спецификация операционно-онтологической семантики при помощи таких систем позволяет применить эти системы к объектно-ориентированным языкам программирования, которые характеризуются богатым набором понятий и концепций.

Отметим, что в силу специфики правил переходов, используемых в этих системах, их можно применять только для описания такого вида операционной семантики как семантика малого шага (small-step semantics).

В дальнейшем предполагается применить подход, основанный на таких системах, к формальной спецификации программных систем и их моделей. Также предполагается разработать новый подкласс систем переходов, ориентированных на программы, — системы переходов, ориентированные на аксиоматическую семантику, и использовать их для унифицированного представления различных видов аксиоматической семантики [2–4, 7–10] в мультиязыковой системе анализа и верификации программ СПЕКТР [5, 11].

Литература

1. *Ануреев И.С.* Операционно-онтологический подход к формальной спецификации языков программирования // Программирование. 2009. №1. С. 50–60.

2. *Ануреев И.С.* Типовые примеры использования языка Atoment // Моделирование и анализ информационных систем. 2011. Том 18, №4. – С. 7–20.

3. *Ануреев И.С., Марьясов И.В., Непомнящий В.А.* Верификация С-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. 2010. Том 17, №3. С. 5–28.

4. *Атучин М.М., Ануреев И.С.* Атрибутные аннотации и их применение в дедуктивной верификации С-программ // Моделирование и анализ информационных систем. 2011. Том 18, №4. – С. 21–33.

5. *Непомнящий В.А., Анureев И.С., Атучин М.М., Маръясов И.В., Петров А.А., Промский А.В.* Верификация С-программ в мультиязыковой системе СПЕКТР // Моделирование и анализ информационных систем. 2010. Том 17, №4. – С. 88–100.

6. *Непомнящий В.А., Анureев И.С., Михайлов И.Н., Промский А.В.* На пути к верификации С программ. Язык C-light и его формальная семантика // Программирование. 2002. №6. С. 1–13.

7. *Непомнящий В.А., Анureев И.С., Промский А.В.* На пути к верификации С программ. Аксиоматическая семантика языка C-kernel // Программирование. 2003. №6. С. 65–80.

8. *Непомнящий В.А., Анureев И.С., Промский А.В., Дубрановский И.В.* На пути к верификации C# программ: трехуровневый подход // Программирование. 2006. №4. С. 4–20.

9. *Шилов Н.В., Анureев И.С., Бодин Е.В.* О генерации условий корректности для императивных программ // Программирование. 2008. №6. С. 5–23.

10. *Anureev I.S.* A three-stage method of C program verification // Joint NCC&IIS Bulletin, Series Computer Science. 2008. Vol. 28. P. 1–29.

11. *Anureev I.S.* Integrated approach to analysis and verification of imperative programs // Joint NCC&IIS Bulletin, Series Computer Science. 2011. Vol. 32. P. 1–18.

12. AsmL: The Abstract State Machine Language. URL: http://research.microsoft.com/en-us/projects/asml/ (дата обращения: 16.10.2012).

13. *Gurevich Y.* Abstract State Machines: An Overview of the Project // Foundations of Information and Knowledge Systems (FoIKS): Proc. Third Internat. Symp. Lect. Notes Comput. Sci. 2004. Vol. 2942. P. 6–13.

14. *Gurevich Y.* Lipari Guide // Specification and Validation Methods. ed. E Börger. Oxford University Press, Inc. New York, NY, USA, 1995. P. 9–36.

15. *Honsell F., Pravato A., Ronchi della Rocca S.* Structured Operational Semantics of the Language Scheme. Tech. Rep., University of Torino, Department of Informatics, 1995.

16. *Huggins J.* Abstract State Machines Web Page. URL: http://www.eecs.umich.edu/gasm (дата обращения: 16.10.2012).

17. *Lucas P., Walk K.* On the Formal Description of PL/1 // Annual Review in Automatic Programming. 1969. Vol. 6(3). P. 105–182.

18. *Morris J.* Algebraic Operational Semantics for Modula-2. PhD Thes. University of Michigan, Ann Arbor, MI, 1988.

19. *Morris J., Pottinger G.* Ada-Ariel Semantics. Tech. Rep. Odyssey Research Associates, 1990.

20. *Norrish M.* C Formalized in HOL. PhD Thes. University of Cambridge, Computer Laboratory, 1998.

21. *Plotkin G.D.* A Structural Approach to Operational Semantics. Tech. Rep. DAIMI FN-19. Computer Science Department, Aarhus University. Aarhus, Denmark, 1981.

22. XasM: An Extensible, Component-Based Abstract State Macnines Language. URL: http://www.xasm.org/ (дата обращения: 16.10.2012).

23. *Wolczko M.* Semantics of Smalltalk-80 // Object-Oriented Programming (ECOOP'87): European Conf. Lect. Notes Comput. Sci. 1987. Vol. 276. P. 108–120.

