УДК 004.432.4

Анализ типов в трансляторе с языка предикатного программирования

Зубарев А.Ю. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Семантика языка предикатного программирования P формализована с использованием трех видов отношений: совместимости, согласованности и тождества. Рекурсивные типы определены через аппарат наименьшей неподвижной точки. Обобщенные типы представлены типовыми ограничениями (концептами). Для конструкций с неявной типизацией сформулированы правила восстановления типов переменных. Разработаны алгоритмы проверки корректности рекурсии, определения типов для языковых конструкций, проверки семантической корректности конструкций.

Ключевые слова: *тип данных, семантический анализ, доказательное программирование, предикатное программирование, обобщенное программирование.*

1. Введение

Язык предикатного программирования P [2] является языком доказательного программирования со статической типизацией. В языках доказательного программирования программа задается вместе с ее спецификацией: предусловиями и постусловиями для всех подпрограмм. Поскольку почти все системы автоматического доказательства базируются на тотальных функциях, типы данных должны быть точно определены. В системе типов становится необходимым использовать *подтип* как множество истинности некоторого предиката. Следствием этого является параметризация типов переменными.

В языках доказательного программирования нет указателей. Вместо них используются рекурсивно определяемые алгебраические типы; во многих языках определяемые конструкцией **datatype**.

Другая особенность – использование произвольных типов как параметров, где тип представлен только именем. Параметризация типами характерна для *обобщенного программирования*, обеспечивающего повторную используемость программных компонент за счёт отделения реализации алгоритмов от конкретных определений типов данных [9].

В языке предикатного программирования P имеются конструкции с *неявной типизацией* – типы переменных в таких конструкциях явно не заданы и при трансляции восстанавливаются из контекста.

Анализ типов с учетом указанных особенностей системы типов становится принципиально сложнее, чем для языков императивного программирования.

Обзор работ. Анализ типов является популярной проблематикой в теории трансляции. В работах [11], [7] описаны основные концепции системы типов разных языков программирования. Объект данных определяется как четверка (L,N,V,T), где L – место нахождения объекта, N – его имя, V – его значение и T – тип объекта. Проверка типов — это процесс определения типа указанного объекта данных. Язык программирования является строго типизированным, если проверка типов происходит во время компиляции. Одной из затрагиваемых проблем в работе [7] является способы преобразования объектов данных одного типа в другой. Выделяют две стратегии преобразования типов – явное и неявное (приведение типов). Рассматриваются два возможных определения эквивалентности типов:

1. Два объекта данных имеют эквивалентный тип, если их типы описывают одно множество значений (структурная эквивалентность).

2. Два объекта данных имеют эквивалентный тип, если они имеют одно имя (именная эквивалентность).

Отношение вложенности типов и основанные на нем правила семантики языков анализируются в работе [10]. Параметризация типов в контексте обобщённого программирования рассматривается в работе [4]. В C++ свободу использования обобщенного программирования дают неограниченные шаблоны. Однако при их использовании возникает ряд проблем, таких как сложность поиска и исправления ошибок, недоступность раздельной трансляции шаблонов классов и функций. Отличие подходов C++ и Java/C# следующее: в шаблонах C++ разрешено все, что не запрещено, а в обобщенных типах Java/C#, наоборот, запрещено все, что не разрешено. В языке Scala присутствуют абстрактные типы-члены. Такие типы могут быть ограничены, при этом существует несколько способов описания этих ограничений. Ограничения на типы используются компилятором для доказательства того, что код удовлетворяет системе типов.

Целью данной работы является разработка алгоритмов анализа типов предикатной программы как части семантического анализа в экспериментальной системе предикатного программирования.

В разделе 2 описывается синтаксис языка P, связный с описанием типовых термов, их параметризации и заданием имени типа. Рассматриваются конструкции языка, описывающие

эквивалентные типы данных. *В разделе 3* описывается понятие языковой конструкции, ее подконструкций и позиций. Вводятся вспомогательные термины и обозначения. *В разделе 4* описаны рекурсивные типы данных. Вводятся ограничения на рекурсивные типы, с использованием аппарата неподвижной точки, доказывается их корректность. *В разделе 5* вводится понятие вложенности типов; на его основе строятся три вида отношений: согласованность, совместимость и тождество типов. *В разделе 6* формализуются ограничения на типы языковых конструкций в терминах трех отношений. *В разделе 7* рассматривается параметризация типов и вводится понятия обобщенных типов. Вводятся ограничения (концепты) для таких типов. В *разделе* 8 рассматриваются конструкции с неявной типизацией, приводятся правила восстановления типов для этих конструкций. *В разделе 9* описываются этапы семантического анализа предикатной программы. *В разделе 10* сформулированы основные результаты работы.

2. Синтаксис типовых термов

Предикатная программа определяет предикат в форме вычислимого оператора. Полная предикатная программа языка P состоит из набора рекурсивных предикатных программ, которые определяются следующей конструкцией:

A **(**x**:** y**) pre** P**(**x**) post** Q**(**x**,** y**) {** S **} ;**

где A – имя предиката, x и y – непересекающиеся наборы переменных (аргументы и результаты), P и Q – логические формулы (предусловие и постусловие), S – оператор.

В экспериментальной системе предикатного программирования программа транслируется во внутреннее представление, из которого генерируется код на языках PVS и C++.

Типы языка P являются примитивными или составными. *Примитивные типы*: натуральный (**nat**), целый (**int**), вещественный (**real**), символьный (**char**) и логический (**bool**). *Составные типы* строятся на базе других типов. Синтаксически различные конструкции языка могут описывать эквивалентные типы.

Произвольный тип в предикатной программе определяется *типовым термом*, представляемым далее конструкцией <TERM>. Задание имени типа для типового терма осуществляется *описанием типа* в виде конструкции <DT>.

Синтаксис языковых конструкций будет описываться на расширенном языке Бэкусовских нормальных форм (БНФ) [8] со следующими особенностями:

- **Жирным** шрифтом выделены терминальные символы
- [*фрагмент*] - определяет возможное отсутствие фрагмента

- (*фрагмент*)* - определяет повторение фрагмента нуль или более раз; круглые скобки могут быть опущены, если фрагмент состоит из одного символа
- (*фрагмент*)+ - определяет повторение фрагмента один или более раз
- (*фрагмент*)^ - определяет список вида: *фрагмент* (, *фрагмент*)*

Ниже приведено описание синтаксиса конструкций языка P описывающих тип.

Описание типа:

DT ::= **type** IT [**(**PARAM**)**] = TERM | **type** IT

Параметры типа:

PARAM ::= TERM ID^ [, PARAM] | **type** ID^ [, PARAM]

где ID - *идентификатор*

Типовой терм:

TERM ::= PRIM |
[ID **.**] IT [**(**CARG^**)**] |
EXPR**..**EXPR |
subtype (TERM ID: EXPR | **var** ID: EXPR**)** |
predicate DP |
struct ((TERM ID^)^**)** |
union ((ID [(TERM ID^)^])^) |
enum (ID^**)** |
set (TERM**)** |
array (TERM **,** TERM (**,** TERM)*) |
class [**extends** ID] **{** DC (**;** DC)* **}** |
list (TERM**)** |
string

где IT – *имя типа,* DC – *описание класса,* EXPR – *выражение*

Примитивный тип:

PRIM ::= **nat** [DIGI] |**int** [DIGI] |**real** [DIG]| **bool** |**char**

Разрядность:

DIGI ::= 1 | 2 | ... | 64

DIG ::= 32 | 64 | 128

Предикатная программа:

DP ::= **((**[ARG]**:**RES**)** [**pre** F][**post** F]**) |**
(([ARG]**:**[RES][**#**M](**:** [RES][**#**M])+**)**[pre F] [(**pre** M **:** F)*] [(**post** M **:** F)*]

где M – *метка*

Аргументы и результаты:

ARG ::= TERM ID[`] ( , ID[`])* [, ARG] | **type** ID [, ARG]

RES ::= TERM ID[`] ( , ID[`])* [, RES]

Аргумент вызова:

CARG := EXPR | TERM

Формула:

F ::= EXPR | (F) | ID**(**EXPR^**)** | **!**F | F **&** F | F **or** F | F **=>** F | F **<=>** F |
(Q (TERM ID)^)+ . F

Q ::= **forall** | **exists**

Множеством значений типа **subtype** является подмножество основного указываемого типа, для которого выполняется некоторое логическое выражение. В частности, тип натуральных чисел определяется следующим образом: **subtype** (**int** i: i>=0).

Значением типа **union** является значение одного из конструкторов, перечисленных в списке определения **union**. Конструктор определяется именем конструктора и набором полей.

Изображение отдельных типов языка P представлено в особом синтаксисе. Они определяются через базисные типовые термы следующим образом:

- Тип диапазона a**..**b эквивалентен типу **subtype** (T i: i>=a & i<=b).
- Тип **enum**(A) эквивалентен типу **union**(A).
- Тип **list**(T) вводится определением:

  type list(T) = **union** (**nil**, **cons** (T **car** , **list** (T) **cdr**)).
- Тип **string** эквивалентен типу **list**(**char**)
- Типы **union** с разными порядками одинаковых конструкторов следует считать эквивалентными.

3. Иерархия конструкций в языке P

Языковая конструкция – независимая часть программы. Синтаксис и семантика языка определяют множество допустимых текстов данной конструкции. Например, оператор и выражение – это конструкции. Тип может быть атрибутом конструкции.

Конструкция обычно составляется из *подконструкций*. Место подконструкции в составе объемлющей конструкции называется *позицией* подконструкции. Позиция может иметь ограничения на тип соответствующей ей конструкции. В простейшем случае позиция допускает определенный тип конструкций. Например, условный оператор имеет три

позиции. Позиция условия допускает только логический тип конструкции. Если конструкция составлена из нескольких подконструкций, то их позиции являются *соседними*.

Обозначение A<B> означает, что B является подконструкцией конструкции A. Разумеется, конструкция A может содержать другие подконструкции, но мы выделяем только B. Обозначение A[B] означает, что конструкция B *входит* в конструкцию A, т.е. A[B] эквивалентно A<B> $\vee$ $\exists$C. A<C> & C[B]; иначе говоря, B является частью A и находится на некотором уровне иерархии в структуре конструкции A. Обозначение $A[B_1...B_n]$ означает, что конструкции $B_1...B_n$ входят в конструкцию A.

Пусть S[x] некоторая конструкция, тогда под S[X] будем подразумевать конструкцию S[x], где x = X; под $S^2[X]$ конструкцию S[x], где x = S[X]; под $S^k[X]$ конструкцию S[x], где $x = S^{k-1}[X]$; под $S^0[X]$ конструкцию X.

4. Рекурсивные типы

Совокупность определений типов вида **type** IT [**(**PARAM**)**] = TERM может оказаться рекурсивной. Тип A *непосредственно зависит* от типа B при наличии определения вида **type** A = T[B], где T некоторый составной тип. A *определяется* через тип B, если существуют типы $X_1,...,X_m$ (m > 1) такие, что $A = X_1$, $B = X_m$ и X_i непосредственно зависит от X_{i+1} для i = 1...m-1. Тип B является *рекурсивным*, если B определяется через B.

Рекурсивным кольцом типа A является совокупность типов B, таких что A определяется через B и B определяется через A. Позиция в определении типа A называется *рекурсивной,* если в этой позиции находится рекурсивный тип и этот тип принадлежит рекурсивному кольцу типа A.

Корректность рекурсивных определений типов может быть обеспеченна аппаратом наименьшей неподвижной точки, который рассматривается в работе [5].

Набор определений типов рекурсивного кольца $X_1,...,X_N$ можно представить в виде:

type $X_k = f_k[X_1...X_N]$; k = 1..n ,

где f_k – типовой терм. Отношение включения $\subseteq$ определяет нижнюю полурешетку на типах, описывающих множества данных, с пустым типом, обозначаемым как $\emptyset$. Перепишем систему в векторной форме:

X = f[X],

где $X=(X_1...X_n)$ – вектор типов, $f=(f_1...f_n)$ вектор типовых термов.

Введем отношение $\sqsubseteq$ на типовых векторах: $X \sqsubseteq Y \cong \forall k=1..n\ (X_k \subseteq Y_k)$. Вектор $\Theta = (\emptyset, \emptyset,...,\emptyset)$ является минимальным элементом полурешетки определяемой отношением $\sqsubseteq$.

Пусть $(D, \sqsubseteq, \perp)$ полная решетка с наименьшим элементом $\perp$. Приведем некоторые определения математических понятий, используемых в дальнейшем изложении.

Последовательность $\{a_m\}_{m\geq 0}$ является *возрастающей цепью*, если $a_0 \sqsubseteq a_1 \sqsubseteq \ldots \sqsubseteq a_m \sqsubseteq \ldots$. Для наименьшей верхней грани цепи $\{a_m\}_{m\geq 0}$ будем использовать обозначение: $\cup_{m\geq 0} a_m$.

Лемма. Пусть $\{A_m\}_{m\geq 0}$ – возрастающая цепь, $A^{\sim} = \cup_{m\geq 0} A_m$. Пусть $a \in A^{\sim}$, тогда $\exists k\ \forall m\geq k.\ a \in A^m$.

Тотальная функция $F{:}D\to D$ называется *непрерывной*, если для любой возрастающей цепи $\{a_m\}_{m\geq 0}$ выполняется равенство $F(\cup_{m\geq 0} a_m) = \cup_{m\geq 0} F(a_m)$.

Неподвижной точкой функции F называется решение уравнения $x = F(x)$.

Пусть F произвольная функция, для натурального n определим $F_0(x)=x$, $F_{n+1}(x)=F_n(F(x))$

Лемма. $\{F_n(\perp)\}_{n\geq 0}$ для непрерывной функции F определяет возрастающую цепь.

Теорема Клини. Пусть F – непрерывная функция. Тогда $\cup_{n\geq 0}\{F_n(\perp)\}$ является наименьшей неподвижной точкой F. [3]

Рассмотрим последовательность типовых векторов $\{X^m\}_{m\geq 0}$ определяемую рекуррентно следующим образом: $X^0 = \Theta$, $X^{m+1} = f[X^m]$, $m\geq 0$. Естественно ожидать, что предел последовательности $\{X^m\}_{m\geq 0}$ (если он существует) даст нам неподвижную точку – решение системы $X = f[X]$.

Лемма. Вектор-функция f системы $X = f[X]$ определений рекурсивных типов является непрерывной относительно **struct** и **union**.

Доказательство.

Пусть типовой терм f имеет вид **struct** $(X_1\ g_1,\ X_2\ g_2,\ \ldots,\ X_n\ g_n)$, элементами соответствующего ему типа будем считать кортежи вида $(x_1 \ldots x_n)$, где x_i элемент множества X_i. Докажем, что f непрерывна относительно типов $X_1, X_2, \ldots, X_n$.

Требуется доказать, что

$\cup_{m\geq 0}(\mathbf{struct}\ (X_1^m\ g_1, \ldots, X_n^m\ g_n)) = \mathbf{struct}\ (\cup_{m\geq 0}(X_1^m)\ g_1, \ldots, \cup_{m\geq 0}(X_n^m)\ g_n)$

Докажем сначала, что тип левой части равенства содержится в типе правой части. Пусть $(x_1 \ldots x_n) \in \cup_{m\geq 0}(\mathbf{struct}\ (X_1^m\ g_1, \ldots, X_n^m\ g_n))$. Тогда, согласно рассмотренной лемме, существует такое k, что $(x_1 \ldots x_n) \in \mathbf{struct}\ (X_1^m\ g_1, \ldots, X_n^m\ g_n)$ для всех $m\geq k$. Далее, $x_i \in \cup_{m\geq 0} X_i^m$ и, следовательно, $(x_1 \ldots x_n) \in \mathbf{struct}\ (\cup_{m\geq 0}(X_1^m)\ g_1, \ldots, \cup_{m\geq 0}(X_n^m)\ g_n)$.

Допустим теперь, что $(x_1 \ldots x_n) \in \mathbf{struct}\ (\cup_{m\geq 0}(X_1^m)\ g_1, \ldots, \cup_{m\geq 0}(X_n^m)\ g_n)$ и $x_i \in \cup_{m\geq 0} X_i^m$. Существуют такие k_i, что $x_i \in X_i^m$ для $m\geq k_i$. Пусть $k = \max(k_1, \ldots, k_n)$, тогда $(x_1 \ldots x_n) \in \mathbf{struct}\ (X_1^m g_1, \ldots, X_n^m g_n)$ для $m\geq k$ и, следовательно, $(x_1 \ldots x_n) \in \cup_{m\geq 0}(\mathbf{struct}\ (X_1^m\ g_1, \ldots, X_n^m\ g_n))$.

Пусть типовой терм f имеет вид **union (**C_1, C_2, ..., C_n**)**. Необходимо доказать, что терм f непрерывен относительно типов полей конструктора $X_1, X_2, ..., X_m$. непрерывность конструкторов C_1, C_2, ..., C_n относительно типов $X_1, X_2, ..., X_m$ доказывается аналогично непрерывности **struct**. Докажем, что терм f непрерывен относительно конструкторов C_1, C_2, ..., C_n.

Требуется доказать, что

$\cup_{m\geq 0}$(**union (**C_1^m, ..., C_n^m**)**) = **union (**$\cup_{m\geq 0}(C_1^m)$, ..., $\cup_{m\geq 0}(C_n^m)$**)**

Пусть x $\in \cup_{m\geq 0}$(**union (**C_1^m, ..., C_n^m**)**). Тогда существует такое k, что x $\in$ (**union (**C_1^m, ..., C_n^m**)**) для всех $m\geq k$. Так как, значением типа **union** является значение одного из конструкторов, то $x \in \cup_{m\geq 0}C_i^m$ и, следовательно, x $\in$ **union (**$\cup_{m\geq 0}(C_1)$, ..., $\cup_{m\geq 0}(C_n)$**)**.

Допустим теперь, что x $\in$ **union (**$\cup_{m\geq 0}(C_1)$, ..., $\cup_{m\geq 0}(C_n)$**)** и $x \in \cup_{m\geq 0}C_i^m$. Существует такое k, что $x \in C_i^m$ для $m\geq k$, тогда x $\in$ **union (**C_1, ..., C_n**)** для $m\geq k$ и, следовательно, $x \in \cup_{m\geq 0}$(**union (**C_1, ..., C_n**)**). □

В соответствии с теоремой Клини о неподвижной точке решением системы X = f[X] является неподвижная точка функции f.

Для построения списков и деревьев достаточно рекурсии с использованием **union** и **struct**. Возможны другие, экзотические формы рекурсии, однако они бесполезны при разработке реальных алгоритмов.

Пусть **type** A(x) = T[x] рекурсивный тип. Позиция K терма T *допускает рекурсию*, если выполнено одно из следующих условий:

- T является структурой и K является позицией типа поля;
- T является объединением и K является позицией типа поля конструктора;
- K является позицией типа поля конструктора объединения, стоящего в допускающей рекурсию позиции;
- K является позицией типа поля структуры, стоящей в допускающей рекурсию позиции.

Рекурсивная позиция должна быть позицией, допускающей рекурсию. Для непустого решения рекурсивного уравнения необходимо присутствие в рекурсивном кольце типа **union** с конструктором, не имеющим рекурсивных позиций.

5. Отношения на типах

Прежде чем определить отношение на типах введем вспомогательные обозначения и понятия.

Позицию K типа T будем называть *ковариантной,* если выполнено одно из следующих условий:

- T является структурным типом и K – позиция типа поля;
- T является типом объединения и K – позиция типа поля конструктора;
- T является типом массива и K – позиция типа элементов;
- T является типом множества подмножеств и K позиция базисного типа;
- K является ковариантной позицией некоторого типа, стоящего в ковариантной позиции типа T.

Последнее условие означает, что в иерархически определяемом типе все подуровни вверх от ковариантной позиции должны быть ковариантны.

Определим *предпорядок* на типовых термах. Бинарное отношение вложенности типов $<:$ определяет подмножество декартова произведения TERM×TERM. Если $(t_1,t_2) \in\, <:$ то тип t_1 не превосходит тип t_2; в этом случае будем использовать традиционную запись вида $t_1 <: t_2$. Если $t_1 <: t_2$ и $t_2 <: t_1$, то будем писать $t_1 \sim t_2$.

По определению предпорядка это отношение должно удовлетворять следующим условиям:

- Рефлексивность: $\forall t.\ t <: t$
- Транзитивность: $\forall t_1,t_2,t_3.\ t_1 <: t_2\ \&\ t_2 <: t_3 \Rightarrow t_1 <: t_3$

Для системы типов языка P определим отношение вложенности типов $<:$ следующим набором правил:

Примитивные типы:

1. **int** <: **real**
2. **nat** <: **int**
3. **int** ~ **int 32**
4. **real** ~ **real 64**
5. **nat** ~ **nat 32**
6. **int** d_1 <: **int** d_2, если $d_1 \le d_2$
7. **nat** d_1 <: **int** d_2, если $d_1 + 1 \le d_2$
8. **int** d_1 <: **real** 32, если $d_1 \le 24$
9. **int** d_1 <: **real** 64, если $d_1 \le 53$
10. **int** d_1 <: **real** 128, $\forall\, d_1$
11. **nat** d_1 <: **nat** d_2, если $d_1 \le d_2$

12. **real** d_1 <: **real** d_2, если $d_1 \le d_2$

Составные типы:

13. Если P_1, P_2 – логические выражения и $P_1(x) \Rightarrow P_2(x)$ $\forall x \in T_1$, T_1<:T_2, тогда **subtype**(T_1 x: $P_1(x)$) <: **subtype**(T_2 x: $P_2(x)$).
14. **subtype**(T x: P(x)) <: T $\forall T \in$ TERM.
15. T ~ **subtype**(T x: **true**) $\forall T \in$ TERM.
16. Пусть m параметр, типа T_1. Если P_1, P_2 – логические выражения, $P_1(x,m) \Rightarrow P_2(x,m)$ $\forall m$ ($m \in T_1$, $x \in T_2$) и T_2<:T_3, тогда **subtype**(T_2 x: $P_1(x, m)$) <: **subtype**(T_3 x: $P_2(x,m)$).
17. Пусть type A = **class** {…}, type B = **class** extends A {…}, тогда A <: B.
18. Тип **predicate** ((A_1: R_1) pre S_1 post F_1) <: **predicate** ((A_2: R_2) pre S_2 post F_2), если количество аргументов и результатов совпадают, типы из R_1 тождественны типам из R_2, типы из A_1 не превосходят соответствующие типы из A_2, $S_2 \Rightarrow S_1$ и $F_1 \Rightarrow F_2$.
19. Пусть T_1 = **struct** (...) и T_2 получен из T_1 перестановкой полей, тогда T_1 ~ T_2.
20. Пусть T_1 = **struct** (...) и T_2 получен из T_1 добавлением новых полей, тогда T_1 <: T_2.
21. Пусть типовой терм x находятся в ковариантной позиции типового терма T[x]. Тогда если X <: Y, то T[X] <: T[Y].

Типы по имени и рекурсивные типы.

22. Пусть **type** A_1**(**x**)** = T_1[x] и **type** A_2**(**y**)** = T_2[y]. Тогда если T_1[X] <: T_2[Y], то A_1(X) <: A_2(Y).
23. Пусть **type** A_1**(**x**)** = S_1[x] и **type** A_2**(**y**)** = S_2[y]. T_1[A_1(X)] <: T_2[A_2(Y)], если T_1[S_1[X]] <: T_2[S_2[Y]]
24. Пусть **type** A**(**x**)** = T_1 [x], где T_2 некоторый типовой терм. Тогда если T_1[X] <: T_2, то A(X) <: T_2, а если T_2 <: T_1[X], то T_2 <: A(X).

    Пусть даны два рекурсивных типа **type** A_1**(**x**)** = T_1[x] и **type** A_2**(**y**)** = T_2[y], обозначение A_1**(**X**)** $\sqsubset$: A_2**(**Y**)** означает, что:

    a) типы совпадают за возможным исключением ковариантных позиций

    b) типы, стоящие в нерекурсивных ковариантных позициях T_1[X], меньше либо равны соответствующим типам из T_2[Y].

25. Пусть **type** A_1**(**x**)** = T[x, A_1(S[x])] и **type** A_2**(**y**)** = T[y, A_2(L[y])] два рекурсивных типа. Если $\forall k \ge 0$. A_1**(**S^k[X]**)** $\sqsubset$: A_2**(**L^k[Y]**)**, то A_1(X) <: A_2(Y).

Заметим, что если x = S[x] и y = L[y], то из A_1**(**S[X]**)** $\sqsubset$: A_2**(**L[Y]**)** следует A_1(X) <: A_2(Y). Иначе ограничимся следующими случаями:

- На место параметра типа или параметра переменной в рекурсивном вызове было поставлено значение или типовой терм не зависящий от параметров, тогда из $A_1(X)$ ⊏: $A_2(Y)$ и $A_1(S[X])$ ⊏: $A_2(L[Y])$ следует $A_1(X)$ <: $A_2(Y)$.
- В S имеет место преобразование параметра переменного (n ⇒ f(n)). Все типы **subtype** из T_1, в которых используется этот параметр, сравниваются с типами из T_2, отличным от **subtype**, тогда из $A_1(X)$ ⊏: $A_2(Y)$ следует $A_1(X)$ <: $A_2(Y)$.

На основе описанного нами частичного порядка определим три вида отношений между типами.

1. Совместимость
2. Согласованность
3. Тождество

Совместимость – отношение между типами правой и левой части оператора присваивания, а также между типами формальных параметров и соответствующих аргументов вызова. Тип t_1 *совместим с типом* t_2, если t_1 <: t_2.

Согласованность – отношение между альтернативами условного выражения, позициями большинства бинарных операций. Типы t_1 и t_2 *согласованы*, если они имеют общую *мажоранту*, т.е. ∃ тип t. t_1 <: t & t_2 <: t.

Тождество – отношение между типами результатов вызова и соответствующими формальными результатами. Тип t_1 тождественен типу t_2, если t_1 <: t_2 и t_2 <: t_1.

6. Детальная семантика языка P

Детально рассмотрим конструкции языка P, позиции которых имеют ограничения на типы соответствующих подконструкций.

6.1. Вызов предиката

Исследуем вызов предиката на примере вызова предиката-функции. Для вызова функции и вызова предиката-гиперфункции рассуждения будут аналогичными.

Синтаксис: CALL := ID **(** [CARG^] **:** CRES^ **)**

Аргумент вызова: CARG := EXPR | TERM

Аргументы вызова предиката представлены в виде списка выражений. Типы конструкций в позициях элементов списка должны быть **совместимы** с соответствующими формальными параметрами.

Результат вызова: CRES := ID | TERM ID | **var** ID

Результаты вызова представлены в виде списка переменных (локальных и глобальных). Типы конструкций в позициях результатов вызова предиката должны быть **тождественны** соответствующим типам, указанным при определении предиката.

6.2. Унарные выражения

Синтаксис: UE := O1 EXPR2

Унарная операция: O1 := **+** | **-** | **!** | **~**

Позиция выражения (EXPR2) может быть *полиморфной* т.е. в зависимости от реализации, конструкции в этой позиции могут иметь разные типы. Так, например, конструкция с унарной операцией «**+**» в позиции выражения может содержать конструкции типа **int** или **real**. Типы конструкции в позиции выражения должны быть **совместимы** с типами, представленными в таблице 5.1.

Таблица 6.1

Операция	*Типы позиций*	*Назначение*
+, -	**int, real**	Унарный плюс и минус
!	**bool**	Логическое отрицание
	Set(T)	Поэлементное дополнение
~	**int**	Побитовое дополнение

6.3. Бинарные выражения

Синтаксис: BE := EXPR O2 EXPR

Бинарная операция: O2 := ***** | **/** | **%** | **+** | **-** | **<<** | **>>** | **in** | **<** | **>** | **<=** | **>=** | **=** | **!=** | **&** | **^** | **or** | **xor** | **=>** | **<=>**

Подобно унарным выражениям во многих бинарных выражениях позиции (EXPR) являются полиморфными.

Типы конструкций в позициях выражений должны быть **совместимы** с типами, указанными в таблице 5.2.

Таблица 6.2

№	*Операция*	*Типы позиций*	*Назначение*
1	<, >, <=, >=	**(nat, nat) (int, int) (real, real)**	арифметическое сравнение
2	=, !=	**(nat, nat) (int, int) (real, real) (list, list)**	проверка на равенство и неравенство
3	/	**(nat, nat) (int, int)**	целая часть от деления
4		**(real, real)**	деление
5	%	**(nat, nat) (int, int)**	остаток от целочисленного деления
6	+	**(nat,nat) (int, int) (real, real)**	арифметическое сложение
7		**(list(T), list(T))**	конкатенация списков
8		**(list(T), T) (T, list(T))**	
9		**(Set(T), Set(T))**	объединение множеств
10		**(array(T, T_1, T_2...) array(T, T_1, T_2...))**	объединение массивов с непересекающимися типами индексов
11	-	**(int, int) (real, real)**	арифметическое вычитание
12		**(Set(T), Set(T))**	разность множеств
13	*	**(nat, nat) (int, int) (real, real)**	арифметическое умножение
14	=>	**(bool, bool)**	импликация
15	<=>	**(bool, bool)**	логическое тождество
16	&	**(Set(T), Set(T))**	пересечение
17		**(nat, nat) (int, int)**	побитовое и
18		**(bool, bool)**	конъюнкция
19	xor	**(Set(T), Set(T))**	симметрическая разность
20		**(nat, nat) (int, int)**	побитовое исключительное или
21		**(bool, bool)**	исключительное или
22	or	**(Set(T), Set(T))**	объединение
23		**(nat, nat) (int, int)**	побитовое или
24		**(bool, bool)**	дизъюнкция
25	^	**(nat, int) (int, int) (real, int)**	возведение в степень
26	in	**(T , Set(T))**	проверка вхождения элемента в множество
27	<<, >>	**(int, int), (nat, nat)**	побитовый сдвиг влево и вправо

В операциях 1-7, 9-25, 27 типы подконструкций в соседних позициях должны быть **согласованы**. Бинарное выражение (операции 1, 2, 26) имеет тип **bool**, в остальных случаях – тип первой позиции.

6.4. Условное выражение

Синтаксис: IFE ::= EXPR ? EXPR : EXPR

Конструкция, стоящая в первой позиции должна быть **тождественна** типу **bool**. Две последние позиции полиморфны по всем типам. Типы конструкций этих позиций должны быть **согласованы**. Условное выражение является единственным тернарным выражением в языке P.

6.5. Условный оператор

Синтаксис: IFOP := **if (**EXPR**)** OP [MOVE] **else** OP [MOVE]

где MOVE – оператор перехода, OP – оператор.

В условном операторе конструкция, стоящая в позиции условия, должна быть **тождественна** типу **bool**.

6.6. Оператор присваивания

Синтаксис: ASSIGN ::= R **=** L

В операторе присваивания тип конструкции правой части должен быть **совместим** с типом левой.

6.7. Инициализация переменной

Синтаксис: INI ::= TERM (ID **=** EXPR)^

При инициализации тип конструкции выражения должен быть **совместим** с типом конструкции переменной.

6.8. Оператор присваивания для struct

Синтаксис: ASSIGN ::= R **= (**(EXPR)^**)**

Переменная в правой части имеет тип **struct**. Типы конструкций выражений списка в левой части должны быть совместимы с соответствующими типами полей в типовом терме **struct**.

Несмотря на то, что типы **struct** с разными порядками полей эквивалентны, в данной конструкции порядок, заданный в описании типа **struct**, важен. Полям структуры присваиваются соответствующее порядку значения из списка выражений.

6.9. Оператор выбора

Синтаксис: **switch (**EXPR**) {**

case EXPR^ : OP [MOVE]

[**default :** OP [MOVE]]

}

Позиции выражений в операторе выбора являются полиморфными. Типы соответствующих конструкций должны быть **совместимы** с одними из следующих типов: (**nat**, **nat,...**) (**int**, **int,...**) (**real**, **real,...**).

Имеется вариант конструкции **switch** для объединений:

switch (EXPR**) {**

case ID [([TP] ID)^]: OP [MOVE]

[**default :** OP [MOVE]]

}

Тип переменной: TP ::= TERM | **var**

В этом случае значением альтернативы будет являться конструктор. Конструкция в позиции выражения должна быть типа **union**.

6.10. Типовые термы

Тип по имени: [ID.] ID [**(**CARG^**)**]

Правила подстановки фактических параметров на место формальных в списке аргументов те же самые, что и для вызова предиката.

Подтип: **subtype (**TERM ID: EXPR | **var** ID: EXPR**)**

Конструкции, стоящие в позиции выражений, должны быть тождественны типу **bool**.

Диапазон: EXPR**..**EXPR

Позиции выражений являются полиморфными. Типы соответствующих конструкций должны быть **совместимы** с одними из следующих типов: (**int**, **int**), (**enum**, **enum**), (**char**, **char**). Первая и вторая конструкции должны быть **согласованы** между собой

6.11. Импорт модуля

Синтаксис: INM::= **import** ID [**(**CARG^**)**] [**as** ID]

Правила подстановки фактических параметров на место формальных те же, что и для вызова предиката.

6.12. Вызов формулы

Синтаксис: CALLF ::= ID (EXPR^)

Типы конструкций в позициях элементов списка выражений должны быть **совместимы** с соответствующими формальными параметрами.

6.13. Вызов процесса

Синтаксис: CALLPROC ::=IPROC **(** [CARG^] (:[CRES^] [MOVE])* **)** ,

где IPROC – имя процесса

При вызове процесса ограничения на типы конструкций аргументов и результатов те же, что и для предикатов.

6.14. Отправка сообщения

Синтаксис: SENDMES ::= **send** IM [(EXPR^)]

где IM – имя сообщения

Типы конструкций в позициях элементов списка выражений должны быть **совместимы** с соответствующими формальными параметрами.

6.15. Литералы

Литералы, представляют собой фиксированное значение, определенного типа данных. Конструкции, являющиеся литералами, имеют следующий синтаксис:

Таблица 6.3

Конструкция	*Тип*
SIGN (DIG)+	**subtype (int** x: x = X**)**
SIGN **0x**(XDIG)+	**subtype (int** x: x = X**)**
[SIGN] (DIG)+ [**.** (DIG)+] [DEG]	**subtype (real** x: x = X**)**
[SIGN] **inf**	**real**
nan	**real**
‘SYMB**’**	**char**
“ (SYMB)* **“**	**string**
true | **false**	**bool**
nil	**string**

DIG ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

XDIG ::= DIG | **A** | **B** | **C** | **D** | **E** | **F**

SIGN ::= **+** | **-**

DEG ::= **e** [SIGN] (DIG)+ | **E** [SIGN] (DIG)+

SYMB ::= … | **\"** | **\'** | **** | **\0** | **\n** | **\r** | **\t** | **\x** XDIG [XDIG [XDIG [XDIG]]]

REG ::= (EXPR)^

6.16. Агрегаты

Агрегат определяет конструктор составного объекта. Конструкции, являющиеся агрегатами, имеют следующий синтаксис:

Таблица 6.4

Конструкция	*Тип*
[TERM] **[** REG **]**	**array** (1..n , T) (n равно количеству выражений в REG)
[TERM] **{** REG **}**	**set** (T)
[TERM] **[[** REG**]]**	**list** (T)

REG ::= (EXPR)^

7. Обобщенные типы

Обобщенными типами будем называть:

- Параметры-типы [**type** T]
- Типы **subtype**, в выражение p(x, n) которых явно или неявно будут входить параметры-переменные n [**subtype** (T x: p(x, n))]

В процессе семантического анализа для обобщенного типа накапливаются ограничения на ассоциированный с ним тип. Ограничения будут представлены как совокупность верхних и нижних граней в терминах рассмотренного выше предпорядка типов. Символами "\", "/" будем определять соответственно ограничение сверху и снизу. Далее, **MAX** и **MIN** будут обозначать, соответственно, супер-тип (∀T. T <: **MAX**) и пустой тип (∀T. **MIN** <: T). Каждое ограничение будем заключать в круглые скобки. Если требуется одновременное выполнение ограничений (A) и (B) будем писать (A) ∧ (B). Если требуется выполнение одного из ограничений (A) или (B) будем писать (A) ∨ (B). При определении типа по имени без инициализации ему автоматически будет заданы следующие ограничения {(**MAX**) ∧ (/**MIN**)}. При определении типа **subtype** (T x: p(x,n)) – ограничения {(/**MIN**) ∧ (\T)}. При каждом использовании экземпляров данного типа его ограничения будет изменяться. Сами ограничения также могут содержать обобщенные типы. Ограничения в этом случае будут наследоваться, и при изменении ограничений обобщенного типа будут меняться все ограничения, в которых он используется.

Совокупность ограничений для типа будем называть *концептом.* Концепты позволяют выявлять ошибки в ходе статического анализа типа до инициализации некоторых переменных и типов-переменных. Например, тип-параметр будет некорректным при

следующем ограничении {(/**real**) ∧ (**nat**)}, а тип с ограничением {(**MIN**)} будет пустым, о чем имеет смысл предупредить разработчика. Концепты упрощают использование отдельных модулей программы. Разработчику возможно предоставить формальные ограничения на типы, которые ему разрешено использовать в параметрах модуля. Анализ корректности фактических типов впоследствии будет сведен к проверке соответствующих ограничений на обобщенные типы.

Обобщенные типы обеспечивают языку P поддержку обобщённого программирования [9], которое, не ограничиваясь определенными типами данных, позволяет повторно использовать код, реализующий некоторый алгоритм.

Пусть T – некоторый обобщенный тип с текущим концептом {(P)}, рассмотрим ограничения на тип T для введенных нами ранее отношений на типах.

Таблица 7.1

Отношение на типах	***Концепт***
Тип A совместим с T	{(P) ∧ (/A)}
T совместим с типом A	{(P) ∧ (\A)}
T согласован с типом A с общей мажорантой B	{(P) ∧ (\B)}
T тождественен типу A	{(P) ∧ (\A) ∧ (/A)}

Если последний концепт не противоречив он будет допускать единственный тип A.

В определении концепта чаще всего используются дизъюнкции, но в случаях полиморфных позиций могут использоваться и конъюнкции. Например, для полиморфного оператора «минус» тип операнда будет иметь следующее ограничение: {(**int**) ∨ (**real**) ∨ (**Set**(MAX))}, которое, в свою очередь, эквивалентно ограничению: {(**real**) ∨ (**Set**(MAX))}.

При изменении ограничений для типа следует проверить их непротиворечивость. В этих целях предполагается использование SMT-решателя [6]. Для корректного построения концептов исключается использование ограничений, связанных с рекурсивными типами, следовательно, обобщенные типы не могут быть ассоциированы с рекурсивными типами.

8. Неявная типизация

Язык P имеет элементы неявной типизации, которая, в частности, характеризуется ключевым словом **var**. Существует четыре вида конструкций с неявной типизацией. Для каждой такой конструкции следует восстановить соответствующие типы.

1. **Конструкциях определения массива**

   Определение массива: DM ::= **for** (([TP] ID)^) EXPR

Ключевое слово **var** в TP заменяется типовым термом, который соответствует типу индексов конструкции DM типа массив.

Пример:

```
array (int, 1..5) a;
a = for (var i) i*i;
```

2. **Конструкция локальных переменных-результатов**

   CRES := ID | TERM ID | **var** ID

   Типовой терм в данном случае определяется соответствующим типом, указанным при определении вызываемого объекта.

3. **Конструкции case для union**

```
switch (EXPR)
{case ID [([TP] ID)^] : OP [MOVE]
[default : OP [MOVE]
}
```

   Типы параметров восстанавливаются по типу полей соответствующих конструкторов в типе **union** (EXPR в **switch**).

4. **Конструкция описания переменных**

   Тип переменной определяется из контекста дальнейшего использования переменной. Для такой переменной в этом случае будет создан новый обобщенный тип, который при использовании этой переменной будет строить свой концепт. По выходу из блока, где использовалась эта переменная, **var** примет значение одной из верхних граней, описанных концептом.

В первых трёх случаях тип восстанавливается непосредственно из контекста конструкции. Для восстановления неявного типа в конструкции 4, используется инструмент обобщенных типов. При обнаружении соответствующей переменной ей присваивается новый обобщенный тип, который по мере использования переменной строит концепт. По выходу из соответствующего блока переменной присваивается тип, который является верхней гранью концепта. Заметим, что концепт должен быть непротиворечив и иметь единственную верхнюю грань, которая отлична от **MIN** и **MAX**.

Описатель **var** в некоторых конструкциях может отсутствовать, в любом случае тип следует восстановить.

9. Задача анализа типов

В процессе трансляции для некоторой синтаксически корректной конструкции требуется установить, является ли эта конструкция семантически правильной. Для этого необходимо провести *семантический анализ*, в частности определить типы для всех подконструкций, выполнить проверку, допустимы ли эти конструкции в данных позициях, определить тип данной конструкции.

Семантический анализ языка P имеет четыре стадии:

1. Выявление неявной типизации и восстановление типов.
2. Определение типов для идентификаторов, литералов и агрегатов.
3. Проверка корректности рекурсивных типов
4. Иерархическая проверка корректности всех подконструкций на соответствующие типовые ограничения позиций.

Вторая стадия, называемая идентификацией, осуществляется при помощи таблиц идентификаторов. Для каждого нового блока или определения предиката создается новая таблица, куда заносятся данные о встречающихся определений переменных и типов.

Для анализа корректности рекурсии параллельно идентификации строиться ориентированный граф. Вершины графа помечены именами типов, дуги типовыми термами. Существование дуги (AB) в этом графе эквивалентно тому, что A непосредственно зависит от B. Если из вершины A существует путь к A, то тип A – рекурсивный. Для рекурсивного типа необходимо найти рекурсивное кольцо и проверить его корректность.

Последняя стадия – проверка типов. Алгоритм проверки типов следующий:

1. Если конструкция имеет подконструкции, то запускаем проверку типов для этих подконструкций.
2. Если конструкция имеет ограничения на тип, то проверяем эти ограничения
3. Для конструкции определяем её тип с использованием типов подконструкций.

Все конструкции, имеющие ограничения на тип, описаны выше. Проверка ограничений возможна, так как все подконструкции будут проверены на корректность и для них будет установлен тип.

10. Заключение

В настоящей работе представлена модель типов языка предикатного программирования, которая включает в себя:

1. Определения рекурсивных типов на базе аппарата наименьшей неподвижной точки рекурсивных типовых уравнений;
2. Предпорядок на типах языка P;
3. Обобщенные типы, построенные на базе концептов – наборов ограничений для типов;
4. Конструкции с неявной типизацией и правила восстановления неявных типов.

Определены три вида отношений: согласованность, совместимость и тождество для пары типов. С помощью данных отношений была формализована семантика конструкций языка предикатного программирования и разработаны правила определения типов выражений и агрегатов. В соответствии с этими правилами разработаны алгоритмы проверки корректности рекурсии, установки типов для языковых конструкций, проверки семантической корректности конструкций [1].

В настоящее время ведется работа по завершению программной реализации анализа типов в экспериментальной системе предикатного программирования.

Список литературы

1. Зубарев А. Ю. Анализ типов в трансляторе с языка предикатного программирования // Материалы 54-й международной научной студенческой конференции МНСК-2016 Математика. Новосибирск: Новосиб. гос. ун-т., 2016. С. 201.
2. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск: 2013. 52 с. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf (дата обращения: 7.12.2016).
3. Клини С. К. Введение в метаматематику: Пер. с англ. М.: 1957. 526 с.
4. Пеленицын А.М. Ассоциированные типы и распространение ограничений на параметры-типы для обобщенного программирования на Scala // Программирование. 2015. №4. С. 13-22.
5. Шелехов В.И. Предикатное программирование: Учеб. пособие. Новосибирск: Новосиб. гос. ун-т, 2009. 109 с.
6. Barrett C., Fontaine P., Tinelli C. The SMT-LIB Standard Version 2.5. 2015. 94 p.
7. Dershem H.L., Jipping M.J. Programming Languages: Structures and Models. Boston: Wadsworth, 1990. 413 p.
8. ISO/IEC 14977:1996(E) Информационная технология – Синтаксический метаязык – Расширенная Форма Бэкуса-Наура (Extended BNF).

9. Musser D.A., Stepanov A.A. Generic Programming // Proceeding of International Symposium on Symbolic and Algebraic Computation. V. 358 of Lecture Notes in Computer Science. Rome. Italy. 1988. P. 1325.
10. Pierce B.C. Types and Programming Languages . Massachusetts: The MIT Press Cambridge, 2002. 623 p.
11. Watt D.A. Programming language concepts and paradigms . Upper Saddle River, New Jersey: Prentice Hall, 1990. 322 p.

УДК 004.43

Предикатная программа вставки в АВЛ-дерево

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Операции с АВЛ-деревьями компактно и элегантно представляются в языках функционального программирования. Однако функциональные программы для операций вставки или удаления вершины заведомо неэффективны, поскольку определяют построение нового дерева, а не модификацию исходного.

Описывается построение двух версий предикатных программ вставки в АВЛ-дерево, допускающих автоматическую трансформацию в эффективные императивные программы. В языке предикатного программирования введена эффективно реализуемая операция доступа вершины по пути в дереве.

Ключевые слова: *АВЛ-дерево, функциональное программирование, трансформации программ, алгебраический тип данных.*

1. Введение

Принципиальная сложность императивного программирования обнаруживается особенно при работе с указателями. Показателем такой сложности является чрезвычайная трудность дедуктивной верификации программ, оперирующих указателями, например, в алгоритме реверсирования списка [14].

В предикатном программировании [6, 7, 16] нет таких языковых конструкций, как циклы и указатели, серьезно усложняющие программу. Вместо циклов используются рекурсивные программы, а вместо указателей – объекты алгебраических типов (списки и деревья). Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением следующих оптимизирующих преобразований [3], переводящих программу на императивное расширение языка P [4]:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;

• кодирование алгебраических типов (списков и деревьев) с помощью массивов и указателей для всех видов операций с объектами алгебраических типов [9]. Отметим, что для алгоритма реверсирования списка используется одна нетривиальная трансформация.

Такая структура данных как граф непредставима алгебраическими типами данных. В предикатном программировании граф представляется массивом вершин. Индекс вершины в массиве является аналогом указателя.

Операции с АВЛ-деревьями компактно и элегантно представляются в языках функционального программирования [10, 13]. Имеется более десятка разных работ (см. например [16]) по дедуктивной верификации и доказательному построению простейших функциональных программ, реализующих операции с АВЛ-деревьями. Однако функциональные программы для операций вставки или удаления вершины заведомо неэффективны, поскольку определяют построение нового дерева, а не модификацию исходного.

В данной работе делается попытка построения таких предикатных программ вставки в АВЛ-дерево, чтобы применением оптимизирующих трансформаций получить эффективные императивные программы, подобные представленным в [1, 2, 15]. До сих пор в технологии предикатного программирования удавалось воспроизвести любую реализацию, проводимую в императивном программировании, для обширного набора алгоритмов из класса задач дискретной и вычислительной математики. Однако при реализации алгоритмов работы с АВЛ-деревьями, особенно нерекурсивного алгоритма вставки нового элемента, обнаруживается недостаток существующих средств. В настоящей работе в языке P вводятся новые конструкции, в частности, средства доступа вершины по некоторому пути в дереве.

В разделе 2 определяется языковые и технологические особенности предикатного программирования. Представление АВЛ-деревьев описывается в разделе 3. Вводятся дополнительные конструкции языка P для эффективной работы с деревьями. В разделе 4 приведены предикатные программы для классического рекурсивного алгоритма вставки в АВЛ-дерево, а также эффективного нерекурсивного алгоритма. В разделе 5 описываются методы применения оптимизирующих трансформаций с получением эффективных императивных программ для двух версий алгоритма вставки в АВЛ-дерево. В заключении отмечаются особенности реализации и приводятся сравнения с реализациями на форуме [1].

2. Предикатное программирование

Полная предикатная программа состоит из набора рекурсивных *предикатных* программ на языке P [4] следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
    pre <предусловие>
    post <постусловие>
    { <оператор> }
```

Необязательные конструкции предусловия и постусловия являются формулами на языке исчисления предикатов; они используются для улучшения понимания программ и для дедуктивной верификации [6, 7, 16].

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [6, 7, 16]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [12]) оптимизация программы полностью возлагается на транслятор, в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду. Разумеется, функциональное программирование существенно уступает в эффективности, поскольку даже применением изощренных методов оптимизации невозможно автоматически воспроизвести серию оптимизаций, совершаемых программистом вручную.

Гиперфункции. Вызов программы A(x, y) с аргументами x и результатами y записывается в виде A(x**:** y). *Гиперфункция* – программа с несколькими *ветвями* результатов. Гиперфункция A(x**:** y**:** z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются.

Рассмотрим предикатную программу следующего вида:

```
A(x: y, z, c)
pre P(x)
post c = C(x) & (C(x) ⇒ S(x, y)) & (¬C(x) ⇒ R(x, z))
{ … };
```

Здесь x, y и z – непересекающиеся возможно пустые наборы переменных; P(x), C(x), S(x, y) и R(x, z) – логические утверждения. Предположим, что все присваивания вида c = **true** и c = **false** – последние исполняемые операторы в теле программы. Программа А может быть заменена следующей программой в виде *гиперфункции*:

```
hyper A(x: y #1: z #2)
pre P(x)   pre 1: C(x)
post 1: S(x, y) post 2: R(x, z)
{ … };
```

В теле гиперфункции каждое присваивание c = **true** заменено оператором перехода #1, а c = **false** – на #2.

Гиперфункция A имеет две *ветви* результатов: первая ветвь включает набор переменных y, вторая ветвь – z. *Метки* 1 и 2 – дополнительные параметры, определяющие два различных *выхода* гиперфункции. *Спецификация гиперфункции* состоит из двух частей. Утверждение после “**pre** 1” есть предусловие первой ветви; предусловие второй ветви – отрицание предусловия первой ветви. Утверждения после “**post** 1” и “**post** 2” есть постусловия для первой и второй ветвей, соответственно.

Ветви *вызова гиперфункции* выходят в разные места программы, содержащей вызов. Вызов гиперфункции записывается в виде A(x**:** y #M1**:** z #M2). Здесь M1 и M2 – метки программы; операторы перехода #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2. Вызов вида A(x**:** y #M1**:** z #M2); M1**: ...** может быть представлен в виде A(x**:** y**:** z #M2).

Аппарат *гиперфункций* является более общим и гибким по сравнению с известным механизмом обработки исключений, например, в таких языках, как Java и C++. Использование гиперфункций делает программу короче, быстрее и проще для понимания [7, 8].

Императивные конструкции. *Модифицируемой* является переменная, являющаяся аргументом и результатом некоторой предикатной программы. Наряду с оператором вида x′ = x + 1, где подразумевается, что x′ склеивается с x, в предикатной программе допускается оператор вида x = x + 1, а также привычная его форма в виде x++.

На базе операции модификации [4] для значений структурных типов строится *оператор модификации*. Оператор A[i] = x является эквивалентом A′ = A **with** [i**:** x]. Аналогично, оператор B.f = x эквивалентен B′ = B **with** (f**:** x). Дополнительно, поля конструктора типа объединения подобны полям структуры, и для них также следует разрешить операцию модификации и эквивалентный оператор модификации. Следующий шаг – это возможность использования переменных вида A[i] и B.f в качестве результатов в вызовах предиката: подобные вызовы нетрудно заменить легальными конструкциями вставкой дополнительного

оператора модификации за вызовом. Например, G(…**:** A[i]) заменяется на G(…**:** X x); A′ = A **with** [i**:** x].

Следует предоставить возможность заменить оператор вида b′ = b пустым оператором. Вследствие замены оператора вида b′ = b пустым оператором появляется укороченный условный оператор **if** (E(x)) A(x**:** y).

В функциональном программировании внесение в программу императивных конструкций реализуется неявно через аппарат монад. Без монад функциональные программы потеряли бы свою компактность и привлекательность. В предикатном программировании императивные конструкции определены явно. Программа с императивными конструкциями легко приводима к правильной предикатной программе.

Дополнительные конструкции для работы с деревьями представлены в разделе 3.

3. АВЛ-деревья

Двоичное дерево – дерево, в котором каждая вершина имеет не более двух потомков. Двоичное дерево используется для представления *таблицы* для хранения множества данных вместе с их *ключами*, используемыми для поиска. Основные операции: включение нового данного, исключение данного и поиск данного в таблице. Ключи и данные представлены следующими типами:

```
type Tkey;
type Tinfo;
```

Для типа ключей Tkey определено отношение линейного порядка «<». Типы Tkey и Tinfo – произвольны и являются параметрами модуля, реализующего АВЛ-деревья.

Элемент таблицы является структурой из двух полей:

```
type ElTab = struct(Tkey key, Tinfo info);
```

Двоичное дерево представляется структурой типа Tree:

```
type BAL = -2..2;
type Tree = union (
  leaf,
  node (Tkey key, Tinfo info, BAL balance, Tree left, right)
);
```

Лист дерева соответствует конструктору leaf. Вершина дерева, соответствующая листу, не хранит никакой информации. Конструктор node определяет вершину, не являющуюся листом. Полями конструктора являются ключ key и ассоциированное с ним данное info.

Левое и правое поддеревья, исходящие из данной вершины, определяются полями left и right. Назначение поля balance будет определено ниже.

Высота heigh дерева N определяется следующей формулой:

```
formula heigh(Tree N: nat) = (N == leaf)? 0 : max(heigh(N.left), heigh(N.right);
```

Совокупность элементов таблицы, хранящихся в двоичном дереве N, характеризуется предикатом isin, определяющим принадлежность элемента (k, x) таблице:

```
formula isin(Tkey k, Tinfo x,  Tree N) =
  (N == leaf)? false : N.key == k & N.info == x ∨ isin(N.left) ∨ isin(N.right);
```

В соответствии с данной формулой для непустого дерева элемент (k, x) либо хранится в корневой вершине, либо принадлежит одному из поддеревьев.

Двоичное дерево поиска – двоичное дерево со следующими свойствами:

- оба поддерева – левое и правое, являются двоичными деревьями поиска;
- у всех вершин левого поддерева произвольной вершины X значения ключей данных меньше, нежели значение ключа данных самой вершины X;
- у всех вершин правого поддерева той же вершины X значения ключей данных больше, нежели значение ключа данных вершины X.

Двоичное дерево поиска N удовлетворяет следующему отношению упорядоченности isord:

```
formula isord(Tree N) =
    (N == leaf)?  true :  isord(N.left) & isord(N.left) &
                          (∀ Tkey k, Tinfo x. isin(k, x, N.left) ⇒ k < N.key) &
                          (∀ Tkey k, Tinfo x. isin(k, x, N.right) ⇒ N.key < k);
```

АВЛ-дерево – сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота двух поддеревьев вершины различается не более чем на 1. Свойство isbal сбалансированности дерева N определяется следующей формулой:

```
formula isbal(Tree N) =
    (N == leaf)?  true :  isbal(N.left) & isbal(N.left) &
                          (heigh(N.left) == heigh(N.right) ∨
                           heigh(N.left) + 1 == heigh(N.right) ∨
                           heigh(N.left) == heigh(N.right) +1 );
```

Поле balance – разница высот правого и левого поддеревьев вершины N: N.balance = heigh(N.right) - heigh(N.left). Дерево, в котором поле balance в каждой вершине равно разнице высот поддеревьев, удовлетворяет предикату, представленному формулой:

```
formula withbal(Tree N) =
(N == leaf)?  true :  N.balance == heigh(N.right) - heigh(N.left) &
                      withbal(N.left) & withbal(N.right);
```

Тип АВЛ-дерева определяется следующим образом:

```
formula isAVL(Tree N) = isord(N) & isbal(N) & withbal(N)
type AVLtree = subtype (Tree N: isAVL(N)};
```

Дополнительные операции с деревьями. С алгебраическим типом Tree считаются ассоциированными следующие типы:

```
type __Tree = enum (left, right);
type _Tree = list(__Tree);
```

Переменная типа __Tree называется *динамическим полем*. Значение типа _Tree определяет *путь* в дереве в виде последовательности полей, ведущих от корня дерева в некоторую его вершину. Для динамического поля f, принадлежащего типу __Tree, конструкция N.f определяет доступ по чтению и записи определяется следующим образом:

```
N.f ≡ f == left? N.left : N.right;
N.f = x ≡ if (f == left) N.left = x else N.right = x;
```

Доступ к вершине, идентифицируемой путем p в дереве N, реализуется конструкцией N.p.

```
N.p ≡ p == nil? N : N.(p.car).(p.cdr);
N.p = x ≡ if (p == nil) N = x else N.(p.car).(p.cdr) = x;
```

Конструкция N.p определена лишь при условии корректности пути p. Путь p в дереве N является *корректным*, если он существует в дереве N. Корректность пути определяется предикатом valid:

```
formula valid(_Tree p, Tree N) = p == nil ? true : N != leaf & valid(p.cdr, N.(p.car));
```

Для пути p операция p.left означает присоединение поля left к пути p. Иначе говоря, значение p.left есть p + left, где «+» понимается как операция конкатенации списков.

В трансформации операций с деревьями конструкция N.p обычно представляется указателем на переменную, соответствующую последнему полю, ссылающемуся на требуемую вершину.

4. Программы вставки в АВЛ-дерево

Описываются два алгоритма вставки элемента в АВЛ-дерево. Первый рекурсивный алгоритм является классическим. Второй, нерекурсивный алгоритм, ранее был представлен лишь в виде императивной программы [2, 15].

4.1. Рекурсивный алгоритм

Гиперфункция AVLinsert реализует вставку значения ainfo с ключом akey в АВЛ-дерево tree. Выход гиперфункции #plus1 реализуется в случае, когда после вставки высота дерева

tree увеличивается на 1; выход #same соответствует случаю, когда высота дерева остается прежней. Наличие «*» у аргумента tree означает, что tree является модифицируемой переменной, т.е. является результатом, причем на обеих ветвях гиперфункции AVLinsert.

```
formula Q_insert(Tree tree, tree′, Tkey akey, Tinfo ainfo) =
        ∀ Tkey k, Tinfo x. ( isin(k, x, tree′) ≡ k = akey & x = ainfo ∨ isin(k, x, tree));
hyper AVLinsert(AVLtree tree*, Tkey akey, Tinfo ainfo: #plus1 : #same)
pre plus1: heigh(tree′) == heigh(tree) +1
pre same: heigh(tree′) == heigh(tree)
post  Q_insert(tree, tree′, akey, ainfo)
{  if (tree == leaf) { tree′ = node(akey, ainfo, 0, leaf, leaf) #plus1 }
elsif (tree.key > akey) {
        AVLinsert(tree.left*, akey, ainfo:  : #same);
        switch (tree.balance) {
           case 1:  tree.balance = 0
           case 0:  tree.balance = -1 #plus1
           case -1: RotateRight(tree*)
        }
} elseif (tree.key < akey) {
        AVLinsert(tree.right*, akey, ainfo:  : #same);
        switch (tree.balance) {
           case -1: tree.balance = 0
           case 0:  tree.balance = 1 #plus1
           case 1:  RotateLeft(tree*)
        }
} else  tree.info = ainfo;
#same
};
```

Поясним некоторые правила для гиперфункций. Если исполнение рекурсивного вызова AVLinsert завершается второй ветвью, то и программа AVLinsert завершается второй ветвью. Если исполнение вызова AVLinsert завершается первой ветвью, то далее исполняется следующий оператор после вызова, поскольку в позиции результатов первой ветви нет оператора перехода.

Если высота левого поддерева увеличивается после срабатывания первого рекурсивного вызова AVLinsert (что соответствует первому выходу гиперфункции), поле tree.balance следует уменьшить на единицу. Если при этом получим tree.balance=-2, реализуется ротация дерева вправо, показанная на рис.1а и 1б. В результате получим правильное АВЛ-дерево, содержащее то же множество вершин, что и дерево до ротации.

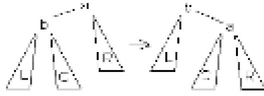


Рис 1а

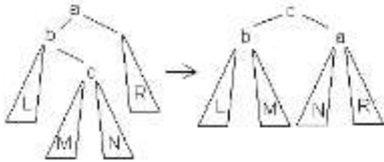


Рис 1б

Ротация на рис1.а реализуется при условии, что высота поддерева L больше, чем высота поддерева C. В противном случае проводится ротация, показанная на рис 1б.

Реализация ротации представлена предикатом:

```
formula eq(Tree tree, tree’) =∀ Tkey k, Tinfo x. isin(k, x, tree) ≡ isin(k, x, tree’);
pred RotateRight(Tree tree: AVLtree tree’)
    pre  tree != leaf & isAVL(tree.left) & isAVL(tree.right) &
          heigh(tree.left) +2 = heigh(tree.right)
    post eq(tree, tree’) & isAVL(tree’)
{   AVLtree L = tree.left;
    if (L.balance == -1)
       tree’ = L with (right: tree with (balance: 0, left: L.right))
    else  {
       AVLtree LR = L.right;
       tree’ = LR with (  left: L with (balance: (LR.balance=-1)? 1 : 0,
                                             right: LR.left),
                         right: tree with (balance: (LR.balance=1)? -1 : 0,
                                                 left: LR.right));
    };
    tree.balance = 0
};
```

Алгоритм ротации для случая, когда значение ainfo с ключом akey вставляется в правое поддерево, аналогичен представленному выше алгоритму для левого поддерева. Соответствующие ротации показаны на рис.2а и 2б.

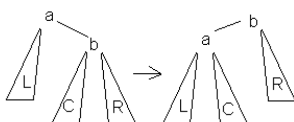


Рис 2а

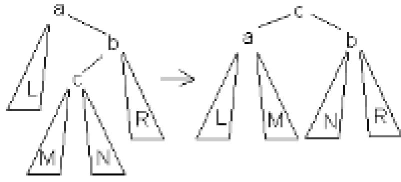


Рис 2б

```
pred RotateLeft(Tree tree: AVLtree tree')
    pre  tree != leaf & isAVL(tree.left) & isAVL(tree.right) &
         heigh(tree.left) = heigh(tree.right) +2
    post eq(tree, tree') & isAVL(tree')
{   AVLtree R = tree.right;
    if (R.balance == 1)
       tree' = R with (left: tree with (balance: 0, right: R.left))
    else {
       AVLtree RL = R.left;
       tree' = RL with ( left: tree with (balance: (RL.balance=1)? -1 : 0,
                                          right: RL.left),
                          right: R with (balance: (RL.balance=-1)? 1 : 0,
                                          left: RL.right));
    };
    tree.balance = 0
}
```

4.2. Нерекурсивный алгоритм

Алгоритм реализует вставку элемента ainfo с ключом akey в дерево N. Если в дереве присутствует вершина с ключом akey, то существующий элемент заменяется на ainfo, при этом реализуется выход гиперфункции #replace. В противном случае в дерево вставляется новый элемент с выходом #new.

Алгоритм реализуется следующим образом. Находится путь q до листа дерева N, куда надо вставить новую вершину, чтобы сохранить упорядоченность по ключам (отношение isord). Дополнительно определяется путь y, являющийся начальной частью пути q, до вершины с ненулевым значением поля balance при условии, что все вершины далее по пути q имеют нулевой balance. Нетрудно показать, что достаточно провести изменения лишь на отрезке пути от конца y до конца q, а остальная часть дерева останется неизменной. На втором шаге корректируется поле balance на найденном отрезке пути. Наконец, в случае, когда для вершины, идентифицируемой путем y, скорректированное поле balance имеет значение -2 или +2, проводится соответствующая ротация дерева в позиции y.

```
hyper AVLinsert1(AVLtree N*, Tkey akey, Tinfo ainfo: #new : #replace)
    pre new: ∀ Tinfo x. ¬ isin(akey, x, tree)
    post Q_insert(tree, tree′, akey, ainfo)
{  Search(N, akey, ainfo: _Tree y, q: N′ #replace);
   N.q = node(akey, ainfo, 0, leaf, leaf);
   updateBalance(N*, y, q);
   if (N.y.balance == -2) RotateRight (N.y*)
   elseif (N.y.balance == 2) RotateLeft(N.y*);
   #new
};
```

Гиперфункция Search определяет путь q до листа для вставки новой вершины и подпуть y до минимального поддерева, в котором надо провести балансировку, если только не обнаружится вершина с ключом akey, в случае чего реализуется выход #replace.

```
hyper Search(AVLtree N, Tkey akey, Tinfo ainfo: _Tree y, q #new : N′ #replace)
    pre new: ∀ Tinfo x. ¬ isin(akey, x, tree)
    post new: Qsearch(N, akey, y, q)
    post replace: ∃ _Tree r. N.r.key = akey & N′ = N with (r: N.r with (info: ainfo))
{  Search1(N, akey, ainfo, nil, nil: y, q #new: N′#replace) }
```

Приведенное определение есть сведение к более общей программе Search1, в которой дополнительные два параметра фиксируют начальные значения пустых путей для y и q. Отметим, что при y = nil поле balance для корневой вершины N может оказаться нулевым.

Постусловие для выхода replace фиксирует, что в дереве N есть вершина с ключом akey и в итоговом дереве N′ отличается от N заменой поля info.

Постусловие для выхода new определяет условия на пути q и y.

```
formula Qsearch(Tree N, Tkey akey, _Tree y, q) =
        PSearch(N, akey, y, q) & N.q == leaf;
```

Предикат PSearch используется в качестве предусловия для программы Search1. Второй конъюнкт постулирует, что путь q достигает листа дерева N.

```
formula Psearch(Tree N, Tkey akey, _Tree y, q) =
        valid(q, N) & valid(y, N) &
        (N.y.balance !=0 ∨ y == nil) & ordered(N, akey, q) &
        ∃ _Tree r. q == y + r & ZeroBal(N.y, r);
```

Утверждается, что пути q и y являются корректными, путь q соответствует порядку ключей (предикат ordered), путь y либо пустой, либо заканчивается на вершине с ненулевым полем balance, путь y является начальной частью пути q, причем ниже находятся вершины с нулевым полем balance.

```
formula ordered(Tree N, Tkey akey, _Tree q) =
        q == nil? true : fiord(q.car, N.key, akey) & ordered(N.(q.car), akey, q.cdr)′
formula fiord(__Tree d, Tkey k, akey) = d == left? akey < k : k < akey;
```

В предикате ordered утверждается, что путь q реализуется движением по дереву N в соответствии с порядком ключей, что гарантирует правильную позицию в дереве для вставки новой вершины.

```
formula ZeroBal(Tree B, _Tree r) = B == leaf ∨ ZeroBal1(B.(r.car), r.cdr);
formula ZeroBal1(Tree B, _Tree r) =
      B == leaf ∨ r = nil ∨ ∃ Tree B1 == B.(r.car). B1.balance == 0 & ZeroBal1(B1, r.cdr);
```

В предикате ZeroBal утверждается, что все вершины на пути r, кроме, возможно, начальной, имеют поле balance = 0.

Программа Search1 строит пути q′ и y′ в предположении, что их начальная часть (q и y) уже построены. Алгоритм реализуется разбором случаев для вершины N.q на пути q.

```
hyper Search1(AVLtree N, Tkey akey, Tinfo ainfo, _Tree y, q:
                                              _Tree y′, q′ #new : N′ #replace)
     pre PSearch(N, akey, y, q)
     pre new: ∀ Tinfo x. ¬ isin(akey, x, tree)
     post new: Qsearch(N, akey, y, q)
{  if (N.q == leaf) #new;
if (N.q.balance !=0 ) y = q;
if (N.q.key > akey) Search1(N, akey, ainfo, y, q.left: y′, q′ #new: N′ #replace)
elsif (N.q.key < akey) Search1(N, akey, ainfo, y, q.right: y′, q′ #new: N′ #replace)
else { N.q.info = ainfo #replace}
};
```

Программа updateBalance модифицирует поле balance для всех вершин на пути от y до q исключая лист в конце пути q. В итоговом дереве поле balance является корректным, т.е. соответствует предикату withbal.

```
pred updateBalance(Tree N, Tkey akey, _Tree y, q : Tree N′)
pre isAVL(N with (q: leaf)) & isord(N)
post withbal(N′)
{  if (y != q) {
        if (N.y.key > akey) {N.y.balance--; updateBalance(N, akey, y.left, q)}
        else { N.y.balance++; updateBalance(N, akey, y.right, q)}
}
};
```

5. Трансформация операций с деревьями

Определим сначала трансформацию рекурсивного алгоритма. Сначала проводятся очевидные склеивания переменных типа tree′ → tree. В программах RotateRight и

RotateLeft декомпозируются иерархические операции модификации: каждая вложенная операция модификации выносится перед оператором в форме X = X **with** (…). Подобное вынесение корректно, если X далее нигде не используется, т.е. не является живой [3]; в противном случае необходимо будет сохранить значение X в дополнительной рабочей переменной. Декомпозируем модификации для программы RotateRight.

```
pred RotateRight(Tree tree: AVLtree tree)
{   Tree L = tree.left;
    if (L.balance == -1) {
       tree = tree with (balance: 0, left: L.right);
       L = L with (right: tree);
       tree = L
    } else  {
       Tree LR = L.right;
       tree = tree with (balance: (LR.balance=1)? -1 : 0, left: LR.right);
       L = L with (balance: (LR.balance=-1)? 1 : 0, right: LR.left);
       LR = LR with (  left: L, right: tree);
       tree = LR;
    };
    tree.balance = 0
};
```

Далее реализуется замена операторов вида X = X **with** (…) на присваивания отдельным полям.

```
pred RotateRight(Tree tree: AVLtree tree)
{   Tree L = tree.left;
    if (L.balance == -1) {
       tree.balance = 0; tree.left =  L.right;
       L.right = tree;
       tree = L
    } else  {
       Tree LR = L.right;
       tree.balance = (LR.balance=1)? -1 : 0;   tree.left = LR.right;
       L.balance = (LR.balance=-1)? 1 : 0;    L.right = LR.left;
       LR.left = L; LR.right = tree;
       tree = LR;
    };
    tree.balance = 0
};
```

Кодирование алгебраического типа Tree реализуется следующим образом. Значением типа дерево является указатель (типа TREE) на корневую вершину дерева. Лист дерева кодируется нулевым указателем. Тип вершины кодируется структурой типа Tree, определяющей поля конструктора node. Правое и левое поддеревья вершины представляются указателями на поддеревья.

```
type TREE = Tree*;
type Tree = struct (Tkey key, Tinfo info, BAL balance, TREE left, right);
```

Определим трансформации типов и конструкций в соответствии с данным способом кодирования алгебраического типа дерева:

```
Tree → TREE
leaf → null
N == leaf → N == null;
N.right →   N->right
```

Переменная tree в программе AVLinsert является аргументом и результатом. Вместо подстановки результатом используется подстановка через указатель. Поэтому используется переменная trEE типа TREE*. Предполагается, что программы RotateRight и RotateLeft открыто подставляются на место вызовов. При этом присваивания tree = L и tree = LR в RotateRight должны быть заменены на trEE = &L и trEE = &LR.

```
hyper AVLinsert(TREE* trEE, Tkey akey, Tinfo ainfo: #plus1 : #same)
{  TREE tree = trEE*;
   if (tree == null) { tree = node(akey, ainfo, 0, null, null) #plus1 }
   elseif (tree->key > akey) {
        AVLinsert(&(tree->left), akey, ainfo:  : #same);
        switch (tree->balance) {
           case 1:  tree->balance = 0
           case 0:  tree->balance = -1 #plus1
           case -1: RotateRight(tree)
        }
   } elseif (tree->key < akey) {
        AVLinsert(&(tree->right), akey, ainfo:  : #same);
        switch (tree->balance) {
           case -1: tree->balance = 0
           case 0:  tree->balance = 1 #plus1
           case 1:  RotateLeft(tree)
        }
   } else  tree->info = ainfo;
   #same
};
```

Поскольку вызовы AVLinsert нельзя подставить открыто, применяется общий способ реализации выходов гиперфункции через аргумент – переменную типа LABEL. Один из выходов гиперфункции, в нашем случае, это выход #plus1, можно реализовать как обычный возврат из процедуры. Самый внешний вызов вида AVLinsert(N, ke, inf : N′: N′), определяющий выход на следующий оператор после вызова для обеих ветвей гиперфункции, реализуется следующим образом:

```
AVLinsert(&N, ke, inf , SAME); SAME: ;
```

Отметим, что оператор перехода #same реализует переход непосредственно на метку SAME, минуя всю иерархию рекурсивных вызовов. Очевидно, что использование гиперфункции вместо результата типа **bool** дает выигрыш в эффективности. Процедура AVLinsert, реализующая выходы гиперфункции, представлена ниже.

```
AVLinsert(TREE* trEE, Tkey akey, Tinfo ainfo, LABEL same)
{  TREE tree = trEE*;
   if (tree == null) { tree = node(akey, ainfo, 0, null, null); return }
   elseif (tree->key > akey) {
        AVLinsert(&(tree->left), akey, ainfo, same);
        switch (tree->balance) {
           case 1:  tree->balance = 0
           case 0:  { tree->balance = -1 ; return }
           case -1: RotateRight(tree)
        }
   } elseif (tree->key < akey) {
        AVLinsert(&(tree->right), akey, ainfo, same);
        switch (tree->balance) {
           case -1: tree->balance = 0
           case 0:  { tree->balance = 1 ; return }
           case 1:  RotateLeft(tree)
        }
   } else  tree->info = ainfo;
   #same
};
```

Трансформация программы RotateRight, подставляемой в AVLinsert, представлена ниже.

```
pred RotateRight(TREE tree: TREE tree)
{    TREE L = tree->left;
     if (L->balance == -1) {
        tree->balance = 0; tree->left =  L->right;
        L->right = tree;
        trEE = &L
     } else  {
        AVLtree LR = L->right;
        tree->balance = (LR->balance=1)? -1 : 0;   tree->left = LR->right;
        L->balance = (LR->balance=-1)? 1 : 0;    L->right = LR->left;
        LR->left = L; LR->right = tree;
        trEE = &LR;
     };
     tree->balance = 0
};
```

Далее представим трансформацию нерекурсивного алгоритма AVLinsert1. Сначала заменим хвостовую рекурсию циклом в программах Search1 и updateBalance.

```
hyper Search1(AVLtree N, Tkey akey, Tinfo ainfo, _Tree y, q:
                                              _Tree y′, q′ #new : N′ #replace) {
   for(;;) {
        if (N.q == leaf) #new;
        if (N.q.balance !=0 ) y = q;
        if (N.q.key > akey) q = q.left
        elsif (N.q.key < akey) q = q.right
        else { N.q.info = ainfo #replace}
   }
};

pred updateBalance(Tree N, Tkey akey, _Tree y, q : Tree N′) {
   for(;;) {
        if (y != q) {
              if (N.y.key > akey) {N.y.balance--; y = y.left}
              else { N.y.balance++; y = y.right}
        }
   }
};
```

Подставим программу Search1 в Search.

```
hyper Search(AVLtree N, Tkey akey, Tinfo ainfo: _Tree y, q #new : N′ #replace)
{    _Tree y = nil, q = nil;
     for(;;) {
        if (N.q == leaf) #new;
        if (N.q.balance !=0 ) y = q;
        if (N.q.key > akey) q = q.left
        elsif (N.q.key < akey) q = q.right
        else { N.q.info = ainfo #replace}
     }
};
```

Подставим программы Search и updateBalance в AVLinsert1.

```
hyper AVLinsert1(AVLtree N*, Tkey akey, Tinfo ainfo: #new : #replace)
{  _Tree y = nil, q = nil;
   for(;;) {
      if (N.q == leaf) #new1;
      if (N.q.balance !=0 ) y = q;
      if (N.q.key > akey) q = q.left
      elsif (N.q.key < akey) q = q.right
      else { N.q.info = ainfo #replace}
   };
   new1:
   N.q = node(akey, ainfo, 0, leaf, leaf);
   for( ;y != q; ) {
          if (N.y.key > akey) {N.y.balance--; y = y.left}
          else { N.y.balance++; y = y.right}
   }
   if (N.y.balance == -2) RotateRight (N.y*)
   elsif (N.y.balance == 2) RotateLeft(N.y*);
   #new
};
```

Переход по метке #new1 можно заменить на **break**.

Будем считать, что любой путь, значение типа _Tree, строится только для одного объекта типа Tree. Путь кодируется указателем на поле вершины дерева. Это поле соответствует концу пути в дереве. Пустой путь кодируется указателем на переменную, значением которого является дерево. Путь кодируется значением типа PTREE.

```
type PTREE = TREE*;  // тип переменных q и y
```

Реализуются трансформации:

```
_Tree → PTREE;
N.q → q*;
N.y → y*;
N.q == leaf → q* == null
N.q.key → q*->key;
q = q.left → q = &(q*->left);
```

Применение трансформаций дает следующую программу.

```
hyper AVLinsert1(AVLtree N*, Tkey akey, Tinfo ainfo: #new : #replace)
{  PTREE y = &N, q = &N;
   for(;;) {
      if (q* == null) break;
      if (q*->balance !=0 ) y = q;
      if (q*->key > akey) q = &(q*->left)
      elsif (q*->key < akey) q = &(q*->right)
      else { q*->info = ainfo #replace}
   };
   q* = node(akey, ainfo, 0, leaf, leaf);
   for( ;y != q; ) {
         if (y*->key > akey) {y*->balance--; y = &(y*->left) }
         else { y*->balance++; y = &(y*->right }
   }
   if (y*->balance == -2) RotateRight (y*)
   elsif (y*->balance == 2) RotateLeft(y*);
   #new
};
```

Вместо двойного указателя (q или y) можно использовать одинарный. В использующих позициях переменных q и y применим трансформации:

$$q^* \to Nq;$$
$$y^* \to Ny;$$

Однако после присваивания переменной q или y необходим синхронный пересчет значения переменной. Итоговая программа представлена ниже.