Ануреев Игорь Сергеевич — к.ф.-м.н.; с.н.с. лаборатории теоретического программирования Института систем информатики имени А.П. Ершова СО РАН (ИСИ СО РАН). Область научных интересов: компьютерные языки, верификация программных систем, спецификация программных систем, семантика программных систем, автоматическое доказательство, безопасность программных систем, онтологические модели, семантика компьютерных языков, предметно-ориентированные языки. Число научных публикаций — 70. anureev@iis.nsk.su; ИСИ СО РАН, пр. Академика Лаврентьева, д. 6, г. Новосибирск, 630090, РФ; р.т. +7(383)330-6360, факс +7(383)332-3494.

Igor S. Anureev — PhD in Computer Science; senior researcher of Theoretical Programming Laboratory of A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences (IIS SB RAS). Research area: computer languages, verification of program systems, specification of program systems, semantics of program systems, automated proving, safety of program systems, ontological models, semantics of computer languages, domain-specific languages. Number of publications — 70. anureev@iis.nsk.su; IIS SB RAS, 6, Acad. Lavrentjev pr., Novosibirsk 630090, Russia; office phone +7(383)330-6360, fax +7(383)332-3494.

Поддержка исследований. Работа выполнена при финансовой поддержке РФФИ, проект №11-01-00028-а, и междисциплинарного интеграционного проекта СО РАН №3.

Рекомендовано лабораторией технологий и систем программирования. Зав. лаб. В. И. Шкиртиль, к.т.н., доцент.

Статья поступила в редакцию 5.10.2012.

РЕФЕРАТ

Ануреев И.С. **На пути к технологии разработки операционной семантики компьютерных языков: унифицированный формат помеченных систем переходов.**

В статье предлагается новый подход к разработке формальных спецификаций компьютерных языков. Он базируется на новом классе помеченных систем переходов — систем переходов, ориентированных на операционную семантику. Цель введения этого класса систем переходов заключается, во-первых, в унификации процесса разработки операционной семантики, и во-вторых, в ускорении этого процесса за счет выделения и формального описания отдельных его шагов.

В начале проводится анализ существующих формальных спецификаций компьютерных языков, базирующихся на системах переходов. Формулируется важная проблема разнородности этих спецификаций, заключающаяся в том, что вид систем переходов, система обозначений, понятийный аппарат, методология и техники разработки операционной семантики на основе систем переходов, специфичны для каждой отдельной спецификации. В качестве решения этой проблемы предлагается использовать предметно-ориентированные системы переходов для области разработки операционной семантики, которые унифицируют формат состояния, формат инструкций компьютерного языка и формат и семантику правил перехода и, тем самым, делают разработку операционной семантики компьютерных языков более технологичной.

Затем исследуется место систем переходов, ориентированных на операционную семантику, среди систем переходов, которые используются для формализации различных аспектов работы с программами (определение семантики, задание стратегий верификации программ и т.п.). С этой целью выделяется и определяется более общий класс систем — системы переходов, ориентированные на программы, которые покрывают вышеперечисленные аспекты. Для этого класса систем описывается общий базис и набор классифицирующих признаков, согласно которому можно выделять подклассы, ориентированные на тот или иной аспект работы с программами.

Далее определяются системы переходов, ориентированные на операционную семантику, за счет выбора подходящих значений классифицирующих признаков.

И наконец, применение систем переходов, ориентированных на операционную семантику, иллюстрируется на примере модельного компьютерного языка.

SUMMARY

Anureev I.S. **Towards technology of development of operational semantics of computer languages: unified format of labelled transition systems.**

A new approach to the development of formal specifications of computer languages is suggested in this paper. It is based on a new class of labelled transition systems — operational semantics specific transition systems. The purpose of introducing this class of transition systems is, first, to unify the development of operational semantics, and, second, to accelerate this process by selection and formal description of its separate steps.

At first, analysis of current formal specifications of computer languages, which are based on transition systems, is performed. An important problem of heterogeneity of these specifications is stated. It consists in the fact that the form of transition systems, the notation, the conceptual apparatus, methodology and techniques of development of the operational semantics based on transition systems are specific to each individual specification. As a solution to this problem, we propose to use domain-specific transition systems for field of development of operational semantics which unify the state format, the format of computer language instructions and the format and semantics of the transition rules, and thus make the development of operational semantics of computer languages more technological.

Then the place of operational semantics specific transition systems among transition systems, which are used to formalize various aspects of the program handling (defining semantics, development of program verification strategies, etc.), is studied. For this purpose, more general class of systems — program specific transition systems that cover these aspects — is selected and defined. A common basis and a set of classifiers is described for this class of systems. The classifiers allow us to select the subclasses, focused on a particular aspect of the program handling.

Further, operational semantics specific transition systems are defined by choosing the appropriate values of classifiers.

Finally, the application of operational semantics specific transition rules is illustrated by an example of a toy computer language.