```
hyper AVLinsert1(AVLtree N*, Tkey akey, Tinfo ainfo: #new : #replace)
{  PTREE y = &N, q = &N;
   TREE Nq = N; // = q*
   for(;;) {
      if (Nq == null) break;
      if (Nq->balance !=0 ) y = q;
      if (Nq->key > akey) q = &(Nq->left)
      elsif (Nq->key < akey) q = &(Nq ->right)
      else { Nq->info = ainfo #replace};
      Nq = q*;
   };
   q* = node(akey, ainfo, 0, leaf, leaf);
   TREE Ny = y*;
   for( ;y != q; ) {
         if (Ny->key > akey) { Ny->balance--; y = &(Ny->left) }
         else { Ny->balance++; y = &(Ny->right };
         Ny = y*;
   }
   if (Ny->balance == -2) RotateRight (y*)
   elsif (Ny->balance == 2) RotateLeft(y*);
   #new
};
```

Заключение

Предпосылкой появления данной работы стала дискуссия на форуме [1] о том, какая из программ вставки в АВЛ-дерево лучше: на языке Оберон или графическом языке Дракон [5]. Эргономические методы, применяемые в языке Дракон, существенно улучшают восприятие программы. Тем не менее, программа не выглядит проще. Причина – исходная сложность императивной программы. Методы предикатного программирования: использование рекурсивных программ вместо циклов, алгебраических типов вместо указателей и др. позволяют существенно снизить сложность программы по сравнению с аналогичной императивной программой, в частности с программами на форуме [1].

Доступ к вершине дерева реализован через указатель на поле (в некоторой вершине), в котором хранится ссылка на требуемую вершину. При передаче через параметр программы возникает двойной указатель. В описании библиотеки libavl [15] используется однократный указатель, однако при этом дополнительно поддерживается указатель на предыдущую вершину-отца. Как следствие, алгоритм получается более громоздким и менее эффективным в сравнении с приведенным в настоящей работе. В нашей версии, тем не менее, для каждого двойного указателя заводится соответствующий одинарный. Реализация такой техники в трансформациях может оказаться нетривиальной. Поэтому в начальном релизе следует ограничиться только двойным указателем.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. АВЛ-дерево. Алгоритм добавления вершины. [Электронный ресурс]. URL: **http://forum.oberoncore.ru/viewtopic.php?f=78&t=4003**
2. Википедия. АВЛ-дерево. [Электронный ресурс]. URL: http://ru.wikipedia.org/wiki/%D0%90%D0%92%D0%9B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE
3. Каблуков И. В. Реализация оптимизирующих трансформаций предикатных программ // XIV Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям. Томск, 2013. 7с. URL: http://conf.nsc.ru/files/conferences/ym2013/fulltext/175069/177104/Опт.%20трансформации.pdf
4. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 52с. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
5. Паронджанов В. Д. Язык ДРАКОН. Краткое описание. М., 2009. 124 с.
6. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.

7. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
8. Шелехов В.И. Разработка автоматных программ на базе определения требований // Системная информатика, №4, 2014. ИСИ СО РАН, Новосибирск. С. 1-29. URL: http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf
9. Шелехов В.И. Предикатное программирование. Учебное пособие. Новосибирск: НГУ, 2009. 109с.
10. AVL Tree in Haskell. [Электронный ресурс]. URL: **https://gist.github.com/gerard/109729**
11. Clochard M. Automatically Verified Implementation of Data Structures Based on AVL Trees // 6th Working Conference on Verified Software: Theories, Tools, and Experiments, 2014. P. 167-180.
12. Cooke D. E., Rushton J. N. Taking Parnas's Principles to the Next Level: Declarative Language Design. *Computer*, 2009, vol. 42, no. 9. P. 56-63.
13. Hettler R., Nazareth D., Regensburger F., Slotosch O. AVL trees revisited: A case study in Spectrum. LCNS, vol. 1009, 1995. P. 128-147.
14. Meyer B. Towards a Calculus of Object Programs // Patterns, Programming and Everything, Judith Bishop Festschrift, eds. Karin Breitman and Nigel Horspool, Springer-Verlag, 2012. P. 91-128.
15. Pfaff B. GNU libavl 2012. An Introduction to Binary Search Trees and Balanced Trees. URL: ftp://ftp.gnu.org/pub/gnu/avl/avl-2.0.2.pdf.gz
16. Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. Vol. 45, No. 7. P. 421–427.

УДК 004.43, 004.27

Вычисления на сетях Слепцова

Зайцев Д.А. (Международный гуманитарный университет, Одесса, Украина)

Выполнен обзор работ, формирующих теоретические основы вычислений на сетях Слепцова и представляющих особенности рисования, компиляции и компоновки программ на языке сетей Слепцова, а также массово параллельные архитектуры вычисляющей памяти для реализации процессоров сетей Слецова. Сеть Петри выполняется экспоненциально медленнее и является частным случаем сети Слепцова. Рассмотрена универсальная сеть Слепцова, содержащая 13 позиций и 26 переходов, представляющая собой прототип процессора сетей Слепцова. Приведены примеры программ на языке сетей Слепцова для эффективного умножения, RSA шифрования/дешифрования, вычисления функции нечёткой логики и решения уравнения Лапласа. Преимуществами вычислений на сетях Слепцова являются наглядный графический язык, сохранение естественного параллелизма предметной области, мелкая грануляция параллельных вычислений, формальные методы верификации параллельных программ, быстрые массово-параллельные архитектуры, реализующие модель вычислений.

Ключевые слова: *сеть Слепцова, сеть Петри, машина Тьюринга, клеточный автомат, параллельные вычисления, универсальные вычислительные модели*

1. Введение

В последнее время многие исследователи предлагают новые модели гипер-вычислений, такие как квантовые вычисления, вычисления на мембранах клеток, на импульсных нейронах и ДНК [10], способные преодолеть препятствие неподдающихся задач. Сети Петри известны более полувека как модель параллельных систем и вычислений [1,4], однако их универсальные расширения выполняются экспоненциально медленнее машины Тьюринга, особенно при реализации арифметических операций. Концепция сети Слепцова, предложенная четверть века назад, недавно обрела своё второе рождение [13] благодаря своей способности к быстрой реализации основных арифметических операций. Запуск перехода в нескольких экземплярах на шаге приводит к универсальным структурам, которые выполняются за полиномиальное время [16]. В вычислениях на сетях Слепцова программа, написанная на языке сетей Слепцова с сохранением естественного параллелизма предметной

области, выполняется на процессоре сетей Слепцова, который реализует параллельное срабатывание переходов в нескольких экземплярах, обеспечивая ультра-быстродействие.

Концепция алгоритма была впервые формализована Аланом Тьюрингом в 1936 году в форме абстрактной машины, которую традиционно называют машиной Тьюринга. Универсальную машину Тьюринга, которая выполняет заданную машину Тьюринга, рассматривают как прототип традиционного компьютера. Кроме машины Тьюринга появились другие универсальные модели вычислений: рекурсивные функции Клини, нормальные алгорифмы Маркова, системы перезаписи тегов Поста, регистровые машины Минского и другие. Разнообразие моделей объясняется специфическими требованиями различных областей применения. Новые модели задействуют возможности массивно-параллельных вычислений, присущие таким простым структурам как элементарные клеточные автоматы, универсальность которых была доказана Мэтью Куком в 2004 году. Кроме того, Турлок Нири и Демием Вудз построили в 2008 году универсальные машины Тьюринга малого размера, которые выполняются в полиномиальное время. Однако способ программирования клеточных автоматов, описанный Мэтью Куком, не задействует массово-параллельные вычисления. Концепция сети Слепцова [13] выправляет основной недостаток сети Петри [4] состоящий в инкрементном характере вычислений, что делает вычисления на сетях Слепцова [13,14] перспективным подходом для достижения ультра-быстродействия параллельных вычислений. В статье [13] представлен обзор работ, которые используют сети Слепцова (сети позиций/переходов с многоканальными переходами или множественной стратегией запуска переходов).

2. Определение сети Слепцова

Сеть Слепцова – это двудольный ориентированный мультиграф, дополненный динамическим процессом [13]. Обозначим сеть Слепцова как $N=(P,T,W,R,\mu_0)$, где P и T это непересекающиеся множества вершин называемых *позиции* и *переходы* соответственно, отображение F описывает *дуги* соединяющие вершины, отношение R представляет приоритеты переходов, и отображение μ_0 задаёт начальное состояние (*маркировку*).

Отображение $W:(P\times T)\to \mathrm{N}\bigcup\{-1\},(T\times P)\to \mathrm{N}$ задаёт дуги, их тип и кратность; нулевое значение соответствует отсутствию дуги, положительное значение – *регулярной дуге* с указанной кратностью, а минус один – *ингибиторной дуге* которая проверяет маркировку позиции на ноль. N обозначает множество неотрицательных целых чисел. Чтобы избежать

вложенных индексов, обозначим $w^{-}_{j,i} = w(p_j, t_i)$ и $w^{-}_{i,j} = w(t_i, p_j)$. Отображение $\mu : P \to \mathrm{N}$ представляет маркировку позиций.

В графической форме позиции изображают в виде кругов, а переходы – в виде прямоугольников (квадратов). Ингибиторную дугу представляют небольшим полым кругом на её конце, а маленький заполненный круг обозначает аббревиатуру цикла. Кратность регулярных дуг большая, чем единица, подписывается над дугой, и маркировка позиции большая нуля записывается внутри позиции. Примеры сетей Слепцова, реализующих основные арифметические и логические операции, приведены на Рис. 6, который будет обсуждаться в последующих разделах.

Чтобы оценить *кратность возбуждения* на каждой входящей дуге позиции, введём следующую вспомогательную операцию

$$x \succ y = \begin{cases} x / y, if \;\; y > 0 \\ 0, ify = -1, x > 0, \\ \infty, ify = -1, x = 0. \end{cases}$$

Во избежание аномалий с бесконечным числом экземпляров перехода, запретим в дальнейшем использование переходов, не содержащих входящих регулярных дуг.

Поведение (динамика) сети Слепцова может быть описана соответствующим уравнением состояний аналогично [6]. Настоящая работа рассматривает поведение сети как результат применения следующего *правила запуска перехода*:

– количество экземпляров перехода t_i возбужденных на текущем шаге равняется

$$v_i = v(t_i) = \min_j (\mu_j \succ w^{-}_{j,i}), 1 \le j \le m, w^{-}_{j,i} \ne 0$$

– когда переход $t_i, v_i > 0$ срабатывает, при $u_i \le v_i$, он
  - ✓ извлекает $u_i \cdot w^{-}_{j,i}$ фишек из каждой своей входной позиции p_j для регулярных дуг $w^{-}_{j,i} > 0$;
  - ✓ добавляет $u_i \cdot w^{+}_{i,k}$ фишек в каждую свою выходную позицию p_k, $w^{+}_{i,k} > 0$;

– сеть останавливается, если отсутствуют возбужденные переходы.

Когда переход, имеющий единственную регулярную входящую дугу кратности a из позиции p и единственную исходящую дугу кратности b в позицию p', срабатывает, при $u_i = v_i$, он осуществляет следующие вычисления: $\mu(p) = \mu(p) \bmod a$; $\mu(p') = \mu(p') + b \cdot (\mu(p) \; div \; a)$. А именно, он реализует деление на a с остатком и умножение на b. Выбирая либо a, либо b равными единице, получаем чистое умножение либо чистое деление соответственно.

В сети Петри только один переход срабатывает на шаге, в то время как в сети Салвицки [8] (или синхронной сети в соответствии с [4]) на шаге срабатывает максимальное множество возбужденных переходов. Однако в обеих сетях (Петри и Салвицки) лишь один экземпляр каждого перехода срабатывает на шаге.

Переход может восприниматься как некоторое виртуальное событие (действие). Количество реально запущенных действий зависит лишь от количества доступных ресурсов, представленных входными позициями перехода. Зачем же необходимо ограничивать количество экземпляров перехода до единицы, когда имеющиеся в наличии ресурсы позволяют запустить действия одновременно? Более того, классический последовательный порядок запуска переходов может быть получен как частный случай путем присоединения к каждому переходу посредством цикла отдельной позиции, содержащей одну фишку.

Известны разнообразные расширения сетей Петри, такие как приоритетные, ингибиторные, временные, нагруженные (раскрашенные), иерархические и вложенные сети Петри [4,5,9,11]. Иногда они лишь усложняют восприятие основной модели. Наша цель состоит в получении гипер-производительности за счёт минимальной модификации, которая состоит во множественной стратегии запуска перехода. Что касается других моделей близких к сетям Петри, следует отметить системы перезаписи множеств, а также системы сложения и замещения векторов. Кроме того, для изучения протоколов используются последовательные взаимодействующие процессы Хоара, близкие к сетям Петри и позволяющие взаимные преобразования [11]. Существенным преимуществом сетей Слепцова (и сетей Петри) является графическое представление структуры и поведения систем.

3. Универсальная сеть Слепцова как процессор

Для программирования на языке сетей Слепцова (Петри, Салвицки) используем концепцию выделенных контактных позиций. Перед запуском сети загружаем исходные данные в контактные позиции, а когда сеть остановиться, извлекаем выходные данные из контактных позиций. Иногда более детальная специализация, различающая отдельные подмножества входных и выходных позиций, является удобной. В этом случае используем декомпозицию сети на кланы (функциональные подсети) [11]. Заметим, что данные различных типов следует закодировать целыми неотрицательными числами для их последующей обработки сетью.

Полнота по Тьюрингу расширенных сетей Петри подразумевает существование универсальной сети, которая выполняет произвольную заданную сеть. В последнее время была построена серия универсальных сетей малого размера [10,12,16], каждая из которых может рассматриваться как прототип процессора в вычислениях на сетях Слепцова [13]. Программа такого компьютера, представленная в форме (ингибиторной, приоритетной) сети Слепцова и закодированная в целых числах, подаётся в качестве исходных данных на процессор, который представляет собой аппаратную реализацию универсальной сети. Рис. 1 иллюстрирует описанный подход.

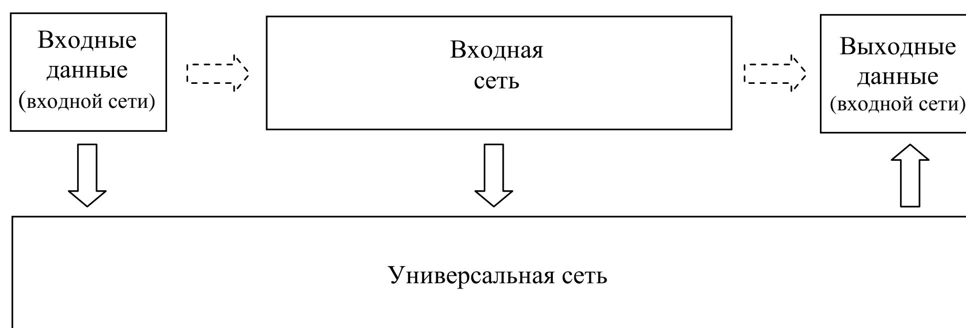


Рис. 1. Универсальная сеть как процессор

Главным препятствием на пути широкой реализации парадигмы вычислений на сетях Петри [7] является тот факт, что все известные универсальные сети малого размера выполняются в экспоненциальное время (в зависимости от числа шагов заданной сети). В статье [16] построена универсальная сеть Слепцова малого размера, которая выполняется в полиномиальное время; сеть содержит 13 позиций и 26 переходов. Техника композиции универсальной сети Слепцова [16] использует обратный поток управления и подсети умножения и деления на константу, которые в последующем являются удобным средством разработки технологии программирования на сетях Слепцова [14]. Существенная разница в вычислительной сложности между универсальными сетями Слепцова и Петри достаточно легко объяснима: во время обработки кода ленты машины Тьюринга, который представляет собой экспоненту от ширины рабочей зоны, универсальная сеть Петри извлекает одну фишку на шаге, в то время как универсальная сеть Слепцова извлекает все фишки. В результате мы приходим к быстрым арифметическим операциям, а именно умножению и делению, которые являются основными для кодирования/декодирования заданной сети.

Известным примером графической технологии программирования является р-технология [2], однако её средства не содержат формальные модели параллельных вычислений, что затрудняет верификацию параллельных программ.

В статье [7] предложена концепция вычисляющей памяти для массово параллельной реализации аппаратных процессоров сетей Слепцова, которая обеспечит ультра-производительность при выполнении программ написанных на языке сетей Слепцова. Отдельно от проблемы эффективной аппаратной реализации стоят вопросы разработки технологии программирования на сетях Слепцова сохраняющей естественный параллелизм предметной области. Несомненным преимуществом таких программ является мелкая грануляция параллельных процессов, а также возможность применения формальных методов верификации параллельных программ [1,4,11] в процессе управляемой моделью разработки.

4. Универсальная сеть Слепцова выполняющаяся за полиномиальное время

Представленная в статье [16] универсальная сеть Слепцова получена путём моделирования слабо-универсальной машины Тьюринга с 2 состояниями и 4 символами ленты, построенной Нири и Вудз и обозначенной СУМТ(2,4). Так как поведение машины Тьюринга детерминировано, используем детерминированные сети Слепцова, в которых все переходы занумерованы и на шаге срабатывает переход с наименьшим индексом в максимальном числе экземпляров. Для визуального отображения порядка (приоритета) позиций используем дуги, соединяющие переходы; если существует дуга из перехода t в переход t', это означает, что индекс перехода t должен быть меньше индекса перехода t' (меньший индекс обозначает более высокий приоритет). Особенностью программирования на сетях Слепцова является использование обратного потока управления, представленного движением нулевой маркировки; таким образом, начальная маркировка позиций потока управления равна 1. Для проверки нулевой маркировки используется ингибиторная дуга, которая не ограничивает кратность срабатывания перехода.

Исходной информацией для моделирования является функция переходов СУМТ(2,4) и кодирование состояний и символов ленты [16], заданные Табл. 1. Изначально на ленте СУМТ(2,4) записано бесконечное повторение пустых слов: $b_l = 0001$ влево и $b_r = 010001$ вправо от рабочей зоны. В соответствии с функцией кодирования ленты

$s(x_{k-1}x_{k-2}...x_0) = \sum_{i=0} s(x_i) \cdot r^i$, коды левого и правого пустых слов равняются: $s(b_l) = 167$ и $s(b_r) = 13596$.

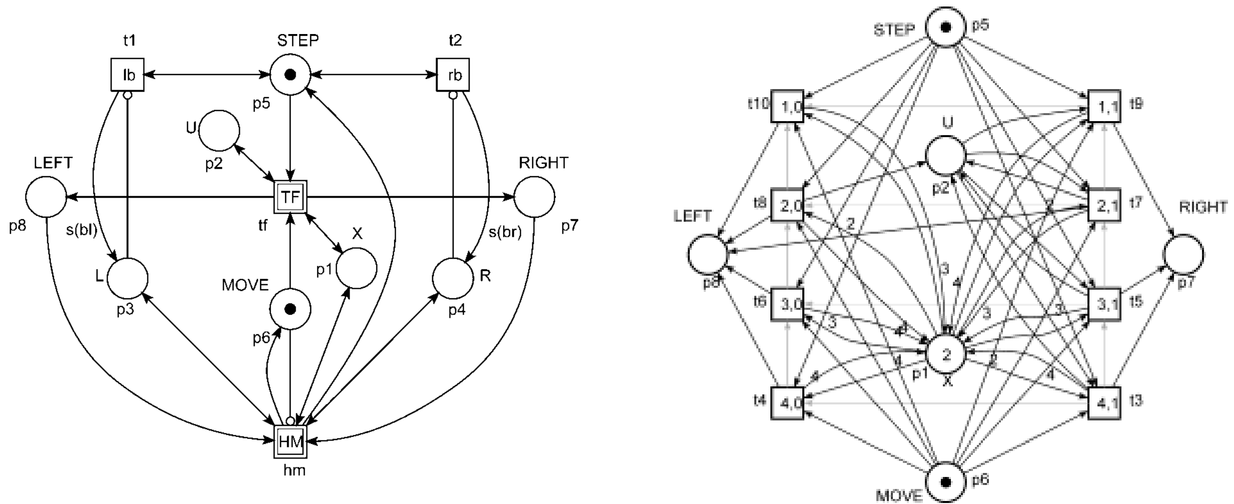
Построенная универсальная сеть Слепцова обозначена как УСС(13,26) в соответствии с количеством её позиций и переходов соответственно. Общая схема УСС(13,26) представлена на Рис. 2 а). Подсети изображены как квадраты с двойной линией границ. Для некоторых вершин кроме их номеров указаны также мнемонические имена. Использованные подсети TF и HM представлены на Рис. 2 б), в) соответственно. Вершины с одинаковыми именами (номерами) логически обозначают одну и ту же вершину и при окончательной компоновке объединяются по всем компонентам сети. Изображение полученной в результате сети выглядит довольно запутанно [16] и для её формального представления рекомендуется использовать параметрические выражения, использованные для описания бесконечных сетей [15], в частности, универсальных сетей, полученных в результате моделирования клеточных автоматов.

Таблица 1. Слабо универсальная машина Тьюринга СУТМ (2,4) и ее кодирование

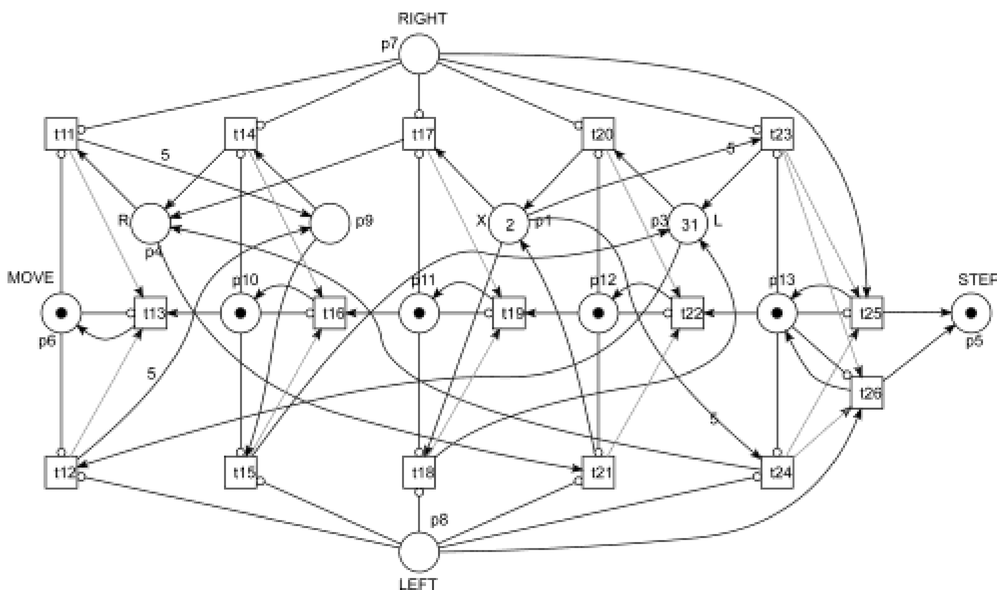
$\Sigma \setminus \Omega$		u_1	u_2
	$s(\Sigma) \setminus s(\Omega)$	0	1
0	1	3,left,0	4,right,0
1	2	4,left,1	3,left,1
0̸	3	4,left,0	1,right,1
1̸	4	4,left,0	2,right,1

Позиция U содержит код состояния $s(u)$, позиция X содержит код текущего символа ленты $s(x)$, а позиции L и R содержат коды левой и правой частей рабочей зоны ленты соответственно по отношению к текущей ячейке. При моделировании текущего шага машины, позиция STEP запускает подсеть TF, которая моделирует функцию переходов СУМТ(2,4). Подсеть TF получает коды нового состояния головки $s(u')$ и нового символа текущей ячейки ленты $s(x')$ в позициях U и X соответственно. Подсеть TF также помещает фишку в позицию RIGHT или позицию LEFT для указания правого либо левого движения головки соответственно. Фишка извлекается из позиции MOVE при завершении работы подсети TF и запускает подсеть HM, которая моделирует требуемое движение управляющей головки вдоль ленты. В конце моделирования текущего шага, фишка возвращается в

позицию STEP, что позволяет начать моделировать следующий шаг. Заметим, что позиции LEFT и RIGHT очищаются заключительными переходами подсети HM.



а) общая схема; б) подсеть функции переходов TF;



в) подсеть движений управляющей головки HM.

Рис. 2. Покомпонентное представление УСС(13,26)

Лента машины представлена позициями L, X, и R содержащими коды левой части рабочей зоны, символа текущей ячейки и правой части рабочей зоны соответственно. Движение головки влево моделируется как (a) $R'' = 5 \cdot R' + X'$, (b) $X'' = L'\%5$, (c) $L'' = L'/5$ а движение головки вправо, как (a) $L'' = 5 \cdot L' + X'$, (b) $X'' = R'\%5$, (c) $R'' = R'/5$; фактически выполняются одинаковые последовательности вычислений, в которых L и R меняются местами. HM

представляет собой объединение трёх частей, индуцированных следующими последовательностями переходов: переходы $t_{11}, t_{14}, t_{17}, t_{20}, t_{23}$ моделируют движение влево; переходы $t_{12}, t_{15}, t_{18}, t_{21}, t_{24}$ моделируют движение вправо; переходы $t_{13}, t_{16}, t_{19}, t_{22}, t_{25}, t_{26}$ представляют инверсный поток управления. Например, при движении влево: переход t_{11} реализует умножение на константу 5: $\mu(p_9) := 5 \cdot R, R := 0$; переход t_{14} перемещает результат обратно из позиции p_9 в позицию R: $R := 5 \cdot R, \mu(p_9) := 0$; переход t_{17} добавляет X к R с очисткой X: $R := 5 \cdot R + X, X := 0$; переход t_{20} подготавливает операнд для деления, перемещая L в X: $X := L, L := 0$; переход t_{23} реализует деление с остатком на константу 5: $X := L\%5, L := L/5$.

В статье [16] доказано, что УСС(13,26) моделирует СУМТ(2,4) за время $O(12 \cdot k) \approx O(k)$. Емкостная сложность оценивается как $\log_2 5^k = k \cdot \log_2 5 \approx O(k)$, где k – это число шагов СУМТ(2,4). Принимая во внимание полиномиальную сложность выполнения заданной машины Тьюринга на УСС(13,26) и полиномиальную сложность выполнения сети Слепцова на машине Тьюринга, получаем общую полиномиальную сложность представленной универсальной сети Слепцова.

Заметим, что УСС(13,26) моделирует элементарный клеточный автомат номер 110. В статье [15] построена серия сетей, непосредственно моделирующих клеточные автоматы, для получения универсальных конструкций с массовым параллелизмом выполнения.

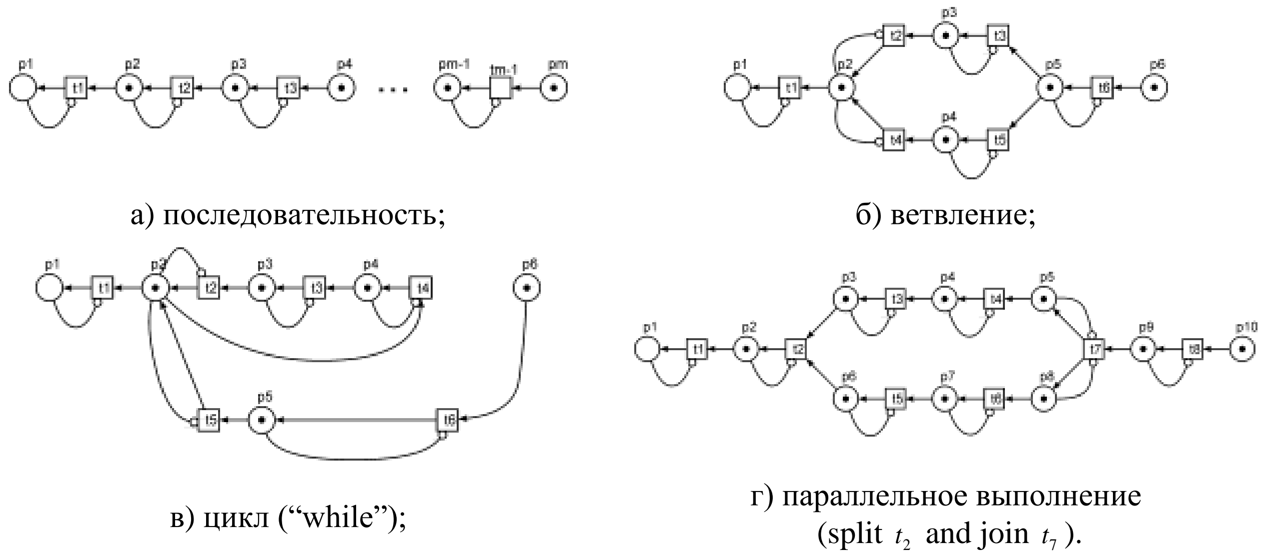
5. Принципы программирования на сетях Слепцова

Программирование на сетях Слепцова [14] представляет собой композицию данных и потоков управления, дополненную подстановкой перехода для описания иерархической структуры программы (сети). Предполагаем, что для комбинирования потока управления с подсетью, которая подставляется вместо перехода, используется специальная пара позиций: *Start* – чтобы запустить подсеть и *Finish* – чтобы индицировать ее завершение.

Потоки управления представлены в инверсной форме «с движущимся нулём» потому что нулевая маркировка легко проверяется ингибиторной дугой и не ограничивает количество возбужденных экземпляров перехода. Таким образом, изначально все позиции потока управления содержат фишку.

Для моделирования стандартных потоков управления классических языков программирования предложены шаблоны, представленные на Рис. 3 для последовательности а), ветвления б), цикла в), и параллельного выполнения г). Кроме того, произвольная подсеть

с маркировками, принадлежащими $\{0,1\}^m$, снабженная парой позиций запуска/завершения, может рассматриваться как поток управления.



а) последовательность; б) ветвление;

в) цикл (“while”); г) параллельное выполнение (split t_2 and join t_7).

Рис. 3. Стандартные шаблоны потока управления

Каждый из переходов на Рис. 3 может быть замещён (подставлен) подсетью, использующей входную позицию для запуска и выходную позицию для завершения. Кроме того, дополнительные инцидентные переходам позиции могут быть добавлены для представления требуемых входные и выходных переменных. В шаблонах с альтернативными переходами (Рис. 3 б, в) предполагается либо внешнее управление посредством переменных, содержащих условия в случае детерминированного выбора, либо недетерминированный выбор.

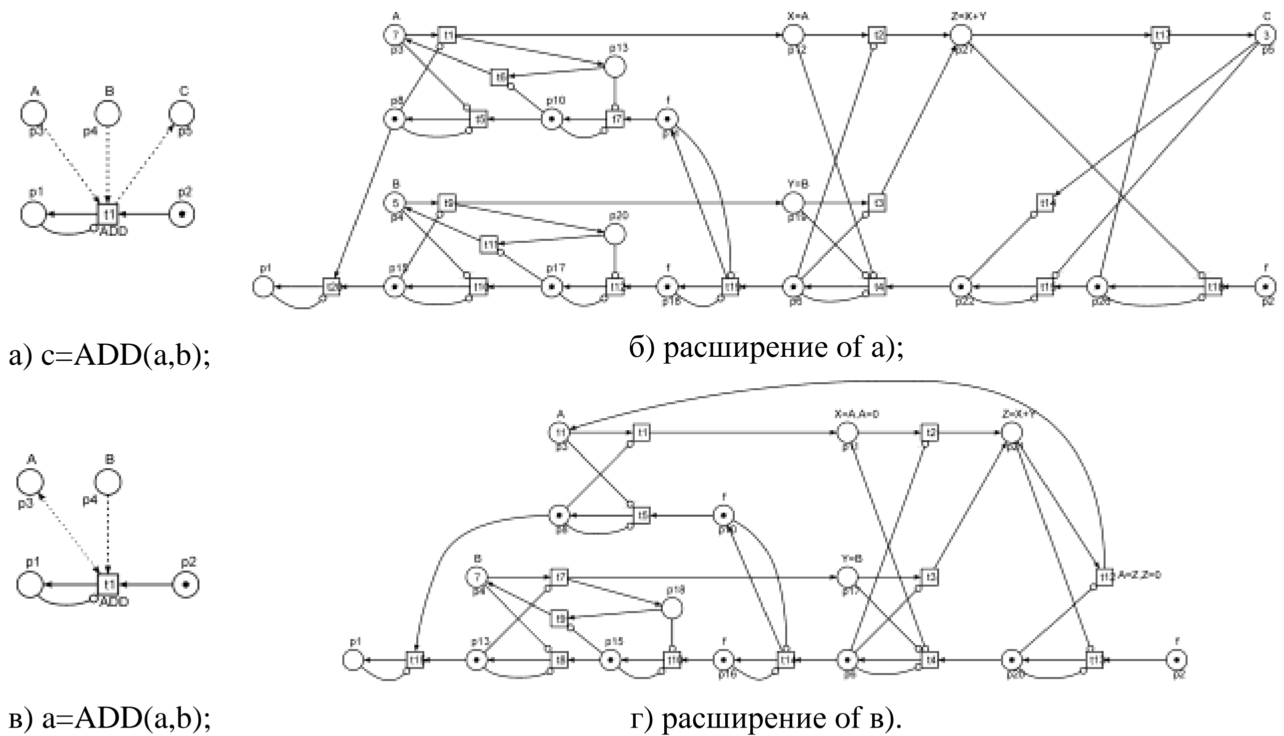
Для вычисления выражений подход управления потоками данных может быть более эффективным, и соответствующая подсеть запускается и завершается с помощью внешнего потока управления. Это вопрос грануляции вычислений: либо использовать строгую композицию подсетей, запускаемых потоком управления, либо рисовать заново сильно взаимосвязанное переплетение потоков данных и управления каждый из которых может быть чётко неразличим.

Программа, нарисованная на языке сетей Слепцова, получается путём композиции потоков управления и данных с использованием модульного подхода. Подстановка перехода модулем определяется указанием имени модуля (подсети) и соединения (отображения) его входных и выходных позиций с позициями исходной сети которые представляют переменные и (или) поток управления.

Также предполагается, что перед запуском модуля все его входные данные скопированы в его входные позиции, а после завершения работы модуля или пред его следующим запуском

все выходные данные перемещены из его выходных позиций. Для этого предложено использовать пунктирные дуги, введенные в [6] для обозначения краткого и удобного способа работы с данными.

Штриховая входящая дуга модуля обозначает подсеть COPY используемую для копирования значения входной переменной в соответствующую входную позицию модуля перед запуском его работы посредством позиции *Start*. А штриховая исходящая дуга модуля обозначает последовательность подсетей CLEAN, MOVE для замещения значения переменной результатом, полученным в соответствующей выходной позиции модуля. В случае если модуль имеет несколько входных (выходных) позиций, подсети COPY (CLEAN, MOVE) вставляются с использованием параллельного шаблона выполнения. В примере, представленном на Рис. 4 а) и б), подсеть а), содержащая штриховые дуги расширяется в соответствующую низкоуровневую сеть б).



а) c=ADD(a,b); б) расширение of а);

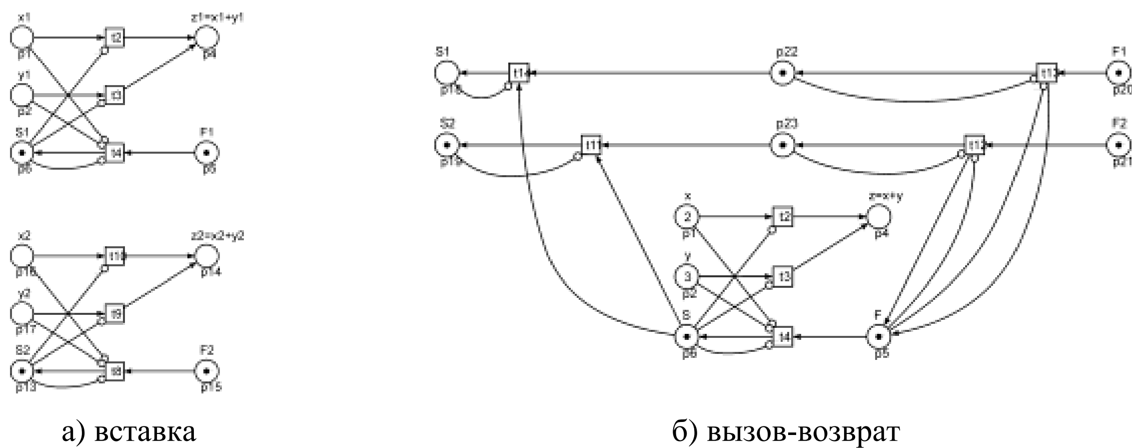
в) a=ADD(a,b); г) расширение of в).

Рис. 4. Примеры расширения пунктирных дуг

Введенные для сокращения размера вспомогательных подсетей пунктирные дуги обозначают MOVE как для входных, так и для выходных переменных; подсеть MOVE является сокращенной и обеспечивает очистку значения своей входной переменной. Таким образом, значение входной переменной не сохраняется и значение выходной переменной добавляется. Когда переменная является как входной, так и выходной, используется пунктирная двунаправленная дуга, которая расширяется как MOVE перед запуском модуля и

MOVE после завершения модуля. Соответствующий пример представлен на Рис. 4 в) и г), где подсеть в) содержащая пунктирную дугу расширяется в низкоуровневую сеть г).

Таким образом, модуль сети может рассматриваться как процедура языка программирования, чьи контактные позиции соответствуют формальным параметрам. Копирование переменных штриховыми и пунктирными дугами соответствует подстановке фактических входных параметров и извлечению фактических выходных параметров. Остается открытым вопрос о способе реализации обращения к модулю, которое может быть выполнено либо в стиле вставки полной копии подсети, либо в стиле вызов-возврат с единственной копией модуля и переключением потоков управления при вызове-возврате из различных мест. Разница между двумя указанными подходами проиллюстрирована Рис. 5 для двух обращений к модулю ADD (без рассмотрения передачи используемых данных). В то время как стиль вызова-возврата является более компактным из-за наличия единственной копии модуля, стиль вставки является более привлекательным с точки зрения параллельного исполнения [14].



а) вставка б) вызов-возврат

Рис. 5. Примеры реализации подстановки перехода для z1=ADD(x1,y2); z2=ADD(x2,y2)

Программа, представляющая собой композицию модулей, в конечном счёте компилируется в простую («плоскую») ингибиторную приоритетную сеть Слепцова, которая загружается на компьютер сетей Слепцова. Наилучшая производительность достигается в случае, когда все переходы запускаются независимо на основе их локальных условий возбуждения; для разрешения конфликтов предложено использовать арбитра, который блокирует инцидентные позиции срабатывающего перехода. Описанный способ исполнения требует рисования программ инвариантных к порядку запуска переходов.

6. Эффективная реализация арифметических и логических операций

Для формального описания операции подстановки перехода с учётом внешнего потока управления и входных/выходных переменных введена концепция *модуля* [6,13,14]. Модуль представляет собой подсеть с контактными (входными и выходными) позициями, которые представляют собой единственный способ взаимодействия с окружающим миром для замкнутого модуля либо комбинируется с использованием глобальных переменных для открытого модуля. Работа модуля контролируется парой выделенных позиций: позиция *Start* (*s*) запускает модуль и позиция *Finish* (*f*) индицирует завершение работы модуля. Извлечение фишки из первой позиции потока управления *Start* запускает движение нулевой маркировки до тех пор пока ноль ни прибудет в последнюю позицию потока управления *Finish*.

На графы потоков управления наложены следующие ограничения. Предположим, что все действия модуля контролируются его потоком управления, и ингибиторные дуги используются для запуска переходов. Таким образом, когда все позиции потока управления содержат фишку, переходы модуля отключены. В результате, перед извлечением фишки из позиции *Start*, а также после прибытия нулевой маркировки в позицию *Finish* все переходы модуля неактивны.

Для обеспечения реентерабильности (повторного входа) модуля предполагается, что когда модуль запускается, все его позиции данных, за исключением входных позиций, имеют нулевую маркировку, а когда модуль завершается, все его позиции данных, за исключением выходных позиций, имеют нулевую маркировку. На Рис. 6 представлены сети, реализующие основные операции: копирование, логические и арифметические. По сравнению со статьёй [6] подсети сокращены на одну позицию и один переход в предположении, что не разрешено использовать входную позицию *Start* для управления переходами внутри подсети, но позиция *Finish* не используется для управления внутри подсети.

Используя метод анализа и классификации всех разрешенных последовательностей переходов, использованную в [13], доказано, что каждая из подсетей, представленных на Рис. 6 реализует соответствующую операцию.

Для (ингибиторных) сетей Петри операции умножения и деления являются наиболее сложными с точки зрения времени вычисления; соответствующие подсети были изучены в статье [13]. Именно их сложность обуславливает экспоненциальную медлительность вычислений на сетях Петри. Представленные в [16] компоненты универсальной сети Слепцова эффективно реализуют умножение и деление на константу (равную 5). Рассмотрим

умножение и деление для произвольных пар заданных целых неотрицательных чисел, используя умножение и деление на константу (равную 2).

Мы выбираем простые школьные алгоритмы умножения и деления «в столбик». Лучшие известные алгоритмы умножения и деления могут также быть закодированы сетью Слепцова. В алгоритме умножения для нахождения текущей цифры множителя используется остаток от деления на два: $d = y \,\%\, 2$; затем множитель перевычисляется как $y /= 2$ для работы со следующей цифрой на следующем проходе основного цикла. Множимое умножается на два: $x *= 2$, что представляет собой его сдвиг влево. Когда текущая цифра d множителя равна 1, сдвинутое множимое добавляется к результату. Алгоритм может быть оптимизирован, чтобы избежать перевычисления, когда новое значение y равно нулю, но это приводит к усложнению сети. Алгоритм закодирован сетью Слепцова и представлен на Рис. 7.

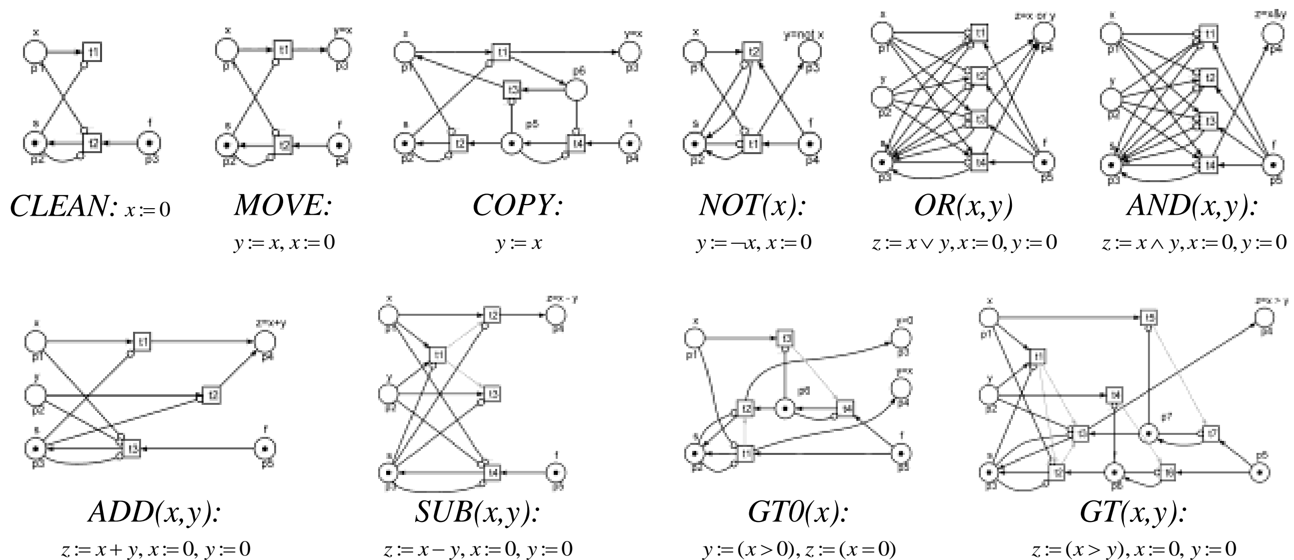


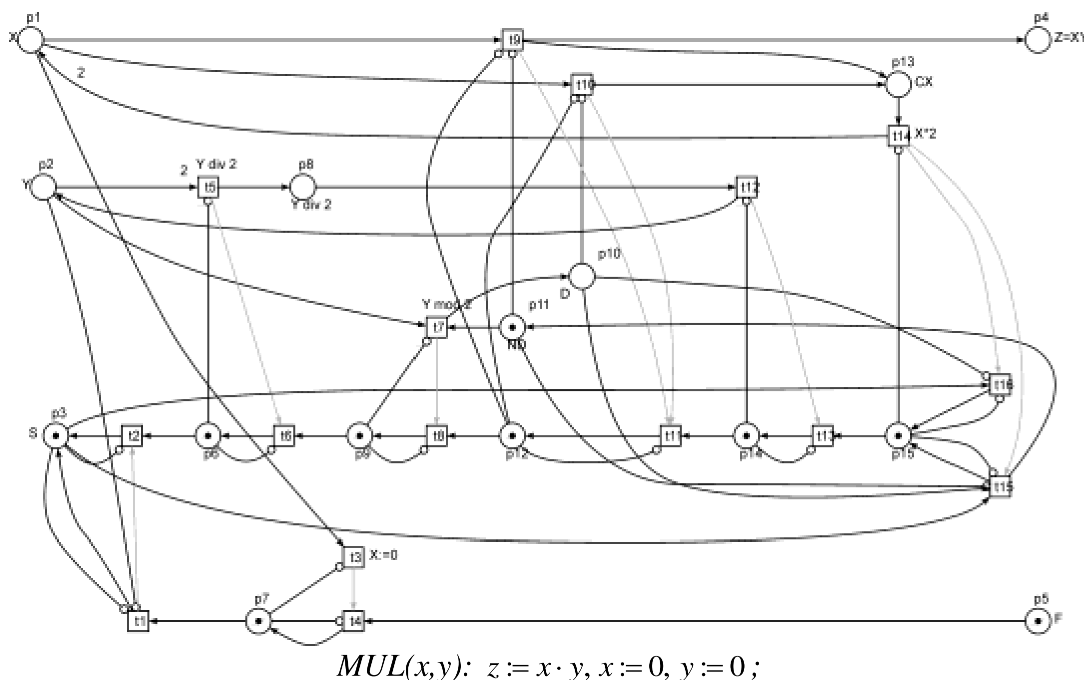
Рис. 6. Модули, реализующие основные операции: копирование, логические и арифметические

Для сетей Слепцова изображенных на Рис. 6, 7 доказано [13] что:

– CLEAN, MOVE, COPY, NOT, OR, AND, ADD, SUB, GT0, GT реализуют соответствующие операции с временной и емкостной сложностью равной константе;

– MUL реализует умножение неотрицательных целых чисел x и y с временной сложностью $O(11 \cdot \log_2 y + 3)$ и постоянной емкостной сложностью равной 15 (по линейной шкале);

– DIV [13] реализует деление неотрицательных целых чисел x и y с временной сложностью $39\cdot(\log_2 x - \log_2 y) + 19$ и постоянной емкостной сложностью равной 48.



MUL(x,y): $z := x \cdot y,\ x := 0,\ y := 0$;

Рис. 7. Модуль умножения за полиномиальное время

Например, чтобы вычислить $3 \cdot 2 = 6$ множитель 3 загружается в позицию X (p_1) и число 2 загружается в позицию Y (p_2). Для запуска вычислений фишка извлекается из позиции S (p_3). Тогда единственная разрешённая последовательность срабатывания переходов $t_1t_5t_6t_8t_{10}^3t_{11}t_{12}t_{13}t_{14}^3t_{16}t_1t_6t_7t_8t_9^6t_{11}t_{13}t_{14}^6t_{15}t_2t_3^{12}t_4$ приводит к получению результата 6 в позиции Z (p_4) что индицируется изъятием фишки из позиции F (p_5).

Заметим, что модуль сети Слепцова для умножения, показанный на Рис. 7, умножает заданные натуральные числа X и Y, используя для этих целей умножение и деление на константу 2 выполняемые одним переходом сети Слепцова; умножение и деление на константу 2 обычно эффективно реализуют как сдвиг двоичного кода влево и вправо соответственно.

В реальных реализациях сети Слепцова (также как сети Петри), программных или аппаратных, мы должны рассматривать сложность указанных процедур. Предположение последовательного вычислительного устройства [12,13], просматривающего переходы и позиции в цикле даёт множитель $O(|P| \cdot |T| \cdot \log_2 l)$ для сетей Петри, где l – это максимальное число шагов. Для сетей Слепцова дополнительная сложность умножения и деления на

константу равную степени 2 также оценивается как $O(\log_2 l)$; для произвольной константы сложность сублинейна [13]. Если мы предполагаем устройство вычисляющей памяти, которое независимо реализует каждый переход [7], мы избавляемся от множителя $|P| \cdot |T|$ и получает ультра-параллельный процессор сетей Слепцова со сложностью шага $O(\log_2 l)$.

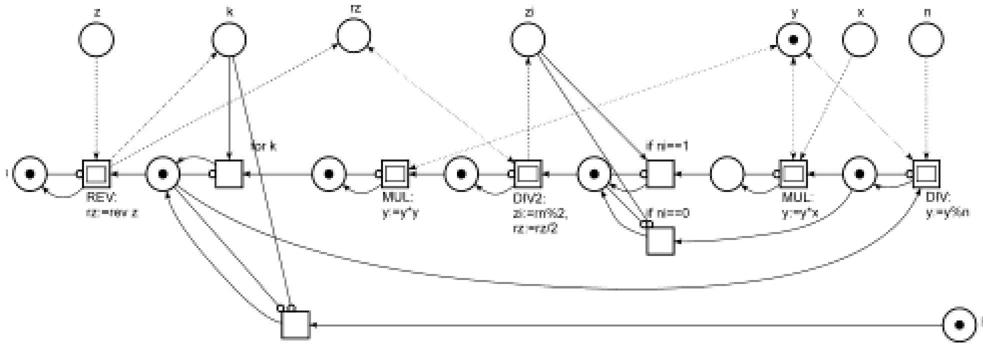
Рассмотренные оценки слишком скрупулёзны для обычного сравнения арифметических алгоритмов. Предполагая, что x и y содержат около n двоичных цифр $x \approx y < 2^n$ и используя логарифмическую шкалу с множителем $\log_2(x \cdot y) \approx 2 \cdot n$, получаем сложность умножения $O(2^{2 \cdot n})$ для сетей Петри и $O(n^2)$ для сетей Слепцова. Для операции деления предположим, что x содержит $2 \cdot n$ двоичных цифр и y содержит n двоичных цифр. Тогда мы получаем аналогичную сложность деления $O(2^{2 \cdot n})$ для сетей Петри и $O(n^2)$ для сетей Слепцова. Таким образом, сети Слепцова выполняются экспоненциально быстрее в сравнении с сетями Петри. Для операций умножения и деления сети Слепцова выполняются за полиномиальное время, в то время как сети Петри требуют экспоненциальное время.

Заметим, что некоторые сети Слепцова могут интерпретироваться как сети Петри, дающие тот же самый результат, но с определенным замедлением вычислений. Исследование указанного типа эквивалентности является направлением дальнейших работ.

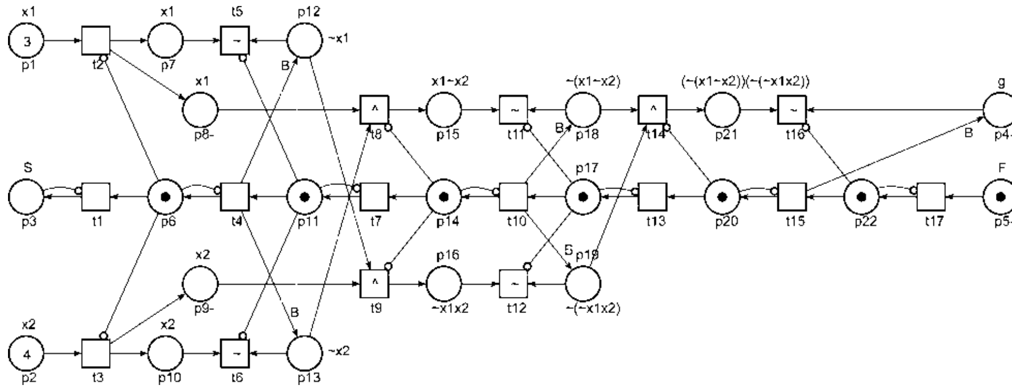
7. Перспективы практической реализации

На основе описанных ранее универсальной сети и сетей, выполняющих арифметические операции, сделан основной вывод, что сети Слепцова выполняются экспоненциально быстрее сетей Петри [13], что позволяет рекомендовать их в качестве модели параллельных вычислений для последующей практической реализации.

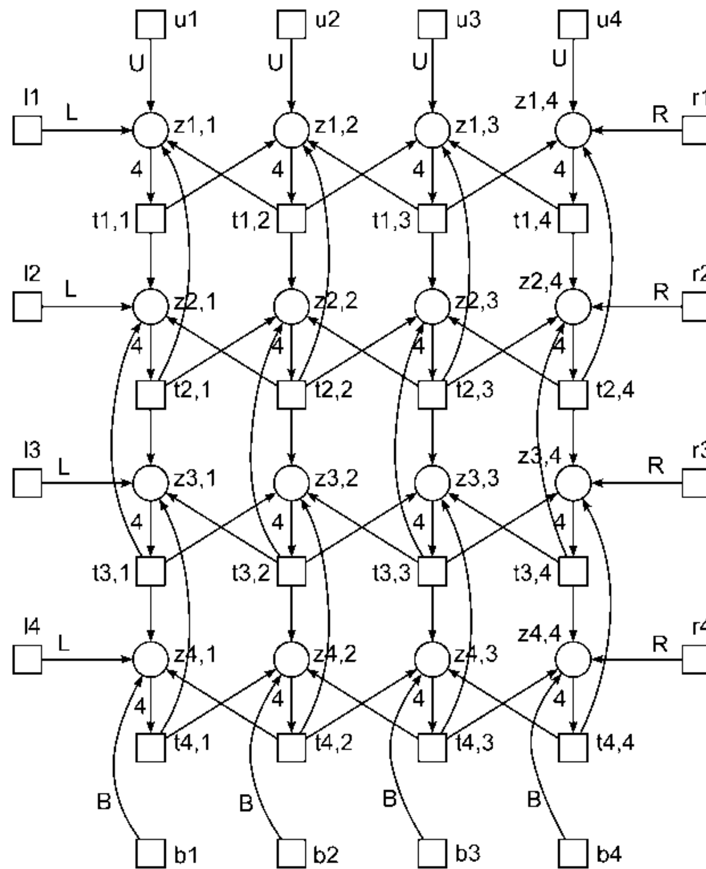
Вычисления на сетях Слепцова обретают все новые области применения, представленные в работах [13,14]. Мы завершаем настоящую работу тремя примерами программ на языке сетей Слепцова для достаточно разнообразных областей приложения, таких как шифрование данных с открытым ключом, мягкие вычисления на основе нечёткой логики и решение уравнений математической физики (Рис. 8 а–в). Первая из сетей использует изученные подсети для арифметических операций и подсеть REV для инверсии побитового представления z в то время как вторая и третья сети реализуют требуемые логические и арифметические операции непосредственно соответствующими переходами сети Слепцова.



а) RSA шифрование/дешифрование $y = x^z \bmod n$



б) вычисление функции нечёткой логики $\varphi = x_1\bar{x}_2 \vee \bar{x}_1 x_2 = \overline{(\overline{(x_1\bar{x}_2)} \wedge \overline{(\bar{x}_1 x_2)})}$;



в) решение уравнения Лапласа $\frac{\delta^2\varphi}{\delta x^2} + \frac{\delta^2\varphi}{\delta y^2} = 0$.

Рис. 8. Примеры программ на языка сетей Слепцова

В первую очередь мы рекомендуем использовать вычисления на сетях Слепцова для тех областей применения, в которых параллельный стиль программирования может принести значительные ускорения вычислений.

Эффективная практическая реализация вычислений на сетях Слепцова требует разработки соответствующих специализированных систем автоматизации программирования и аппаратной реализации процессоров сетей Слепцова. Кроме того, необходимо дальнейшее развитие теоретических методов доказательства корректности программ на языке сетей Слепцова и разработка универсальных сетей, которые используют массовый параллелизм.

Преимуществами вычислений на сетях Слепцова являются наглядный графический язык, сохранение естественного параллелизма предметной области, мелкая грануляция параллельных вычислений, формальные методы верификации параллельных программ, быстрые массово-параллельные архитектуры, реализующие модель вычислений.

Список литературы

1. Ачасова С.М., Бандман О.Л. Корректность параллельных вычислительных процессов. Н.: Наука, 1990, 254 с
2. Вельбицкий И.В. Технология программирования. Киев: Техіка, 1984. 279 с.
3. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб: БХВ-Петербург, 2002, 608 с.
4. Котов В.Е. Сети Петри. М: Наука, 1984, 160 с.
5. Ломазова И.А. Вложенные сети Петри: моделирование и анализ распределенных систем с объектной структурой, Науч. мир, 2004, 207 с.
6. Зайцев Д.А. Универсальная сеть Петри. Кибернетика и системный анализ, № 4, 2012, 24-39.
7. Зайцев Д.А. Парадигма вычислений на сетях Петри. Автоматика и телемеханика, № 8, 2014, 19–36.
8. Burkhard H.-D. Ordered Firing in Petri Nets, Jornal of Information Processing and Cybernetics, no. 2, 1981, 71-86.
9. Jensen K., Kristensen L.M. Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, Berlin, 2009.
10. Neary T., Cook M. (ed.) Proceedings Machines, Computations and Universality (MCU 2013), Zurich, Switzerland, Electronic Proceedings in Theoretical Computer Science 128, 2013.
11. Zaitsev D.A. Clans of Petri Nets: Verification of protocols and performance evaluation of networks, LAP LAMBERT Academic Publishing, 2013, 292 p.
12. Zaitsev D.A. Toward the Minimal Universal Petri Net. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 44 (1), 2014, 47–58.

13. Zaitsev D.A. Sleptsov Nets Run Fast, IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2016, Vol. 46(5), 682–693.
14. Zaitsev D.A., Jürjens J. Programming in the Sleptsov Net Language for Systems Control, Advances in Mechanical Engineering, 2016, Vol. 8(4), 1–11.
15. Zaitsev D.A. Simulating Cellular Automata by Infinite Petri Nets, Journal of Cellular Automata, 2017.
16. Zaitsev D.A. Universal Sleptsov Net, International Journal of Computer Mathematics, 2017.

УДК 004.43

Оптимизирующие трансформации списков и деревьев в системе предикатного программирования

Булгаков К.В. (Новосибирский государственный университет),
Каблуков И.В., Тумуров Э.Г. (Институт систем информатики СО РАН),
Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описываются оптимизирующие трансформации для операций над списками и деревьями в системе предикатного программирования. Кодирование операций представлено набором правил, определяющих замену исходной операции на ее образ в императивном языке. Результатом трансформаций является императивная программа по эффективности сравнимая с написанной вручную.

Ключевые слова: *функциональное программирование, трансформации программ, алгебраический тип данных.*

1. Введение

Оперирование указателями является весьма сложной и опасной процедурой в императивном программировании. Показателем такой сложности является чрезвычайная трудность дедуктивной верификации программ, оперирующих указателями, например, для алгоритма реверсирования списка [15].

В языке предикатного программирования P [12] нет указателей, серьезно усложняющих программу. Вместо указателей используются объекты алгебраических типов: списки и деревья. Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением оптимизирующих трансформаций. Они определяют оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу. Эта оптимизация отлична от классической.

Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной [2, 7];
- замена хвостовой рекурсии циклом;

- подстановка определения предиката на место его вызова;
- кодирование списков и деревьев при помощи массивов и указателей.

Цель данной работы – определить эффективные способы кодирования языковых конструкций с объектами алгебраических типов. Итоговая программа по эффективности должна быть сравнима с программами на императивных языках C или C++.

Во втором разделе дается описание алгебраических типов в системе предикатного программирования. В третьем разделе проводится обзор использования алгебраических типов в других языках программирования. В четвертом разделе описывается реализация трансформации конструкций с алгебраическими типами в императивное расширение языка предикатного программирования P [9]. В пятом разделе описаны правила трансформации для конструкций с алгебраическими типами. В шестом разделе приведены примеры кодирования алгебраических структур и анализ производительности сгенерированного кода по сравнению с кодом, написанным вручную.

2. Предикатное программирование

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
pre <предусловие>
post <постусловие>
{ <оператор> }
```

Необязательные конструкции предусловия и постусловия являются формулами на языке исчисления предикатов; они используются для улучшения понимания программ и для дедуктивной верификации [8, 10, 18]. Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

```
<переменная> = <выражение>
{<оператор1>; <оператор2>}
if (<логическое выражение>) <оператор1> else <оператор2>
<имя программы>(<список аргументов>: <список результатов>)
<тип> <пробел> <список имен переменных>
```

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по

эффективности не уступает написанной вручную и, как правило, короче [1, 8, 10, 18]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [14]) оптимизация программы полностью возлагается на транслятор, в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду. Разумеется, функциональное программирование существенно уступает в эффективности, поскольку даже применением изощренных методов оптимизации невозможно автоматически воспроизвести серию оптимизаций, совершаемых программистом вручную.

Описание типа связывает имя типа с его изображением. Тип может быть параметризованным.

```
<описание типа> ::=
    type <имя типа> [( <параметры типа> )] =
        <изображение типа> | <предописание типа>
```

В языке P алгебраические типы определяются следующей конструкцией:

```
<описание типа> ::= union ( <описание конструкторов> )
<описание конструкторов> ::= <описание конструктора>
                              [, <описание конструкторов>]*
<описание конструктора> ::= <идентификатор> [( <описание полей> )]
<описание полей> ::= <изображение типа> <идентификатор>
                              [, <описание полей>]
```

Значением типа объединения является значение одного из конструкторов, перечисленных в списке описаний конструкторов, вместе с набором полей конструктора. Алгебраический тип может быть рекурсивно определяемым.

2.1. Списки

Тип «список» – встроенный алгебраический тип со следующим определением [11]:

```
type list (type T) = union (
     nil,
     cons(T car, list(T) cdr)
);
```

Здесь T – тип элемента списка, nil и cons – конструкторы. Канонический способ работы со списками определяется оператором выбора:

```
switch (s) {
     case nil: <оператор1>
     case cons(head, tail): <оператор2>
};
```

Вхождения переменных head и tail являются определяющими, причем head – начальный элемент списка s, а tail – «хвост» списка s. Оператор выбора эквивалентен следующему оператору:

if (nil?(s)) <оператор1> **else** { { T c = s.car || list(T) y = s.cdr}; <оператор2> };

Определены следующие операции со списками:

- s.car – первый элемент списка
- s.cdr – список без первого элемента
- last(s) – последний элемент списка
- s[m] – элемент под номером m
- len(s) – длина списка
- nil?(s) – проверка на пустоту списка
- s == nil – проверка на пустоту списка
- cons?(s) – проверка на непустоту списка
- s != nil – проверка на непустоту списка
- prec(s) – список без последнего элемента
- s + t – конкатенация списков
- s + e – добавление элемента в конец списка
- s[m..n] – вырезка списка от номера m до номера n
- s[m..] – вырезка списка от m до конца

Здесь s – выражение типа список, t – терм типа список, e –выражение типа элемента списка, m, n – выражение типа nat. Элементы списка нумеруются с нуля.

Конструкторы списка:

- nil
- cons(head, tail)
- consLeft (list, m) |
- consRight (list) |
- consRight (list, n)

Здесь nil и cons(head, tail), где head – элемент списка и tail – список, являются стандартными конструкторами в соответствии с определением типа list. Описание специальных конструкторов consLeft и consRight дано в разделе 4.2.

Для изображения типа списка допускается использование следующих типовых термов:

list(T)
list(T, L)

Здесь T – тип элемента списка, L – максимальная длина списка. Размер памяти, отводимой для переменной типа list(T, L), будет достаточным для размещения L элементов списка.

Для значения списковой переменной не допускается выход значения за границы памяти, отведенной для переменной. Соответствующий контроль возлагается на программиста. Для этой цели предусмотрены следующие конструкции.

```
max_len(s)
store(s)
left_store(s)
resize(s, n)
```

Здесь s – переменная типа список. Значением функции max_len(s) является максимальное число элементов списка, которое можно разместить в массиве для переменной s. Функция store(s) определяет число элементов, которое можно разместить справа от значения s в свободной части памяти. Функция left_store(s) определяет число элементов, которое можно разместить слева от значения s в памяти. Оператор resize(s, n) отводит новую память размера n элементов, переписывает туда значение переменной s, освобождая старую память.

В дополнение к основному режиму, в котором программист полностью контролирует распределение памяти для списковых переменных, следует также предусмотреть режим, задаваемый прагмой, в котором контроль выхода за границы и заказ памяти большего размера реализуется автоматически.

2.2. Строковый тип

Строковый тип string является предопределенным в языке предикатного программирования P [11]. Его определение имеет вид:

type string = list(**char**);

Набор конструкций, определенный для списков, применим также и для строк с некоторыми ограничениями. В дополнении к этому в языке P определены строковые константы.

Основным представлением строкового объекта является массив литералов, завершающийся терминальным нулем, причем нуль не входит в значение строки. Иначе говоря, для строкового типа фактически действует следующее определение:

type string = **subtype**(list(**char**) s: s != nil & last(s) == 0);

Проверка строки s на пустоту реализуется оператором s.car == 0, а не s == nil. В принципе, возможна реализация, в которой конструктор nil кодируется значением из единственного нулевого элемента, однако такое решение приведет к потере эффективности. В итоге, типы list и string несовместимы: со строковым объектом нельзя работать как со

списком, в частности, нельзя подставлять строковый объект параметром типа list. Как следствие, библиотеки для списков неприменимы для строковых объектов.

Для строкового типа, как и для списков, существует возможность указать размер памяти выделяемой для строковых объектов. Для этого используется конструкция string(L), где L количество литер, которые вмещает память значения типа string(L). Для строкового типа также возможно использование следующих конструкций:

```
max_len(s)
store(s)
resize(s, n)
```

Исключается возможность сдвига вправо значения строки относительно начала памяти, т.е значение строки всегда размещается с начала памяти.

Вводятся дополнительные конструкции для строковых объектов. Для определения числа элементов строки s вместо функции len(s) используется length(s). Конструктор nil, распознаватель nil?(s), а также отношения s == nil и s != nil не используются для строковых объектов. В качестве пустой строки используется конструктор empty, значением которого является строка из единственного нулевого элемента.

2.3. Двоичные деревья

Двоичное дерево – дерево, в котором каждая вершина имеет не более двух потомков. Двоичное дерево используется для представления табличных данных и хранения множества данных вместе с их ключами, используемыми для поиска. Основные операции двоичного дерева это добавление нового элемента, удаление существующего элемента и поиск элемента по ключу. Ключи и данные представлены следующими типами:

```
type Tkey(<);
type Tinfo;
```

Для типа ключей Tkey определено отношение линейного порядка «<». Типы Tkey и Tinfo – произвольны и являются параметрами модуля, реализующего АВЛ-деревья. [5, 6, 16].

АВЛ-деревья выбраны в языке предикатного программирования P для реализации сбалансированных деревьев, хотя в большинстве современных реализаций сбалансированных деревьев используются красно-черные деревья. Данный выбор был основан на результатах работы Performance Analysis of BSTs in System Software [17]

Двоичное дерево представляется алгебраическим типом Tree:

```
type Tree = union (
  leaf,
  node (Tkey key, Tinfo info, Tree left, right)
);
```

Лист дерева соответствует конструктору leaf. Вершина дерева, соответствующая листу, не хранит никакой информации. Конструктор node определяет вершину, не являющуюся листом. Полями конструктора являются ключ key и ассоциированное с ним данное info. Левое и правое поддеревья, исходящие из данной вершины, определяются полями left и right.

3. Обзор реализации списков и строк

Для языков программирования C++, Java, Lisp и других имеются библиотеки для поддержки операций со списковыми и строковыми объектами. Стандартными способами реализации списков являются их реализация в виде односвязных и двусвязных списков.

В C++ существует несколько способов работы со строками. Один из них, это способ унаследованный от языка C, когда строки представляются как char-массивы с терминальным нулем в конце. Для работы с таким представлением используется библиотека cstring. Стандартная библиотека C++ предоставляет более удобный формат работы со строками в объектно-ориентированном стиле — класс string. Тип string представляет собой последовательность символов, где каждый символ может быть получен по его позиции в последовательности. Обычно реализуется при помощи динамического массива, который позволяет произвольный доступ к элементам по индексу.

В Java класс для работы со строками — String. Список операций и внутреннее представление этого класса очень схож с классом string в C++, но есть одно важное отличие. Строки в Java неизменяемые. Такие операции, как получение подстроки или конкатенация, всегда будут возвращать новый объект класса String. Для того, чтобы эффективно реализовать комбинации и модификации уже существующих строк, следует использовать классы StringBuilder и StringBuffer.

В системе предикатного программирования строки кодируются массивами с терминальным нулем, что аналогично реализациям в императивных языках C и C++. Для списков имеется два способа реализации: через массивы и через односвязные списки. Для списков и строк, представляемых массивами, используются разные способы реализации в зависимости от контекста, определяемого с помощью потокового анализа.

4. Реализация трансформаций кодирования операций с алгебраическими типами

При преобразовании сложной иерархической конструкции с объектами алгебраических типов преобразование начинается с вложенных подконструкций.

Для оператора присваивания списковой переменной выполняется определение *вида присваивания* для списковой переменной, и на основании этих данных выбирается преобразование.

4.1. Определение вида присваивания для списковой переменной

В соответствии с формальной семантикой языка P вычисление нового значения списка как результата некоторой операции сопровождается выделением памяти для этого значения. Буквальная реализация этого положения оказалась бы весьма расточительной. Например, при исполнении оператора s = x + y + z предполагается отведение памяти для результатов выражений x + y и (x + y) + z, а также для нового значения переменной s. Если заранее подсчитать длину списка x + y + z и отвести достаточную память для переменной s, то исходный оператор заменяется последовательностью ″s = x; s = s + y; s = s + z″, исполнение которой не требует дополнительной памяти.

В целях эффективного кодирования списков различаются три вида присваивания списковым переменным:

– **Присваивание вида копирование** реализуется копированием результата спискового выражения в переменную слева от знака равенства. При этом переменная s не участвует в выражении e.
– **Присваивание вида модификация** изменяет значение списковой переменной, добавляя к изначальным данным слева или справа дополнительные элементы списка.
– **Сканирование** не модифицирует значение списковой переменной, осуществляя только анализ списка с продвижением по нему без модификации.

Присваивание вида копирования для оператора s = e реализуется копированием списка, вычисленного выражением e, в память для значения переменной s. Для оператора s = e + d в массив для хранения переменной s копируется значение e и вслед за ним копируется значение d.

Присваивание вида модификации[1] реализует изменение значения, размещаемого памяти для переменной s. Итоговое значение помещается в тот же участок памяти. Оператор s = s + e копирует список (значение выражения e) вслед за значением списка s в памяти. Оператор s = e + s копирует список (значение выражения e) перед значением списка s в памяти. Оператор s = s.car реализует отсечение хвоста списка. Операторы s = s[m..n] и s = s[m..] реализуют сужение значения списка к вырезке исходного значения s. Вместо сдвига значения s в начало массива проводится соответствующая корректировка позиции значения списка внутри памяти для списковой переменной. Операторы s = s.cdr и s = prec(s) также реализуются коррекцией позиции в памяти.

Списковая переменная, все действия с которой реализуют лишь анализ списка с продвижением по нему без его модификации[2] в памяти, называется *переменной сканирования.* Для переменной сканирования s присваивание вида s = y реализуется не копированием значения y, а создание *объекта сканирования*, ассоциированного с переменной y, и присваиванию его переменой s. Операторами сканирования являются s = s.cdr, s = prec(s), s = s[m..n] и s = s[m..]. Их отличие от соответствующих операторов присваивания в режиме модификации в том, что они не модифицируют переменную y. Операторы сканирования являются аналогами итераторов в императивных языках.

4.2. Представление алгебраических типов

Основным способом представления списка является массив.

Для представления значения типа list(T) в императивном расширении используется следующая структура.

```
struct list {
 T *data;
 int max_len;
 int m;
 int n;
}
```

Здесь T – тип элемента списка, max_len – максимальная длина списка, m, n – индексы начала и конца текущей вырезки массива, 0 ≤ m ≤ n ≤ max_len, data – массив данных.

---

[1] Модификация переменных возможна как в исходной предикатной программе, так и в результате склеивания переменных после проведения трансформации склеивания переменных.
[2] Точнее, допускается модификация отдельных элементов списка без изменения их состава.

Другими возможными альтернативами кодирования списка являются: односвязный список, двунаправленный список, кольцевой список. Представление в виде кольцевого буфера следует считать модификацией представления в виде массива.

Представление списка через односвязный список.

```
struct list {
 list *next;
 T data;
}
```

Здесь T – тип элемента списка, next – указатель на следующий элемент.

Для любых видов списковых выражений возможен такой способ реализации, при котором удается отложить отведения памяти до момента присваивания списковой переменной, находящейся в левой части оператора присваивания. Поэтому отведение памяти далее рассматривается по отношению к списковым переменным.

Оптимальной реализацией является использование одного экземпляра памяти для всех присваиваний одной списковой переменной. Такое возможно, если известно верхнее ограничение L числа элементов списка: list(a, L).

Допустим, отведенная для списковой переменной память есть массив A с индексами в диапазоне 0..N. Тогда значение списковой переменной можно представить вырезкой A[m..n], где $0 \le m \le n \le N$, однако в случае пустого списка $m > n$. В большинстве случаев $m = 0$ и свободное место в памяти остается слева в диапазоне m+1..N. Однако бывают случаи, когда свободную часть памяти надо оставить справа для того, чтобы реализовать присваивание вида s = y + s, как, например, в работе [2], где используется присваивание buf = stf + buf.

Для формирования нестандартного размещения значения списка используется специальные конструкторы consLeft, consRight. Конструктор consLeft(list, m), где m – номер элемента, формирует представление списка s в массиве, сдвинутое на m элементов относительно начала массива. Конструктор вида consRight(list) формирует представление списка s в массиве прижатым вправо, т.е. последний элемент списка находится в конце массива. Конструктор вида consRight(list, n) формирует представление списка s в массиве длины n прижатым вправо. При исполнении оператора s = consRight(list, n) отводится новая память для строковой переменной s; если до присваивания переменная s уже имела некоторое значение, то транслятор с языка P должен обеспечить возврат старой памяти переменной s.

В языке P нет операторов отведения и освобождение памяти для списковых переменных. Вставка в код соответствующих действий реализуется транслятором. Отведение памяти реализуется при первом присваивании переменной. Размер памяти определяется по значению правой части оператора присваивания, если он явно не указан описанием типа.

Далее рассматривается лишь представление списка в виде массива.

Представление значений типа string.

```
struct string {
 char *data;
 int max_len;
 int len;
}
```

Здесь max_len – максимальная длина строки, len – индекс конца текущей строки, len < max_len, по индексу len значение ноль, data – массив данных.

Для строкового типа используется два вида объектов сканирования: один – для сканирования с начала строки, второй – с конца. В первом случае объект сканирования может быть представлен указателем на начальный элемент строки, во втором – дополнительно требуется длина строки, при этом строка, представленная объектом сканирования, нулем не завершается.

Представление значений типа Tree через указатели.

```
struct Tree {
 Tree *left;
 Tree *right;
 Tkey key;
 Tinfo info;
}
```

Здесь left, right – указатели на левое и правое поддеревья, key – ключ типа Tkey, info – значение типа Tinfo.

Представление значений типа Tree через через массивы [3].

```
struct TreeValue {
    Tkey key;
    Tinfo info;
}
struct Tree {
      TreeValue *data;
 int left;
 int right;
}
```

Здесь data – указатель на массив, где размещается дерево, left, right – индексы левого и правого поддеревьев в массиве.

4.3 Потоковый анализ программы

При кодировании рекурсивных структур используется потоковый анализ программы [4], который включает построение графа вызовов, нахождение аргументов и результатов операторов и нахождение живых переменных операторов. Потоковый анализ определяет время жизни переменных: для каждого вхождения переменной определяется, будет значение этой переменной использоваться при дальнейшем исполнении программы или нет. На основании этих данных определяются эффективные способы преобразования операций алгебраических типов.

Алгоритм определения времени жизни переменных реализуется следующим образом. Для каждого оператора определяются преемники – операторы, исполняющиеся сразу после него. Строятся цепочки преемников для каждого оператора. Переменная жива после исполнения оператора, если ее текущее значение используется в одной из цепочек преемников этого оператора.

Предикат isAlive(variable_name) по отношению к текущему оператору истинен, если переменная с именем variable_name является живой переменной после исполнения оператора. Этот предикат используется в правилах трансформации, чтобы избавиться от избыточного копирования данных неживых переменных. Считается, что результаты программы являются живыми для каждого оператора его тела, так как предположительно будут использоваться после завершения исполнения программы.

5. Правила трансформации

5.1. Описание правил трансформации

Правила трансформации имеют следующую структуру:

if(<условия применимости>)
<конструкция на предикатном языка> → <код в императивном расширении>

Условия применимости это логическое выражение. Трансформация применима лишь при истинном значении условия. Переменные, встречающиеся в левой части правила, являются *параметрами* правила. При использовании правила вместо параметра может быть подставлена любая переменная соответствующего типа.

5.2. Правила трансформации списка при кодировании через массив

Напомним представление списка через массив:

```
struct list {
    T *data;
    nat max_len;
    nat m;
    nat n;
}
```

Приведем правила трансформации для основных операций списка при кодировании через массив.

```
list(T) l;          →   list *l = new list();
                        l→max_len = DEFAULT_MAX_LEN;
                        l→data = new T[l→max_len];
                        l→m = 0;
                        l→n = 0;
list(T, L) l;       →   list *l = new list();
                        l→max_len = L;
                        l→data = new T[l→max_len];
                        l→m = 0;
                        l→n = 0;
l = consLeft(y, m); →   delete[] l→data;
                        l→max_len = y→max_len + m;
                        l→data = new T[l→max_len];
                        l→m = m;
                        l→n = len(y) + m;
                        <копирование массива из y в l со сдвигом на m
                        относительно начала>
l = consRight(y);   →   delete[] l→data;
                        l→max_len = y→max_len;
                        l→data = new T[l→max_len];
                        l→n = l→max_len;
                        l→m = l→n – len(y);
                        <копирование из массива y в l прижатым
                        вправо>
l = consRight(y, L); →  delete[] l→data;
                        l→max_len = L;
                        l→data = new T[l→max_len];
                        l→n = l→max_len;
                        l→m = l→n – len(y);
                        <копирование из массива y в l прижатым
                        вправо>
max_len(s);         →   s→max_len;
store(s);           →   s→max_len – s→n;
```

```
left_store(s); →    s→m;
resize(s, n);  →    T *new_data = new T[n];
                    <копируем данные из s→data в new_data>
                    delete[] s→data;
                    s→data = new_data;
                    s→max_len = n;
len(s);        →    s→n – s→m;
nil?(s);       →    s→data == NULL;
cons?(s);      →    s→data != NULL;
```

Для присваивания вида копирования используются следующие правила.

```
s = s1 + .. + sn;   →   nat result_len = len(s1) + … + len(sn);
                        if (s→max_len < result_len) {
                                list(T, result_len) new_list;
                                delete s;
                                s = new_list;
                        }
                        nat shift = 0;
                        <копирование s1 в s со сдвигом на shift
                        относительно начала>
                        shift += len(s1);
                        …
                        <копирование sn в s со сдвигом на shift
                        относительно начала>
```

Для присваивания вида модификация используются следующие правила.

```
s = s + e;     →    if (store(s) >= len(e)) {
                            <копировать e вслед за значением s>
                    } else {
                            resize(s, len(s) + len(e));
                            <копировать e вслед за значением s>
                    }
s = e + s;     →    if (left_store(s) >= len(e)) {
                            <копировать e перед значением s>
                    } else {
                            s = consRight(s, len(s) + len(e));
                            <копировать e перед значением s>
                    }
s = s.car;     →    s→n = s→m + 1;
s = s[m..n];   →    s→m = m;
                    s→n =n;
s = s[m..];    →    s→m = m;
s = s.cdr;     →    s→m = s→m + 1;
s = prec(s);   →    s→n = s→n - 1;
s = last(s);   →    s→m= s→n – 1;
```

Для присваивания вида сканирование используются следующие правила.

```
s = s.cdr;      →    s→m = s→m + 1;
s = prec(s);    →    s→n = s→n - 1;
s = s[m..n];    →    s→m = m;
                     s→n = n;
s = s[m..];     →    s→m = m;
```

5.3. Правила трансформации списка при кодировании через указатели

Напомним представление списка через указатели:

```
struct list {
    list *next;
    T data;
}
```

Приведем правила трансформации для основных операций списка при кодировании через список.

```
list(T) l;        →    list *l = new list();
                       l→next = NULL;
max_len(s);       →    sizeof(nat);
store(s);         →    max_len(s) – len(s);
left_store(s);    →    max_len(s) – len(s);
len(s);           →    nat len = 0;
                       list *tmp = s;
                       while (tmp != NULL) {
                           ++len;
                           tmp = tmp→next;
                       }
nil?(s);          →    s == NULL;
cons?(s);         →    s != NULL;
```

Для присваивания вида копирования используются следующие правила.

```
if(!isAlive(s1) && … && !isAlive(sn))
s = s1 + .. + sn;    →    s = s1;
                          while (s1→next != NULL) {
                              s1 = s1→next;
                          }
                          s1→next = s2;
                          while (s2→next != NULL) {
                              s2 = s2→next;
                          }
                          ….
```

Для присваивания вида модификация используются следующие правила.

```
if(!isAlive(e))
s = s + e;      →    list *tmp = s;
                     while (tmp→next != NULL) {
                             tmp = tmp→next;
                     }
                     tmp→next = e;
if(!isAlive(e))
s = e + s;      →    list *tmp = e;
                     while (tmp→next != NULL) {
                             tmp = tmp→next;
                     }
                     tmp→next = s;
                     s = e;
s = s.car;      →    <освободить память начиная с s→next>
                     s→next = NULL;
s = s[m..n];    →    list *tmp;
                     for (nat j = 0; j < m; ++j) {
                             tmp = s→next;
                             delete s;
                             s = tmp;
                     }
                     nat size = n – m + 1;
                     for (nat j = 0; j < size; ++j) {
                             tmp = tmp→next
                     }
                     <удалить элементы после tmp>
                     tmp→next = NULL;
s = s[m..];     →    list *tmp;
                     for (nat j = 0; j < m; ++j) {
                             tmp = s→next;
                             delete s;
                             s = tmp;
                     }
s = s.cdr;      →    list *tmp = s;
                     s = s→next;
                     delete tmp;
s = prec(s);    →    list *tmp = s;
                     while (tmp→next→next != NULL) {
                             tmp = tmp→next;
                     }
                     delete tmp→next;
                     tmp→next = NULL;
s = last(s);    →    list *tmp = s;
                     while (tmp→next != NULL) {
                             tmp = tmp→next;
                             delete s;
                             s = tmp;
```

```
}
```

Для присваивания вида сканирование используются следующие правила.

```
s = s[m..];   →   for (nat j = 0; j < m; ++j) {
                      s = s→next;
                  }
s = s.cdr;    →   s = s→next;
```

При кодировании списка через указатели существуют дополнительные правила трансформации для парных операций.

```
s = u.car + s; u = u.cdr;   →   list *a = u→next;
                                u→next = s;
                                s = u;
                                u = a;
```

5.4. Правила трансформации для строкового типа

Напомним представление строки:

```
struct string {
    char *data;
    nat max_len;
    nat len;
}
```

Приведем правила трансформации для строкового типа.

```
string(L) s;    →   string *s = new string();
                    s→max_len = L;
                    s→data = new char[s→max_len];
                    s→len = 0;
                    s→data[s→len] = 0;
s = empty;      →   string *s = new string();
                    s→max_len = 1;
                    s→data = new char[s→max_len];
                    s→len = 0;
                    s→data[s→len] = 0;
max_len(s);     →   s→max_len;
store(s);       →   max_len(s) – len(s) - 1;
length(s);      →   s→len;
resize(s, n);   →   char *new_data = new char[n];
                    <копируем данные из s→data в new_data>
                    delete[] s→data;
                    s→data = new_data;
                    s→max_len = n;
s.car == 0;     →   s→data[0] == 0;
```

Для присваивания вида копирование используется следующее правило.

```
s = s1 + .. + sn;    →    nat result_len = length(s1) + … + length(sn);
                          if (s→max_len < result_len) {
                                  string(result_len) new_string;
                                        delete s;
                                  s = new_string;
                          }
                          nat shift = 0;
                          <копирование s1 в s со сдвигом на shift
                          относительно начала>
                          shift += length(s1);
                          …
                          <копирование sn в s со сдвигом на shift
                          относительно начала>
```

Для присваивания вида модификация используются следующие правила.

```
s = s + e;      →    if (store(s) >= len(e)) {
                             <копировать e вслед за значением s>
                     } else {
                             resize(s, len(s) + len(e));
                             <копировать e вслед за значением s>
                     }
s = s.car;      →    s→len = 1;
                     s→data[s→len] = 0;
s = prec(s);    →    s→len = s→len – 1;
                     s→data[s→len] = 0;
```

5.5. Правила трансформации для дерева

Напомним представление дерева через указатели:

```
struct Tree {
    Tree *left;
    Tree *right;
    BAL balance;
    Tkey key;
    Tinfo info;
}
```

Приведем правила трансформации для деревьев при кодировании дерева через указатели.

```
Tree tree(leaf);                →    Tree *tree = NULL;
Tree tree(key, info, bal, leaf, leaf);
                                →    Tree *tree = new Tree();
                                     tree→key = key;
                                     tree→info = info;
                                     tree→balance = bal;
                                     tree→left = NULL;
                                     tree→right = NULL;
if(!isAlive(subtree1) && !isAlive(subtree2))
Tree tree(key, info, bal, subtree1, subtree2);
                                →    Tree *tree = new Tree();
                                     tree→key = key;
                                     tree→info = info;
                                     tree→balance = bal;
                                     tree→left = subtree1;
                                     tree→right = subtree2;
if(!isAlive(subtree1) && isAlive(subtree2))
Tree tree(key, info, bal, subtree1, subtree2);
                                →    Tree *tree = new Tree();
                                     tree→key = key;
                                     tree→info = info;
                                     tree→balance = bal;
                                     tree→left = subtree1;
                                     tree→right = copy_tree(subtree2);
if(isAlive(subtree1) && !isAlive(subtree2))
Tree tree(key, info, bal, subtree1, subtree2);
                                →    Tree *tree = new Tree();
                                     tree→key = key;
                                     tree→info = info;
                                     tree→balance = bal;
                                     tree→left = copy_tree(subtree1);
                                     tree→right = subtree2;
if(isAlive(subtree1) && isAlive(subtree2))
Tree tree(key, info, bal, subtree1, subtree2);
                                →    Tree *tree = new Tree();
                                     tree→key = key;
                                     tree→info = info;
                                     tree→balance = bal;
                                     tree→left = copy_tree(subtree1);
                                     tree→right = copy_tree(subtree2);
```

6. Примеры кодирования рекурсивных структур

6.1. Кодирование списка через указатели

6.1.1. Вычисление суммы значений всех элементов списка

Дан список элементов, для которых определена операция сложения. Нужно вычислить сумму значений всех элементов списка. Код программы на исходном языке:

```
type T;
type ListT = list(T);
addItems(ListT l, T t : T s) {
    if ( l = nil )
        s = t;
    else
        addItems(l.cdr, t + l.car : s);
}
```

Код после замены хвостовой рекурсии циклом, открытой подстановки и склеивания переменных:

```
addItems(ListT l : T s) {
    s = 0;
    while ( l != nil ) {
        s = s + l.car;
        l = l.cdr;
    }
}
```

Процесс преобразования предикатной программы в императивное расширение осуществляется путем последовательного обхода инструкций в исходном тексте программы и их преобразовании в императивное расширение. Рассмотрим этот процесс применительно к текущему примеру.

В программе выделяются три конструкции со списками: l != nil, l.car, l = l.cdr. Для первых двух конструкций процесс кодирования реализуется единственным образом без каких-либо условий применимости. Более подробно рассмотрим конструкцию l = l.cdr. По результатам потокового анализа все действия с переменной l реализуют лишь анализ списка с продвижением по нему, без его модификации в памяти, точнее изменяется указатель на текущий элемент, но не изменяются значения элементов списка. Поэтому в данном случае применяется преобразование для присваивания вида сканирование.

В результате применения найденных преобразований получим следующую программу на императивном расширении.

```
addItems (ListT *l : T s) {
    s = 0;
    while ( l != NULL ) {
        s = s + l→data;
        l = l→next;
    }
}
```

Код, написанный вручную:

```
int64_t calc_sum_manuall(List *list) {
    int64_t result = 0;
    while (list != nullptr) {
        result += list→value;
        list = list→next;
    }
    return result;
}
```

6.1.2. Обращение списка

Дан список, необходимо его инвертировать, т.е. обратить порядок элементов так, чтобы первый элемент стал последним, а последний первым. Пусть список представлен последовательностью элементов: $s_1,s_2,\ldots,s_n$. Тогда результатом инвертирования будет список $s_n,\ldots s_2,s_1$. Проблема дедуктивной верификации программы инвертирования списка чрезвычайно сложна и относится к категории Verification Grand Challenges. Графически задачу можно представить следующим образом (рис. 3).

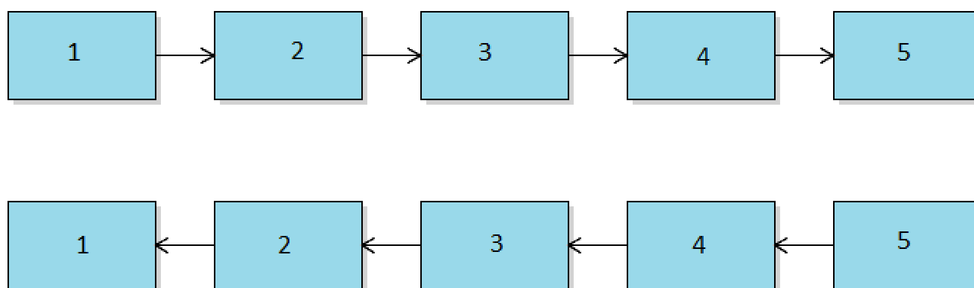


Рисунок 1

Код программы на исходном языке:

```
type T;
type ListT = list(T);
reverse(ListT s : ListT s') pre s != nil {
    reverseIn([s.car], s.cdr : s');
}
reverseIn(ListT s, u : s') {
    if (u = nil)
            s' = s;
    else
            reverseIn(u.car + s, u.cdr : s');
}
```

В программе reverse аргумент s – исходный список, результат s' – инвертированный список. Программа reverseIn является обобщением программы reverse. Исходный список представлен в reverseIn из двух частей. Аргумент s есть начальная часть списка в инвертированном виде, аргумент u – оставшаяся часть списка.. Графически аргументы программы reverseIn можно представить следующим образом (рис. 4).

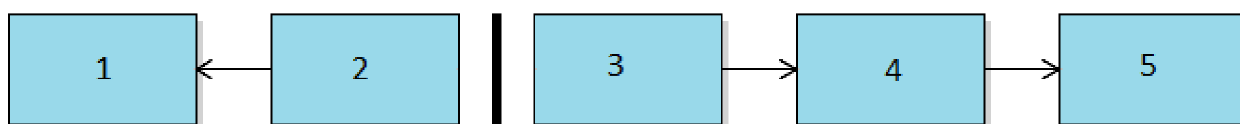


Рисунок 2

Код после замены хвостовой рекурсии циклом, открытой подстановки и склеивания переменных:

```
reverse(ListT s : ListT s) {
     ListT u = s.cdr;
     s = [s.car];
     while ( u != nil ) {
             s = u.car + s;
             u = u.cdr;
     }
}
```

В программе выделяются пять конструкций со списками: u = s.cdr, s = [s.car], u != nil, s = u.car + s, u = u.cdr. Для конструкций s = [s.car] и u != nil процесс кодирования реализуется единственным образом без каких-либо условий применимости. Рассмотрим ListT u = s.cdr. По результатам потокового анализа все действия с переменной u реализуют лишь анализ списка с продвижением по нему, без его модификации в памяти, точнее изменяется указатель на текущий элемент, но не изменяются значения элементов списка. Поэтому в данном случае применяется преобразование для присваивания вида сканирование. Дополнительных условий применимости не требуется. Рассмотрим конструкцию s = u.car + s; В данном случае изменяется значение, размещаемое в памяти s, поэтому для кодирования данной конструкции используется присваивание вида модификация. Переменная u.car не является живой, поэтому выбирается преобразование без копирования переменной u.car. Рассмотрим конструкцию u = u.cdr. Как говорилось ранее, по результатам потокового анализа все действия с переменной u реализуют лишь анализ списка с продвижением по нему, поэтому в данном случае имеет место присваивание вида сканирование. В результате, для данных конструкций используется правило трансформации для парного кодирования. Т.е. применяется следующее правилом

```
s = u.car + s; u = u.cdr; →    list *a = u→next;
                               u→next = s;
                               s = u;
                               u = a;
```

В результате применения соответствующих правил трансформации получаем следующий код.

```
reverse(list *s) {
      list *u = s→next;
      s→next = NULL;
     while ( u != NULL ) {
             list *a = u→next;
             u→next = s;
             s = u;
             u = a;
     }
}
```

Код, написанный вручную:

```
List *revert_manuall(List *list) {
    List *result = nullptr;
    while (list != nullptr) {
        List *tmp = list;
        list = list→next;
        tmp→next = result;
        result = tmp;
    }
    return result;
}
```

6.2. Кодирование списка через массив

6.2.1. Вычисление суммы значений всех элементов списка

Дан список элементов, для которых определенны операция сложения. Нужно вычислить сумму значений всех элементов списка. Код программы на исходном языке:

```
type T;
type ListT = list(T);
addItems(ListT l, T t : T s) {
    if (len(l) = 0)
        s = t;
    else
        addItems(l.cdr, t + l.car : s);
}
```

Код после замены хвостовой рекурсии циклом, открытой подстановки и склеивания переменных:

```
addItems(ListT l : T s) {
    s = 0;
    while (len(l) != 0) {
        s = s + l.car;
        l = l.cdr;
    }
}
```

Процесс преобразования предикатной программы в императивное расширение осуществляется путем последовательного обхода инструкций в исходном тексте программы и их преобразовании в императивное расширение. Рассмотрим этот процесс применительно к текущему примеру.

В программе выделяются три конструкции со списками: len(l), l.car, l = l.cdr. Для первых двух конструкций процесс кодирования реализуется единственным образом без каких-либо условий применимости. Более подробно рассмотрим конструкцию l = l.cdr. По результатам потокового анализа все действия с переменной l реализуют лишь анализ списка с продвижением по нему, без его модификации в памяти, точнее изменяется указатель на текущий элемент, но не изменяются значения элементов списка. Поэтому в данном случае применяется преобразование для присваивания вида сканирование.

```
addItems (ListT *l : T s) {
    s = 0;
    while ( (l→n – l→m) != 0) {
        s = s + l→data[l→m];
        l→m += 1;
    }
}
```

Код, написанный вручную:

```
int64_t calc_sum_manuall(ArrayList *list) {
    int64_t result = 0;
    for (int64_t i = list→m; i < list→n; ++i) {
        result += list→data[i];
    }
    return result;
}
```

6.3. Анализ производительности примеров

В приведенных выше примерах показаны коды программ на императивном расширенные, которые получены при помощи оптимизирующих преобразований, в том числе применением правил трансформации. Необходимо оценить производительность программ, которые получаются в результате компиляции исходных кодов, полученных при помощи оптимизирующих преобразований. А также сравнить их производительность, с программами полученными при помощи компиляции программ, которые изначально были написаны на императивном языке C++.

Для анализа и сравнения производительности сгенерированного и написанного вручную кода использовалась библиотека Celero [13]. Программы компилировались при помощи стандартного компилятора IDE Visual Studio 2013 со стандартными настройками. В ходе тестирования выполнялось несколько итераций программ с разным размером входных данных. Результаты тестирования следующие (Таблица 1):

Таблица 1. Тест производительности

Timer resolution: 0.410529 us

Group	Experiment	Prob. Space	Baseline	us/Iteration	Iterations/sec
ElemSumList	Manual	16	1.00000	0.03337	29964451.05
ElemSumList	Manual	128	1.00000	0.25748	3883852.16
ElemSumList	Manual	1024	1.00000	1.92963	518235.21
Revert	Manual	16	1.00000	0.02475	40408331.57
Revert	Manual	128	1.00000	0.21030	4755128.63
Revert	Manual	1024	1.00000	1.58906	629301.42
ElemSumArrayLis	Manual	16	1.00000	0.04778	20929556.24
ElemSumArrayLis	Manual	128	1.00000	0.25167	3973508.08
ElemSumArrayLis	Manual	1024	1.00000	1.72695	579054.14
ElemSumList	Generated	16	0.99491	0.03320	30117863.32
ElemSumList	Generated	128	0.99054	0.25504	3920961.98
ElemSumList	Generated	1024	1.02137	1.97086	507391.48
Revert	Generated	16	1.23557	0.03058	32704124.76
Revert	Generated	128	1.23033	0.25874	3864920.05
Revert	Generated	1024	1.25464	1.99371	501578.47
ElemSumArrayLis	Generated	16	0.99901	0.04773	20950360.14
ElemSumArrayLis	Generated	128	1.00148	0.25204	3967648.13
ElemSumArrayLis	Generated	1024	1.00366	1.73328	576939.74

Столбец Group содержит названия решаемых задач, где ElemSumList — нахождение суммы элементов при кодировании списка через указатели, Revert — обращение списка, ElemSumArrayLis - нахождение суммы элементов при кодировании списка через массив. Столбец Experiment содержит обозначение способа, каким был получен код: Manual — написан вручную на императивном языке, Generated — получен при помощи оптимизирующих преобразований. Столбец Prob. Space содержит информацию о размере входных данных, на которых проводилось тестирование. Столбец Baseline отображает соотношение времени, затраченного на выполнения кода, для программы написанной вручную и для программы сгенерированной при помощи оптимизирующих трансформаций. us/Iteration — время одной итерации программы в миллисекундах. Iterations/sec — количество итераций в секунду.

Для того чтобы сравнить производительность программ, написанных вручную и сгенерированных при помощи оптимизирующих трансформаций, рассмотрим столбец Baseline. По результатам измерений производительности можно увидеть, что в одних случаях быстрее выполняется код, написанный в ручную, а в других — сгенерированный код, при этом колебания производительности достаточно малы. Это говорит о том, что производительность кода, написанного вручную и сгенерированного при помощи оптимизирующих преобразований практически идентична.

По результатам тестирования можно сказать, что поставленная в начале работы задача выполнена. Определены эффективные способы кодирования языковых конструкций использующих алгебраические структуры, которые позволяют добиться для оттранслированной программы на языке P производительности, сравнимой с кодом, изначально написанным на императивном языке программирования, таком как C++.

7. Заключение

В данной работе описывается трансформация кодирования объектов алгебраических типов через массивы и указатели. Кодирование операций над объектами алгебраических типов представлено набором правил, определяющих замену исходной операции на ее образ в языке императивного расширения. Кодирование эффективно для простых типовых случаев и позволяет получить программы по эффективности сравнимые с написанными вручную. С этой целью проводится потоковый анализ программы, в частности определяется время жизни переменных.

В дальнейшем планируется реализовать полный набор кодирования списков: через односвязный список, двусвязный список, очередь, массив и деку. По набору операций, используемых в программе, для конкретного объекта можно было бы автоматически определять один из указанных способов кодирования, наиболее подходящий для набора используемых операций.

Список литературы

1. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т. 4 (33) , 2008. С. 79-94.
2. Каблуков И. В., Шелехов В.И. Реализация склеивания переменных в предикатной программе. — Новосибирск, 2012. — 6с. — (Препр. / ИСИ СО РАН; N 167).
3. Каррано Ф. М. Абстракция данных и решение задач на C++. Стены и зеркала. Вильямс, 2003. 848 с.
4. Касьянов В. П. Оптимизирующие преобразования программ. М.: Наука, 1988. 336 с.
5. Кнут Д. Э. Искусство программирования. Том 1. Основные алгоритмы. Вильямс, 2010. 720 с.
6. Кормен Т. Х. Алгоритмы. Построение и анализ. Вильямс, 2013. 1328 с.

7. Петров Э. Ю. Склеивание переменных в предикатной программе // Методы предикатного программирования. ИСИ СО РАН, Новосибирск, 2003. С. 48–61.

8. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.

9. Шелехов В.И. Предикатное программирование. Учебное пособие. НГУ, Новосибирск, 2009. 109 с.

10. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).

11. Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, №3, 2014. ИСИ СО РАН, Новосибирск. С. 25-43. [Электронный ресурс] URL: http://persons.iis.nsk.su/files/persons/pages/String.pdf

12. Шелехов В.И. Язык предикатного программирования P. Описание языка. - Новосибирск, 2013, [Электронный ресурс] URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf.

13. Celero - A C++ Benchmark Authoring Library. [Электронный ресурс] URL: http://www.codeproject.com/Articles/525576/Celero-A-Cplusplus-Benchmark-Authoring-Library

14. Cooke D. E., Rushton J. N. Taking Parnas's Principles to the Next Level: Declarative Language Design. *Computer*, 2009, vol. 42, no. 9, P. 56-63.

15. Meyer B. Towards a Calculus of Object Programs // Patterns, Programming and Everything, Judith Bishop Festschrift, eds. Karin Breitman and Nigel Horspool, Springer-Verlag, 2012. P. 91-128.

16. Pfaff B. An Introduction to Binary Search Trees and Balanced Trees. [Электронный ресурс] URL: https://ftp.gnu.org/gnu/avl/avl-2.0.2.pdf.gz

17. Pfaff B. Performance Analysis of BSTs in System Software. [Электронный ресурс] URL: http://benpfaff.org/papers/libavl.pdf

18. Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. Vol. 45, No. 7, P. 421–427.

УДК 81`33, 004.8

Подход к извлечению информации из протоколов клинических испытаний на основе медицинской онтологии

Кононенко И.С. (Институт систем информатики СО РАН),
Сидорова Е.А. (Институт систем информатики СО РАН),
Боровикова О.И. (Институт систем информатики СО РАН)

В статье описан подход к организации процесса извлечения информации из протоколов клинических испытаний под управлением онтологии. Рассмотрены отдельные компоненты модели знаний, включая семантический словарь, жанровую модель текста, онтологию клинических испытаний, и приведены примеры извлечения конкретных ситуаций.

Ключевые слова: *извлечение информации, онтология предметной области, предметный словарь, жанр текста, модель факта, клинические испытания, медицинская онтология.*

1. Введение

В последнее время наблюдается резкий рост числа медицинских текстов, посвященных проводимым во всем мире клиническим исследованиям (КИ) лекарственных средств и медицинских технологий. Ежегодно в медицинской литературе появляется порядка 10 000 отчетов о новых завершенных рандомизированных клинических исследованиях [15]. Для получения представления о методике и результатах отдельных клинических исследований необходимо обращение к различным источникам, например, к базе медицинских публикаций MEDLINE. Информация о проводимых испытаниях фиксируется в виде протоколов, которые хранятся в специализированных базах, таких как международный реестр клинических исследований Национального института здоровья США www.clinicaltrials.gov, реестр Минздрава России www.grls.rosminzdrav.ru. Несмотря на свободный доступ к этим базам, поиск необходимой информации затруднен, поскольку отсутствует необходимая структуризация данных, а объемы выдачи исчисляются сотнями документов. С этой точки зрения представляют интерес системы, которые, работая с базой протоколов КИ, обеспечат: а) автоматическую индексацию текстов протоколов на основе онтологии, б) анализ и

структурирование описаний исследований в виде набора фактов, в) содержательный информационный поиск в базе на основе семантического анализа запроса пользователя.

Для обеспечения адекватного поиска необходимой информации в текстах статей, аннотаций и отчетов по КИ используются методы машинного обучения и автоматической обработки текста. В англоязычной литературе представлены работы этого направления, ориентированные на извлечение ключевых элементов КИ: [6-12,19]. Создаваемые системы опираются на существующие медицинские лексиконы и тезаурусы, такие как UMLS и MeSH (см., например, [10,19]). Конкретный набор извлекаемых ключевых элементов варьируется и во многом определяет применяемые методы обработки текста. Так, в [10] рассматривается задача извлечения информации из рефератов базы MEDLINE. Модули извлечения описаний пациентов, заболеваний, основных и сравнительных вмешательств используют правила, созданные вручную, в то время как модуль извлечения исходов основан на методах машинного обучения. Авторы мотивируют это тем, что заболевания и вмешательства описываются в тексте наименованиями, которые напрямую соответствуют концептам медицинского метатезауруса UMLS, описания пациентов представляются в виде шаблонов, включающих соответствующие концепты, в то время как описания исходов не имеют предсказуемой структуры и выходят за рамки именных групп, представляя собой большие фрагменты текста длиной от одного до восьми предложений.

Работа [7] посвящена поиску в текстах аннотаций КИ ключевых предложений, содержащих информацию о вмешательстве, параметрах исходов и участниках, с целью облегчить пользователю поиск релевантных фактов об экспериментальном дизайне КИ. Классификация осуществлена с помощью CRF-метода, для обучения использовался корпус структурированных рефератов. Описанная автоматическая разметка документов может далее использоваться не только для поиска и аннотирования, но и как первый шаг для идентификации и структурирования информации в рамках выбранных предложений. Именно такая двухступенчатая архитектура характеризует системы извлечения информации, описанные в [8,11,19]. В [8] целевая информация охватывает 23 информационных элемента (критерии отбора пациентов, размер выборки, параметры вмешательства, значения параметров исходов и т.п.), представленных в полнотекстовых публикациях о рандомизированных клинических исследованиях. Архитектура системы сочетает в себе текстовый SVM-классификатор, который осуществляет отбор предложений, предположительно содержащих искомую информацию, и поиск и извлечение целевых фрагментов текста, шаблоны которых описаны в виде простых регулярных выражений.

Особенностью содержания протоколов рандомизированных КИ является сопоставление двух (или более) вмешательств – экспериментального препарата и препарата сравнения – и соответствующих групп участников. Эта информация представляется в текстах посредством сравнительных и однородных конструкций, требующих более глубокого лингвистического анализа. В [11] для конструкций, представляющих сравнение видов применяемой терапии и сравнение сущностей, характерное для описаний параметров исходов, применяется семантический анализ. В [6] производится полный синтаксический анализ однородных конструкций, которые часто описывают сопоставляемые типы лечения. Полученные в результате синтаксические признаки используются статистическим классификатором.

Исследовательская работа, требующая анализа протоколов КИ, затруднена разнородностью формального представления информации в различных клинических областях. Вариантом решения этой проблемы является использование технологии Semantic Web для интеграции разнородных приложений на основе онтологии КИ, определяющей общие словарь и семантику [14]. Единое решение проблемы предложено в рамках проекта по созданию банка данных КИ (Trial Bank Project, http://rctbank.ucsf.edu/), которому посвящена серия публикаций: [8,15-18]. В [17] клинические исследования рассматриваются как разновидность научной деятельности человека. Соответствующим образом выстроена онтология OCRe, разработанная как OWL-онтология сущностей и отношений, представленных в протоколах КИ. Онтология не зависит от дизайна и клинической области исследования и ориентирована на интеллектуальную поддержку планирования и анализа КИ, включая поиск протоколов и оценку уже проведенных исследований по конкретной проблеме.

В данной работе предлагается подход к извлечению информации из протоколов КИ в рамках онтологического направления: информация словарей, семантико-синтаксических моделей и правил извлечения существенным образом опирается на структуру онтологии КИ. Второй особенностью предлагаемого подхода является ориентация анализа на специфику жанровой структуры протоколов, которые написаны на естественном языке, но подчиняются строгим требованиям не только к используемым наименованиям препаратов, но и к структуре изложения при описании процесса испытаний. Это дает возможность значительно ограничить область поиска информации путем использования условий на жанровый сегмент в правилах извлечения, благодаря чему высокая точность извлекаемых данных достигается без предварительного этапа классификации для поиска релевантных предложений. Структура извлекаемой информации, а также связь с жанровыми особенностями протоколов

задаются проблемной онтологией, которая фиксирует схему БД и способ ее наполнения данными, полученными в результате анализа текста.

2. Информационные потребности пользователя

Целевую информацию, отвечающую на основные вопросы доказательной медицины, принято представлять в виде фрейма PICO [9]: patient/problem (характеристики субъектов, отобранных для исследования/заболевание), intervention (вмешательство: диагностический тест, лекарственный препарат, терапевтическая процедура), comparison (с чем сравнивается исследуемое вмешательство: отсутствие вмешательства, другой препарат или процедура, плацебо), outcome (исход вмешательства – совокупность контролируемых параметров исходов). К базе MEDLINE обеспечен многоязыковой PICO-интерфейс [13]. Однако даже при формулировке запроса в формате PICO результаты поиска не удовлетворяют клиницистов ввиду огромных объемов выдаваемых ссылок и невозможности формулировки более детализированных информационных запросов.

Разрабатываемая информационная система ориентирована на поиск протоколов прошедших клинических испытаний, удовлетворяющих поисковым запросам различной сложности. Для русскоязычных исследователей актуален двуязычный поиск – как на русском, так и на английском языках.

Поисковые задачи:

- Поиск по ключевым терминам и тегам с учетом синонимов по всей структуре протокола;
- Поиск по сочетанию параметров/фактов;
- Поиск с учетом родо-видовых отношений и других отношений между сущностями (например, препарат – заболевание).

Аналитические задачи:

- Обработка количественных запросов по контролируемым параметрам (биостатистические показатели);
- Анализ успешности испытаний (доказанность основной статистической гипотезы).

3. Модель знаний

Знания о предметной области, используемые в предлагаемом подходе, опираются на модель предметной области, которая фиксирует понятия и отношения между ними в виде онтологии. Онтология КИ (см. Рис.1) содержит классы понятий *Клиническое испытание*, *Препарат*, *Заболевание*, *Группа*, *Цель*, *Результаты*, определяющие состав и условия

проведения испытаний и служащие для представления участников, объектов, целей и результатов КИ. Для отражения специальных знаний из области медицины, анатомии и фармакологии, неявно заданных в описаниях протоколов, в онтологию включены иерархии понятий *Лекарственных средств*, *Анатомических объектов* и *Химических веществ*, связанные с заболеваниями и между собой ассоциативными отношениями. В состав онтологии входят также понятия, относящиеся непосредственно к проведению и организации научной деятельности КИ, такие как *Организации*, *Персоны*, *Географические объекты*, *События*, *Документы*, *Методы исследования*.

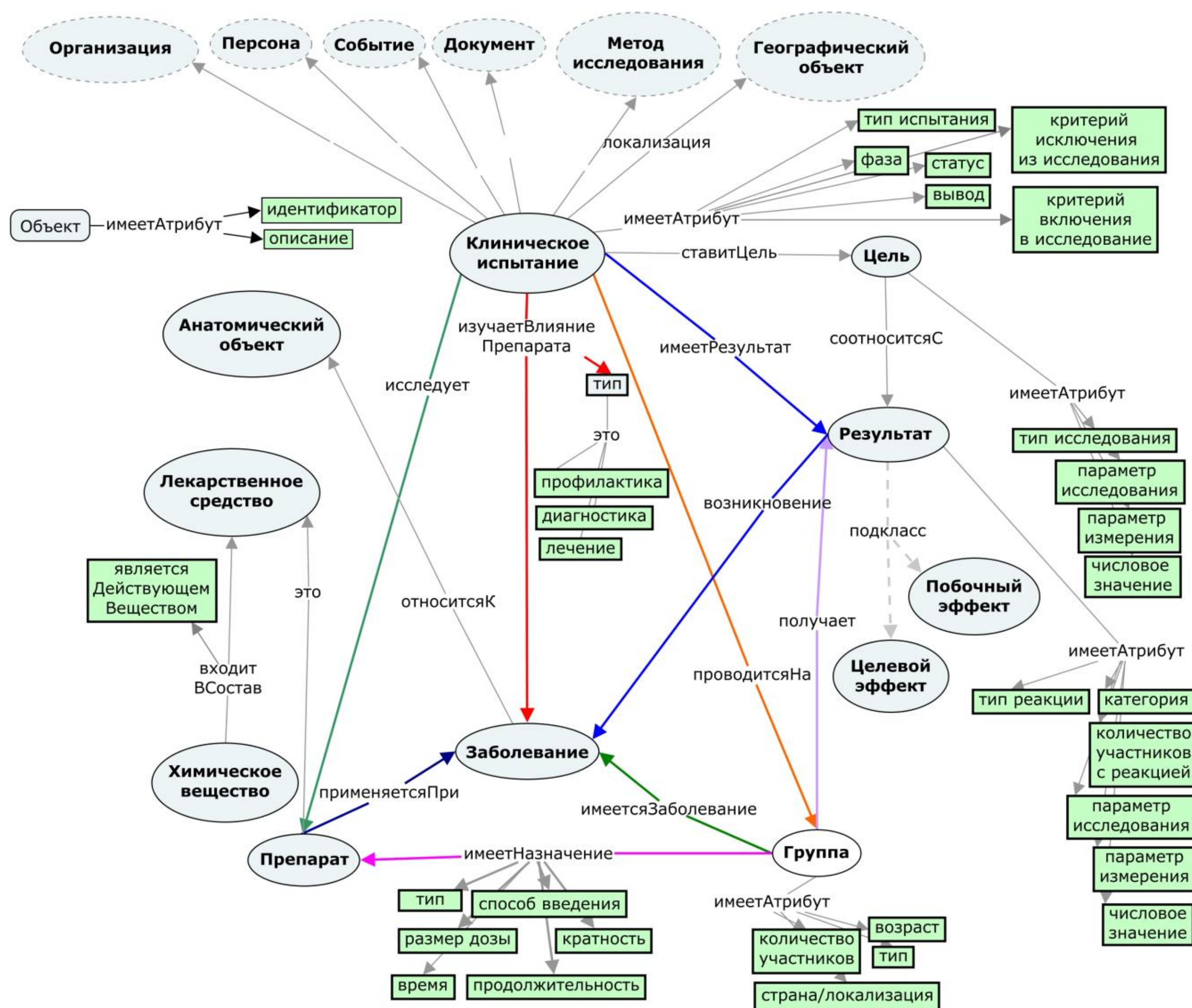


Рис. 1. Фрагмент онтологии предметной области "Клинические испытания".

Понятия онтологии КИ связаны между собой следующими основными отношениями:

«исследует» – связывает непосредственно данное КИ и исследуемый препарат;

«изучаетВлияниеПрепарата» – задает связь между КИ и заболеванием с указанием типа вмешательства: профилактика, диагностика, лечение, исследование качества жизни и т.п.

«проводитсяНа» – связывает КИ и группу участников, включенных в исследование;

«имеетНазначение» – определяет атрибутированную связь между группой участников-пациентов и исследуемым препаратом с заданием характеристик и условий приема препарата;

«имеетРезультат» – задает связь между КИ и полученными результатами (исходами) испытания;

«соотноситсяС» – позволяет задать связь между заданными целями и результатами, планируемыми и полученными в ходе испытаний.

На основе онтологии определяются характеристики информации для извлечения из доступных источников и способ (формат) ее представления для организации хранения и поиска.

Модель знаний о подъязыке предметной области представлена семантическими словарями (словарь предметной лексики, словари лексических шаблонов и семантико-синтаксических моделей управления), моделями фактов, описывающими способы выражения информации, принятые в рассматриваемой области знаний, а также знаниями об особенностях жанра рассматриваемых текстовых источников.

3.1. Текстовая коллекция

Корпус текстов содержит более 200 тыс. xml-документов, извлеченных из базы протоколов КИ, доступной на онлайн-ресурсе ClinicalTrials.gov. В базе представлены данные о клинических исследованиях широкого диапазона препаратов по различным показаниям. Формат и содержание протокола соответствуют принятым стандартам и положениям ICH GCP. В каждом протоколе представлены предопределенные форматом содержательные блоки (цель, задачи, дизайн, методология, статистические показатели исходов и др.), размеченные тегами и расположенные в строгой иерархической последовательности. Приведем фрагмент текста протокола, описывающий дизайн исследования:

```
<study_design_info>
    <allocation>Randomized</allocation>
    <intervention_model>Parallel Assignment</intervention_model>
    <primary_purpose>Prevention</primary_purpose>
    <masking>Double Blind (Participant, Care Provider, Investigator, Outcomes Assessor)</masking>
</study_design_info>
```

Такая формальная схема разметки позволяет построить жанровую модель [1,5] рассматриваемых текстов в виде следующей упрощенной схемы.

```
PageGenre
   Block genre_segment
         Block genre_segment
         …
         Block genre_segment
   Block genre_segment
   …
```

Модель включает жанровые сегменты, маркированные жанровыми тегами, на основе которых извлекаются фрагменты текста для поиска той или иной информации.

Перечислим основные типы жанровых тегов рассматриваемых протоколов:

<brief_title>,<official_title>,<brief_summary>,<detailed_description> – описания типа, целей и содержания проводимых клинический испытаний;

<primary_outcome> – ожидаемые (целевые) результаты исследования;

<arm_group> – описание групп;

<intervention> – описание вмешательства;

<clinical_results>, <outcome> – описание результатов;

<reported_events>, <event> – побочные эффекты и т.д.

Жанровые особенности анализируемых текстов можно представить как совокупность следующих признаков.

Структурные особенности:

- Описание представлено иерархически организованными текстовыми блоками;
- Семантические единицы привязаны к структурным фрагментам текста.

Лексические особенности:

- Однозначность и единообразие терминов благодаря использованию номенклатурной лексики – наименований препаратов, заболеваний;
- Использование принятых аббревиатур и сокращений *ACAM200, US., PFU/ml, PRNT50.*

Грамматические особенности:

- Целевая информация представлена преимущественно редуцированными и полными именными группами, параметрическими и однородными конструкциями;
- Анафорические отсылки используются редко, представлены относительными местоимениями *which, who, that* и не выходят за рамки предложения.

Использование знаний о жанровых особенностях текста позволяет значительно ограничить разнообразие способов передачи информации, учитываемых в моделях фактов.

3.2. Словарь

Словарь системы создается путем обучения на представительном корпусе КИ, но ядро словаря составят термины тезауруса MeSH (https://www.nlm.nih.gov/mesh/), который в

версии 2016 г. содержит 27 883 дескрипторов, более 87 тыс. входных терминов-синонимов и 232 тыс. дополнительных концепт-записей – наименований конкретных химикатов, болезней и медикаментов.

Словарь состоит из основного словаря и словаря лексических шаблонов. В этих словарях фиксируется семантически значимая лексика, представляющая элементы целевой информации. Система семантических признаков в словарях основана на структуре онтологии клинических испытаний, отражая иерархию ее объектов и отношений. Объектные термины представлены преимущественно существительными (нарицательными и собственными именами), именными группами, лексическими конструкциями (аббревиатурами и более сложными буквенно-символьными конструкциями). С помощью семантических признаков объектные термины распределены по основным классам:

- Деятельность
  - Вмешательство (*intervention*)
  - Лечение (*therapeutics, acupuncture therapy, radiation treatment*)
  - Профилактика (*prophylaxis, preventive therapy, preventive procedure, vaccination, vaccinate*)
  - Диагностика/Обследование (*diagnostic procedure, diagnostic test, investigative techniques, blood chemical analysis*)
- Клиническое_испытание (*clinical trial, clinical study*)
- Препарат (*drug, organic chemicals, pharmaceutical, insulin lente, biological product, vaccine, herpesvirus vaccine, smallpox vaccine, ACAM2000, gD-Alum/MPL vaccine*)
- Болезнь
  - Вирусная_болезнь (*herpes henitalis, smallpox, hepatitis A*)
- Патологическое_состояние/признак/симптом (*asthenia, cyanosis, swelling, papule, pain, burning, itching, tingling, dysuria*)
- Состояние_здоровья (*healthy*)
- Анатомический объект
  - Система (*Cardiovascular System*)
  - Орган (*Heart, Miocardium*)
  - Локализация (*head, ear*)
- Участник (*participant, population*)
  - Персона (*patient, subject*)
  - Группа (*arm, cohort*)
- Организация
- Географический объект
  - Регион (*region, Europe*)
  - Страна (*Japan*)
  - Город (*city, New York, Moscow*)
- Временной объект

- Дата (*March 31, 2003*)
- Период (*from 10 January 2003 to 14 April 2003*)

Отдельный семантический класс Параметр составляют лексические единицы, описывающие параметрические характеристики объектов: доза, кратность, время и др. Объекты класса Участник описываются с помощью характеристик “этническая группа” (*african american, arab*), “пол” (*male, female, woman*), “возраст” (*age, aged, adult, adolescent, child, baby, 28 years, 50-59 years*). Группа и Клиническое испытание характеризуются признаком “тип” (*experimental, active comparator, placebo comparator*)**.** Назначение препарата характеризуют “доза” (*1.0x10-8th plaque-forming units/mL*), “время” (*on Day 0*), “способ” (*orally, parenteral, intramuscular*), “кратность” (*single, twice*) и “продолжительность”.

Более детальная классификация терминов обусловлена такими онтологическими свойствами объектов, которые проявляются на уровне репрезентации в языковых конструкциях. Так, дополнительный семантический признак “колич” характеризует параметры, значения которых могут представляться нумеративной конструкцией. Признак “эталон” характеризует лексемы, представляющие стандартные оценки количественных параметров (*adult* vs. *24 to 34 years old, standard-dose, high dose* vs. *0.5 mL*). Специальные признаки выделяют элементы языковых конструкций параметрической семантики: “число” (*five*), “функция” (*more, equal, above, to*), “мера” (*milli-International Units, milliliter, microgram, mL*). Кроме того, отдельными признаками выделяются показатели временных отношений (*between <months 2 and 3>, preceeding, after*), однородности (*or, and*) и отрицание (*free <of>, without, not*).

Лексические признаки “тип”, “знач”, “имя” фиксируют особенности сочетаемости терминов в языковых конструкциях. Так, признаком “тип” выделены существительные-классификаторы, называющие объекты того или иного класса в общем виде (*vaccine*). Признак “имя” характеризует имена собственные (например, наименования препаратов *qHPV***,** *Dryvax*®, *ACAM2000*).

Для извлечения дат и временных интервалов, сокращенных и стандартных наименований препаратов (*qHPV***,** *ACAM2000*), значений параметров, представленных числовыми конструкциями (**<***temperature*> *above 99.0°F, <dose:> 2.0x10-7th PFU/ml*), используется словарь лексических шаблонов. Шаблоны позволяют задать порядок следования элементов конструкций, описывающих наименования объектов, и учесть их написание с заглавной буквы, курсивом, латиницей, через дефис /тире или в кавычках. Так, типичная конструкция для дозы препарата представляется с помощью следующих шаблонов:

[кратное_число] = [число](_)x(_)10(_)-(_) [цел_число] (_) (th)
[мера] =
plaque(_)(-)(_)forming unit…(_)(/)(_)ml
PFU(_)(/)(_)ml
[доза] = **[кратное_число] [мера]**
2.0x10-7th plaque-forming units/mL

Помимо семантически значимой лексики, словарь лексических шаблонов содержит класс Жанровой лексики. Это теги, содержащие слова и словосочетания (в том числе конструкции с подчерком), которые могут рассматриваться как индикаторы целевой информации. В процессе анализа используется структурированность блоков содержания с помощью разметки. Извлечение конкретных элементов целевой информации происходит в пределах выделенных индикаторами жанровых сегментов.

5. Поиск информации

При извлечении информации мы, помимо онтологии клинических испытаний, будем опираться на типичные информационные потребности пользователя-исследователя. Особый интерес представляют «содержательные» ситуации, описывающие проводимый эксперимент и его результаты. На текущий момент мы выделили четыре типа запросов:

1) Запросы, обеспечивающие поиск по характеристикам участников (пациентов) исследования, т.е. запросы на условия, применяемые к группам или когортам.
   Найти испытания, которые проводились над участниками
   - *с заболеванием* D,
   - *с расовой принадлежностью* R,
   - *возрастом до* A *лет*,
   - *живущих в стране* C,
   - *в которых применялась терапия типа* T.
2) Запросы, обеспечивающие поиск по особенностям применения препарата.
   Найти испытания, при которых применялся способ лечения препаратом P
   - *размер дозы меньше* X / *больше* X, *в интервале от* X1 *до* X2,
   - *вводится* Y *раз* / *однократно* / *многократно*,
   - *в течение времени* T / *меньше* T / *больше* T,
   - *способ доставки* W.
3) Запросы, обеспечивающие поиск по сочетанию ключевых элементов.
   Найти испытания, которые проводились
   - *для лечения заболевания* D/*для профилактики заболевания* D/*заболевания типа* TD,
   - *используя препарат* P/ *препарат типа* TP,
   - *с применением терапии типа* T.

4) Запросы, обеспечивающие поиск по результатам исследований.
Найти успешные исследования / с серьезными побочными эффектами
   - *с применением терапии типа* T / *препарата* P,
   - *для лечения заболевания* D / *для профилактики заболевания* D / *заболевании типа* TD,
   - *с наличием эффекта* X.

Таким образом, в соответствии с представленными типичными запросами, мы будем рассматривать следующие типы ситуаций: это, во-первых, описание участников испытаний и их деление по группам (*мужчины пожилого возраста, группа плацебо*), во-вторых, информация о препаратах, способах их применения, характере и особенностях проводимого лечения (*применение препарата в течение месяца 2 раза в день*), в-третьих, ситуации, характеризующие цели проводимых испытаний (*исследование безопасности дозы препарата*), и, наконец, описание полученных результатов (*положительный эффект был достигнут в 85% случаев*) и их соответствие поставленным целям.

В соответствии с представленными ситуациями сформирована схема универсальной ситуации, в которой имена классов выступают в качестве параметров поискового запроса:

Группа участвовала в *Испытании* с использованием *Препарата* для лечения/профилактики *Заболевания* с результатом *Результат* и побочным эффектом *Эффект*.

Найденные и распознанные ситуации можно представить в виде набора фактов, которые формально описываются экземплярами классов онтологии, значениями их атрибутов и связями. Для поиска и извлечения фактографической информации применяется технология анализа текста FATON [4], использующая ряд лингвистических ресурсов – терминологические словари, снабженные системой семантических признаков, а также лингвистическую модель предметной области КИ, содержащую набор *моделей фактов*, позволяющих в терминах семантических и грамматических признаков описывать способы выражения требуемой онтологической информации.

Каждая модель факта описывается схемой (правилом), которая включает набор аргументов структуры факта (arg1, arg2, …), их семантические/грамматические признаки, условия на семантико-синтаксическую сочетаемость характеристик аргументов, и набор объектов, который фиксирует структуру факта в онтологическом представлении. Рассмотрим несколько примеров и набор необходимых моделей для извлечения из них целевой информации.

4.1. Инициализация объектов. Как показано выше, система семантических признаков словаря формируется на основе онтологических сущностей, что позволяет инициализировать начальное формирование объектов непосредственно на основании словарных признаков.

Объект класса *Препарат* может быть представлен в тексте аппозитивной именной группой, в которой опорным словом является родовое слово или словокомплекс (тип), а имя примыкает к нему в постпозиции. Например,

Вакцина <Препарат, SemClass: тип> *"ACAM2000"* <Препарат, SemClass: имя> [1]

извлекается с помощью модели:

Scheme Препарат3 : segment *Клауза* **(1)**
arg1: Term::Препарат(SemClass: тип)
arg2: Term:: Препарат(SemClass: имя)
Condition PrePos(arg1,arg2), Contact(arg1,arg2)
⇨ Object :: Препарат(Тип: arg1.Class & arg2.Class, Наименование: arg2.Norm)

В данной схеме термины должны иметь семантических класс *Препарат*, с учетом иерархии наследования признаков в словаре, а также первый термин должен обладать семантическим признаком *тип*, а второй – *имя*. На основе схемы создается объект – экземпляр понятия онтологии Препарат, тип препарата (например, фармакологическая группа) может уточниться в соответствии с семантическим признаком первого или второго термина, атрибут *Наименование* у объекта заполняется предпочтительным наименованием второго термина (Norm), заданным в тезаурусе для данного дескриптора (при наличии других входных терминов, синонимичных данному). Аналогичным образом могут извлекаться объекты *Заболеваний*, *Организаций* и т.п., если их названия присутствуют в словаре.

Инициализация объектов типа *Группа* возможна не только по названию, но и по присвоенному индексу, например, из фрагмента вида *“<group group_id="P5">”*.

Особо следует отметить случаи, когда на основе лексического шаблона (LexTerm) выделяется фрагмент текста в кавычках и формируется гипотеза о том, что это имя объекта, но уточнение его класса возможно только при наличии термина-классификатора, либо при последующей сборке ситуации (например, на основе семантической роли в ситуации).

Scheme Новый_объект : segment *Клауза* **(2)**
arg1: Term:: (SemClass: тип)
arg2: LexTerm::Именованный объект()
Condition PrePos(arg1,arg2), Contact(arg1,arg2)
⇨ Object :: Object (Тип: arg1.Class, Наименование: arg2.Name)

Появление таких объектов объясняется либо неполнотой базы знаний (например, при употреблении новых наименований препаратов, которые еще не зафиксированы в

---

[1] В примерах в скобках указываются признаки терминов, заданные в словаре.

онтологиях), либо наличием ошибок в тексте (в этом случае объект можно сопоставить с другими объектами того же типа, встречающимися в тексте рассматриваемого протокола).

4.2. Извлечение фактов. При поиске и выявлении характеристик объектов и их связей, как правило, требуется проверить сочетаемость семантических и/или грамматических признаков объектов. Для описания сочетаемости предикатных лексем разрабатывается словарь семантико-синтаксических конструкций (аналог моделей управления), который фиксирует семантические валентности предикатов, описывая их в терминах грамматических и семантических признаков актантов. Это позволяет проверять наличие управления в анализируемом фрагменте текста, т.е. согласованность семантических и синтаксических признаков предиката и актантов.

Рассмотрим примеры формирования с помощью моделей фактов фрагмента онтологии, описывающего характеристики объектов в рамках ситуации клинического испытания.

Извлечение характеристик объектов. Рассмотрим примеры схем, используемых для извлечения атрибутов объектов.

Scheme ТипКИсследования: genre_segment *<brief_title>* or *<official_title>* **(3)**
arg1: Term::Параметр(SemClass: цель_исследования)
arg2: Object::Препарат()
Condition PrePos (arg1,arg2), Contact(arg1,arg2), Упр(arg1,arg2)
⇨ Relation::изучаетВлияниеПрепарата (исследование: $this_CT, Препарат: arg2)
$obj1 = Object::Цель (тип: arg1.Name)
Relation::ставитЦель(исследование: $this_CT, цель: $obj1)

Данная схема позволяет извлекать информацию о цели проводимого исследования, зафиксированную в жанровых полях протокола *<brief_title>* или *<official_title>*. Так, из фрагмента *“Dose Study of ACAM2000 Smallpox Vaccine in Previously Vaccinated Adults …”* будет извлечен факт о том, что исследование посвящается изучению дозы вакцины от оспы.

Следующая схема позволит уточнить тип группы пациентов, принимающих участие в исследованиях.

Scheme ТипКогорты: genre_segment *<arm_group>* **(4)**
arg1: Object:: Группа(), genre_segment *<arm_group_label>*
arg2: Term::Параметр(), genre_segment *<arm_group_type>*
⇨ arg1: Группа (тип: arg2.Name)

Данная схема применяется для фрагментов вида:

<arm_group>
<arm_group_label>Group 5: Dryvax®</arm_group_label>
<arm_group_type>Active Comparator</arm_group_type>
</arm_group>

Создание отношений. Рассмотрим пример схемы построения отношения в соответствии с рассматриваемой ситуацией.

Scheme УсловияПримененияПрепарата: genre_segment <*group*> **(5)**
arg1: Object::Группа()
arg2: Object::Препарат()
arg3: Term::Параметр(SemClass: кратность)
arg4: LexTerm::Доза()
arg5*: Term::Параметр(SemClass: время)
Condition genre_segment (arg2, arg3, arg4, arg5) <*description*>,
Contact(arg2, arg3), Contact_weak(arg2, arg4), Contact_weak(arg2, arg5)
⇨ Relation::имеетНазначение (группа: arg1, препарат: arg2, тип: «препарат»,
размер дозы: arg4.value, кратность: arg3.value, время: arg5.value)

Условия клинических испытаний для конкретной группы участников испытаний при назначении препарата описывается такими характеристиками, как наименование препарата, его дозировка, кратность применения и время приема. Данная информация в соответствии с принятым стандартом содержится строго в определенных жанровых фрагментах, однако в рамках фрагмента <*description*> описание параметров разворачивается в виде текста из одного-двух предложений, что требует применения более сложного лингвистического анализа (особенно в случаях комплексного применения препаратов).

Данная схема покрывает фрагменты вида:

<group group_id="P5">
<title>Dryvax® Vaccine</title>
<description> Participants received a single dose of Dryvax® smallpox vaccine, 1.0x10-8th plaque-forming units/mL on Day 0
</description>
</group>

В результате применения схемы к данному фрагменту текста будет создано описание ситуации терапевтического вмешательства для конкретной группы участников.

Приведенный набор моделей фактов демонстрирует подход к извлечению информации о проводимом клиническом исследовании на основе структуры протокола.

Разрешение кореференции объектов. Важной проблемой анализа текста является установление кореферентности объектов при их повторном упоминании. В общем случае онтологический подход позволяет разрешить кореференцию после основного анализа текста в процессе сравнения и идентификации объектов (относительно онтологии). Эквивалентными с точки зрения онтологии считаются объекты с непротиворечивыми классами и наборами атрибутов [2].

Особенности протоколов КИ, содержащих однозначную номенклатурную лексику, распределенную по разным структурным блокам, упрощают процедуру поиска эквивалентных с точки зрения онтологии объектов.

Заключение

В статье описан подход к организации процесса извлечения информации под управлением онтологии. Рассмотрены отдельные компоненты системы и приведены примеры извлечения конкретных ситуаций, описывающих клинические испытания.

В нашей лаборатории создан ряд инструментов, поддерживающих все этапы разработки и эксплуатации систем извлечения информации с опорой на онтологию [3]. Для разработки информационной базы клинических испытаний используются следующие инструменты: а) технология построения предметных словарей KLAN, поддерживающая методы машинного обучения, тематической и жанровой классификации, морфологического и поверхностно-синтаксического анализа текстов и обеспечивающая эксперта-лингвиста широким набором инструментов для отладки словаря; б) технология построения лексических шаблонов DigLex, обеспечивающая поиск в тексте несловарных конструкций, таких как сокращения, буквенно-числовые обозначения препаратов, химические названия веществ и т.п.; в) система жанровой сегментации текстов; г) система фактографического анализа текстов FATON, которая реализует обработку текста на основе схем фактов и обеспечивает пополнение БД системы; д) технология построения портала знаний, обеспечивающая доступ пользователей к информационному наполнению базы данных, содержательный поиск и навигацию на основе онтологии.

Планируется апробировать предложенный подход на публичной базе английских текстов клинических испытаний, представленных на сайте *ClinicalTrials.gov*, и в дальнейшем направить усилия на создание аналогичной системы для русскоязычного контента.

Список литературы

1. Кононенко И. С., Сидорова Е. А. Жанровые аспекты классификации веб-сайтов // Программная инженерия. 2015. № 8. С. 32–40.
2. Серый А.С., Сидорова Е.А. Поиск референциальных отношений между информационными объектами в процессе автоматического анализа документов // Труды XIV Всероссийской научной конференции RCDL-2012 Электронные библиотеки: перспективные методы и технологии, электронные коллекции. Переславль-Залесский, 2012. С. 206-212.
3. Сидорова Е.А. Разработка лингвистического обеспечения информационных систем на основе онтологических моделей знаний // Известия Томского политехнического университета. 2013. Т. 322. № 5. С. 143-147.
4. Сидорова Е.А. Фактографический анализ текста в контексте интеллектуальных информационных систем // Информационные и математические технологии в науке и

управлении: тр. XVIII Байкальской Всероссийской конференции. Иркутск: Институт систем энергетики им Л.А. Мелентьева СО РАН, 2013. Т.3. С. 79-85.

5. Сидорова Е.А., Кононенко И.С. Представление жанровой структуры документов и ее использование в задачах обработки текста // Труды Седьмой Международной конференции памяти академика А.П. Ершова "Перспективы систем информатики". Рабочий семинар «Наукоемкое программное обеспечение». Новосибирск: Сибирское Научное Издательство, 2009. С. 248-254.
6. Chung G.Y. Towards identifying intervention arms in randomized controlled trials: extracting coordinating constructions // Journal of Biomedical Informatics. Vol. 42. 2009. P. 790–800.
7. Chung G.Y., Coiera E. A study of structured clinical abstracts and the semantic classification of sentences // Proceedings of the Workshop on BioNLP 2007: Biological, Translational, and Clinical Language Processing. 2007. P. 121–128.
8. De Bruijn B., Carini S., Kiritchenko S., Martin J., Sim I. Automated information extraction of key trial design elements from clinical trial publications // Proceedings of AMIA Annual Symposium. 2008. P. 141-155.
9. Demner-Fushman D., Lin J. Answering clinical questions with knowledge-based and statistical techniques. Computatinal Linguistics. Vol.33 (1). 2007. P. 63-103.
10. Demner-Fushman D, Lin J. Knowledge Extraction for Clinical Question Answering: Preliminary Results // Proceedings of AAAI Workshop on Question Answering in Restricted Domains. 2005. P. 1–9.
11. Fiszman M, Demner-Fushman D, Lang FM, Goetz P, Rindflesch T. Interpreting comparative constructions in biomedical text // Proceedings of the BioNLP workshop, association for computational linguistics. 2007. P. 137–44.
12. Ke-Chun Huang, I-Jen Chiang, Furen Xiao, et al. PICO element detection in medical text without metadata: Are first sentences enough? // Journal of Biomedical Informatics. Vol.46. 2013. P. 940-946.
13. PICO Linguist. [Electronic resource]. URL: http://babelmesh.nlm.nih.gov/pico.php (Accessed: 9/05/2017).
14. Ravi D. Shankar, Susana B. Martins, MD, Martin O’Connor, David B. Parrish, Amar K. Das. An Ontology-based Architecture for Integration of Clinical Trials Management Applications. //Proceedings of AMIA Annual Symposium. 2007. P. 661-665.
15. Sim I. The Trial Bank Project. [Electronic resource]. URL: http://grantome.com/grant/NIH/R01-LM006780-10 (Accessed: 9/05/2017).
16. Sim I., Olasov B., Carini S. An ontology of randomized controlled trials for evidence-based practice: content specification and evaluation using the competency decomposition method. Journal of Biomedical Informatics. Vol.37. 2004. P. 108-119.

17. Sim I., Tu Samson W., Carini S. et al. The Ontology of Clinical Research (OCRe): an informatics foundation for the science of clinical research // Journal of Biomedical Informatics. Vol.52. 2014. P. 78-91.
18. Tu Samson W., Peleg M., Carini S.et al. A practical method for transforming free-text eligibility criteria into computable criteria // Journal of Biomedical Informatics. Vol.44, 2011. P. 239-250.
19. Xu R, Garten Y, Supekar KS, Das AK, Altman RB, Garber AM. Extracting subject demographic information from abstracts of randomized clinical trial reports // Studies in Health Technology and Informatics. Vol.129. 2007. P. 550–554.

УДК 004.052, 519.179.2

Онтологический подход к организации шаблонов требований в рамках системы поддержки формальной верификации распределенных программных систем

Н.О. Гаранина (Институт систем информатики им. А.П. Ершова),

В.Е. Зюбин (Институт автоматики и электрометрии),

Т.В. Лях (Институт автоматики и электрометрии)

В статье описывается структура онтологии шаблонов требований, извлекаемых из текстов технической документации. Эта онтология комбинирует шаблоны известных классификаций требований с новыми шаблонами. Язык онтологии допускает запись булевых комбинаций шаблонов следующих типов: качественных, реального и ветвящегося времени, с комбинированными событиями, количественными характеристиками событий и простыми утверждениями о данных. Приведены примеры требований к реальной системе управления вакуумированием Большого солнечного вакуумного телескопа. Изложена схема интеллектуальной системы поддержки формальной верификации программных распределенных систем.

Ключевые слова: шаблоны требований, инженерия требований, темпоральные логики, онтология

1. Введение

Данная работа выполняется в рамках проекта “Методы извлечения формальных спецификаций программных систем из текстов технических заданий и их верификация”. Проект посвящён проблеме обеспечения качества программных систем с помощью формальных методов. В рамках проекта планируется разработать комплексный подход к извлечению формальных моделей и свойств распределенных программных систем из текстов технической документации с последующей их верификацией. Под распределенной программной системой (РПС) здесь мы понимаем систему, состоящую из множества параллельно исполняемых взаимодействующих компонент.

Комплексный подход подразумевает создание системы поддержки формальной верификации распределенных программных систем. Известны системы поддержки разработки требований на основе шаблонов [1, 13, 16, 18–20], но они позволяют только ручное

формулирование требований с последующей визуализацией и определением точной формальной семантики. Разрабатываемая нами система предполагает автоматическое порождение требований к системе с последующей ручной коррекцией. Порождаемые требования также имеют формальную семантику, выраженную формулами некоторой темпоральной или модальной логики, определение на естественном языке и визуализацию. В настоящей работе разработана онтология требований и определена их формальная семантика. Онтология требований сочетает известные шаблоны требований, адаптированные и расширенные новыми шаблонами для более выразительного и единообразного представления.

Систематизированное задание требований и их формальной семантики как формул темпоральных логик является хорошо исследованной задачей. Известны различные классификации формул темпоральных логик. Наиболее ранняя классификация [12] является синтаксической и касается самых общих свойств программных систем (живость, безопасность, справедливость, блокировка). Ставшая классической система шаблонов из [4] отражает наиболее характерные качественные требования к системам различного назначения. При этом каждый шаблон описан на естественном языке и дана его формализация на языках CTL, LTL [3], квантифицированных регулярных выражений и графического представления GIL. В работах [8, 11] эти шаблоны были расширены на случай систем реального времени и вероятностных систем соответственно. В работах [13, 16] предложены варианты шаблонов составных событий. В [2] авторы дополняют качественные и временные шаблоны шаблонами, отражающими количественные характеристики появления событий, а также шаблоном данных, впервые упомянутым в [9]. Все упомянутые работы оперируют только шаблонами, выразимыми в логике линейного времени LTL и её реально-временных и вероятностных расширениях, однако в статье [14] отмечается необходимость в некоторых случаях использовать ветвящееся время и логику CTL с соответствующими расширениями. Недавняя работа [1] комбинирует описания классических шаблонов с вероятностными и шаблонами реального времени и даёт их описание на ограниченном английском языке. Такие описания можно использовать для генерации правил извлечения требований из текстов технической документации. В работе [20] классификация шаблонов также представлена в виде онтологии, однако набор шаблонов свойств систем весьма ограничен, а формализация семантики самих паттернов вовсе отсутствует.

В рамках разработки онтологии требований полезно организовать уже известные системы шаблонов в единую онтологическую классификацию, добавив некоторые шаблоны

и области действия, и выработать соответствующую онтологию собственно требований (классы, отношения, домены), наполнение которой уникально для каждого отдельного комплекта технической документации. Такой комплексный систематизированный подход к представлению шаблонов спецификаций позволяет с помощью небольшого набора атрибутов описывать широкий спектр свойств (требований) взаимодействующих параллельных систем. Эта широта важна потому, что для одной и той же системы бывает необходимо задавать как простые, легко верифицируемые свойства типа достижимости выделенного состояния системы, так и сложные свойства, зависящие от реального времени работы компонент системы. Возможность формулировать такие разнообразные свойства в рамках одного формализма повышает качество поддержки процесса разработки таких систем, поскольку позволяет целиком охватывать всю картину требований к системе. Онтология требований будет представлена на языке OWL.

Шаблоны требований нашей онтологии имеют строгую семантику, выраженную формулами темпоральных логик CTL, LTL и их расширений реального времени, а также мюисчисления (MuC), что позволяет, с одной стороны, чётко выражать требования, а с другой — однозначно определять подходящий метод верификации. Онтология требований допускает расширение свойствами систем, которые не описываются темпоральными логиками, однако опускают эффективную верификацию. Классификация практически-значимых поведенческих свойств распределенных программных систем, используемых в технической документации, которые могут быть эффективно верифицированы в моделях, позволит упростить процедуру проверки соответствия модели РПС и требований к ней. Онтология требований пополняется из текстов технической документации. На настоящий момент она содержит 13 классов требований и 15 отношений между ними с параметрами, конкретные значения которых извлекаются из технической документации.

В данной работе представлена онтология требований, которые могут задаваться булевой комбинацией шаблонов следующих типов: качественных; реального времени; ветвящегося времени; допускающих комбинированные события; позволяющих учитывать количественные характеристики событий, а также простые утверждения о данных. Добавлены шаблоны, задающие свойства оптимальности и устойчивости разрабатываемой системы к нежелательному поведению среды. Следующий раздел 2 описывает классы и отношения онтологии требований с примерами их формальной семантики. В разделе 3 приведены примеры требований к реальной системе управления вакуумированием Большого солнеч-

ного вакуумного телескопа. В следующем разделе 4 изложена схема интеллектуальной системы поддержки разработки программных распределенных систем. В заключении 5 мы обсуждаем дальнейшие планы.

Благодарности. Исследование поддержано Российским Фондом Фундаментальных Исследований (грант № 17-07-01600).

2. Онтология требований

Мы считаем *онтологией* структуру, включающую следующие элементы: (1) конечное непустое множество *классов*, (2) конечное непустое множество *атрибутов-данных и атрибутов-отношений*, и (3) конечное непустое множество *доменов атрибутов-данных*. Каждый класс определяется набором атрибутов. Атрибуты-данные принимают значения из домена, а значениями атрибутов-отношений являются экземпляры классов. *Экземпляр класса* определяется набором значений атрибутов этого класса. Класс c_2 является *подклассом* c_1 если и только если все экземпляры класса c_2 являются также экземплярами класса c_1. Подкласс наследует все атрибуты родительского класса. *Информационный контент* онтологии — это набор экземпляров её классов. Задача пополнения заданной онтологии состоит в извлечении информационного контента этой онтологии из входных данных. В нашем случае такими данными для пополнения онтологии программной системы и онтологии требований является техническая документация. Для пополнения обеих онтологий мы планируем использовать нашу систему семантического извлечения информации из текстов на естественном языке [5–7]. Кроме того, контент онтологии требований также может пополняться из онтологии программной системы с помощью специального транслятора, разработка которого запланирована на ближайшее время.

В следующих подразделах дано подробное описание классов и доменов нашей онтологии требований. В таблицах, перечисляющих атрибуты классов, имена классов выделены курсивом, имена доменов и их значения – телетайпом. Для пояснения семантики требований определим следующие понятия. Мы рассматриваем *модель системы* в духе CSP [10] как параллельное исполнение последовательных процессов, каждый из которых задаётся сменой своих состояний. *Состояния объекта-процесса* определяются набором значений его переменных. Взаимодействие объектов-процессов происходит путём обмена сообщениями и через изменение разделяемых переменных. *Состоянием системы* является набор состояний объектов-процессов, т.е. мгновенное описание значений их перемен-

Table 1

Спецификации

<table>
<tr><td rowspan="2">SimSpec</td><td>Operation</td><td>S1</td><td>S2</td></tr>
<tr><td rowspan="2">$\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$</td><td>Proposition
SimSpec</td><td>Proposition
SimSpec</td></tr>
<tr><td>Spec</td><td>Pattern
Spec</td><td>Pattern
Spec</td></tr>
</table>

ных. Далее мы будем использовать понятие *события-утверждения* (или просто *события*), которое может иметь место в определённых состояниях системы. В данной статье мы рассматриваем примеры формальной семантики шаблонов, а задача полного описания семантики имеет технический характер.

2.1. Спецификации

Спецификации представленные в таб.2.1, задают требования к системе. Простые спецификации требований, задаваемые классом *SimSpec*, являются булевыми комбинациями событий-утверждений, представленных классом *Proposition*, который описан ниже. Эти комбинации строятся с помощью атрибута-данного “Operation”, используемого для булевых операторов, и пары атрибутов-отношений “S1” и “S2” для двух операндов. Например, в классе *SimSpec* экземпляр ss = (Operation: $\rightarrow$, S1: pr_{Start}, S2: $pr_{AllReady}$), где pr — экземпляры класса *Proposition*, обозначает спецификацию требования $Start \rightarrow AllReady$. Такие простые спецификации используются при задании сложных спецификаций, включающих шаблоны, и ограничений на исполнение простых событий класса *Proposition*. Класс *Spec* является подклассом *SimSpec*. Он представляет спецификации, являющиеся булевой комбинацией простых спецификаций *SimSpec* и шаблонов, представленных классом *Pattern*. Например, в классе *Spec* экземпляр s = (Operation: $\rightarrow$, S1: pr_{Start}, S2: $ptU_{AllRight}$), где pr — экземпляр класса *Proposition*, а pt_U — экземпляр класса *Pattern*, обозначает спецификацию требования, которая выражается в логике LTL как $Start \rightarrow \mathbf{G}AllRight$.

Table 2

Шаблоны

<table>
<tr><th>Pattern</th><th>Kind</th><th>S1</th><th>S2</th><th>Space</th><th>T.Type</th><th>Fr.Sco</th><th>Fr.Time</th><th>Fr.Qnt</th><th>Cstr</th></tr>
<tr><td>Occurence</td><td><code>Univ</code>
<code>Exist</code>
<code>Abs</code></td><td rowspan="2">Prop</td><td></td><td></td><td rowspan="2"><code>linear</code>
<code>branch</code></td><td rowspan="2">Scope</td><td rowspan="2">Bound</td><td rowspan="2">Bound</td><td rowspan="2">Prop</td></tr>
<tr><td>Order</td><td><code>Prec</code>
<code>Resp</code></td><td>Prop</td><td><code>Novlap</code>
<code>Rovlap</code>
<code>Lovlap</code></td></tr>
</table>

2.2. Шаблоны

Шаблоны, представленные в таб.2.2 основаны на классических шаблонах появления *Occurence* и порядка *Order* из [4]. Они снабжены рядом атрибутов-спецификаторов, позволяющих задавать различные дополнительные ограничения, такие как время появления относительно определённых событий-утверждений (“FrameScope”), время выполнения как самого шаблона, так и его событий, по логическим часам (“FrameTime”) и количество повторений выполнения самого шаблона, либо его событий (“FrameQuantity”). Шаблоны также могут быть снабжены атрибутом “TimeType” для определения линейного, либо ветвящегося времени шаблона. Утверждения, которые не должны выполняться одновременно с данным шаблоном, задаются значением атрибута “Constraint”.

Шаблоны появления описывают появление некоторого события в процессе работы системы с помощью класса *Occurence*, который является наследником класса *Pattern* и может использовать все его атрибуты. Событие задаётся атрибутом-отношением “S1” со значением в классе *Proposition*, а тип появления события — атрибутом “Kind”∈{`Absence, Existence, Universality`}. Семантика этих шаблонов для линейного времени и при отсутствии ограничений описывается следующим образом:

- `Universality`: S_1 имеет место всегда.
  - LTL: $\mathbf{G}S_1$.
- `Existence`: S_1 когда-нибудь случится.
  - LTL: $\mathbf{F}S_1$.
- `Absence`: S_1 никогда не случается.
  - LTL: $\mathbf{G}\neg S_1$.

С помощью *шаблонов порядка* можно выразить относительное появление двух событий в процессе работы системы используя класс $Order$, который является наследником класса $Pattern$. Первое событие шаблона задаётся атрибутом-отношением "S1", а второе — атрибутом-отношением "S2" со значениями в классе $Proposition$. Тип порядка задаётся с помощью атрибута "Kind" $\in$ {`Precedence, Response`}. Семантика этих шаблонов для линейного времени и при отсутствии ограничений описывается следующим образом:

- `Precedence`: если случилось S_1, то до этого когда-то случилось S_2.
  – LTL: $\mathbf{F}S_1 \to \neg S_1\mathbf{U}(S_2 \wedge \neg S_1)$.
- `Response`: если выполнено S_1, то потом когда-нибудь выполнится S_2.
  – LTL: $\mathbf{G}(S_1 \to \mathbf{F}S_2)$.

Кроме того для шаблонов из $Order$ на события "S1" и "S2" могут накладываться ограничения одновременности выполнения "Space": {`NonOverlap, RightOverlap, LeftOverlap`}:

- `NonOverlap`: S_1 и S_2 никогда не случаются одновременно.
  – LTL: $\mathbf{G}\neg(S_1 \wedge S_2)$.
- `RightOverlap`: S_1 всегда случается раньше S_2 и некоторое конечное время они имеют место одновременно.
  – LTL: $\mathbf{G}(S_1 \wedge \neg S_2\mathbf{U}(S_1 \wedge S_2\mathbf{U}\neg S_1 \wedge S_2))$.
- `LeftOverlap`: S_2 всегда случается раньше S_1 и некоторое конечное время они имеют место одновременно.
  – LTL: $\mathbf{G}(S_2 \wedge \neg S_1\mathbf{U}(S_2 \wedge S_1\mathbf{U}\neg S_2 \wedge S_1)))$.

Семантика всех предыдущих шаблонов описывалась формулами логики линейного времени LTL, с использованием значения `linear` спецификатора "TimeType". Однако бывает полезно использовать шаблоны семантика которых описывается формулами логики ветвящегося времени CTL, но не формулами LTL [14]. Шаблоны *ветвящегося времени* задаются с помощью значения `branch` спецификатора "TimeType". Их семантика при отсутствии ограничений описывается следующим образом:

- `Universality+branch`: всегда, рано или поздно, есть хотя бы один вариант работы системы, в котором S_1 имеет место всегда.
  – CTL: $\mathbf{AFEG}S_1$.
- `Existence+branch`: всегда есть хотя бы один вариант работы системы, в котором S_1 когда-нибудь имеет место.
  – CTL: $\mathbf{AGEF}S_1$.

Table 3

Утверждения-события

$Variable$	Name	Domain		
	`string`	`Dom`		
$Data$	Var1	Var2	Opr	Quantifier
	$Variable$	$Variable$	$\{=, <, >\}$	$\forall, \exists$
$Case$	Kind	Name	Condition	
	`state` `event`	`string`	$SimpleSpec$	
$Complex$	Kind	Type	Cases	
	`one` `parallel` `serial` `eventual`	`strict` `free` `hold`	$Data$ $Case$	

- `Absence+branch`: всегда, рано или поздно, есть хотя бы один вариант работы системы, в котором S_1 никогда не имеет места.
  – CTL: $\mathbf{AFEG}\neg S_1$.
- `Precedence+branch`: если случилось S_2, тогда есть хотя бы один вариант работы системы, при котором S_1 случилось раньше S_2.
  – CTL: $\mathbf{AF}S_2 \rightarrow (\neg S_2 \mathbf{EU}(S_1 \wedge \neg S_2))$.
- `Response+branch`: если случилось S_1, тогда есть хотя бы один вариант работы системы, при котором случится S_2.
  – CTL: $\mathbf{AG}(S_1 \rightarrow \mathbf{EF}S_2)$.

2.3. Утверждения

Экземпляры класса $Proposition$ (таб. 2.3) описывают выделенные состояния системы. Семантикой этих экземпляров является то, что утверждения о данных (класс $Data$), события (класс $Case$), либо составные события (класс $Complex$) имеют место в некоторых состояниях системы. Все эти классы являются подклассами класса $Proposition$. Одинаковых наследуемых атрибутов данных эти классы не имеют, но наследуют все отношения родительского класса.

С помощью служебного класса Var задаются переменные системы, значения которых могут определять состояния или события. Константами являются переменные с одноэлементной областью определения. $Data$ — это утверждения о переменных системы, представленными атрибутами отношениями “Var1” и “Var2”. Они задают сравнение их значений с помощью атрибута “Opr”, возможно, с учётом квантора из значений атрибута “Quantifier”. Утверждения об именованных состояниях и событиях системы с помощью класса $Case$, где атрибуты “Kind” и “Name” определяют тип и строковое имя, а атрибут-отношение “Condition” служит для задания условия пуска, представленного экземпляром класса $SimSpec$. Составные события определяются с помощью класса $Complex$, который использует значения атрибута “Kind”, задающего вид комбинации событий, и атрибута “Type”, задающего её тип, а также атрибута-отношения “Cases”, задающего собственно множество комбинируемых событий. Семантику комбинаций событий можно описать следующим образом, основываясь на работах [13, 16]. Пусть $G = \{e_1, \dots e_n\}$ – это множество утверждений о переменных и событиях, представленные экземплярами классов $Data$ и $Case$.

- `one`(G):
  - `strict`: хотя бы одно из событий множества G имеет место.
    - LTL: $orE = e_1 \vee \ldots \vee e_n$.
  - `free`: хотя бы одно из событий множества G будет иметь место.
    - LTL: $notE \wedge notE\mathbf{U}orE$, где $notE = \neg e_1 \wedge \ldots \wedge \neg e_n$.
- `parallel`(G):
  - `strict`: все события множества G имеют место.
    - LTL: $andE = e_1 \wedge \ldots \wedge e_n$.
  - `free`: все события множества G будут иметь место.
    - LTL: $notE \wedge notE\mathbf{U}andE$.
- `serial`(G):
  - `strict`: события множества G имеют место в заданном порядке, по одному в последовательных состояниях.
    - LTL: $xE = e_1 \wedge (\mathbf{X}e_2 \wedge (\mathbf{X}e_3 \ldots \wedge \mathbf{X}e_n))$.
  - `hold`: события множества G имеют место в заданном порядке, по одному в последовательных состояниях, и все следующие события не имеют места в этот момент.
    - LTL: $xhE = e_1 \wedge notE_2^n \wedge (\mathbf{X}e_2 \wedge notE_3^n \wedge (\mathbf{X}e_3 \wedge notE_4^n \ldots \wedge \mathbf{X}e_n))$, где $notE_i^n =$

$\bigwedge_{j=i}^{n} \neg e_j$.

- `free`: события множества G будут иметь место по одному в заданном порядке.
  - LTL: $notE \wedge notE\mathbf{U}xhE$.

- `eventual`(G):
  - `strict`: события множества G имеют место в заданном порядке, в различных и возможно не последовательных состояниях.
    - LTL: $fE = e_1 \wedge \mathbf{X}(\neg e_2\mathbf{U}e_2 \wedge \mathbf{X}(\neg e_3\mathbf{U}e_3 \ldots \wedge \mathbf{X}(\neg e_n\mathbf{U}e_n)))$.
  - `hold`: события множества G будут иметь место в заданном порядке, в различных и возможно не последовательных состояниях, и все следующие события не имеют места в этот момент.
    - LTL: $fhE = e_1 \wedge notE_2^n \wedge (notE_2^n\mathbf{U}e_2 \wedge notE_3^n \wedge (notE_3^n\mathbf{U}e_3 \wedge notE_4^n \ldots \wedge \neg e_n\mathbf{U}e_n))$.
  - `free`: события множества G будут иметь место в заданном порядке, в различных и возможно не последовательных состояниях.
    - LTL: $notE \wedge notE\mathbf{U}fhE)$.

Представленное множество шаблонов составных событий мы планируем расширить шаблонами, которые учитывают выполнимость событий после их однократного выполнения.

2.4. Границы

Экземпляры классов, приведённых в таблице 2.4, задают ограничения на выполнение шаблонов относительно событий (класс $Scope$), временных рамок (класс $Time$) и количества выполнений ($Quantity$).

Класс $Scope$ определяет область действия шаблонов, используя атрибуты-отношения “S1” и “S2” для спецификации событийных границ, а тип ограничений задаётся атрибутом “Kind”$\in$ `Scopes ={globally, before, after, between, after-until, start, regular, final}`. Семантика событийных границ выполнения шаблонов относительно событий основана на работах [4, 17]. Приведём семантику событийных границ для шаблона универсальности, когда событие S выполняется всегда:

- `globally`: шаблон выполняется в течении всего времени работы системы.
  - LTL: $\mathbf{G}S$.
- `before`: шаблон выполняется в течении всего времени работы системы до первого появления события S_1.
  - LTL: $\mathbf{F}S_1 \rightarrow S\mathbf{U}S_1$.

Table 4

Границы

<table>
<tr><td rowspan="2">Scope</td><td>Kind</td><td>Space</td><td>S1</td><td>S1</td></tr>
<tr><td><code>Scopes</code></td><td><code>Overlapping</code></td><td>Props</td><td>Props</td></tr>
<tr><td rowspan="2">Bound</td><td>Kind</td><td>Bound1</td><td>Bound2</td><td>Bound3</td></tr>
<tr><td><code>Situation</code></td><td>Quantity
Time</td><td>Quantity
Time</td><td>Quantity
Time</td></tr>
<tr><td rowspan="2">Quantity</td><td>Kind</td><td>Num1</td><td>Num2</td><td rowspan="2"></td></tr>
<tr><td><code>point</code>
<code>minimum</code>
<code>maximum</code>
<code>interval</code></td><td><code>Integer</code></td><td><code>Integer</code></td></tr>
<tr><td rowspan="2">Time</td><td>Duration</td><td>Periodic</td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td>Quantity</td><td>Quantity</td></tr>
</table>

- `after`: шаблон выполняется выполняется в течении всего времени работы системы после первого появления события S_1.
  – LTL: $\neg S\mathbf{U}(S_1 \wedge \mathbf{G}S)$.
- `between`: шаблон выполняется между первым появлением события S_1 и следующим после этого появлением события S_2.
  – LTL: $\neg S\mathbf{U}(S_1 \wedge \neg S_2 \wedge (S\mathbf{U}S_2))$.
- `after-until`: шаблон выполняется выполняется между первым появлением события S_1 и следующим после этого появлением события S_2 либо в течении всего последующего времени работы системы.
  – LTL: $\neg S\mathbf{U}(S_1 \wedge \neg S_2 \wedge (S\mathbf{W}S_2))$.
- `start`: шаблон выполняется, пока событие S_1 не обозначит конец начальной фазы работы системы
  – LTL: $\neg S_1 \rightarrow S\mathbf{W}S_1$.
- `regular` шаблон всегда выполняется выполняется между появлением события S_1 и следующим после этого появлением события S_2 либо в течении всего последующего времени работы системы.
  – LTL: $\mathbf{G}(S_1 \wedge \neg S_2 \rightarrow (S\mathbf{W}S_2))$.

- `final` шаблон выполняется после того, как событие S_1 обозначит начало финальной фазы работы системы.

  – LTL: $(\mathbf{FG}S_1 \wedge \mathbf{F}S) \rightarrow \neg S\mathbf{U}S_1)$.

Для дополнительных ограничений на одновременное исполнение событийный границ “S1”, “S2” и событий шаблона используется атрибут “Space”$\in$ $\mathtt{Overlapping} = \{\mathtt{NonOverlap}(i,j), \mathtt{RightOverlap}(i,j), \mathtt{LeftOverlap}(i,j) \mid i,j \in \{S_{1^s}, S_{2^s}, S_{1^p}, S_{2^p}\}\}$, где S_{1^s} и S_{2^s} — событийные границы, а S_{1^p} и S_{2^p} — события шаблона, и семантика конструкций из `Overlapping` аналогична семантике ограничений одновременности выполнения шаблонов “Space” из предыдущего раздела.

В отличие от событийных границ, которые накладываются на исполнение шаблона в целом, временные и количественные границы могут определяться как для шаблона в целом, так и для его событий. Для спецификации этого используется класс *Bound*, где атрибут “Kind”$\in$ $\mathtt{Situation} = \{\mathtt{whole}, \mathtt{S1}, \mathtt{S2}, \mathtt{whole+S1}, \mathtt{whole+S2}, \mathtt{S1+S2}, \mathtt{whole+S1+S2}\}$ определяет сочетание накладываемых ограничений, а сами ограничения задаются атрибутами-отношениями “Bound1”, “Bound2” и “Bound3”, и могут быть как количественными (класс *Quantity*), так и временными (класс *Time*).

Число появлений шаблона или события задаётся экземпляром класса *Quantity* с помощью атрибута-спецификатора “Kind” и натуральных числовых атрибутов “Num1” и “Num2”. Приведём семантику ограничений на число появлений при Num1=2 и Num2=3 (легко обобщается на произвольные натуральные числа) для шаблона существования, когда событие S выполнится когда-нибудь:

- `point` (ровно Num1 раз):

  – S выполнится когда-нибудь ровно 2 раза.

  – LTL: $Pt_2 = Now_2 \vee (\neg S\mathbf{U}Now_2)$, где $Now_2 = S \wedge \mathbf{X}Pt_1$ и $Pt_1 = Now_1 \vee \neg S\mathbf{U}Now_1$, а $Now_1 = S \wedge \mathbf{XG}\neg S$;

- `minimum` (хотя бы Num1 раз):

  – S выполнится когда-нибудь хотя бы 2 раза.

  – LTL: $Min_2 = \mathbf{F}(S \wedge \mathbf{XF}S)$;

- `maximum` (не более Num1 раз):

  – S выполнится когда-нибудь не более 2 раз.

  – LTL: $Max_2 = G\neg S \vee Pt_1 \vee Pt_2$;

- `interval` (от Num1 до Num2 раз):

– S выполнится когда-нибудь не более 3 и не менее 2 раз.

– LTL: $Int_{2,3} = Min_2 \wedge Max_3$.

Класс $Time$ позволяет задавать временные границы (атрибут-отношение “Duration” с классом $Quantity$) и временную периодичность (атрибут-отношение “Periodic” с классом $Period$) появления шаблона или события. Приведём семантику временных ограничений, выраженную формулами логики MTL [15], для шаблона существования, когда событие S выполнится когда-нибудь:

- Duration:
  - `point` (ровно через Num1):
    - S выполнится через Num_1 единиц времени.
    - MTL: $\neg S \mathbf{U}_{Num_1} S$;
  - `minimum` (минимум через Num1):
    - S выполнится не раньше Num_1 единиц времени.
    - MTL: $Min^d_{Num_1}(S) = \neg S \mathbf{U}_{Num_1}(\neg S \rightarrow \mathbf{F}S)$,
  - `maximum` (максимум через Num1):
    - S выполнится не позже Num_1 единиц времени.
    - MTL: $Max^d_{Num_1}(S) = \mathbf{F}_{Num_1} S$.
  - `interval` (между Num1 и Num2):
    - S выполнится не раньше Num_1 и позже Num_2 единиц времени.
    - MTL: $Min^d_{Num_1}(S) \wedge Max^d_{Num_2}(S)$.
- Periodic:
  - `point` (ровно каждые Num1):
    - S выполняется каждые Num_1 единиц времени.
    - MTL: $\mathbf{G}(\neg S \mathbf{U}_{Num_1}(S \wedge \mathbf{X} \neg S))$.
  - `minimum` (минимум каждые Num1):
    - S выполняется не реже каждых Num_1 единиц времени.
    - MTL: $Min^p_{Num_1}(S) = \mathbf{GF}_{Num_1} S$,
  - `maximum` (максимум каждые Num1):
    - S выполняется не чаще каждых Num_1 единиц времени.
    - MTL: $Max^p_{Num_1}(S) = \mathbf{G}(\neg S \mathbf{U}_{Num_1}(\neg S \rightarrow \mathbf{F}(S \wedge \mathbf{X} \neg S)))$.
  - `interval` (между Num1 и Num2):
    - S выполняется не реже Num_2 и не чаще Num_1 единиц времени.

Table 5

Среда

Environment	Kind	Env	Sys
	`{BadBeh,Opt}`	*Spec*	*Spec*

– MTL: $G(Min^{p}_{Num_2}(S) \wedge Max^{p}_{Num_2}(S))$.

Отметим, что могут быть одновременно заданы все ограничения, как событийные, так и количественные с временными.

2.5. Окружение

Класс *Environment* позволяет моделировать отношения поведений системы и её окружения. Атрибут “Kind” задаёт вид этих отношений, а атрибуты “Env” и “Sys” — поведение окружения и системы, соответственно, со следующей семантикой:

- `BadBeh` (плохое поведение): если окружение всегда может следовать плохой спецификации Env, то система всё равно следует хорошей спецификации Sys.

  – CTL: $\mathbf{EF}Env \wedge \mathbf{AG}Sys$.

- `Opt` (оптимальность): как бы ни вело себя окружение, система может реагировать так, что оптимальное спецификация Sys для системы и спецификация Env для окружения будут достигнуты

  – Свойство невыразимо в логиках CTL или LTL, но выразимо в $\mu-$исчислении. Доказательство этого факты выходит за рамки данной работы.

Неформально говоря, первый тип отношений окружения и системы полезен, чтобы убедиться, что определённый (нежелательный) сценарий поведения окружения действительно смоделирован, но ошибок системы при этом не возникает. Второй тип, выражающий свойство оптимального поведения системы заключается в том, что при любом поведении окружения система может достигнуть нужного состояния.

Заметим, что не всякая комбинация значений атрибутов классов построенной онтологии имеет смысл. Например, шаблон *ограниченной универсальности BU* события P мог бы быть специфицирован следующим образом: $BU = Occurrence(Kind : universality, S1 : P, TimeType : linear, FrameQuantity : (Kind : whole, Bound3 : (Kind : point, Num1 = 5)))$, но это понятие не имеет осмысленной семантики.

Онтология требований может быть представлена в графическом виде. На рис.1 классы

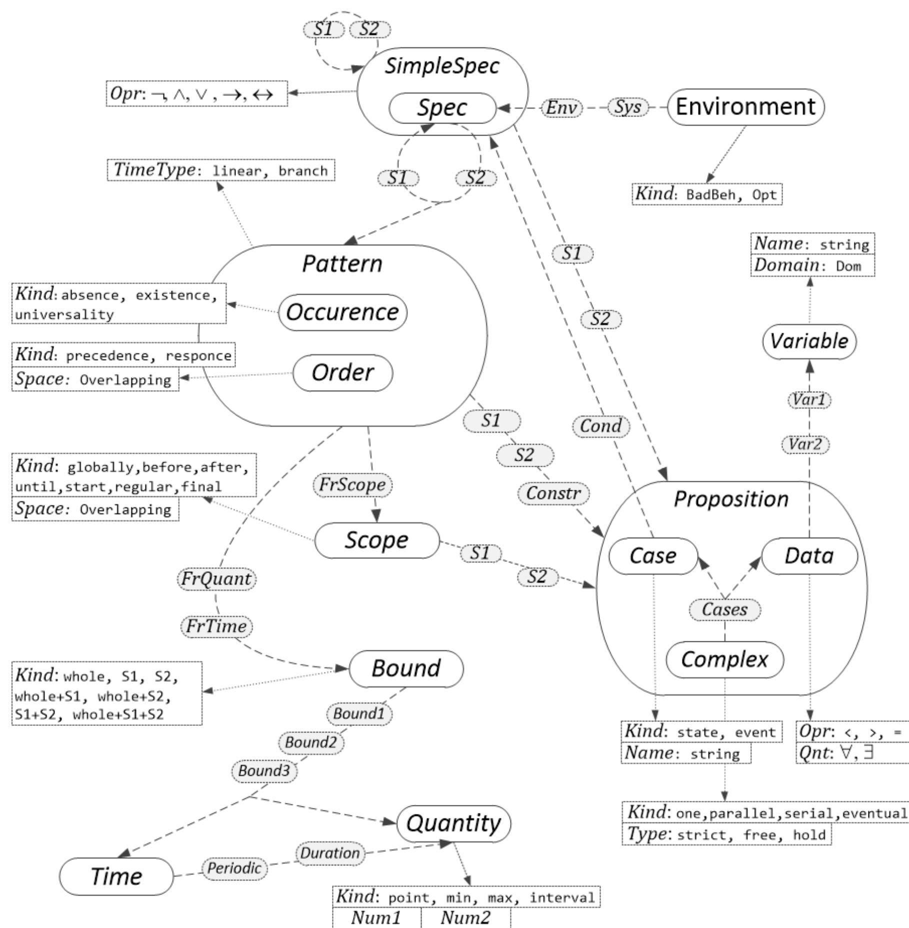


Рис. 1. Онтология требований

представлены в виде белых овалов и их подклассы находятся внутри них. Отношения между классами помечены штриховыми стрелками, а названия отношений помещены внутри серых овалов. Атрибуты классов располагаются в прямоугольниках и связаны с соответствующими классами штрихпунктирными стрелками.

3. Примеры требований

Приведём примеры требований к автоматической системе управления вакуумированием Большого солнечного вакуумного телескопа (БСВТ)[21].

Система вакуумирования БСВТ (рис. 2) содержит следующие компоненты: 1) труба телескопа, 2) пневмоустройство, 3) датчик давления в трубе телескопа, 4) клапан подключения вакуумного насоса к трубе телескопа, 5) датчик давления в патрубке вакуумного насоса, 6) отсечной клапан соединения вакуумного насоса с атмосферой (сапун), 7) датчик температуры воды в рубашке охлаждения вакуумного насоса, 8) вентилятор, 9) вакуумный насос, 10) датчик температуры воды в системе климат-контроля, 11) насос охлажде-

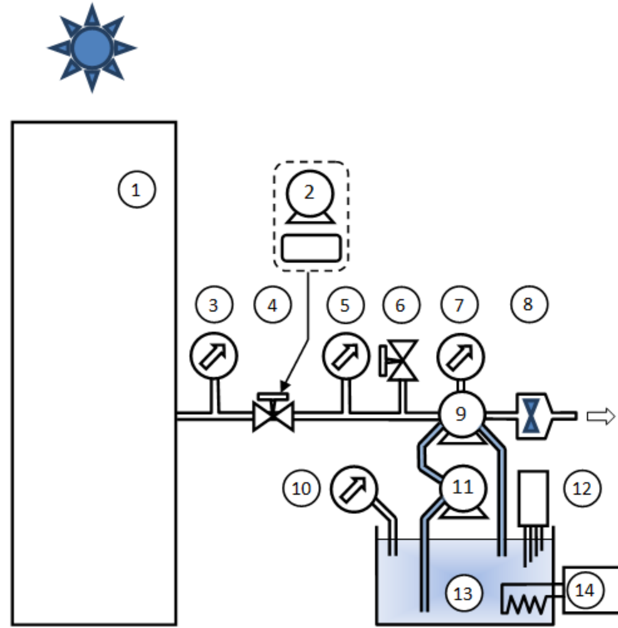


Рис. 2. Подсистема вакуумирования БСВТ

ния, 12) датчики уровня воды в системе климат-контроля, 13) система климат-контроля, 14) нагреватель воды в системе климат-контроля.

Описание требований в нашей работе включает в себя формулирование их на естественном языке (RUS), с помощью темпоральных логик (LTL, MTL) и в виде экземпляра онтологии шаблонов требований (ONT). Если область действия шаблона явно не задана, то мы считаем, что шаблон выполняется всегда.

Пример 1. Требование отсутствия составного события.

RUS: *S1 = Никогда одновременно не включается насос P9 и не открывается затворный клапан V4.*

Пусть состояния системы, в которых включён насос, обозначаются с помощью события $Pump$, выполняющегося в них, а открывание затворного клапана описывается исполнением одного из событий: вызова открытия $V4.in$ и собственно открытия $V4.out$.

LTL: $S1_F = \mathbf{G}\neg(Pump \wedge V4.in \wedge V4.out)$.

ONT: $S1_O = Occurrence(\text{Kind: } \texttt{Absence}, \text{S1: } E_1, \text{TimeType: } \texttt{linear})$, где

- составное событие $E_1 = Complex$

  $(\text{Kind: } \texttt{strict}, \text{Type: } \texttt{parallel}, \text{Cases: } \{Pump, V4.in, V4.out\})$, при этом

  – каждое из простых событий $e \in \{Pump, V4.in, V4.out\}$ – это экземпляр $e = Case(\text{Kind: } \texttt{event}, \text{Name: } e)$.

В этих записях экземпляров и далее опущены имена атрибутов, значения которых не

заданы. Далее будем обозначать экземпляры простых событий именами этих событий.

Пример 2. Требование появления события между двумя другими событиями.

RUS: *Между сигналами датчика уровня воды S12 "Воды недостаточно" и "Воды достаточно" всегда посылается сигнал отключения нагревателя H14.*

Пусть состояния системы, в которых датчик S12 обнаруживает, что воды недостаточно, обозначаются с помощью события $S12.low$, если же воды достаточно, то $S12.norm$, а сигнал отключения нагревателя H14 описывается исполнением события $H14.tooff$.

LTL: $S2_F = \mathbf{G}(\neg H14.tooff \rightarrow$

$$(\neg H14.tooff\,\mathbf{U}(S12.low \wedge \neg S12.norm \rightarrow (\neg S12.norm\mathbf{U}H14.tooff \wedge \neg S12.norm)))).$$

ONT: $S2_O = Occurrence$

(Kind: `Existence`, S1: $H14.tooff$, TimeType: `linear`, FrameScope: Sc_2), где

- область действия $Sc_2 = Scope$(Kind: `after-until`, S1: $S12.low$, S2: $S12.norm$).

Пример 3. Требование реакции на событие в течение заданного времени.

RUS: *Если пришёл сигнал на клапан V5, то он откроется до истечения таймаута TV5.*

Пусть состояния системы, в которых приходит сигнал на клапан V5, обозначаются с помощью события $V5.toOpen$, а в которых этот клапан открыт – $V5.Open$.

MTL: $S3_F = \mathbf{G}(V5.toOpen \rightarrow \mathbf{F}_{\leq TV5}V5.Open)$.

ONT: $S3_O = Order$

(Kind: `Response`, S1: $V5.toOpen$, S2: $V5.Open$, TimeType: `linear`, FrameTime: B_3), где

- ограничение по времени, в течении которого должно выполниться второе событие шаблона $B_3 = Bound$(Kind: `S2`, Bound2: T_3), где
  - время ограничения $T_3 = Time$(Duration: Q_3) с
    - количественными характеристиками времени

$Q_3 = Quantity$(Kind: `maximum`, Num1: $TV5$).

Пример 4. Требование однократной реакции на составное событие в течение заданного времени.

RUS: *Всегда при нормальной работе вакуумного насоса P9, если падение давления в трубе сидеростата (S4) меньше уровня D04 за время ToutD4, и при этом давление в трубе сидеростата (S4) все еще больше критического уровня давления P04, не более, чем за время ToutM4 генерируется сообщение об ошибке. Это сообщение генерируется только один раз.*

Обозначим состояния системы, в которых насос работает нормально, как $PumpNorm$,

изменение давления и значение давления в трубе сидеростата S4 задаётся условно-целочисленными переменными $D4$ и $P4$, локальный счётчик времени — целочисленной переменной $T4$, а событие генерирования сообщения об ошибке обозначается как $AlarmPS4$.

MTL: $S4_F = \mathbf{G}(PumpNorm \wedge T4 > ToutD4 \wedge D4 < D04 \wedge P > P04 \rightarrow \mathbf{F}_{\leq ToutM4} AlarmPS4 \wedge \neg AlarmPS4 \mathbf{U}(PumpNorm \wedge T4 > ToutD4 \wedge D4 < D04 \wedge P > P04))$.

ONT: $S4_O = Order(\text{Kind: } \texttt{Response}, \text{S1: } E_4, \text{S2: } AlarmPS4,$
$\text{TimeType: } \texttt{linear}, \text{FrameTime: } Bt_4, \text{FrameQuantity: } Bq_4)$, где

- составное событие $E_4 = Complex$
  $(\text{Kind: } \texttt{strict}, \text{Type: } \texttt{parallel}, \text{Cases: } \{PumpNorm, D_1, D_2, D_3\})$, где
  - утверждения о данных:
    $D_1 = Data(\text{Var1: } T4, \text{Var2: } ToutD4, \text{Opr: } \texttt{>})$,
    $D_2 = Data(\text{Var1: } D4, \text{Var2: } D04, \text{Opr: } \texttt{<})$, $D_3 = Data(\text{Var1: } P, \text{Var2: } P04, \text{Opr: } \texttt{>})$,
    где переменные и константы:
    $T4 = Variable(\text{Name: } T4, \text{Domain: } int)$,
    $ToutD4 = Variable(\text{Name: } ToutD4, \text{Domain: } \{ToutD4\})$,
    $D4 = Variable(\text{Name: } D4, \text{Domain: } int)$,
    $D04 = Variable(\text{Name: } D04, \text{Domain: } \{D04\})$,
    $P = Variable(\text{Name: } P, \text{Domain: } int)$,
    $P04 = Variable(\text{Name: } P04, \text{Domain: } \{P04\})$;
- ограничение по времени, в течении которого должно выполниться второе событие шаблона, $Bt_4 = Bound(\text{Kind: } \texttt{S2}, \text{Bound2: } T_4)$, где
  - время ограничения $T_4 = Time(\text{Duration: } Qt_4)$ с
    - количественными характеристиками времени
      $Qt_4 = Quantity(\text{Kind: } \texttt{maximum}, \text{Num1: } ToutM4)$,
- количественные характеристики исполнения второго события шаблона
  $Bq_4 = Bound(\text{Kind: } \texttt{S2}, \text{Bound2: } Qq_4)$ с $Qq_4 = Quantity(\text{Kind: } \texttt{point}, \text{Num1: } 1)$.

4. Концепция интеллектуальной системы поддержки разработки распределенных программных систем.

Интеллектуальная система поддержки формальной верификации распределенных программных систем включает следующие компоненты.

1. Системы семантического извлечения информации (СИИ).

2. Онтология программной системы (ОПС).
3. Онтология требований (ОТ).
4. Редактор онтологий ОПС и ОТ.
5. Транслятор содержания ОПС в язык подходящего инструмента верификации.
6. Модуль извлечения требований из ОПС.
7. Транслятор содержания ОТ в высказывания на естественном языке (ТЯ).
8. Транслятор содержания ОТ в графическое представление (ТГ).
9. Транслятор содержания ОТ в формулы логики спецификаций (ТЛ).
10. Онтология логик спецификаций (ОЛС).
11. Инструмент верификации шаблонов требований (ВШ).

Наша система семантического извлечения информации (1) позволяет извлекать данные из текстов на естественном языке в виде экземпляров онтологии предметной области [5–7]. Онтология программной системы (2) содержит описание программной системы как параллельно взаимодействующих последовательных процессов, временных и причинно-следственных отношений между ними. Онтология требований (3) содержит описание требований, относящихся к корректности программной системы. Содержание онтологии ОПС извлекается из технической документации с использованием СИИ, а содержание онтологии ОТ — с помощью модуля извлечения требований из ОПС и из технической документации с использованием СИИ. Транслятор содержания ОПС (5) в язык инструмента верификации служит для задания программной системы на входном языке выбранного инструмента верификации. Модуль извлечения требований (6) из ОПС, основываясь на причинно-следственных и временных отношениях процессов, описанных в ОПС, и онтологии ОТ, извлекает типичные требования корректности. Трансляторы требований ТЯ и ТГ (7,8), представляющие экземпляры ОТ как высказывания на ограниченном подмножестве естественного языка и в графическом виде, служат для облегчения задачи понимания и спецификации требований. Планируется также разработка редакторов результата трансляции и обратных трансляторов в ОТ. Транслятор требований ТЛ (9) в формулы логики спецификаций определяет формальную семантику требований. Формальная семантика позволяет выбрать способ и инструмент верификации требований к программной системе, информация о которых содержится в онтологии логик спецификаций ОЛС (10). В отличие от других онтологий нашей системы, онтология ОЛС пополняется вручную и содержит сведения о темпоральных и модальных логиках спецификаций,

отношениях между ними и известных инструментах верификации. Инструмент верификации шаблонов требований ВШ (11) использует алгоритмы верификации шаблонов, обладающие меньшей трудоёмкостью, чем стандартные алгоритмы верификации общего вида. В силу неоднозначности естественного языка и трудоёмкости точного задания требований корректности программных систем все инструменты системы поддержки формальной верификации, кроме верификатора шаблонов, не являются полностью автоматическими, т.е. результат их работы может требовать дополнительного ручного анализа.

5. Заключение

В работе предложен начальный вариант онтологии требований, которая допускает описание требований следующего типа: качественные; реально-временные; количественные; учитывающие составные события, а также утверждения о данных. Эта онтология может быть расширена несколькими способами. Независимым от предметной области способом является описание не только шаблонов комбинаций событий, но и обобщение этих комбинаций до поведения в духе CSP, предложенное в [19]. Кроме того, специализированные предметные области, такие как безопасность, агентные модели, могут потребовать свои шаблоны спецификаций.

Очевидным следующим шагом является задание полной формальной семантики шаблонов. Особого внимания требует формальная семантика шаблонов, содержащих в том или ином виде оператор Until и его варианты, т.к. можно получить некорректную спецификацию за счёт несоответствия правых границ времён появления событий. Эта семантика будет использоваться в трансляторе содержания ОТ в формулы логики спецификаций. Трансляторы в высказывания на естественном языке и в графическое представление зависят от формальной семантики и должны разрабатываться после её полного определения. Отметим, что за счёт неоднозначности естественного языка возможно неоднозначное соответствие шаблонов и результатов их трансляции в естественный язык. Представление шаблонов требований с помощью графических формальных языков, например GIL, может помочь выявить и исправить некорректные требования. Разработка модуля извлечения требований из онтологии программной системы и инструмента верификации шаблонов требований также зависит от формальной семантики шаблонов. Независимой задачей является построение онтологии логик спецификаций.

Список литературы

1. **Autili M., Grunske L., Lumpe M., Pelliccione P., Tang A.** *Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar* // IEEE Transactions on Software Engineering, Volume: 41, Issue: 7, July 1 2015, P. 620–638.
2. **Bianculli D., Ghezzi C., Pautasso C., Senti P.** *Specification Patterns from Research to Industry: A Case Study in Service-Based Applications* // Proc. of 34th International Conference on Software Engineering (ICSE), 2012, P. 968–976.
3. **Clarke E.M., Grumberg O., Peled D.** *Model Checking.* // MIT Press, 1999. 324 p.
4. **Dwyer M. B., Avrunin G. S., Corbett J. C.** *Patterns in property speciÏcations for finite-state verification* // Proc. of the 21st Int. Conf. on Software Engineering, IEEE Computer Society Press, 1999, P. 411–420.
5. **Garanina N., Sidorova E., Bodin E.** *A Multi-agent Text Analysis Based on Ontology of Subject Domain* // In: Perspectives of System Informatics. LNCS Vol .8974, 2015, P. 102-110.
6. **Garanina N., Sidorova E.** *Context-dependent Lexical and Syntactic Disambiguation in Ontology Population* // Proc. of the 25th International Workshop on CS&P. Rostock, Germany, Sep. 28-30, 2016. – Humboldt-Universitat zu Berlin, 2016, P. 101–112.
7. **Garanina N., Sidorova E., and Kononenko I.** *A Distributed Approach to Coreference Resolution in Multiagent Text Analysis for Ontology Population* // Springer International Publishing AG 2018 A. K. Petrenko and A. Voronkov (Eds.): PSI 2017, LNCS 10742, P. 1–16, 2018.
8. **Grunske L.** *Specification patterns for probabilistic quality properties* // Proc. of the 30th international conference on Software engineering. - ICSE '08. - New York, NY, USA: ACM, 2008. - P. 31-40.
9. **Halle S., Villemaire R., Cherkaoui O.** *Specifying and validating data-aware temporal web service properties* // IEEE Trans. Softw. Eng., 2009, vol. 35, no. 5, P. 669-–683.
10. **Hoare, C. A. R.** *Communicating sequential processes* // Communications of the ACM. 21 (8): P. 666–677
11. **Konrad S., Cheng B. H. C.** *Real-time specification patterns* // Proc. of the 27th International Conference on Software Engineering. ACM, 2005, P. 372—381.
12. **Manna Z., Pnueli A.** *The Temporal Logic of Reactive and Concurrent Systems* // Springer-Verlag, 1991. 427 p.
13. **Mondragon O., Gates A. Q., Roach S.** *Prospec: Support for Elicitation and Formal Specification of Software Properties* // Proc. of Runtime Verification Workshop, ENTCS, Vol. 89, Elsevier, 2004. P. 67–88.
14. **Post A., Menzel I., Podelski A.** *Applying restricted english grammar on automotive requirements: does it work? A case study* // Proc. of 17th international working conference on Requirements engineering: foundation for software quality. Vol. 6606 of LNCS. Berlin, Heidelberg: Springer-Verlag, 2011. P. 166–180.
15. **Koymans R.** *Specifying Real-Time Properties with Metric Temporal Logic* // Real-Time Systems, November 1990, Volume 2, Issue 4, P. 255–299
16. **Salamah S., Gates A. Q., Kreinovich V.** *Validated templates for specification of complex LTL*

formulas // J. of Syst. and Soft., 2012, V. 85, n. 8. P. 1915–1929.

17. **Shoshmina I. V** *Developing formal temporal requirements to distributed program systems* // Proc. of Seven Workshop on Program Semantics, Specification and Verification: Theory and Applications (PSSV 2016) June 14-15, 2014 in St. Petersburg, Russia. - System Informatics. - 2016. - N.8. - P. 21-31.
18. **Smith M.H., Holzmann G.J., Etessami K.** *Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs* // Proc. of Proceedings Fifth IEEE International Symposium on Requirements Engineering 27-31 Aug. 2001, P. 14-22
19. **Wong P.Y.H., Gibbons J.** *Property Specifications for Workflow Modelling* // Proc. of Integrated Formal Methods. IFM 2009. Lecture Notes in Computer Science, vol 5423. Springer, Berlin, Heidelberg
20. **Yu J., Manh T. P., Han J. et al.** *Pattern based property specification and verification for service composition* // Proc. of 7th International Conference on Web Information Systems Engineering (WISE). Vol. 4255 of LNCS. Springer-Verlag, 2006. P. 156–168.
21. **Лях Т.В., Зюбин В.Е., Сизов. М.М.** *Опыт применения языка Reflex при автоматизации Большого солнечного вакуумного телескопа* // Журнал «Промышленные АСУ и контроллеры». 2016. №7. С. 37-43.

УДК 004.8

Conceptual transition systems and their application to development of conceptual models of programming languages*

Anureev I.S. (Institute of Informatics Systems),
Promsky A.V. (Institute of Informatics Systems)

In the paper the notion of the conceptual model of a programming language is proposed. This formalism represents types of the programming language, values, exceptions, states and executable constructs of the abstract machine of the language, and the constraints for these entities at the conceptual level. The new definition of conceptual transition systems oriented to specification of conceptual models of programming languages is presented, the language of redefined conceptual transition systems CTSL is described, and the technique of the use of CTSL as a domain-specific language of specification of conceptual models of programming languages is proposed. The conceptual models for the family of sample programming languages illustrate this technique.

Keywords: *operational semantics, conceptual transition system, programming language, conceptual model, domain-specific language*

1. Introduction

This paper relates to the development of operational semantics of programming languages. Following [1], we distinguish two parts of the operational semantics of a programming language. The structural part defines how the elements of the language relate to runtime elements that an abstract machine of the programming language can use at runtime. The structural part is called instantiation semantics or structure-only semantics [2]. The dynamic part describes the actual state changes that take place at runtime.

In traditional operational semantics approaches [3–6], the main focus is on state changes, while the structural part is defined ad-hoc. The modern programming languages becomes more complex. Therefore, development of formalisms, languages and frameworks to describe the instantiation semantics is very important problem.

* Partially supported by RFBR under grants 15-01-05974 and 17-01-00789 and SB RAS interdisciplinary integration project No.15/10.

The meta-model-based object-oriented approach [1] to description of the instantiation semantics uses MOF (EMF) [7]. The algebraic approach [8] is based on abstract state machines. Abstract state machines are the special kind of transition systems in which states are algebraic systems. The structural part of the operational semantics is flexibly modelled by the appropriate choice of the symbols of the signature of an algebraic system. Rewrite-based approach is implemented in the frameworks K [9] and Maude [10].

These approaches do not take into account the natural conceptual nature of instantiation semantics which is easier to describe in the ontological terms of concepts, their instances and attributes.

In this paper, we introduce the notion of the conceptual model of a programming language. This formalism describes the instantiation semantics at the conceptual level. The conceptual model is specified in terms of conceptual transition systems (CTSs) [11] in the language of conceptual transition systems CTSL [12]. Thus, CTSL acts as a domain-specific language oriented to specification of conceptual models of programming languages.

The paper has the following structure. The preliminary concepts and notation are given in section 2. The new definition of CTSs is presented in section 3. The basic definitions of the theory of CTSs are given in sections 4 and 5. The language CTSL for redefined CTSs is described in section 6. The definition of the conceptual model of a programming language is introduced, and the technique of development of conceptual models of programming languages is illustrated by the sample programming language examples in section 7.

2. Preliminaries

The preliminary concepts and notation are given in this section.

2.1. Sets and sequences

Let $\$w$, $\$w1$, $\$w2$, … denote elements of the sort w, where w is a word, and $\$\$w$ denote the set of all elements of the sort w. For example, if n is a sort of natural numbers, then $\$n$, $\$n1$, … are natural numbers, and $$n is the set of all natural numbers.

Let $\$\$o$ and $\$\$set$ be sets of objects and sets considered in this paper. Let $\$\$i$, $\$\$n$, and $\$\$bo$ be sets of integers, natural numbers (with zero), and boolean values $true$ and $false$.

Let $\$\$se$ denote the set of finite sequences of the form $\$o1$ ... $\$o\$n$. Let $\$\$w^*$ denote the set of finite sequences of the form $\$w1$ … $\$w\$n$, and $\$w^*$, $\$w^*1$, $\$w^*2$, and so on denote the elements of the set $\$\$w^*$. Let $[es]$ denote the empty sequence. Let $\$\$w^+$ denote the set of finite nonempty sequences of the form $\$w1$ … $\$w\$n$, and $\$w^+$, $\$w^+1$, $\$w^+2$, and so on denote the elements of the set $\$\$w^+$.

Let $[repeat\ \$o\ \$n]$ denote the sequence consisting of $\$n$-th occurrences of the object $\$o$.

Let $[\$o \in \$se]$ and $[\$se1 \sqsubseteq \$se2]$ denote $\$o \in \{\$se\}$ and $\{\$se1\} \sqsubseteq \{\$se2\}$. Let $[len\ \$se]$ denote the length of \$se. Let und denote the undefined value. Let $[\$se\ ..\ \$n]$ denote the $\$n$-th element of $\$se$. If $[len\ \$se] < \$n$, then $[\$se\ ..\ \$n] = und$. Let $[\$se\ ..\ \$n := \$o]$ denote the result $\$se1$ of replacement of $\$n$-th element in $\$se$ by $\$o$. If $\$n > [len\ \$se]$, then $\$se1 = \$se\ [repeat\ und\ [[len\ \$se] - \$n - 1]]\ \$o$.

Let $[\$o \in \$se]$ and $[\$se1 \sqsubseteq \$se2]$ denote $\$o \in \{\$se\}$ and $\{\$se1\} \sqsubseteq \{\$se2\}$. Let $[len\ \$se]$ denote the length of \$se. Let und denote the undefined value. Let $[\$se\ ..\ \$n]$ denote the $\$n$-th element of $\$se$. If $[len\ \$se] < \$n$, then $[\$se\ ..\ \$n] = und$. Let $[\$se\ ..\ \$n := \$o]$ denote the result $\$se1$ of replacement of $\$n$-th element in $\$se$ by $\$o$. If $\$n = [len\ \$se] + 1$, then $\$se1 = \$se\ \$o$. If $\$n > [len\ \$se] + 1$, then $\$se1 = und$.

Let $[\$o1 \prec_{[\![\$se]\!]} \$o2]$ denote the fact that there exist $\$o^*1$, $\$o^*2$ and $\$o^*3$ such that $\$se = \$o^*1\ \$o1\ \$o^*2\ \$o2\ \$o^*3$.

Let $[\$o\ \$o1\ \hookleftarrow\ \$o2]$ denote the result of replacement of all occurrences of $\$o1$ in $\$o$ by $\$o2$. Let $[\$se\ \$o\ \hookleftarrow\!*\ \$o1]$ denote the result of replacement of each element $\$o2$ in $\$se$ by $[\$o1\ \$o\ \hookleftarrow\ \$o2]$. For example, $[a\ b\ x \hookleftarrow\!*\ (f\ x)]$ denotes $(f\ a)\ (f\ b)$.

Let $\$o1$, $\$o2 \in \$\$se \cup \$\$set$. Then $[\$o1 =_{set} \$o2]$ denote that the sets of elements of $\$o1$ and $\$o2$ coincide, and $[\$o1 =_{mul} \$o2]$ denote that the multisets of elements of $\$o1$ and $\$o2$ coincide.

The above defined operations on the set $\$\$se$ are also applied to the set $\{(\$se) \mid \$se \in \$\$se\}$. The results of $[(\$se)\ ..\ \$n]$, $[\$o \in (\$se)]$, $[(\$se1) \sqsubseteq (\$se2)]$, $[\$o1 \prec_{[\![(\$se)]\!]} \$o2]$, $[(\$se)\ \$o\ \hookleftarrow^*\ \$o1]$, $[len\ (\$se)]$, $[(\$se)\ ..\ \$n \coloneqq \$o]$ and $[and\ (\$se)]$ are $[\$se\ ..\ \$n]$, $[\$o \in \$se]$, $[\$se1 \sqsubseteq \$se2]$, $[\$o1 \prec_{[\![\$se]\!]} \$o2]$, $[\$se\ \$o\ \hookleftarrow^*\ \$o1]$, $[len\ \$se]$, $[\$se\ ..\ \$n \coloneqq \$o]$ and $[and\ \$se]$.

Let $[(o^*) + (\$o^*1)]$, $[\$o\ .+(o^*)]$ and $[(o^*)\ +.\ \$o]$ denote $(\$o^*\ \$o^*1)$, $(\$o\ \$o^*)$ and $(\$o^*\ \$o)$.

2.2. Contexts

The terms used in the paper can be context-dependent. A context has the form $[\![\$o^*]\!]$. The elements of $\$o^*$ are called embedded contexts. The context in which some embedded contexts are omitted is called a partial context. All omitted embedded contexts are considered bound by the existential quantifier, unless otherwise specified.

Let $\$o[\![\$o^*]\!]$ denote the object \$o in the context $[\![\$o^*]\!]$. The expression 'in $[\![\$o1, \$o^*]\!]$' can be rewritten as 'in $[\![\$o1]\!]$ in $[\![\$o^*]\!]$', if this does not lead to ambiguity.

2.3. Functions

Let $\$\$f$ be a set of functions. Let $\$\$a$ and $\$\$v$ be sets of objects called arguments and values. Let $[\$f\ a^*]$ denote the result of application of $\$f$ to $\$a^*$. Let $[support\ \$f]$ denote the support in $[\![\$f]\!]$, i. e. $[support\ \$f] = \{\$a \mid [\$f\ \$a] \neq und\}$. Let $[image\ \$f\ \$set]$ denote the image in $[\![\$f,\ \$set]\!]$, i. e. $[image\ \$f\ \$set] = \{[\$f\ \$a] : \$a \in \$set\}$. Let $[image\ \$f]$ denote the image in $[\![\$f,\ [support\ \$f]]\!]$. Let $[narrow\ \$f\ \$set]$ denote the function $\$f1$ such that $[support\ \$f1] = [support\ \$f] \cap \$set$, and $[\$f1\ \$a] = [\$f\ \$a]$ for each $\$a \in [support\ \$f1]$. The function $\$f1$ is called a narrowing of $\$f$ to $\$set$. Let $[support\ \$f1] \cap [support\ \$f2] = \emptyset$. Let $\$f1 \cup \$f2$ denote the union $\$f$ of $\$f1$ and $\$f2$ such that $[\$f\ \$a] = [\$f1\ \$a]$ for each $\$a \in [support\ \$f1]$, and $[\$f\ \$a] = [\$f2\ \$a]$ for each $\$a \in [support\ \$f2]$. Let $\$f1 \subseteq \$f2$ denote the fact that $[support\ \$f1] \subseteq [support\ \$f2]$, and $[\$f1\ \$a] = [\$f2\ \$a]$ for each $\$a \in [support\ \$f1]$.

An object $\$u$ of the form $\$a := \$v$ is called an update. The objects $\$a$ and $\$v$ are called an argument and values in $[\![\$u]\!]$. Let $\$\$u$ be a set of updates.

Let $[\$f\ \$u]$ denote the function $\$f1$ such that $[\$f1\ \$a] = [\$f\ \$a]$ if $\$a \neq \$a[\![\$u]\!]$, and $[\$f1\ \$a[\![\$u]\!]] = \$v[\![\$u]\!]$. Let $[\$f\ \$u\ \$u^*]$ be a shortcut for $[[\$f\ \$u]\ \$u^*]$. Let $[\$f\ \$a.\$a1.\ \ldots\ .\$a\$n \coloneqq \$v]$ be a shortcut for $[\$f\ \$a \coloneqq [[\$f\ \$a]\ \$a1.\ \ldots\ .\$a\$n \coloneqq \$v]]$. Let $[\$u^*]$ be a shortcut for $[\$f\ \$u^*]$, where $[support\ \$f] = \emptyset$.

Let $[if\ \$con\ then\ \$o1\ else\ \$o2]$ denote the object $\$o$ such that $\$o = \$o1$ for $\$con = true$, and $\$o = \$o2$ for $\$con = false$.

3. Conceptual transition systems

The notion of conceptual transition systems (CTSs) is based on the notion of conceptual structures.

Let $\$\$ato$ be a set of objects called atoms.

The set $\$\$cs$ of conceptual structures in $[\![\$\$ato]\!]$ is defined as follows:

- $\$ato \in \$\$cs$;
- $(\$cs^*) \in \$\$cs$;
- if the elements of $\$cs^+$ are pairwise distinct, and $\$cs \neq und$, then $\$cs\colon (\$cs^+) \in \$\$s$;
- if the elements of $\$cs^+$ are pairwise distinct, and $\$cs \neq und$, then $(\$cs^+) :: \$cs \in \$\$cs$.

A structure $\$cs$ is atomic if $\$cs \in \$\$ato$.

A structure $\$ccs$ is a compound structure if $\$ccs$ has the form $(\$cs^*)$. The operation $(\ldots)$ is called a sequential composition. A structure $\$cs$ is an element in $[\![\$ccs]\!]$ if $\$cs^* = \$cs^*1\ \$cs\ \$cs^*2$ for some $\$cs^*1$ and $\$cs^*2$. The structure $(\ )$ is called an empty structure. Let $\$\$ccs$ be a set of compound structures.

Let $\$\$t$ and $\$\$v$ be sets of objects called types and values. An object $\$mt$ is a multi-type if $\$mt = (t^+)$. Let $\$\$mt$ be a set of multi-types. An object $\$mv$ is a multi-value if $\$mv = (v^+)$. Let $\$\$mv$ be a set of multi-values.

A structure $\$cs$ is an absolutely typed structure if $\$cs = \$v{::}\$mt$. The operation $\ldots{::}(\ldots)$ is called an absolute typification operation. Let $\$\$atcs$ be a set of absolutely typed structures.

A structure $\$t$ is an absolute type in $[\![\$atcs]\!]$ if $\$atcs = \$v{::}\$mt$, and $\$t \in \$mt$ for some $\$v$ and $\$mt$. A structure $\$atcs$ has an absolute type $\$t$ if $\$t$ is an absolute type in $[\![\$atcs]\!]$. A structure $\$v$ is a value in $[\![atcs]\!]$ if $\$atcs = \$v{::}\$mt$ for some $\$mt$.

A structure $\$mt$ is an absolute multi-type in $[\![\$atcs]\!]$ if $\$atcs = \$v{::}\$mt1$, and $\$mt \subseteq \$mt1$ for some $\$v$ and $\$mt1$. A structure $\$atcs$ has an absolute multi-type $\$mt$ if $\$mt$ is an absolute multi-type in $[\![\$atcs]\!]$.

The absolute typification operation categorizes structures, using absolute types as category names and absolute multi-types as category unions. It also models instance constructors for these categories. For example, the structure $"division\ by\ zero"{::}(exception)$ specifies the value (instance) of the type (category) $exception$ as the result of application of the instance constructor $::(exception)$ to the argument $"division\ by\ zero"$.

A structure $\$cs$ is a relatively typed structure if $\$cs = \$v{:}\$mt$. The operation $\ldots{:}(\ldots)$ is called a relative typification operation. Let $\$\$rtcs$ be a set of relatively typed structures.

A structure $\$t$ is a relative type in $[\![\$v,\ (\$cs^*)]\!]$ if $\$cs^* = \$cs^*1\ \$v{:}\$mt\ \$cs^*2$, and $\$t \in \$mt$ for some $\$cs^*1$, $\$cs^*2$ and $\$mt$. A structure $\$t$ is a relative type in $[\![\$ccs]\!]$ if $\$t$ is a relative type in $[\![\$v,\ \$ccs]\!]$ for some $\$v$. A structure $\$v$ has a relative type $\$t$ in $[\![\$ccs]\!]$ if $\$t$ is a relative type in $[\![\$v,\ \$ccs]\!]$. A structure $\$ccs$ has a relative type $\$t$ if $\$t$ is a relative type in $[\![\$ccs]\!]$. A structure $\$v$ is a value in $[\![\$rtcs]\!]$ if $\$rtcs = \$v{:}\$mt$ for some $\$mt$. A structure $\$v$ is a value in $[\![\$t, \$rtcs]\!]$ if $\$cs^* = \$cs^*1\ \$v{:}\$mt\ \$cs^*2$, and $\$t \in \$mt$ for some $\$cs^*1$, $\$cs^*2$ and $\$mt$.

A structure $\$mt$ is a relative multi-type in $[\![\$v,\ (\$cs^*)]\!]$ if $\$cs^* = \$cs^*1\ \$v{:}\$mt1\ \$cs^*2$, and $\$mt \subseteq \$mt1$ for some $\$cs^*1$, $\$cs^*2$ and $\$mt$. A structure $\$mt$ is a relative multi-type in $[\![\$ccs]\!]$ if $\$mt$ is a relative multi-type in $[\![\$v,\ \$ccs]\!]$ for some $\$v$. A structure $\$v$ has a relative multi-type $\$mt$ in $[\![\$ccs]\!]$ if $\$mt$ is a relative multi-type in $[\![\$v,\ \$ccs]\!]$. A structure $\$ccs$ has a relative multi-type $\$mt$ if $\$mt$ is a relative multi-type in $[\![\$ccs]\!]$. A structure $\$v$ is a value in $[\![\$mt, \$rtcs]\!]$ if $\$cs^* = \$cs^*1\ \$v{:}\$mt1\ \$cs^*2$, and $\$mt \subseteq \$mt1$ for some $\$cs^*1$, $\$cs^*2$ and $\$mt$.

The relative typification operation categorizes elements of compound structures using relative types as category names and relative multi-types as category unions.

A structure $\$cs$ is typed if $\$cs$ is relatively typed, or $\$cs$ is absolutely typed. Let $\$\$tcs$ be a set of typed structures.

Conceptual transition systems are transition systems that have elements, and in which elements and states are conceptual structures.

Let $\$\$s$ be a set of objects called states. A subset of the set $\$\$s \times \$\$s$ is called a transition relation. Let $\$\$tr$ be a set of transition relations. A pair $(\$\$s, \hookrightarrow)$ is a transition system if $\hookrightarrow \in \$\$tr$.

An object $\$cts$ of the form $(\$\$ato,\ \$\$is, \hookrightarrow)$ is a conceptual transition system if $(\$\$cs[\![\$ato]\!], \hookrightarrow)$ is a transition system, and $\$\$is \subseteq \$\$cs$. The elements of $\$\$ato$ and $\$\$is$ are called atoms and initial states in $[\![\$cts]\!]$. The relation $\hookrightarrow$ is called a transition relation in $[\![\$cts]\!]$. Let $\$\$cts$ be a set of CTSs. The sets $\$\$s$ and $\$\$e$ of states and elements in $[\![\$\$cts]\!]$ are defined as follows: $\$\$s = \$\$e = \$\$cs$.

Let $[.\ \$mt]$, $[.\ \$mt \coloneqq \$v]$ and $[.\ \$mt \coloneqq]$ denote $[\$s\ .\ \$mt]$, $[\$s\ .\ \$mt \coloneqq \$v]$ and $[\$s\ .\ \$mt \coloneqq]$ for the current state $\$s$.

4. The CTSL language

Let $\$\$sa$ be a set of syntactic constructs called special atoms.

The CTSL language is a basic language of CTSs. It only defines the syntax of conceptual structures and does not concretize the set $\$\$sa$ and the transition relation $\hookrightarrow$. The extensions of CTSL for the special kinds of CTSs use the CTSL syntax and concretize $\$\$s$a and $\hookrightarrow$.

The set $\$\$ato$ of atoms in CTSL is defined as follows:

- if $\$o$ is a sequence of Unicode symbols except for the whitespace symbols and the symbols ", (,), {, }, ;, ,, and :, then $\$o \in \$\$ato$;
- $\$\$sa \subseteq \$\$ato$;
- if $\$o$ has the form "$\$o1$", $\$o1$ is a sequence of Unicode symbols, each occurrence of the symbol " in $\$o1$ is preceded by the symbol \, and each occurrence of the symbol \ in $\$o1$ is doubled, then $\$o \in \$\$ato$. In this case, the atom $\$o$ is called a string.

The set $\$\$cs$ of conceptual structures in CTSL is defined as follows:

- $\$ato \in \$\$cs$;
- $(\$cs^{*}) \in \$\$cs$;
- if the elements of $\$cs^{+}$ are pairwise distinct, and \$cs $\neq und$, then $\$cs{:}\,(\$cs^{+}) \in \$\$s$;
- if the elements of $\$cs^{+}$ are pairwise distinct, and \$cs $\neq und$, then $(\$cs^{+}){::}\,\$cs \in \$\$cs$.

The whitespace symbols, comma (,) and the semicolon (;) are interchangeable in compound structures in CTSL. For example, $(\$cs1,\ \$cs2)$, $(\$cs1; \$cs2)$ and $(\$cs1\ \$cs2)$ represent the same conceptual structure.

The bracket pairs (,) and {, } are interchangeable in compound structures in CTSL. For example, $(\$cs^*)$ and $\{\$cs^*\}$ represent the same compound conceptual structure.

5. The basic operations on conceptual structures

The conceptual structure access operation $[\$cs\ .\ \$mt]$ makes selection of elements of a compound structure in accordance with their relative types. It is defined as follows:

- if $\$cs = \$v{:}\,\$mt1$, and $\$mt \subseteq \$mt1$, then $[\$cs\ .\ \$mt] = \$v$;
- if $\$cs \in \$\$ccs$, and there exists only one element $\$cs1$ of $\$cs$ such that $\$cs1 = \$v{:}\,\$mt1$, and $\$mt \subseteq \$mt1$, then $[\$cs\ .\ \$mt] = \$v$;
- if $\$cs \in \$\$ccs$, $\$n > 1$, $\$cs1, \ldots, \$cs\$n$ are (ordered from left to right) elements of $\$cs$ such that $\$cs\$n1 = \$v\$n1{:}\,\$mt\$n1$, and $\$mt \subseteq \$mt\$n1$ for each $1 \leq \$n1 \leq \$n$, then $[\$cs\ .\ \$mt] = (\$v1 \ldots \$v\$n){::}\,(multivalue)$;
- otherwise, $[\$cs\ .\ \$mt] = und$.

An element $\$v$ is a value in $[\![\$mt, \$cs]\!]$ if $\$v = [\$cs\ .\ \$mt]$. The value of the form $\$mv{::}\,(multivalue)$ is called a multi-value. Let $[support\ \$cs]$ denote $\{\$mt \mid [\$cs\ .\ \$mt] \neq und\}$.

A structure $\$t$ is an (single-valued) attribute in $[\![\$cs]\!]$ if $[\$cs\ .\ (\$t)]$ does not have the absolute type $multivalue$. A structure $\$t$ is a multi-valued attribute in $[\![\$cs]\!]$ if $[\$cs\ .\ (\$t)]$ has the absolute type $multivalue$. Let $\$\$att$ and $\$\$mvatt$ be sets of attributes and multi-valued attributes.

A structure $\$cs$ is an (single-valued) attribute structure if $\$t$ is an attribute in $[\![\$cs]\!]$ for each $\$t \in \$\$cs$. A structure $\$ccs$ is a multi-valued attribute structure if $\$t$ is a multi-valued attribute in $[\![\$ccs]\!]$ for some $\$t \in \$\$cs$. Let $\$\$as$ and $\$\$mvas$ be sets of attribute structures and multi-valued attribute structures.

For example, the conceptual structure

$(x{:}\,\{variable\}\ int{:}\,\{(type\ x)\}\ 3{:}\,\{(value\ x)\}\ y{:}\,\{variable\}\ bool{:}\,\{(type\ y)\}\ true{:}\,\{(value\ y)\})$

defines the variables x and y by the multi-attribute $variable$, the types int and $bool$ of these variables by the parametric attribute $(type\ \$va)$, where the values of the parameter $\$va$ are variables, and the values 3 and $true$ of these variables by the parametric attribute $(value\ \$va)$.

A structure $\$mt$ is a (single-valued) multi-attribute in $[\![\$cs]\!]$ if $[\$cs\ .\ \$mt]$ does not have the absolute type $multivalue$. A structure $\$mt$ is a multi-valued multi-attribute in $[\![\$cs]\!]$ if $[\$cs\ .\ \$mt]$ has the absolute type $multivalue$. Let $\$\$matt$ and $\$\$mvmatt$ be sets of multi-attributes and multi-valued multi-attributes.

A structure $\$cs$ is an (single-valued) multi-attribute structure if $\$mt$ is a multi-attribute in $[\![\$cs]\!]$ for each $\$mt \in \{(t^+) | t^+ \in \$\$cs^+\}$. A structure $\$ccs$ is a multi-valued multi-attribute structure if

$\$mt$ is a multi-valued multi-attribute in $[\![\$ccs]\!]$ for some $\$mt \in \{(t^+) | t^+ \in \$\$cs^+\}$. Let $\$\$mas$ and $\$\$mvmas$ be sets of mult-attribute structures and multi-valued multi-attribute structures.

The conceptual structure update operation $[\$cs\ .\ \$mt \coloneqq \$v]$ replaces all values in $[\![\$mt,\ \$cs]\!]$ in $\$cs$ by $\$v$ from left to right and deletes these values in case when $\$v = und$. It is defined as follows (the first proper rule is applied):

- if \$mt is not a relative multi-type in $[\![(\$cs^*)]\!]$, and $\$v \neq und$, then $[(\$cs^*)\ .\ \$mt \coloneqq \$v] = (\$cs^*\ \$v{:}\,\$mt)$;
- if \$mt is not a relative multi-type in $[\![\$cs]\!]$, and $\$v \neq und$, then $[\$cs\ .\ \$mt \coloneqq \$v] = (\$cs\ \$v{:}\,\$mt)$;
- if $\$mt \sqsubseteq \$mt1$, and $\$v \neq und$, then $[\$v1{:}\,\$mt1\ .\ \$mt \coloneqq \$v] = \$v{:}\,\$mt1$;
- if $\$mt \sqsubseteq \$mt1$, then $[\$v1{:}\,\$mt1\ .\ \$mt \coloneqq und] = und$;
- $[(\$cs^*)\ .\ \$mt \coloneqq \$v] = ([\$cs^*\ .{:}{:}\{seq\}\,\$mt \coloneqq \$v])$;
- $[\$cs\ .\ \$mt \coloneqq \$v] = \$cs$.

The conceptual structure update operation $[\$cs^*\ .{:}{:}\{seq\}\,\$mt \coloneqq \$v]$ is defined as follows (the first proper rule is applied):

- if $\$mt \sqsubseteq \$mt1$, and $\$v \neq und$, then $[\$v1{:}\,\$mt1\ \$cs^*\ .{:}{:}\{seq\}\,\$mt \coloneqq \$v] = \$v{:}\,\$mt1\ [\$cs^*\ .{:}{:}\{seq\}\,\$mt \coloneqq \$v]$;
- if $\$mt \sqsubseteq \$mt1$, then $[\$v1{:}\,\$mt1\ \$cs^*\ .{:}{:}\{seq\}\,\$mt \coloneqq und] = [\$cs^*\ .{:}{:}\{seq\}\,\$mt \coloneqq und]$;
- $[\$cs\ \$cs^*\ .{:}{:}\{seq\}\,\$mt \coloneqq \$v] = \$cs\ [\$cs^*\ .{:}{:}\{seq\}\,\$mt \coloneqq \$v]$;
- $[[es]\ .{:}{:}\{seq\}\,\$mt \coloneqq \$v] = [es]$.

The conceptual structure update operation $[\$cs\ .\ \$mt1 \coloneqq \$v1\ \ldots\ \$mt\$n \coloneqq \$v\$n]$ is defined as follows:

- $[\$cs\ .\ \$mt \coloneqq \$v\ \$se] = [[\$cs\ .\$mt \coloneqq \$v]\ \$se]$;
- $[\$cs\ .\,[es]] = \$cs$.

The conceptual structure update operation $[\$cs\ .\ \$mt \coloneqq]$ is a shortcut for $[\$cs\ .\ \$mt \coloneqq und]$.

6. The properties of conceptual transition systems

An element $\$tra$ of the form $(\$s1,\ \$s2)$ is called a transition. The states $\$s1$ and $\$s2$ are called input and output states in $[\![tra]\!]$. Let $\$\$tra$ be a set of transitions.

A state $\$s1$ is final if there is no $\$s2$ such that $\$s1 \hookrightarrow \$s2$. Let $\$\$fs$ be a set of final states in $[\![\$cts]\!]$. A system $\$cts$ stops in $[\![\$s]\!]$ if $\$s$ is final.

A state $\$s$ is reachable if there exist $\$n > 0$, $\$s1$, …, $\$s\$n$ such that $\$s1 \in \$is$, $\$s\$n1 \hookrightarrow \$s[\$n1 + 1]$ for each $1 \leq \$n1 < \$n$, and $\$s = \$s\$n$. Let $\$\$rs$ be a set of reachable states in $[\![\$cts]\!]$.

An element $\$t$ is an attribute in $[\![\$cts]\!]$ if $\$t$ is an attribute in $[\![\$s]\!]$ for each $\$s \in \$\$rs[\![\$cts]\!]$. An element $\$t$ is a multi-valued attribute in $[\![\$cts]\!]$ if there exists $\$s \in \$\$rs[\![\$cts]\!]$ such that $\$t$ is a multi-valued attribute in $[\![\$s]\!]$. An element $\$mt$ is a multi-attribute in $[\![\$cts]\!]$ if $\$mt$ is a multi-attribute in $[\![\$s]\!]$ for each $\$s \in \$\$rs[\![\$cts]\!]$. An element $\$mt$ is a multi-valued multi-attribute in $[\![\$cts]\!]$ if there exists $\$s \in \$\$rs[\![\$cts]\!]$ such that $\$mt$ is a multi-valued multi-attribute in $[\![\$s]\!]$.

A system $\$cts$ is a CTS with return values if $value$ is an attribute in $[\![\$cts]\!]$. An element $\$v$ is a value in $[\![\$s]\!]$ if $\$v = [\$s\,.\,\{value\}]$. An element $\$v$ is a value in $[\![tra]\!]$ if $[\$tra\,..\,1] \hookrightarrow [\$tra\,..\,2]$, and $\$v$ is a value in $[\![[\$tra\,..\,2]]\!]$. A transition $\$tra$ returns a value $\$v$ if $\$v$ is a value in $[\![\$tra]\!]$. An element $\$v$ is undefined if $\$v = und$. The set $\$\$v$ of (possible) values is defined as follows: $\$\$v = \$\$e$.

A system $\$cts$ with return values can return exceptions. A value $\$v$ is an exception (an exceptional value) if $\$v$ has the absolute type $exception$. Thus, exceptions are specified by the absolute type $exception$. Let exc be a shortcut for $exception$. Let $\$\$ex$ be a set of exceptions. An element $\$t$ is called a type in $[\![\$ex]\!]$ if $\$t = [\$v\,.\,\{type\}]$, where $\$v$ is a value in $[\![\$ex]\!]$.

A value $\$v$ is abnormal if $\$v$ is undefined, or $\$v$ is an exception. Let $\$\$av$ be a set of abnormal values. A value $\$v$ is normal if $\$v \notin \$\$av$. Let $\$\$nv$ be a set of normal values. A transition $\$tra$ returns (generates) an exception $\$ex$ if $\$ex$ is a value in $[\![\$tra]\!]$. A transition $\$tra$ is normally executed if $\$tra$ does not return exceptions.

A system $\$cts$ is a CTS with programs if $program$ is an attribute in $[\![\$cts]\!]$, and the value of this attribute is a compound structure. A compound structure $\$p$ is a program in $[\![\$s]\!]$ if $\$p = [\$s\,.\,\{program\}]$. Let $\$\$p$ be a set of programs. A program in $[\![\$s]\!]$ is empty if $[\$s\,.\,\{program\}] = ()$. A program in $[\![\$s]\!]$ initiates transitions from $\$s$.

The elements that initiate transitions are called executable elements. Let $\$\$ee$ be a set of executable elements. A program $\$p$ is an executable element, and the elements of $[\![\$p]\!]$ are executable elements.

A system $\$cts$ is a CTS with direct stop if $stop$ is an attribute in $[\![\$cts]\!]$, and $\$s$ is final for each $\$s$ such that $[\$s\,.\,\{stop\}] \neq und$. A state $\$s$ is a stop state if $[\$s\,.\,\{stop\}] \neq und$. The value of the attribute $stop$ specifies why the system $\$cts$ stopped.

An attribute $\$bi$ in $[\![\$s]\!]$ is a backtracking invariant in $[\![\$s]\!]$ if $[\$s\,.\,\{((backtracking\ invariant)\ \$bi)\}] \neq und$. An attribute $\$bi$ is a backtracking invariant in

$[\![\$cts]\!]$ if $\$bi$ is a backtracking invariant in $[\![\$s]\!]$ for some $\$s \in \$\$rs[\![\hookrightarrow [\![\$cts]\!]]\!]$. Backtracking invariants preserves their values after backtracking. Let $\$\$bi$ be a set of backtracking invariants.

Let $[(propagate\ backtracking\ invariants)\ \$s1\ \$s2]$ denote the state $\$s$ such that $[\$s\ .\ \{\$att\}] = [\$s1\ .\ \{\$att\}]$ for each $\$att \in \$\$bi[\![\$s2]\!]$, and $[\$s\ .\ \{\$att\}] = [\$s2\ .\ \{\$att\}]$ for each $\$att \notin \$\$bi[\![\$s2]\!]$.

Let $\$e^*\ \#\ \$v\ \#\ \$s$ and $\$e^*\ \#\ \$s$ denote $[\$s\ .\ \{program\} := (\$e^*),\ \{value\} := \$v]$ and $[\$s\ .\ \{program\} := (\$e^*)]$.

A system $\$cts$ is a CTS with backtracking in $[\![\$\$bi]\!]$ if $\$cts$ is a CTS with return values with programs, $((backtracking\ invariant)\ \$bi)$ is a parametric attribute in $[\![\$cts]\!]$, $\$bi$ is a backtracking invariant in $[\![\$cts]\!]$ for each $\$bi \in \$\$bi$, and $\hookrightarrow [\![\$cts]\!]$ satisfies the following properties:

- if $\$v[\![\$s]\!] \neq und$, then $(backtracking\ \$s1\ \$e^*1)\ \$e^*\ \#\ \$s\ \hookrightarrow \$e^*\ \#\ \$s$;
- if $\$v[\![\$s]\!] = und$, then
  $(backtracking\ \$s1\ \$e^*1)\ \$e^*\ \#\ \$s\ \hookrightarrow$
  $\$e^*1\ \$e^*\ \#\ [(propagate\ backtracking\ invariants)\ \$s1\ \$s]$.

The element $\$e$ of the form $(backtracking\ \$s\ \$e^*)$ is called a backtracking point. The objects $\$s$ and $\$e^*$ are called a state and a program prefix in $[\![\$e]\!]$.

7. Examples of conceptual models of programming languages

Let $\$\$l$ be a set of programming languages. Let $[am\ \$l]$ denotes an abstract machine executing the constructs of $\$l$. A tuple $(\$\$t,\ \$\$v,\ \$\$ex, \$\$s,\ \$\$c, \$\$ax)$ is a conceptual model of $\$l$ in CTSL if $\$\$t[\![\$l]\!]$ is a set of elements in CTSL representing the types of $\$l$, $\$\$v[\![\$l]\!]$ is a set of elements in CTSL representing the values in $[am\ \$l]$ (in particular, the values of the types of $\$l$), $\$\$ex[\![\$l]\!]$ is a set of exceptions in CTSL representing the exceptions in $[am\ \$l]$, $\$\$ex[\![\$l]\!] \sqsubseteq \$\$v[\![\$l]\!]$, $\$\$s[\![\$l]\!]$ is a set of states in CTSL representing the states of $[am\ \$l]$, $\$\$c[\![\$l]\!]$ is a set of executable elements in CTSL representing the executable constructs of $[am\ \$l]$ (in particular, the elements of programs in $\$l$), and $\$\$ax$ is a set of axioms representing the constraints for the conceptual model of $\$l$ (the other elements of the tuple).

Let $[content\ \$t]$ denote the set of values in $[\![\$t]\!]$. The set $[content\ \$t]$ is called the content in $[\![\$t]\!]$. The fact that $\$\$t$ and $\$t$ depend on $\$s$ is denoted by $\$\$t[\![\$s]\!]$ and $\$t[\![\$s]\!]$.

Let **Axiom:** $\$ax$ denote that $\$ax$ is an axiom of the conceptual model of $\$l$.

The family of model programming languages (MPLs) is described and their conceptual models are defined in this section.

7.1. MPL1: types, typed variables and basic statement

The MPL1 language is an extension of CTSL that adds types, typed variables, the variable access operation, and the basic statements such as variable declarations, variable assignments, if statements, while statements and block statements.

7.1.1. Types, values, states

For MPL1, $\$\$t[\![\text{MPL1}]\!] = \{int,\ nat\}$, $\$\$v[\![\text{MPL1}]\!] = \$\$i \cup \$\$n$, and $\$\$ex[\![\text{MPL1}]\!] = \emptyset$, where $[content\ int] = \$\$i$, and $[content\ nat] = \$\$n$.

An element $\$e$ is a name if $\$e$ is normal. Let $\$\$na$ be a set of names.

The attribute $(variable\ \$na)$ specifies variables in MPL1. A name $\$na$ is a variable in $[\![\$s]\!]$ if $[\$s\ .\ \{(variable\ \$na)\}] \neq und$. Let $\$\$va$ be a set of variables.

The attribute $(type\ \$va)$ specifies the type of the variable $\$va$. A type $\$t$ is a type in $[\![\$va,\ \$s]\!]$ if $[\$s\ .\ \{(type\ \$va)\}] = \$t$.

Axiom: If $[\$s\ .\ \{(variable\ \$va)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va)\}] \in \$\$t[\![\$s]\!]$.

The attribute $(value\ \$va)$ specifies the value of the variable $\$va$. A value $\$v$ is a value in $[\![\$va,\ \$s]\!]$ if $[\$s\ .\ \{(value\ \$va)\}] = \$v$.

Axiom: If $[\$s\ .\ \{(value\ \$va)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va)\}] = \$t$, and $[\$s\ .\ \{(value\ \$va)\}] \in [content\ \$t]$ for some $\$t \in \$\$t[\![\$s]\!]$.

7.1.2. Constructs

The MPL1 program is represented by the element $(program\ \$na\ \$c^*)$. It specifies a program with the name $\$na$ and the body $\$c^*$.

The variable declaration is represented by the element $(var\ \$va\ \$t)$. It declares the variable $\$va$ of the type $\$t$.

Axiom: Variable declarations are elements of the program body.

The variable access operation is represented by $\$va$. It returns the value of the variable $\$va$.

The variable assignment is represented by the element $(\$va \setminus\!\!:= \ \$c)$. If $\$v$ is a value of $\$c$, then it assigns $\$v$ to the variable $\$va$.

The block statement is represented by the element $(block\ \$c^*)$. It specifies the block statement with the body $\$c^*$.

The if statement is represented by the element $(\backslash if\ \$c\ then\ \$c^*1\ else\ \$c^*2)$. It specifies the if statement with the condition $\$c$, the then-branch $\$c^*1$ and the else-branch $\$c^*2$. The element $(\backslash if\ \$c\ then\ \$c^*1)$ is a shortcut for $(\backslash if\ \$c\ then\ \$c^*1\ else)$.

The while statement is represented by the element $(\backslash while\ \$c\ do\ \$c^*)$. It specifies the while statement with the condition $\$c$ and the body $\$c^*$.

7.2. MPL2: variable scopes

The MPL2 language is an extension of MPL1 that adds the variable scopes feature.

The relative scope of the variable $\$va$ occuring in the element $\$c$ is the number of blocks surrounding this occurrence of $\$va$ in $\$c$. The value and type of $\$va$ depend on its scope. The variable $\$va$ can be global (with the scope 0) and local. The following example illustrates variable scopes:

$(program\ scopes$ // $x = und,\ y = und,\ scope = 0$

$(var\ x\ int)$ // $x = und,\ y = und,\ scope = 0$

$(x := 0)$ // $x = 0,\ y = und,\ scope = 0$

$(var\ y\ bool)$ // $x = 0,\ y = und,\ scope = 0$

$(y := true)$ // $x = 0,\ y = true,\ scope = 0$

$(block$ // $x = 0,\ y = true,\ scope = 1$

$(var\ x\ bool)$ // $x = und,\ y = true,\ scope = 1$

$(x := false)$ // $x = false,\ y = true,\ scope = 1$

$(block$ // $x = false,\ y = true,\ scope = 2$

$(var\ x\ int)$ // $x = und,\ y = true,\ scope = 2$

$(x := 2)$ // $x = 2,\ y = true,\ scope = 2$

$)$ // $x = false,\ y = true,\ scope = 1$

$(var\ y\ int)$ // $x = false,\ y = und,\ scope = 1$

$(y := 1)$ // $x = false,\ y = 1,\ scope = 1$

$)$ // $x = 0,\ y = true,\ scope = 0$

$)$.

7.2.1. Types, values, states

For MPL2, $\$\$t[\![\text{MPL2}]\!] = \$\$t[\![\text{MPL1}]\!]$, $\$\$v[\![\text{MPL2}]\!] = \$\$v[\![\text{MPL1}]\!]$, and $\$\$ex[\![\text{MPL2}]\!] = \emptyset$.

Let $\$\$sc$ be a set of (relative) variable scopes.

The attribute $(variable\ \$na\ \$sc)$ specifies variables in $[\![\$sc]\!]$. A name $\$na$ is a variable in $[\![\$s,\ \$sc]\!]$ if $[\$s\ .\ \{(variable\ \$na\ \$sc)\}] \neq und$.

A variable $\$va[\![\$s,\ \$sc]\!]$ is global if $\$sc = 0$. A variable $\$va[\![\$s,\ \$sc]\!]$ is local if $\$sc > 0$.

The attribute $(current\ scope)$ specifies the scope of the current block. A scope $\$sc$ is a current scope in $[\![\$s]\!]$ if $[\$s\ .\ \{(current\ scope)\}] = \$sc$. A name $\$na$ is a variable in $[\![\$s]\!]$ if $\$na$ is a variable in $[\![\$s, \$sc]\!]$ for some $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$. A scope $\$sc$ is a scope in $[\![\$va, \$s]\!]$ if

$\$va$ is a variable in $[\![\$s, \$sc]\!]$, $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$, and $\$va$ is not a variable in $[\![\$s, \$sc1]\!]$ for each $\$sc < \$sc1 \leq [\$s\ .\ \{(current\ scope)\}]$.

The attribute $(type\ \$va\ \$sc)$ specifies the type of the variable \$va in $[\![\$sc]\!]$. A type $\$t$ is a type in $[\![\$va, \$s, \$sc]\!]$ if $[\$s\ .\ \{(type\ \$va\ \$sc)\}] = \$t$. A type $\$t$ is a type in $[\![\$va, \$s]\!]$ if $\$t$ is a type in $[\![\$va, \$s, \$sc]\!]$, where $\$sc$ is a scope in $[\![\$va, \$s]\!]$.

Axiom: If $[\$s\ .\ \{(variable\ \$va\ \$sc)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va\ \$sc)\}] \in \$\$t[\![\$s]\!]$.

The attribute $(value\ \$va\ \$sc)$ specifies the value of the variable $\$va$ in $[\![\$sc]\!]$. A value $\$v$ is a value in $[\![\$va, \$s, \$sc]\!]$ if $[\$s\ .\ \{(value\ \$va\ \$sc)\}] = \$v$. A value $\$v$ is a value in $[\![\$va, \$s]\!]$ if $\$v$ is a value in $[\![\$va, \$s, \$sc]\!]$, where $\$sc$ is a scope in $[\![\$va, \$s]\!]$.

Axiom: If $[\$s\ .\ \{(value\ \$va\ \$sc)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va\ \$sc)\}] = \$t$, and $[\$s\ .\ \{(value\ \$va\ \$sc)\}] \in [content\ \$t]$ for some $\$t \in \$\$t[\![\$s]\!]$.

7.2.2. Constructs

For MPL2, $\$\$c[\![\text{MPL2}]\!] = \$\$c[\![\text{MPL1}]\!]$.

Axiom: Variable declarations are elements of the program body or of block bodies.

7.3. MPL3: functions

The MPL3 language is an extension of MPL2 that adds the functions feature: declarations and calls of functions, and the return statement.

Axiom: Function overloading is prohibited.

7.3.1. Types, values, states

The exception $(return{:}\{type\},\ v{:}\{value\}){::}\{exc\}$ specifies the execution of the return statement with the return value v. Let $\$\$ex1$ be a set of such exceptions.

For MPL2, $\$\$t[\![\text{MPL3}]\!] = \$\$t[\![\text{MPL2}]\!]$, $\$\$v[\![\text{MPL3}]\!] = \$\$v[\![\text{MPL2}]\!] \cup \$\$ex[\![\text{MPL3}]\!]$, and $\$\$ex[\![MPL3]\!] = \$\$ex1$.

The attribute $(function\ \$na)$ specifies functions. A name $\$na$ is a function in $[\![\$s]\!]$ if $[\$s\ .\ \{(function\ \$na)\}] \neq und$. Let $\$\$f$ be a set of functions.

The attribute $(arity\ \$f)$ specifies the arity of the function $\$f$. A number $\$n$ is an arity in $[\![\$f, \$s]\!]$ if $[\$s\ .\ \{(arity\ \$f)\}] = \$n$.

The attribute $(argument\ \$f\ \$n)$ specifies the $\$n$-th argument of the function $\$f$. A name $\$na$ is an argument in $[\![\$f, \$n]\!]$ if $[\$s\ .\ \{(argument\ \$f\ \$n)\}] = \$na$, and $1 \leq \$n \leq [\$s\ .\ \{(arity\ \$f)\}]$. Let $\$\$a$ be a set of arguments.

The attribute $((argument\ type)\ \$f\ \$n)$ specifies the type of the $\$n$-th argument of the function $\$f$. A type $\$t$ is a type in $[\![\$f, \$n]\!]$ if $[\$s\ .\ \{((argument\ type)\}\ \$f\ \$n)] \neq und$, and $1 \leq \$n \leq [\$s\ .\ \{(arity\ \$f)\}]$.

The attribute $((return\ type)\ \$f)$ specifies the return type of the function $\$f$. A type $\$t$ is a return type in $[\![\$f]\!]$ if $\left[\$s\ .\ \left\{\left((return\ type)\$f\right)\right\}\right] = \$t$.

The attribute $(body\ \$f)$ specifies the body of the function $\$f$. A sequence $\$c^*$ is a body in $[\![\$f]\!]$ if $[\$s\ .\ \{(body\ \$f)\}] = (\$c^*)$.

A call level is a number of embedded function calls. Let $\$\$cl$ be a set of call levels. The attribute $(current\ call\ level)$ specifies the current call level. A level $\$cl$ is a current call level in $[\![\$s]\!]$ if $\$cl = [\$s\ .\ \{(current\ call\ level)\}]$.

The $(current\ return\ type)$ specifies the return type in the current function call. A type $\$t$ is a current return type in $[\![\$s]\!]$ if $[\$s\ .\ \{(current\ return\ type)\}] = \$t$.

The attribute $(variable\ \$na\ \$sc\ \$cl)$ specifies variables in $[\![\$sc, \$cl]\!]$. A name $\$na$ is a variable in $[\![\$s, \$sc, \$cl]\!]$ if $[\$s\ .\ \{(variable\ \$na\ \$sc\ \$cl)\}] \neq und$, and $\$sc = 0$ implies $\$cl = 0$.

A variable $\$va[\![\$s, \$sc, \$cl]\!]$ is global if $\$sc = 0$, and $\$cl = 0$. A variable $\$va[\![\$s, \$sc, \$cl]\!]$ is local if $\$sc > 0$, and $\$cl > 0$.

A name $\$na$ is a variable in $[\![\$s, \$cl]\!]$ if $\$na$ is a variable in $[\![\$s, \$sc,\ \$cl]\!]$ for some $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$. A scope $\$sc$ is a scope in $[\![\$va, \$s,\ \$cl]\!]$ if $\$va$ is a variable in $[\![\$s, \$sc,\ \$cl]\!]$, $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$, and $\$va$ is not a variable in $[\![\$s,\ \$sc1, \$cl]\!]$ for each $\$sc < \$sc1 \leq [\$s\ .\ \{(current\ scope)\}]$.

The attribute $(type\ \$va\ \$sc\ \$cl)$ specifies the type of the variable $\$va$ in $[\![\$sc,\ \$cl]\!]$. A type $\$t$ is a type in $[\![\$va, \$s, \$sc, \$cl]\!]$ if $[\$s\ .\ \{(type\ \$va\ \$sc\ \$cl)\}] = \$t$.

Axiom: If $[\$s\ .\ \{(variable\ \$va\ \$sc\ \$cl)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va\ \$sc\ \$cl)\}] \in \$\$t[\![\$s]\!]$.

A type $\$t$ is a type in $[\![\$va, \$s, \$cl]\!]$ if $\$t$ is a type in $[\![\$va, \$s, \$sc, \$cl]\!]$, and $\$sc$ is a scope in $[\![\$va, \$cl]\!]$ for some $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$. A type $\$t$ is a type in $[\![\$va, \$s, \$cl]\!]$ if $\$t$ is a type in $[\![\$va, \$s, \$sc, \$cl]\!]$, where $\$sc$ is a scope in $[\![\$va, \$s, \$cl]\!]$.

The attribute $(value\ \$va\ \$sc\ \$cl)$ specifies the value of the variable $\$va$ in $[\![\$sc,\ \$cl]\!]$. A value $\$v$ is a value in $[\![\$va, \$s, \$sc]\!]$ if $[\$s\ .\ \{(value\ \$va\ \$sc)\}] = \$v$. A value $\$v$ is a value in $[\![\$va, \$s,\ \$cl]\!]$ if $\$v$ is a value in $[\![\$va, \$s, \$sc,\ \$cl]\!]$, where $\$sc$ is a scope in $[\![\$va, \$s,\ \$cl]\!]$.

Axiom: If $[\$s\ .\ \{(value\ \$va\ \$sc\ \$cl)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va\ \$sc\ \$cl)\}] = \$t$, and $[\$s\ .\ \{(value\ \$va\ \$sc\ \$cl)\}] \in [content\ \$t]$ for some $\$t \in \$\$t[\![\$s]\!]$.

7.3.2. Constructs

An object $\$o$ is a typed name if $\$o = \$na\ \$t$. Let $\$\$tna$ be a set of typed names.

The function declaration is represented by the element $(function\ \$f\ (\$tna1\ \dots\ \$tna\$n)\ \$t\ \$c^*)$. It specifies the declaration of the function $\$f$, the arguments $\$na[\![\$tna1]\!]$, ..., $\$na[\![\$tna\$n]\!]$ of the types $\$t[\![\$tna1]\!]$, ..., $\$t[\![\$tna\$n]\!]$, the return type $\$t$, and the body $\$c^*$.

Axiom: Function declarations are elements of the program body.

The return statement is represented by the element $(return\ \$c)$. It specifies the return statement with the return element $\$c$. If $\$v$ is a value of $\$c$, then it returns $\$v$.

The function call is represented by the element $(call\ \$f\ \$c^*)$. It specifies the call of the function $\$f$ with the arguments $\$c^*$.

7.4. MPL4: procedures

The MPL4 language is an extension of MPL3 that adds the procedures feature: declarations and calls of procedures, and the exit statement.

Axiom: Procedure overloading is prohibited.

Axiom: The sets of function names and procedure names are disjoint.

7.4.1. Types, values, states

The exception $(exit: \{type\}):: \{exc\}$ specifies the execution of the exit statement. Let $\$\$ex1$ be a set of such exceptions.

For MPL4, $\$\$t[\![\text{MPL4}]\!] = \$\$t[\![\text{MPL3}]\!]$, $\$\$v[\![\text{MPL4}]\!] = \$\$v[\![\text{MPL3}]\!] \cup \$\$ex[\![\text{MPL4}]\!]$, and $\$\$ex[\![MPL4]\!] = \$\$ex[\![MPL3]\!] \cup \$\$ex1$.

The attribute $(procedure\ \$na)$ specifies procedures. A name $\$na$ is a procedure in $[\![\$s]\!]$ if $[\$s\ .\ \{(procedure\ \$na)\}] \neq und$. Let $\$\$pr$ be a set of procedures.

The attribute $(arity\ \$pr)$ specifies the arity of the procedure $\$pr$. A number $\$n$ is an arity in $[\![\$pr, \$s]\!]$ if $[\$s\ .\ \{(arity\ \$pr)\}] = \$n$.

The attribute $(argument\ \$pr\ \$n)$ specifies the $\$n$-th argument of the procedure $\$pr$. A name $\$na$ is an argument in $[\![\$pr, \$n]\!]$ if $[\$s\ .\ \{(argument\ \$pr\ \$n)\}] = \$na$, and $1 \leq \$n \leq [\$s\ .\ \{(arity\ \$pr)\}]$.

The attribute $((argument\ type)\ \$t\ \$pr\ \$n)$ specifies the type of the $\$n$-th argument of the procedure $\$pr$. A type $\$t$ is a type in $[\![\$pr, \$n]\!]$ if $[\$s\ .\ \{((argument\ type)\ \$t\ \$pr\ \$n)\}] \neq und$, and $1 \leq \$n \leq [\$s\ .\ \{(arity\ \$pr)\}]$.

The attribute $(body\ \$pr)$ specifies the body of the procedure $\$pr$. A sequence $\$c^*$ is a body in $[\![\$pr]\!]$ if $[\$s\ .\ \{(body\ \$pr)\}] = (\$c^*)$.

A call level is redefined in MPL4 as a number of embedded function and procedure calls.

7.4.2. Constructs

The procedure declaration is represented by the element $(procedure\ \$pr\ (\$tna1\ \ldots\ \$tna\$n)\ \$c^*)$. It specifies the declaration of the procedure $\$pr$, the arguments $\$na[\![\$tna1]\!]$, ..., $\$na[\![\$tna\$n]\!]$ of the types $\$t[\![\$tna1]\!]$, ..., $\$t[\![\$tna\$n]\!]$, and the body $\$c^*$.

Axiom: Procedure declarations are elements of the program body.

The exit statement is represented by the element $exit$.

The procedure call is represented by the element $(call\ \$pr\ \$c^*)$. It specifies the call of the procedure $\$pr$ with the arguments $\$c^*$.

7.5. MPL5: pointers

The MPL5 language is an extension of MPL4 that adds the pointers feature: the pointer types, the operations of pointer content access, variable address access and pointer deletion, statements of pointer content assignment and pointer deletion.

7.5.1. Types, values, states

An element $(pointer\ \$t)$ is called a pointer type in $[\![\$t]\!]$. An element $\$e$ is a pointer type if $\$e$ is a pointer type in $[\![\$t]\!]$ for some $\$t$. Let $\$\$pt$ be a set of pointer types.

The absolute type $pointer$ specifies pointers in MPL5. An element $\$e \in \$\$ats$ is a pointer if $\$e$ has the absolute type $pointer$, and the value in $[\![\$e]\!]$ belongs to $\$\$n$. Thus, pointers are represented in MPL5 by natural numbers categorized by the type $pointer$. Let $\$\$po$ be a set of pointers.

For MPL5, $\$\$t[\![MPL5]\!] = \$\$t[\![MPL4]\!] \cup \$\$pt$, $\$\$v[\![\mathrm{MPL5}]\!] = \$\$v[\![\mathrm{MPL4}]\!] \cup \$\$po$, and $\$\$ex[\![MPL5]\!] = \$\$ex[\![MPL4]\!]$.

The attribute $(pointer\ \$po)$ specifies pointers in states. A pointer $\$po$ is a pointer in $[\![\$s]\!]$ if $[\$s\ .\ \{(pointer\ \$po)\}] \neq und$.

The attribute $((content\ type)\ \$po)$ specifies the content type of the pointer $\$po$. A type $\$t$ is a content type in $[\![\$po,\ \$s]\!]$ if $\left[\$s\ .\ \left\{\left((content\ type)\$po\right)\right\}\right] = \$t$. It specifies the type of the content to which the pointer $\$po$ refers.

Axiom: If $[\$s\ .\ \{(pointer\ \$po)\}] \neq und$, then $[\$s\ .\ \{((content\ type)\ \$po)\}] \in \$\$t[\![\$s]\!]$.

The pointer $\$po$ has the type $(pointer\ \$t)$ in $[\![\$s]\!]$ if $\$t$ is a content type in $[\![\$po,\ \$s]\!]$. Thus, the type $(pointer\ \$t)$ specifies pointers with the content type $\$t$.

The attribute $(content\ \$po)$ specifies the content of the pointer $\$po$. A value $\$v$ is a content in $[\![\$po,\ \$s]\!]$ if $[\$s\ .\ \{(content\ \$po)\}] = \$v$.

Axiom: If $[\$s\,.\,\{(content\ \$po)\}] \neq und$, then $[\$s\,.\,\{((content\ type)\ \$po)\}] = \$t$, and $[\$s\,.\,\{(content\ \$po)\}] \in [content\ \$t]$ for some $\$t[\![\$s]\!]$.

The attribute $(pointer\ \$va\ \$sc\ \$cl)$ specifies variables in $[\![\$sc, \$cl]\!]$ by the pointers referring to their values. A name $\$na$ is a variable in $[\![\$p,\ \$s, \$sc, \$cl]\!]$ if $[\$s\,.\,\{(pointer\ \$na\ \$sc\ \$cl)\}] = \$p$. A variable $\$va$ represents $\$p$ in $[\![\$s, \$sc, \$cl]\!]$ if $\$va$ is a variable in $[\![\$p,\ \$s, \$sc, \$cl]\!]$. A name $\$na$ is a variable in $[\![\$s, \$sc, \$cl]\!]$ if $\$na$ is a variable in $[\![\$p,\ \$s, \$sc, \$cl]\!]$ for some $\$p \in \$\$p[\![\$s]\!]$. A pointer $\$p$ is a pointer in $[\![\$va, \$s,\ \$sc, \$cl]\!]$ if $\$va$ is a variable in $[\![\$p,\ \$s, \$sc, \$cl]\!]$.

The content of the pointer $\$p[\![\$va]\!]$ coincides with the value of the variable $\$va$. A type $\$t$ is a type in $[\![\$va, \$s, \$sc, \$cl]\!]$ if $[\$s\,.\,\{((content\ type)\ [\$s\,.\,\{(pointer\ \$na\ \$sc\ \$cl)\}])\}] = \$t$. A value $\$v$ is a value in $[\![\$va, \$s, \$sc, \$cl]\!]$ if $[\$s\,.\,\{(content\ [\$s\,.\,\{(pointer\ \$va\ \$sc\ \$cl)\}])\}] = \$v$.

Axiom: If $[\$s\,.\,\{(pointer\ \$va\ \$sc\ \$cl)\}] \neq und$, then $[\$s\,.\,\{(pointer\ \$va\ \$sc\ \$cl)\}] \in \$\$po[\![\$s]\!]$.

7.5.2. Constructs

Pointers are represented by elements of $\$\$po$.

The pointer content access operation is represented by the element $(*\ \$c)$. If $\$po$ is a value of $\$c$, then it returns the content in $[\![\$po]\!]$.

The variable address access operation is represented by the element $(\&\ va)$. It returns the pointer in $[\![\$s, \$va, [\$s\,.\,\{(current\ scope)\}], [\$s\,.\,\{(current\ call\ level)\}]]\!]$.

The pointer addition operation is represented by the element $(new\ \$pt)$. It adds a new pointer of the type $\$pt$.

The pointer content assignment statement is represented by the element $(*\ \$c1 := \ \$c2)$. If $\$po$ and $\$v$ are the values of $\$c1$ and $\$c2$, and they have the types $(pointer\ \$t)$ and $\$t$ for some $\$t$, then it assigns $\$v$ to the content of $\$po$.

The pointer deletion operation is represented by the element $(delete\ \$c)$. If $\$po$ is a value of $\$c$, then it specifies the deletion of the pointer $\$po$.

7.6. MPL6: jump statements

The MPL6 language is an extension of MPL5 that adds the jump statements feature: break statement, continue statement, goto statement and labelled statement.

7.6.1. Types, values, states

The exception $(break{:}\{type\})::\{exc\}$ specifies the execution of the break statement.

The exception $(continue{:}\{type\})::\{exc\}$ specifies the execution of the continue statement.

The exception $(goto{:}\{type\},\ \$l{:}\{label\}){::}\{exc\}$ specifies the execution of the goto statement with the label $\$l$.

Let $\$\$ex1$ be a set of such exceptions.

For MPL6, $\$\$t[\![\text{MPL6}]\!] = \$\$t[\![\text{MPL5}]\!]$, $\$\$v[\![\text{MPL6}]\!] = \$\$v[\![\text{MPL5}]\!] \cup \$\$ex[\![\text{MPL6}]\!]$, and $\$\$ex[\![MPL6]\!] = \$\$ex[\![MPL5]\!] \cup \$\$ex1$.

An element $\$l$ is a label if $\$l$ is a name. Let $\$\$l$ be a set of labels.

7.6.2. Constructs

The label statement with the label $\$l$ is represented by the element $(label\ \$l)$. It specifies the program point labelled by the label $\$l$. The labelled statement is represented by the sequence $(label\ \$l)\ \$c$. It specifies that the statement $\$c$ is labelled by the label $\$l$.

The break statement is represented by the element $break$.

The continue statement is represented by the element $continue$.

The goto statement is represented by the element $(goto\ \$l)$.

7.7. MPL7: dynamic arrays

The MPL7 language is an extension of MPL6 that adds the dynamic arrays feature: dynamic array types, the array element access operation and the array element assignment statement.

7.7.1. Types, values, states

An element $(array\ \$t)$ is called a dynamic array type in $[\![\$t]\!]$. An element $\$e$ is a dynamic array type if $\$e$ is a dynamic array type in $[\![\$t]\!]$ for some $\$t$. Let $\$\$dat$ be a set of dynamic array types.

An element $\$e$ is an array type if $\$e$ is a dynamic array type. Let $\$\$at$ be a set of array types.

The absolute type $(dynamic\ array)$ specifies dynamic arrays. An element $\$dar$ is a dynamic array if $\$dar = ((\$e^*){:}\{content\},\ \$t{:}\{type\}){::}\{(dynamic\ array)\}$, and $\$e^*$ consists of the elements of $[content\ \$t]$. The elements \$se and \$t are called the content and the element type in $[\![\$dar]\!]$. Let $\$\$dar$ be a set of dynamic arrays.

An element $\$e$ is an array if $\$e$ is a dynamic array. Let $\$\$ar$ be a set of arrays.

For MPL7, $\$\$t[\![MPL7]\!] = \$\$t[\![MPL6]\!] \cup \$\$\text{d}at$, $\$\$v[\![\text{MPL7}]\!] = \$\$v[\![\text{MPL6}]\!] \cup \$\$dar$, and $\$\$ex[\![MPL7]\!] = \$\$ex[\![MPL6]\!]$.

The dynamic array $\$dar$ has the type $(array\ \$t)$ if $\$t$ is an element type in $[\![\$dar,\ \$s]\!]$. Thus, the type $(array\ \$t)$ specifies dynamic arrays with the element type $\$t$.

A value $\$v$ is a value in $[\![\$ar,\ \$n]\!]$ if $\left[[\$ar\ .\ \{content\}]..\ \$n\right] = \$v$. The element $\$v$ specifies the value of $\$n$-th element of $\$ar$.

7.7.2. Constructs

The array element access operation is represented by the element ($\$c1$ [$\$c2$]). If $\$ar$ and $\$n$ are the values of $\$c1$ and $\$c2$, then it returns the value in ⟦$\$ar,\ \$n,\ \$s$⟧.

The array element assignment operation is represented by the element ($\$c1$ [$\$c2$] $:=$ $\$c3$). If $\$dar$, $\$n$ and $\$v$ are the values of $\$c1$, $\$c2$ and $\$c3$, and $1 \le \$n \le [len\ [\$dar\ .\ \{content\}]]$, then it replaces the value of the $\$n$-th element of $\$dar$ by $\$v$. If $\$dar$, $\$n$ and $\$v$ are the values of $\$c1$, $\$c2$ and $\$c3$, and $\$n > [len\ [\$dar\ .\ \{content\}]]$, then it replaces the value of the $\$n$-th element of $\$dar$ by $\$v$ and the values of the elements of $\$dar$ from $[len\ [\$dar\ .\ \{content\}]] + 1$ to $\$n - 1$ by und.

The array element assignment operation is represented by the element ($\$c1$ [$\$c2$] $:=$ $\$c3$) where the expressions $\$c1$, $\$c2$ and $\$c3$ have the types $(array\ \$t)$, nat and $\$t$ for some $\$t$. It assigns $\$v$ to the $\$n$-th element of $\$ar$ where $\$ar$, $\$n$ and $\$v$ are the values of $\$c1$, $\$c2$ and $\$c3$ in ⟦$\$s$⟧.

7.8. MPL8: static arrays

The MPL7 language is an extension of MPL6 that adds the static arrays feature: static array types, the array element access operation and the array element assignment statement.

7.8.1. Types, values, states

An element $(array\ \$t\ \$n)$ is called a static array type in ⟦$\$t$⟧. An element $\$e$ is a static array type if $\$e$ is a static array type in ⟦$\$t$⟧ for some $\$t$. Let $\$\$sat$ be a set of static array types.

An element $\$e$ is an array type if $\$e$ is a dynamic array type, or $\$e$ is a static array type. Let $\$\$at$ be a set of array types.

The absolute type $(static\ array)$ specifies arrays. An element $\$sar$ is a static array if $\$sar = ((\$e^*){:}\{content\},\ \$t{:}\{type\}){:}{:}\{(static\ array)\}$, and $\$e^*$ consists of the elements of $[content\ \$t]$. The elements $se and $t are called the content and the element type in ⟦$\$sar$⟧. Let $\$\$sar$ be a set of arrays.

An element $\$e$ is an array if $\$e$ is a dynamic array, or $\$e$ is a static array. Let $\$\$ar$ be a set of arrays

For MPL7, $\$\$t[\![MPL7]\!] = \$\$t[\![MPL6]\!] \cup \$\$sat$, $\$\$v$⟦MPL7⟧ $=$ $\$\$v$⟦MPL6⟧ $\cup\ \$\$sar$, and $\$\$ex[\![MPL7]\!] = \$\$ex[\![MPL6]\!]$.

The array $\$sar$ has the type $(array\ \$t\ \$n)$ if $\$t$ is an element type in ⟦$\$sar,\ \$s$⟧, and $[len\ [\$sar\ .\ \{content\}]] = \$n$. Thus, the type $(array\ \$t\ \$n)$ specifies static arrays with the element type $\$t$ and the content of the length $\$n$.

7.8.2. Constructs

The array element access operation does not depend on the specific features of dynamic arrays. Therefore it is extended for static arrays by simple array redefinition.

The array element assignment operation is extended for static arrays as follows: if $\$sar$, $\$n$ and $\$v$ are the values of $\$c1$, $\$c2$ and $\$c3$, and $1 \le \$n \le [len\ [\$sar\ .\ \{content\}]]$, then $(\$c1\ [\$c2] \coloneqq \$c3)$ replaces the value of the $\$n$-th element of $\$sar$ by $\$v$.

7.9. MPL9: structures

The MPL9 language is an extension of MPL8 that adds the structures feature: the structure types, the structure field access operation, structure declarations, and the structure field assignment statement.

7.9.1. Types, values, states

The attribute $((structure\ type)\ \$na)$ specifies structure types in states. A name $\$na$ is a structure type in $[\![\$s]\!]$ if $[\$s\ .\ \{((structure\ type)\ \$na)\}] \neq und$. Let $\$\$st$ be a set of structure types.

For MPL9, $\$\$t[\![MPL9]\!] = \$\$t[\![MPL8]\!] \cup \$\$st$.

The attribute $(field\ \$na\ \$st)$ specifies the fields of the structure type $\$st$. A name $\$fi$ is a field in $[\![\$st, \$s]\!]$ if $[\$s\ .\ \{(field\ \$fi\ \$st)\}] \neq und$. Let $\$\$fi$ be a set of fields.

The attribute $(type\ \$fi\ \$st)$ specifies the type of the field $\$fi$ of the structure type $\$st$. A type $\$t$ is a type in $[\![\$fi,\ \$st,\ \$s]\!]$ if $[\$s\ .\ \{(type\ \$fi\ \$st)\}] = \$t$.

Axiom: If $\$fi$ is a field in $[\![\$st, \$s]\!]$, then $[\$s\ .\ \{(type\ \$st\ \$fi)\}] \in \$\$t[\![\$s]\!]$.

The absolute type $structure$ specifies structures. An element $\$str$ is a structure in $[\![\$s]\!]$ if $\big((\$v1{:}\{\$fi1\}\ ...\ \$v\$n{:}\{\$fi\$n\}){:}\{content\},\ \$st{:}\{type\}\big){::}\{structure\} = \$str$, $\$n > 0$, the structure type $\$st$ has the fields $\$fi1$, …, $\$fi\$n$ and no other fields in $[\![\$s]\!]$, and the values $\$v1$, …, $\$v\$n$ in $[\![\$s]\!]$ have the types of the fields $\$fi1$, …, $\$fi\$n$ in $[\![\$st, \$s]\!]$. The elements $[\$str\ .\ \{content\}]$ and $\$st$ are called the content and the type in $[\![\$str]\!]$. The elements $\$fi1$, …, $\$fi\$n$ are called the fields in $[\![\$str]\!]$. The elements $\$v1$, …, $\$v\$n$ are called the values of these fields in $[\![\$st]\!]$. Let $\$\$str$ be a set of structures.

For MPL9, $\$\$v[\![MPL9]\!] = \$\$v[\![MPL8]\!] \cup \$\$str$, and $\$\$ex[\![MPL9]\!] = \$\$ex[\![MPL8]\!]$.

7.9.2. Constructs

The structure declaration is represented by the element $(structure\ \$na\ (\$tna1\ \ ...\ \ \$tna\$n))$. It specifies the declaration of the structure type with the name $\$na$, and the fields $\$na[\![\$tna1]\!]$, ..., $\$na[\![\$tna\$n]\!]$ of the types $\$t[\![\$tna1]\!]$, ..., $\$t[\![\$tna\$n]\!]$.

Axiom: structure declarations are elements of the program body.

The structure field access operation is represented by the element ($\$c \setminus .\ \$fi$). If $\$str$ is the value of $\$c$, then it returns the value in $[\![\$fi, \$str,\ \$s]\!]$.

The structure field assignment operation is represented by the element ($\$c1 \setminus .\ \$fi \coloneqq \$c2$). If $\$str$ and $\$v$ are the values of $\$c1$ and $\$c2$, then it assigns $\$v$ to the field $\$fi$ of $\$str$.

8. Conclusion

In the paper the formalism of the conceptual model of a programming language has been proposed. It represents types of the programming language, values (in particular, the values of the types the programming language), exceptions (the special kind of values), states and executable constructs (in particular, the elements of programs in the programming language) of the abstract machine of the language, and the constraints (axioms) for these entities at the conceptual level. The new definition of conceptual transition systems oriented to specification of conceptual models of programming languages has been proposed, the language CTSL for redefined conceptual transition systems has been described, and the technique of the use of CTSL as a domain-specific language for specification of conceptual models of programming languages has been presented. We have conducted the incremental development of the conceptual models for the family of sample programming languages to illustrate this technique.

We plan to use the CTSL language as a domain specific language oriented to the development of the conceptual operational semantics of programming languages defined as the operational semantics of representations of executable constructs of the abstract machines of the programming languages in CTSL.

References

1. Prinz A., Møller-Pedersen B., Fischer J. Object-Oriented Operational Semantics. In: Grabowski J., Herbold S. (eds) System Analysis and Modeling. Technology-Specific Aspects of Models. SAM 2016. Lecture Notes in Computer Science, vol 9959. Springer, Cham. P. 132-147.
2. Wider A. Model transformation languages for domain-specific workbenches // Ph.D. thesis, Humboldt-Universitat zu Berlin. 2015.
3. Felleisen M., Findler R.B., Flatt M. // Semantics Engineering with PLT Redex, 1st edn. The MIT Press, Cambridge. 2009.
4. Kahn G. Natural semantics. In: Brandenburg F.J., Vidal-Naquet G., Wirsing M. (eds.) STACS 1987. LNCS. 1987. Vol. 247, P. 22–39.
5. Mosses P.D. Structural operational semantics modular structural operational semantics // J. Logic Algebr. Program. 2004. Vol. 60. P. 195–228.

6. Plotkin G.D.: A structural approach to operational semantics // Technical report. DAIMI FN-19, AARHUS UNIVERSITY (DK). 1981.
7. OMG Editor. OMG Meta Object Facility (MOF) Core Specification Version 2.4.2. // Technical report, Object Management Group. 2014.
8. 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.
9. Rosu G., Serbanuta T.F. An overview of the K semantic framework // J. Logic Algebr. Program. 2010. 79(6). P. 397–434.
10. Clavel M., Duran F., Eker S., Lincoln P., Marti-Oliet N., Meseguer J., Quesada J.F. Rewriting logic and its applications maude: specification and programming in rewriting logic // Theor. Comput. Sci. 2002. 285(2). P. 187–243.
11. Anureev I.S. Conceptual Transition Systems // System Informatics. 2015. Vol. 5. P. 1–41.
12. Anureev I.S. Kinds and language of conceptual transition systems // System Informatics. 2015. Vol. 5. P. 55–74.

УДК 004.8

Operational conceptual transition systems and their application to development of conceptual operational semantics of programming languages*

Anureev I.S. (Institute of Informatics Systems)

In the paper the notion of the conceptual operational semantics of a programming language is proposed. This formalism represents operational semantics of a programming language in terms of its conceptual model based on conceptual transition systems. The special kind of conceptual transition systems, operational conceptual transition systems, oriented to specification of conceptual operational semantics of programming languages is defined, the extension of the language of conceptual transition systems CTSL for operational conceptual transition systems is described, and the technique of the use of the extended CTSL as a domain-specific language of specification of conceptual operational semantics of programming languages is proposed. The conceptual operational semantics for the family of sample programming languages illustrate this technique.

Ключевые слова: *operational semantics, conceptual transition system, programming language, conceptual model, domain-specific language, conceptual operational semantics*

1. Introduction

This paper relates to the development of operational semantics of programming languages. Following [1], we distinguish two parts of the operational semantics of a programming language. The structural part defines how the elements of the language relate to runtime elements that an abstract machine of the programming language can use at runtime. The structural part is called instantiation semantics or structure-only semantics [2]. The dynamic part describes the actual state changes that take place at runtime.

The notion of the conceptual model of a programming language is proposed in [3]. This formalism describes the instantiation semantics at the conceptual level. The conceptual model is specified in terms of conceptual transition systems (CTSs) in the language of conceptual transition systems CTSL [3].

---

* Partially supported by RFBR under grants 15-01-05974 and 17-07-01600 and SB RAS interdisciplinary integration project No.15/10.

In this paper, we introduce the notion of the conceptual operational semantics of a programming language. This formalism describes the operational semantics of a programming language in terms of its conceptual model. The dynamic part of the operational semantics is defined in terms of the special kind of CTS, operational CTSs, in the extension of CTSL for operational CTSs. Thus, CTSL acts as a domain-specific language oriented to specification of conceptual operational semantics of programming languages.

The paper has the following structure. The preliminary concepts and notation are given in section 2. The concepts and definitions related to the pattern matching which operational CTSs are based on is given in section 3. The operational CTSs is defined in section 4. The extension of the CTSL language for operational CTSs is described in section 5. Semantics of basic executable elements in CTSL is defined in section 6. The definition of the conceptual operational semantics of a programming language is introduced, and the technique of development of conceptual operational semantics of programming languages is illustrated by the sample programming language examples in section 7.

2. Preliminaries

The preliminary concepts and notation are given in this section.

2.1. Sets and sequences

Let $\$w$, $\$w1$, $\$w2$, … denote elements of the sort w, where w is a word, and $\$\$w$ denote the set of all elements of the sort w. For example, if n is a sort of natural numbers, then $\$n$, $\$n1$, … are natural numbers, and $$n is the set of all natural numbers.

Let $\$\$o$ and $\$\$set$ be sets of objects and sets considered in this paper. Let $\$\$i$, $\$\$n$, and $\$\$bo$ be sets of integers, natural numbers (with zero), and boolean values $true$ and $false$.

Let $\$\$se$ denote the set of finite sequences of the form $\$o1$ ... $\$o\$n$. Let $\$\$w^*$ denote the set of finite sequences of the form $\$w1$ … $\$w\$n$, and $\$w^*$, $\$w^*1$, $\$w^*2$, and so on denote the elements of the set $\$\$w^*$. Let $[es]$ denote the empty sequence. Let $\$\$w^+$ denote the set of finite nonempty sequences of the form $\$w1$ … $\$w\$n$, and $\$w^+$, $\$w^+1$, $\$w^+2$, and so on denote the elements of the set $\$\$w^+$.

Let $[repeat\ \$o\ \$n]$ denote the sequence consisting of $\$n$-th occurrences of the object $\$o$.

Let $[\$o \in \$se]$ and $[\$se1 \sqsubseteq \$se2]$ denote $\$o \in \{\$se\}$ and $\{\$se1\} \sqsubseteq \{\$se2\}$. Let $[len\ \$se]$ denote the length of $se. Let und denote the undefined value. Let $[\$se\ ..\ \$n]$ denote the $\$n$-th element of $\$se$. If $[len\ \$se] < \$n$, then $[\$se\ ..\ \$n] = und$. Let $[\$se\ ..\ \$n := \ \$o]$ denote the result $\$se1$ of

replacement of $\$n$-th element in $\$se$ by $\$o$. If $\$n > [len\, \$se]$, then $\$se1 = \$se\ [repeat\ und\ [[len\, \$se] - \$n - 1]]\ \$o$.

Let $[\$o \in \$se]$ and $[\$se1 \sqsubseteq \$se2]$ denote $\$o \in \{\$se\}$ and $\{\$se1\} \sqsubseteq \{\$se2\}$. Let $[len\, \$se]$ denote the length of $se. Let und denote the undefined value. Let $[\$se\ ..\ \$n]$ denote the $\$n$-th element of $\$se$. If $[len\, \$se] < \$n$, then $[\$se\ ..\ \$n] = und$. Let $[\$se\ ..\ \$n := \ \$o]$ denote the result $\$se1$ of replacement of $\$n$-th element in $\$se$ by $\$o$. If $\$n = [len\, \$se] + 1$, then $\$se1 = \$se\ \$o$. If $\$n > [len\, \$se] + 1$, then $\$se1 = und$.

Let $[\$o1 \prec_{[\![\$se]\!]} \$o2]$ denote the fact that there exist $\$o^*1$, $\$o^*2$ and $\$o^*3$ such that $\$se = \$o^*1\ \$o1\ \$o^*2\ \$o2\ \$o^*3$.

Let $[\$o\ \$o1\ \hookleftarrow\ \$o2]$ denote the result of replacement of all occurrences of $\$o1$ in $\$o$ by $\$o2$. Let $[\$se\ \$o\ \hookleftarrow\!*\ \$o1]$ denote the result of replacement of each element $\$o2$ in $\$se$ by $[\$o1\ \$o\ \hookleftarrow\ \$o2]$. For example, $[a\ b\ x \hookleftarrow\!*\ (f\ x)]$ denotes $(f\ a)\ (f\ b)$.

Let $\$o1,\ \$o2 \in \$\$se \cup \$\$set$. Then $[\$o1 =_{set} \$o2]$ denote that the sets of elements of $\$o1$ and $\$o2$ coincide, and $[\$o1 =_{mul} \$o2]$ denote that the multisets of elements of $\$o1$ and $\$o2$ coincide.

The above defined operations on the set $\$\$se$ are also applied to the set $\{(\$se) \mid \$se \in \$\$se\}$. The results of $[(\$se)\ ..\ \$n]$, $[\$o \in (\$se)]$, $[(\$se1) \sqsubseteq (\$se2)]$, $[\$o1 \prec_{[\![(\$se)]\!]} \$o2]$, $[(\$se)\ \$o\ \hookleftarrow^*\ \$o1]$, $[len\ (\$se)]$, $[(\$se)\ ..\ \$n \coloneqq \$o]$ and $[and\ (\$se)]$ are $[\$se\ ..\ \$n]$, $[\$o \in \$se]$, $[\$se1 \sqsubseteq \$se2]$, $[\$o1 \prec_{[\![\$se]\!]} \$o2]$, $[\$se\ \$o\ \hookleftarrow^*\ \$o1]$, $[len\ \$se]$, $[\$se\ ..\ \$n \coloneqq \$o]$ and $[and\ \$se]$.

Let $[(o^*) + (\$o^*1)]$, $[\$o\ .+(o^*)]$ and $[(o^*) +.\ \$o]$ denote $(\$o^*\ \$o^*1)$, $(\$o\ \$o^*)$ and $(\$o^*\ \$o)$.

2.2. Contexts

The terms used in the paper can be context-dependent. A context has the form $[\![\$o^*]\!]$. The elements of $\$o^*$ are called embedded contexts. The context in which some embedded contexts are omitted is called a partial context. All omitted embedded contexts are considered bound by the existential quantifier, unless otherwise specified.

Let o[\![\$o^*]\!]$ denote the object $o in the context $[\![\$o^*]\!]$. The expression 'in $[\![\$o1,\ \$o^*]\!]$' can be rewritten as 'in $[\![\$o1]\!]$ in $[\![\$o^*]\!]$', if this does not lead to ambiguity.

2.3. Functions

Let $\$\$f$ be a set of functions. Let $\$\$a$ and $\$\$v$ be sets of objects called arguments and values. Let $[\$f\ a^*]$ denote the result of application of $\$f$ to $\$a^*$. Let $[support\ \$f]$ denote the support in $[\![\$f]\!]$, i. e. $[support\ \$f] = \{\$a \mid [\$f\ \$a] \neq und\}$. Let $[image\ \$f\ \$set]$ denote the image in $[\![\$f,\ \$set]\!]$, i. e. $[image\ \$f\ \$set] = \{[\$f\ \$a] : \$a \in \$set\}$. Let $[image\ \$f]$ denote the image in $[\![\$f,\ [support\ \$f]]\!]$.

Let $[narrow\ \$f\ \$set]$ denote the function $\$f1$ such that $[support\ \$f1] = [support\ \$f] \cap \$set$, and $[\$f1\ \$a] = [\$f\ \$a]$ for each $\$a \in [support\ \$f1]$. The function $\$f1$ is called a narrowing of $\$f$ to $\$set$. Let $[support\ \$f1] \cap [support\ \$f2] = \emptyset$. Let $\$f1 \cup \$f2$ denote the union $\$f$ of $\$f1$ and $\$f2$ such that $[\$f\ \$a] = [\$f1\ \$a]$ for each $\$a \in [support\ \$f1]$, and $[\$f\ \$a] = [\$f2\ \$a]$ for each $\$a \in [support\ \$f2]$. Let $\$f1 \subseteq \$f2$ denote the fact that $[support\ \$f1] \subseteq [support\ \$f2]$, and $[\$f1\ \$a] = [\$f2\ \$a]$ for each $\$a \in [support\ \$f1]$.

An object $\$u$ of the form $\$a := \$v$ is called an update. The objects $\$a$ and $\$v$ are called an argument and values in $[\![\$u]\!]$. Let $\$\$u$ be a set of updates.

Let $[\$f\ \$u]$ denote the function $\$f1$ such that $[\$f1\ \$a] = [\$f\ \$a]$ if $\$a \neq \$a[\![\$u]\!]$, and $[\$f1\ \$a[\![\$u]\!]] = \$v[\![\$u]\!]$. Let $[\$f\ \$u\ \$u^*]$ be a shortcut for $[[\$f\ \$u]\ \$u^*]$. Let $[\$f\ \$a.\$a1.\ \ldots\ .\$a\$n := \$v]$ be a shortcut for $[\$f\ \$a := [[\$f\ \$a]\ \$a1.\ \ldots\ .\$a\$n := \$v]]$. Let $[\$u^*]$ be a shortcut for $[\$f\ \$u^*]$, where $[support\ \$f] = \emptyset$.

Let $[if\ \$con\ then\ \$o1\ else\ \$o2]$ denote the object $\$o$ such that $\$o = \$o1$ for $\$con = true$, and $\$o = \$o2$ for $\$con = false$.

3. Pattern matching

General CTSs only defines the state structure. The special kinds of CTSs also refine the structure of the transition relation. The refinement of operational CTSs is based on the pattern matching on the state structure.

A function $\$su \in \$\$cs \to \$\$cs^*$ is called a substituton. Let $\$\$su$ be a set of substitutions. A function $sub \in \$\$su \times \$\$cs^* \to \$\$cs^*$ is a substitution function if it is defined by the following rules (the first proper rule is applied):

- if $\$cs \in [support\ \$su]$, then $[sub\ \$su\ \$cs] = [\$su\ \$cs]$;
- $[sub\ \$su\ \$ato] = \$ato$;
- $[sub\ \$su\ \$cs{:}\{\$t^+\}] = [sub\ \$su\ \$cs]{:}\{[sub\ \$su\ \$t^+]\}$;
- $[sub\ \$su\ \$cs{::}\{\$t^+\}] = [sub\ \$su\ \$cs]{::}\{[sub\ \$su\ \$t^+]\}$;
- $[sub\ \$su\ (\$cs^*)] = ([sub\ \$su\ \$cs^*])$;
- $[sub\ \$su\ \$cs^*] = [\$cs^*\ \$e\ \hookleftarrow^*\ [sub\ \$su\ \$e]]$.

A structure $\$p$ is a pattern in $[\![\$cs,\ \$su]\!]$ if $[sub\ \$su\ \$p] = \$cs$. A structure $\$p$ is a pattern in $[\![\$cs]\!]$ if $\$p$ is a pattern in $[\![\$cs,\ \$su]\!]$ for some $\$su$. Let $\$\$p$ be a set of patterns. A structure $\$in$ is an instance in $[\![\$p,\ \$su]\!]$ if $[sub\ \$su\ \$p] = \$in$. A structure $\$in$ is an instance in $[\![\$p]\!]$ if $\$in$ is an instance in $[\![\$p,\ \$su]\!]$ for some $\$su$. Let $\$\$in$ be a set of instances.

A structure $\$cs1$ is weakly equal to a structure $\$cs2$ ($\$cs1 =_w \$cs2$) if the following properties hold:

- if $\$cs1 \in \$\$ato$, then $\$cs2 \in \$\$ato$, and $\$cs1 = \$cs2$;
- if $\$cs1 = v1::\{t^+1\}$, then $\$cs2 = v2::\{t^+2\}$, $v1 =_w v2$, and $\{t^+1\} =_w \{t^+2\}$ for some $\$va2$ and $\$t^+2$;
- if $\$cs1 = v1:\{t^+1\}$, then $\$cs2 = v2::\{t^+2\}$, $v1 =_w v2$, and $\{t^+1\} =_w \{t^+2\}$ for some $\$va2$ and $\$t^+2$;
- if $\$cs1 \in \$\$ccs$, then $\$cs2 \in \$\$ccs$, $[len\, \$cs1] = [len\, \$cs2]$, and $[\$cs1 \,..\, \$n] =_w \$cs2 \,..\, \$n]$ for each $1 \leq \$n \leq [len\, \$cs1]$.

A structure set $\$set1$ is weakly equal to a structure set $\$set2$ ($\$set1 =_w \$set2$) if the following properties hold:

- if $\$set1 = \emptyset$, then $\$set2 = \emptyset$;
- if $\$cs1 \in \$set1$, then there exists $\$cs2 \in \$set2$ such that $\$cs1 =_w \$cs2$, and $\$set1 \setminus \{\$cs1\} =_w \$set2 \setminus \{\$cs2\}$.

An element $\$e$ is linear in $[\![\$e^*]\!]$ if $\$e1$ occurs in $\$e$ exactly once for each element $\$e1$ of $\$e^*$. An element $\$ps$ of the form ($\$p$ ($\$va^*$) ($\$sv^*$)) is a pattern specification if the elements of the sequence $\$va^*\, \$sv^*$ are pairwise distinct, and $\$p$ is linear in $[\![\$va^*\, \$sv^*]\!]$. The elements $\$p$, ($\$va^*$), and ($\$sv^*$) are called a pattern, state variable specification and sequence variable specification in $[\![\$ps]\!]$. The elements of $\$va^*$ and $\$sv^*$ are called state variables and sequence variables in $[\![\$ps]\!]$. Let $\$\$ps$ be a set of pattern specifications. Let $\$\$va$ and $\$\$sv$ be sets of state and sequence variables.

A structure $\$in$ is an instance in $[\![\$ps, \$su]\!]$ if $[support\, \$su] = \{\$va^*\} \cup \{\$sv^*\}$, $[\$su\, \$va] \in \$\$s$ for $\$va \in \{\$va^*\}$, $[\$su\, \$sv] \in \$\$s^*$ for $\$sv \in \{\$sv^*\}$, and $\$in = w\, [sub\, \$p[\![\$ps]\!]\, \$su]$. A structure $\$in$ is an instance in $[\![\$ps]\!]$ if $\$in$ is an instance in $[\![\$ps,\ \$su]\!]$ for some $\$su$.

A function $\$mt \in \$\$s \times \$\$ps \to \$\$su$ is a matching tactic if $[\$mt\, \$s\, \$ps] = \$su$ implies that $\$s$ is an instance in $[\![\$ps,\ \$su]\!]$. A structure $\$in$ is an instance in $[\![\$ps,\ \$mt,\ \$su]\!]$ if $[\$mt\, \$in\, \$ps] = \$su$. A structure $\$in$ is an instance in $[\![\$ps,\ \$mt]\!]$ if $\$in$ is an instance in $[\![\$ps,\ \$mt,\ \$su]\!]$ for some $\$su$.

A substitution $\$su$ is a matching result in $[\![\$ps,\ \$mt,\ \$in]\!]$ if $\$in$ is an instance in $[\![\$ps,\ \$mt,\ \$su]\!]$. A substitution $\$su$ is a matching result in $[\![\$ps,\ \$mt]\!]$ if $\$su$ is a matching result in $[\![\$ps,\ \$mt,\ \$in]\!]$ for some $\$in$. A value $\$v$ is a matching result in $[\![\$va, \$ps,\ \$mt,\ \$su, \$in]\!]$ if $\$in$ is an instance in $[\![\$ps,\ \$mt,\ \$su]\!]$, and $\$v = [\$su\, \$va]$. A value $\$v$ is a matching result in $[\![\$va, \$ps,\ \$mt,\ \$in]\!]$ if $\$v$ is a matching result in $[\![\$va, \$ps,\ \$mt,\ \$su, \$in]\!]$ for some $\$su$. Let $\$\$mr$ be a set of matching results.

4. Operational conceptual transition systems

Operational CTSs (OCTSs) are the special kind of CTSs used to describe operational semantics of program systems and programming languages. Let $\$\$octs$ be a set of OCTSs.

The structure of $\hookrightarrow_{\llbracket \$octs \rrbracket}$ is based on program transition relations, program transition rules and atomic transition relations. Program transition relations and program transition rules include a specification of a pattern and their application is based on the matching of the first element of a program with the pattern. Atomic transition relations are applied in the case of the empty program.

An object $\$ptr$ of the form $(\$ps, \$f)$ is a program transition relation in $\llbracket \$mt \rrbracket$ if $\$f \in \{\$su \cup (\{cstate\}{:}\, \$s,\ \{cvalue\}{:}\, \$v\llbracket \$s \rrbracket) \mid \$su \in \$\$mr\llbracket \$ps,\ \$mt \rrbracket$, and $\$s \in \$\$s\} \to \$\$tr$. Thus, $\$ptr$ specifies a parametric transition relation, where the values of the parameter are the results of the pattern matching. The special elements $\{cstate\}$ and $\{cvalue\}$ refer to the current state and the value in the current state. Let $\$\$ptr$ be a set of program transition relation. The objects $\$p\llbracket \$ps \rrbracket$, $(\$va^{*}\llbracket \$ps \rrbracket)$, $(\$sv^{*}\llbracket \$ps \rrbracket)$ and $\$f$ are called a pattern, state variable specification, sequence variable specification and value in $\llbracket \$ptr \rrbracket$. The elements of $\$va^{*}$ and $\$sv^{*}$ are called state and sequence variables in $\llbracket \$ptr \rrbracket$. If $\$su$ is the result of matching the first element $\$e$ of the program $\$p\llbracket \$s \rrbracket$ with the pattern of $\$ptr$, then a transition from $\$s$ to $\$s1$ initiated by $\$ptr$ and denoted by $\$s \hookrightarrow_{\llbracket \$ptr,\, \$su \rrbracket} \$s1$ is defined as $(\$s, \$s1) \in [\$f\ \$su \cup (\{cstate\}{:}\, \$s,\ \{cvalue\}{:}\, \$v\llbracket \$s \rrbracket)]$. Let $\$s \hookrightarrow_{\llbracket \$f\llbracket \$ptr \rrbracket \rrbracket} \$s1$ denote $\$s \hookrightarrow_{\llbracket \$ptr,\, \$su \rrbracket} \$s1$ for some $\$su$.

A partial function $\$ptrs \in \$e \to \$\$ptr$ is a program transition relation specification if $[support\ \$ptrs]$ is finite. A relation $\$ptr$ is a relation in $\llbracket \$ptrs \rrbracket$ if $[\$ptrs\ \$na] = \$ptr$ for some $\$na \in \$\$e$. An element $\$na$ is a name in $\llbracket \$ptr,\ \$ptrs \rrbracket$ if $[\$ptrs\ \$na] = \$atr$. An element $\$na$ is a name in $\llbracket \$ptrs \rrbracket$ if $\$na$ is a name in $\llbracket \$ptr,\ \$ptrs \rrbracket$ for some $\$atr$. Thus, $\$ptrs$ defines a finite set of named program transition relations.

An element $\$r$ of the form $(\$ps\ \$b)$ is called a (program) transition rule. The objects $\$p\llbracket \$ps \rrbracket$, $(\$va^{*}\llbracket \$ps \rrbracket)$, $(\$sv^{*}\llbracket \$ps \rrbracket)$ and $\$b$ are called a pattern, state variable specification, sequence variable specification and body in $\llbracket \$r \rrbracket$. The elements of $\$va^{*}$ and $\$sv^{*}$ are called state and sequence variables in $\llbracket \$r \rrbracket$. Let $\$\$r$ be a set of transition rules. If $\$su$ is the result of matching the first element $\$e$ of the program $\$p\llbracket \$s \rrbracket$ with the pattern of $\$r$, then a transition from $\$s$ initiated by $\$r$ replaces $\$e$ in $\$p$ by $[sub\ \$su \cup (\{cstate\}{:}\, \$s,\ \{cvalue\}{:}\, \$v\llbracket \$s \rrbracket)\ \$b]$.

A structure $\$rs$ is a rule specification if $[\$rs\ .\ \{\$na\}] \in \$\$r \cup \{und\}$ for each $\$na \in \$\$e$. A rule $\$r$ is a rule in $\llbracket \$rs \rrbracket$ if $[\$rs\ .\ \{\$na\}] = \$r$ for some $\$na \in \$\$e$. An element $\$na$ is a name in

$[\![\$r,\ \$rs]\!]$ if $[\$rs\,.\,\{\$na\}]\ =\ \$r$. An element $\$na$ is a name in $[\![\$rs]\!]$ if $\$na$ is a name in $[\![\$r,\ \$rs]\!]$ for some $\$r$. Thus, $\$rs$ defines a finite set of named transition rules.

A structure $\$rs$ is a rule specification in $[\![\$s]\!]$ if $\$rs = [\$s\,.\,\{rules\}]$, and $[\$rs\,.\,\{\$na\}] \in \$\$r \cup \{und\}$ for each $\$na \in \$\$e$. It specifies the set of named transition rules in $[\![\$s]\!]$.

Let $[support\ \$ptrs] \cap [support\ \$rs] = \emptyset$.

A structure $\$pto$ of the form $(\$na^*)$ is a program transition order in $[\![\$ptrs,\ \$rs]\!]$ if $\{\$na^*\} \subseteq [support\ \$ptrs] \cup [support\ \$rs]$, and the elements of $\$na^*$ are pairwise distinct. It specifies the order of application of program transition relations and transition rules, i. e. the order of matching the first element of the program with their patterns.

A structure $(\$na^*)$ is a program transition order in $[\![\$s]\!]$ if $(\$na^*) = [\$s\,.\,\{(program\ transition\ order)\}]$, and the elements of $\$na^*$ are pairwise distinct. It specifies the order of application of program transition relations and transition rules in $[\![\$s]\!]$.

A relation $\$atr \in \$\$tr$ is called an atomic transition relation. Let $\$\$atr$ be a set of atomic transition relations. If $\$p[\![\$s]\!] = ()$, then a transition from $\$s$ to $\$s1$ initiated by $\$atr$ and denoted by $\$s \hookrightarrow_{[\![\$atr]\!]} \$s1$ is defined as $(\$s,\ \$s1) \in \$atr$.

A partial function $\$atrs \in \$\$s \to \$\$atr$ is an atomic transition relation specification if $[support\ \$atrs]$ is finite. A relation $\$atr$ is a relation in $[\![\$atrs]\!]$ if $[\$atrs\ \$na]\ =\ \$atr$ for some $\$na \in \$\$e$. An element $\$na$ is a name in $[\![\$atr,\ \$atrs]\!]$ if $[\$atrs\ nm] = \$atr$. An element $\$na$ is a name in $[\![\$atrs]\!]$ if $\$na$ is a name in $[\![\$atr,\ \$atrs]\!]$ for some $\$atr$. Thus, $\$atrs$ defines a finite set of named atomic transiton relations.

A structure $\$ator$ of the form $(\$na^*)$ is an atomic transition order in $[\![\$atrs]\!]$ if $\{\$na^*\} \subseteq [support\ \$atrs]$, and the elements of $\$na^*$ are pairwise distinct. It specifies the order of application of atomic transition relations.

A structure $(\$na^*)$ is an atomic transition order in $[\![\$s]\!]$ if $(\$na^*) = [\$s\,.\,\{(atomic\ transition\ order)\}]$, and the elements of $\$na^*$ are pairwise distinct. It specifies the order of application of atomic transition relations in $[\![\$s]\!]$.

Let $\$\$bi$ be a set of elements called backtracking invariants.

Let $\$e * \#\,\$v\,\#\,\$s$ denote $[\$s\ program{:}\,(\$e^*)\ value{:}\,\$v]$. Let $\$e^*\,\#\,\$s$ denote $[\$s\ program{:}\,(\$e^*)]$.

A tuple $\$octs$ of the form $(\$ato,\ \$\$is, \$ptrs,\ \$rs,\ \$pto,\ \$atrs,\ \$ator,\ \$\$bi,\ \$mt)$ is an operational CTS if the system $\$cts$ of the form $(\$\$at,\ \$\$is, \hookrightarrow)$ is a CTS with backtracking in $[\![\$\$bi]\!]$ and with direct stop, $[support\ \$ptrs]\ \cap\ [support\ \$rs]\ =\ \emptyset$, $\$s$ is consistent for each $\$s \in \$\$rs[\![\hookrightarrow]\!]$, and $\hookrightarrow$ is defined by the following rules (the first proper rule is applied):

- if $\$ptr = [\$ptrs\ \$na]$, $\$e$ is an instance in $[\![\$ps[\![\$ptr]\!], \$mt, \$su]\!]$, and $\$s \hookrightarrow_{[\![\$ptr, \$su]\!]} \$s1$, then
  $((execute\ program\ element)\ \$e\ (\$na\ \$na^*))\ \$e^* \#\ \$s\ \hookrightarrow$
  $(bactracking\ \$s\ ((execute\ program\ element)\ \ \$e\ (\$na^*)))\ \$e^*\ \#\ \$s1;$
- if $\$ptr = [\$ptrs\ \$na]$, and $\$e$ is not an instance in $[\![\$ps[\![\$ptr]\!], \$mt]\!]$, then $((execute\ program\ element)\ \$e\ (\$na\ \$na^*))\ \$e^*\ \#\ \$s\ \hookrightarrow$
  $((execute\ program\ element)\ \$e\ (\$na^*))\ \$e^*\ \#\ \$s;$
- if $(\$r = [\$rs\ .\ \{\$na\}]$, or $\$r = [[\$s\ .\ \{rules\}]\ .\ \{\$na\}])$, and $\$e$ is an instance in $[\![\$ps[\![\$r]\!], \$mt, \$su]\!]$, then
  $((execute\ program\ element)\ \$e\ (\$na\ \$na^*))\ \$e^*\ \#\ \$s$
  $\hookrightarrow_{[sub\ \$su \cup (\{cstate\}:\$s, \{cvalue\}:\$v[\![\$s]\!])\ \$b[\![\$r]\!]]}$
  (backtracking $s ((execute program element) $e ($na^*))) $e^* # $s;
- if $(\$r = [\$rs\ .\ \{\$na\}]$, or $\$r = [[\$s\ .\ \{rules\}]\ .\ \{\$na\}])$, and $\$e$ is not an instance in $[\![\$ps[\![\$r]\!], \$mt]\!]$, then
  $((execute\ program\ element)\ \$e\ (\$na\ \$na^*))\ \$e^*\ \#\ \$s \hookrightarrow$
  $((execute\ program\ element)\ \$e\ (\$na^*))\ \$e^*\ \#\ \$s;$
- $((execute\ program\ element)\ \$e\ ())\ \$e^*\ \#\ \$s \hookrightarrow \$e^*\ \#\ und\ \#\ \$s;$
- if $\$atr = [\$atrs\ \$na]$, $\$s \hookrightarrow [\![\$atr]\!]\ \$s1$, and $\$p[\![\$s1]\!] \neq ()$, then $((execute\ atomic\ transition)\ (\$na\ \$na^*))\ \#\ \$s \hookrightarrow \$s1;$
- if $\$atr = [\$atrs\ \$na]$, $\$s \hookrightarrow [\![\$atr]\!]\ \#\ \$v\ \#\ \$s1$, and $\$p[\![\$s1]\!] = ()$, then $((execute\ atomic\ transition)\ (\$na\ \$na^*))\ \#\ \$s \hookrightarrow$
  $(backtracking\ \$s\ ((execute\ atomic\ transition)\ (\$na^*)))\ \#\ \$s1;$
- $((execute\ atomic\ transition)\ ())\ \#\ \$s \hookrightarrow [\$s\ true{:}\{stop\}];$
- $\$e\ \$e^*\ \#\ \$s\ \hookrightarrow$
  $((execute\ program\ element)\ \$e\ [\$s\ ..\ (program\ transition\ order)])\ \$e^*\ \#\ \$s;$
- $\#\ \$s\ \hookrightarrow ((execute\ atomic\ transition),\ [\$s\ ..\ (atomic\ transition\ order)])\ \#\ \$s.$

A state $\$s$ is consistent in $[\![\$cts]\!]$ if the following properties hold:

- the set of bactracking invariants in $[\![\$s]\!]$ is finite;
- $[support\ \$ptrs] \cap [support\ [\$s\ .\ \{rules\}]] = \emptyset;$
- $[support\ \$rs] \cap [support\ [\$s\ .\ \{rules\}]] = \emptyset;$
- if $\$na1 \prec_{[\![\$pto]\!]} \$na2$, $\$na1 \in [\$s\ .\ \{(program\ transition\ order)\}]$, and $\$na2 \in [\$s\ .\ \{(program\ transition\ order)\}]$, then $\$na1 \prec_{[\![[\$s\ .\ \{(program\ transition\ order)\}]]\!]} \$na2;$

if $\$na1 \prec_{[\![\$ator]\!]} \$na2$, $\$na1 \in [\$s\,.\,\{(atomic\ transition\ order)\}]$, and $\$na2 \in [\$s\,.\,\{(atomic\ transition\ order)\}]$, then $\$na1 \prec_{[\![[\$s\,.\,\{(atomic\ transition\ order)\}]]\!]} \$na2$.

5. The CTSL language

The CTSL language is extended for operational CTSs by adding transition rules and extended transition rules.

The transition rule $((\$p,\ (\$va^*),\ (\$sv^*)),\ \$b)$ with the name $\$na$ is represented in CTSL[o] by the state $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ then\ \$b)::\{\$na\}$.

Extended transition rules are transition rules enriched by the mechanisms of evaluation of pattern variable matching results, imposition of constraints on pattern variable matching results and their values and propagation of abnormal values (undefined values and exceptions) from pattern variable matching results and the attribute $value$.

Let $\{\$eva^*\} \subseteq \{\$va^*\}$, the elements of the sequence $\$eva^*$ are pairwise disjoint, $\$set = \{\$eva :: \{*\} \mid \$eva \in \$eva^*\}$, $\{\$va^*1\} \cup \{\$va^*2\} \cup \{\$va^*3\} \subseteq \{\$va^*\} \cup \$set$, the elements of the sequence $\$va^*1\ \$va^*2\ \$va^*3$ are pairwise disjoint, and $\$se \in \{[se],\ und,\ exc,\ abn\}$. The state

$(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se$
$where\ \$co\ then\ \$b)$

is called an extended rule. It is defined as follows:

- If $\$co \neq true$, then
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)\ exc\ (\$va^*2)\ abn\ (\$va^*3)\ \$se$
  $where\ \$co\ then\ \$b)$
  is a shortcut for
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)\ exc\ (\$va^*2)\ abn\ (\$va^*3)\ \$se$
  $where\ true\ then\ (if\ \$co\ then\ \$b\ else\ und))$.
- The rule
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)\ exc\ (\$va^*2)$
  $abn\ (\$va^*31\ \$eva::\{*\}\ \$va^*32)\ \$se\ where\ true\ then\ \$b)$
  is a shortcut for
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)\ exc\ (\$va^*2)$
  $abn\ (\$va^*31\ \$va^*32)\ \$se\ where\ true\ then\ (if\ (\$eva :: \{*\}\ is\ abnormal)$
  $then\ \$eva::\{*\}\ else\ \$b))$.
- If $\{\$va^*3\} \cap \$set = \emptyset$, then
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)$

$exc\ (\$va^{*}21\ \$eva{:}{:}\{*\}\ \$va^{*}22)\ abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ \$b)$

is a shortcut for

$(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*})\ und\ (\$va^{*}1)\ exc\ (\$va^{*}21\ \$va^{*}22)$

$abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ (if\ (\$eva{:}{:}\{*\}\ is\ exception)\ then\ \$eva{:}{:}\{*\}\ else\ \$b)).$

- If $(\{\$va^{*}2\} \cup \{\$va^{*}3\}) \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*})\ und\ (\$va^{*}11\ \$eva :: \{*\}\ \$va^{*}12)$

  $exc\ (\$va^{*}2)\ abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ \$b)$

  is a shortcut for

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*})\ und\ (\$va^{*}11\ \$va^{*}12)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ (if\ (\$eva{:}{:}\{*\}\ is\ undefined)\ then\ \$eva{:}{:}\{*\}\ else\ \$b)).$

- If $(\{\$va^{*}1\} \cup \{\$va^{*}2\} \cup \{\$va^{*}3\}) \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*}\ \$eva :: \{*\})\ und\ (\$va^{*}1)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ \$b)$

  is a shortcut for

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*})\ und\ (\$va^{*}1)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ (let\ w\ be\ \$eva\ in\ (subst\ (\$eva :: \{*\}{:}\ w)\ \$b))),$

  where w is a new state that does not occur in the initial form.

- If $(\{\$va^{*}1\} \cup \{\$va^{*}2\} \cup \{\$va^{*}31,\ \$va,\ \$va^{*}32\}) \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\$va^{*}1)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}31\ \$va\ \$va^{*}32)\ \$se\ where\ true\ then\ \$b)$

  is a shortcut for

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\$va^{*}1)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}31\ \$va^{*}32)\ \$se\ where\ true\ then\ (if\ (\$va\ is\ abnormal)\ then\ \$va\ else\ \$b)).$

- If $(\{\$va^{*}1\} \cup \{\$va^{*}21,\ \$va,\ \$va^{*}22\}) \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\$va^{*}1)\ exc\ (\$va^{*}21\ \$va\ \$va^{*}22)$

  $abn\ (\,)\ \$se\ where\ true\ then\ \$b)$

  is a shortcut for

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\$va^{*}1)\ exc\ (\$va^{*}21\ \$va^{*}22)\ abn\ (\,)\ \$se$
  $where\ true\ then\ (if\ (\$va\ is\ exception)\ then\ \$va\ else\ \$b)).$

- If $\{\$va^{*}11,\ \$va,\ \$va^{*}12\} \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\$va^{*}11\ \$va\ \$va^{*}12)\ exc\ (\,)\ abn\ (\,)\ \$se$
  $where\ true\ then\ \$b)$

  is a shortcut for

$(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\,)\ und\ (\$va^*11\ \$va^*12)\ exc\ (\,)\ abn\ (\,)\ \$se$
$where\ true\ then\ (if\ (\$va\ is\ undefined)\ then\ \$va\ else\ \$b))$.

- The rule
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ abn\ where\ true\ then\ \$b)$ is a shortcut for
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ where\ true$
  $then\ (if\ (cvalue\ is\ abnormal)\ then\ else\ \$b)$.
- The rule
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ exc\ where\ true\ then\ \$b)$ is a shortcut for
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ where\ true$
  $then\ (if\ (cvalue\ is\ exception)\ then\ else\ \$b)$.
- The rule
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ und\ where\ true\ then\ \$b)$ is a shortcut for
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ where\ true$
  $then\ (if\ (cvalue\ is\ undefined)\ then\ else\ \$b)$.

A pattern variable $\$va$ is evaluated if the matching result for \$va is evaluated. The sequence $\$eva^*$ contains evaluated pattern variables. The special variable $\$eva::\{*\}$ references to the value of the matching result for \$va. A pattern variable $\$va$ is quoted if the matching result for \$va is not evaluated.

The state $\$co$ imposes of constraints on the values of the variables $\$va^*$, $\$sv^*$, $\$eva::\{*\}^*$.

The undefined value und is propagated through the variables $\$v^*1$. Exceptions are propagated through the variables $\$v^*2$. Abnormal values are propagated through the variables $\$v^*3$.

The sequence $\$se$ specifies propagation of abnormal values through the attribute $value$. The undefined value is propagated through the attribute $value$ when $\$se = und$. Exceptions are propagated through the attribute $value$ when $\$se = exc$. Abnormal values are propagated through the attribute $value$ when $\$se = abn$.

The executable elements $(if\ \$con\ then\ \$e^*1\ else\ \$e^*2)$ and $(let\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ are defined in section 6.7. The executable elements $(\$e\ is\ abnormal)$, $(\$e\ is\ exception)$ and $(\$e\ is\ undefined)$ are defined in section 7.4.

Let $\$\$er$ be a set of extended transition rules.

The objects $var\ (\$va^*)$, $seq\ (\$sv^*)$, $val\ (\$eva^*)$, $abn\ (\$va^*1)$, $und\ (\$va^*2)$, $exc\ (\$va^*3)$ and $where\ \$co$ in extended transition rules can be omitted. The omitted objects correspond to $var\ (\ )$, $seq\ (\ )$, $val\ (\ )$, $abn\ (\ )$, $und\ (\ )$, $exc\ (\ )$ and $where\ true$.

6. Semantics of executable elements in CTSL

To define operational semantics of executable elements in CTSL the special denotations for program and atomic transition relations are introduced.

Let $(transition\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ then\ \$f)::\{\$na\}$ denote the program transition relation $((\$p,\ (\$va^*),\ (\$sv^*)),\ \$f)$ with the name $\$na$. The objects $var\ (\$va^*)$ and $seq\ (\$sv^*)$ can be omitted. The omitted objects correspond to $var\ ()$ and $seq\ ()$.

Let $(atomic\ transition\ \$f)::\{\$na\}$ denotes the atomic transition relation defined by the characteristic function $\$f \in \$s \times \$s \to \$b$ with the name $\$na$.

For simplicity, we omit the names of transition relations and transition rules.

6.1. Values

The executable elements handling the transition value are defined in this section.

An element $\$e$ of the form $\$v::\{q\}$ is called a quoted element. It is defined as follows:

$(rule\ v::\{q\}\ \ var\ (v)\ abn\ then\ v::\{q\}::\{transition\})$;

$(transition\ v::\{q\}::\{transition\}\ var\ (v)\ then\ \$f)$,

where $\$v::\{q\}::\{transition\}; \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ \$v\ \#\ \$s$.

The value v is called a quoted value in $[\![\$e]\!]$. The element $\$e$ returns the quoted value v.

The element und is defined by the rule

$(rule\ und\ abn\ then\ und::\{q\})$.

The element ex is defined by the rule

$(rule\ v::\{exc\}\ var\ (v)\ abn\ then\ v::\{exc\}::\{q\})$.

The element $(\$e\ is\ undefined)$ specifies that $\$e$ equals und. It is defined by the rule

$(rule\ (e\ is\ undefined)\ var\ (e)\ abn\ then\ (e::\{q\} = und))$.

The element $(\$e\ is\ defined)$ specifies that $\$e$ does not equal und. It is defined by the rule

$(rule\ (e\ is\ defined)\ var\ (e)\ abn\ then\ (e::\{q\}\ ! = und))$.

The element $(\$e\ is\ exception)$ specifies that $\$e$ is an exception. It is defined by the rule

$(rule\ (e\ is\ exception)\ var\ (e)\ abn\ then\ (e\ is\ exception)::\{transition\})$;

$(transition\ (e\ is\ exception)::\{transition\}\ var\ (e)\ then\ \$f)$,

where

$(e\ is\ exception) :: \{transition\}; \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [\$e \in \$\$exc]\ then\ true\ else\ und]\ \#\ \$s.$

The element $(\$e\ is\ abnormal)$ specifies that $\$e$ is abnormal. It is defined by the rule

$(rule\ (e\ is\ abnormal)\ var\ (e)\ abn\ then\ ((e\ is\ undefined)\ or\ (e\ is\ exception)));$

The element $(\$e\ is\ normal)$ specifying that $\$e$ is normal. It is defined by the rule

$(rule\ (e\ is\ normal)\ var\ (e)\ abn\ then\ ((e\ is\ defined)\ and\ (not\ (e\ is\ exception))));$

The element $\$e$ of the form $(catch::\{und\}\ \$va\ \$e^*)$ is called a value handler. It is defined as follows:

$(transition\ (catch::\{und\}\ va\ e_s)\ var\ (va)\ seq\ (e_s)\ then\ \$f),$

where $(catch::\{und\}\ \$va\ \$e^*); \$e^*1\ \#\ \$v\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} (sub\ (\$va{:}\ \$v)\ \$e^*)\ e^*1\ \#\ true\ \#\ \$s.$

The elements $\$va$ and $\$e^*1$ are called a variable and body in $[\![\$e]\!]$. The element $\$e$ replaces all occurences of the variable $\$va$ in the body $\$e^*1$ by the current value, resets the current value to $true$ and executes the modified body.

The element $\$e$ of the form $(catch\ \$va\ \$e^*)$ is called an exception handler. It is defined as follows:

$(rule\ (catch\ va\ e_s)\ var\ (va)\ seq\ (e_s)\ und\ then\ (catch::\{und\}\ va\ e_s)),$

The elements $\$va$ and $\$e^*$ are called a variable and body in $[\![\$e]\!]$. If the current value is defined, the element $\$e$ replaces all occurences of the variable $\$va$ in the body $\$e^*1$ by the current value, resets the current value to $true$ and executes the modified body. It propagates und.

The element $(current\ value)$ returns the current value. It is defined by the rule

$(rule\ (current\ value)\ abn\ then\ cvalue::\{q\}).$

The element $((to\ value)\ \$e)$ replaces the current value to $\$v$, where $\$v$ is the value of $\$e$. It is defined as follows:

$(rule\ ((to\ value)\ e)\ var\ (e)\ val\ (e)\ then\ ((to\ value)\ e::\{*\})::\{transition\});$

$(transition\ ((to\ value)\ v)::\{transition\}\ var\ (v)\ then\ \$f),$

where $((to\ value)\ \$v)::\{transition\};\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ \$v\ \#\ \$s.$

The element ((catch exception) t) catches an exception of the type $\$t$. It is defined by the rule

$(rule\ ((catch\ exception)\ t)\ var\ (t)\ und$

$then\ (catch\ va\ ($

$if\ ((va\ is\ exception)\ and\ ((va::\{q\}\ .\ \{type\})\ = t::\{q\}))$

$then\ ((to\ value)\ true)\ else\ ((to\ value)\ va::\{q\})))).$

6.2. Integers

The executable elements handling integers are defined in this section.

The element ($\$e\ is\ nat$) specifies that $\$e$ is a natural number. It is defined as follows:

$(rule\ (e\ is\ nat)\ var\ (e)\ abn\ then\ (e\ is\ nat)::\{transition\});$

$(transition\ (e\ is\ nat)::\{transition\}\ var\ (e)\ then\ \$f),$

where

$(\$e\ is\ nat) :: \{transition\}\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [if\ [\$e \in \$\$n]\ then\ true\ else\ und]\ \#\ \$s.$

The element ($\$e\ is\ int$) specifies that $\$e$ is an integer. It is defined as follows:

$(rule\ (e\ is\ int)\ var\ (e)\ abn\ then\ (e\ is\ int)::\{transition\});$

$(transition\ (e\ is\ int)::\{transition\}\ var\ (e)\ then\ \$f),$

where

$(\$e\ is\ int)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [if\ [\$e \in \$\$in]\ then\ true\ else\ und]\ \#\ \$s.$

The element $\$i$ is defined by the rules

$(rule\ i\ var\ (i)\ abn\ where\ (i\ is\ int)\ then\ i::\{q\}).$

If $\$v1$ and $\$v2$ are values of $\$e1$ and $\$e2$, then the element ($\$e1 + \$e2$) returns [$\$v1 + \$v2$]. It is defined as follows:

$(rule\ (e1 + e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$then\ (e1::\{*\}\ +::\{integer\}\ e2::\{*\})::\{transition\});$

$(transition\ (i1\ +::\{integer\}\ i2)::\{transition\}\ var\ (i1,\ i2)\ then\ f),$

where $(\$i1\ +::\{integer\}\ \$i2)::\{transition\};\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [\$i1 + \$i2]\ \#\ \$s.$

The elements ($\$e1\ \$op\ \$e2$), where $\$op \in \{-,\ *,\ div,\ mod\}$, specifying the integer operations $-$, $*$, div and mod, are defined in the similar way.

If $\$v1$ and $\$v2$ are values of $\$e1$ and $\$e2$, then the element ($\$e1 < \$e2$) specifies that [$\$v1 < \$v2$]. It is defined as follows:

$(rule\ (e1 < e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$then\ (e1::\{*\}\ <::\{integer\}\ e2::\{*\})::\{transition\});$

$(rule\ (e1::\{*\}\ <::\{integer\}\ e2::\{*\})::\{transition\}\ var\ (e1,\ e2)\ then\ f),$

where $(\$i1\ <::\{integer\}\ \$i2)::\{transition\};\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [\$i1 < \$i2]\ \#\ \$s.$

The elements ($\$e1\ \$op\ \$e2$), where $\$op \in \{<=,\ >,\ >=\}$, specifying the integer relations $\leq$, $>$ and $\geq$, are defined in the similar way.

6.3. Boolean values

The executable elements handling boolean values are defined in this section.

The element $true$ is defined by the rule:

$(rule\ true\ abn\ then\ true::\{q\})$.

If $\$v1$ and $\$v2$ are values of $\$e1$ and $\$e2$, then the element $(\$e1\ and\ \$e2)$ specifies the conjunction of $\$v1$ and $\$v2$. It is defined by the rule:

$(rule\ (e1\ and\ e2)\ var\ (e1,\ e2)\ abn\ then\ (if\ e1\ then\ e2\ else\ und))$.

If $\$v1$ and $\$v2$ are values of $\$e1$ and $\$e2$, then the elements $(\$e1\ \$op\ \$e2)$, where $\$op \in \{or,\ =>,\ <=>\}$ specifying the disjunction, implication and equivalence of $\$v1$ and $\$v2$ are defined in the similar way.

If $\$v1$, $\$v2$, ..., $\$v\$n$ are values of $\$e1$, $\$e2$, ..., $\$e\$n$, then the element $(\$e1\ and\ \$e2\ and \ldots\ and\ \$e\$n)$ specifies the conjunction of $\$v1$, $\$v2$, ..., $\$v\$n$. It is defined by the rule

$(rule\ (e1\ and\ e2\ and\ e_s)\ var\ (e1,\ e2)\ seq\ (e_s)\ abn\ then\ ((e1\ and\ e2)\ and\ e_s)$.

If $\$v1$, $\$v2$, ..., $\$v\$n$ are values of $\$e1$, $\$e2$, ..., $\$e\$n$, then the element $(\$e1\ or\ \$e2\ or\ \ldots\ or\ \$e\$n)$ specifying the disjunction of $\$v1$, $\$v2$, ..., $\$v\$n$ is defined in the similar way.

If $\$v$ is a value of $\$e$, then the element $(not\ \$e)$ specifies the negation of $\$v$. It is defined by the rule $(rule\ (not\ e)\ var\ (e)\ abn\ then\ (if\ e\ then\ und\ else\ true))$.

6.4. Conceptual structures

The executable elements handling conceptual structures are defined in this section.

The element $(\$e\ is\ atom)$ specifies that $\$e$ is an atom. It is defined as follows:

$(rule\ (e\ is\ atom)\ var\ (e)\ abn\ then\ (e\ is\ atom)::\{transition\})$;

$(transition\ (e\ is\ atom)::\{transition\}\ var\ (e)\ then\ \$f)$,

where

$(e\ is\ atom)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [if\ [e \in \$\$ato]\ then\ true\ else\ und]\ \#\ \$s$.

The element $(\$e\ is\ compound)$ specifies that $\$e$ is a compound structure. It is defined by the rule

$(rule\ ((e_s)\ is\ compound)\ seq\ (e_s)\ abn\ then\ true)$.

The element $(\$e\ is\ (absolutely\ typed))$ specifies that $\$e$ is an absolutely typed structure. It is defined by the rule

$(rule\ (e::\{t_s\}\ is\ (absolutely\ typed))\ var\ (e)\ seq\ (e_s)\ abn\ then\ true)$.

The element $(\$e\ is\ (relatively\ typed))$ specifies that $\$e$ is a relatively typed structure. It is defined by the rule

$(rule\ (e:\{t_s\}\ is\ (relatively\ typed))\ var\ (e)\ seq\ (e_s)\ abn\ then\ true)$.

The element $(\$e\ is\ empty)$ specifies that $\$e$ is an empty structure. It is defined by the rule

$(rule\ (()\ is\ empty)\ abn\ then\ true)$.

The element $(\$e\ is\ nonempty)$ specifies that $\$e$ is not an empty structure. It is defined by the rule

$(rule\ (e\ is\ empty)\ var(e)\ abn\ then\ (not\ (e\ is\ empty)))$.

The empty structure is defined by the rule

$(rule\ ()\ abn\ then\ ()::\{q\})$.

The element $(\$ccs\ is\ (\$t\ *))$ specifies that the value of $(\$e\ is\ \$t)$ does not equal und for each element $\$e$ of $\$ccs$. It is defined by the rule

$(rule\ ((e\ e_s)\ is\ (t\ *))\ var\ (e,\ t)\ seq\ (e_s)\ abn\ then\ ((e\ is\ t)\ and\ ((e_s)\ is\ (t\ *)))$;

$(rule\ (()\ is\ (t\ *))\ var\ (t)\ abn\ then\ true)$.

If $\$cs$ is a value of $\$e$, then the element $(len\ \$e)$ specifies the length of $\$cs$. It is defined as follows:

$(rule\ (len\ e)\ var\ (e)\ val\ (e)\ abn\ then\ (len\ e :: \{*\}) :: \{transition\})$;

$(transition\ (len\ cs) :: \{transition\}\ var\ (cs)\ then\ \$f)$,

where $(len\ \$cs) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [len\ \$cs] \# \$s$.

If $\$cs1$ and $\$cs2$ are values of $\$e1$ and $\$e2$, then the element $(\$e1\ = \$e2)$ specifies the equality of $\$cs1$ and $\$cs2$. It is defined as follows:

$(rule\ (e1 = e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (e1 :: \{*\} = e2 :: \{q\}) :: \{transition\})$;

$(transition\ (cs1\ = cs2) :: \{transition\}\ var\ (cs1,\ cs2)\ then\ \$f)$,

where $(\$cs1\ = \$cs2) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$cs1 = \$cs2] \# \$s$.

If $\$cs1$ and $\$cs2$ are values of $\$e1$ and $\$e2$, then the element $(\$e1\ != \$22)$ specifies the inequality of the structures $\$cs1$ and $\$cs2$. It is defined by the rule

$(rule\ (e1\ != e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (not\ (e1 = e2)))$.

If $\$cs$ is a value of $\$e$, then the conceptual structure access operation $(\$e\ .\ \$mt)$ returns $[\$cs\ .\ \$mt]$. It is defined as follows:

$(rule\ (e\ .\ mt)\ var\ (e,\ t)\ val\ (e)\ abn\ then\ (e :: \{*\}\ .\ mt): \{transition\})$;

$(transition\ (cs\ .\ mt) :: \{transition\}\ var\ (cs,\ mt)\ then\ \$f)$,

where $(\$cs\ .\ \$mt) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$cs\ .\ \$mt] \# \$s$.

If $\$ccs$ and $\$n$ are values of $\$e1$ and $\$e2$, then the conceptual structure access operation $(\$e1\ ..\ \$e2)$ returns $[\$ccs\ ..\ \$n]$. It is defined as follows:

$(rule\ (e1\ ..\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$where\ ((e1 :: \{*\}\ is\ compound)\ and\ (e2 :: \{*\}\ is\ nat)\ and\ (e2 :: \{*\} > 0))$

$then\ (e1 :: \{*\}\ ..\ e2 :: \{*\}): \{transition\})$;

$(transition\ (cs\ ..\ n) :: \{transition\}\ var\ (cs,\ n)\ then\ \$f)$,

where $(\$cs \,..\, \$n) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$cs \,..\, \$n] \# \$s$.

If $\$cs$ and $\$v$ are values of $\$e$ and $\$e1$, then the conceptual structure update operation $(\$e \,.\, \$mt := \$e1)$ returns $[\$cs \,.\, \$mt \coloneqq \$v]$. It is defined as follows:

$(rule\ (e \,.\, mt1\ \coloneqq e1)\ var\ (e,\ mt1,\ e1)\ val\ (e,\ e1)\ abn)$

$then\ (e::\{*\} \,.\, mt1\ \coloneqq e1::\{*\})::\{transition\}));$

$(transition\ (cs \,.\, mt \coloneqq\ v)::\{transition\}\ var\ (cs, mt, v)\ abn\ then\ \$f)$,

where $(\$cs \,.\, \$mt := \$v)::\{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$cs \,.\, \$mt \coloneqq \$v] \# \$s$.

The conceptual structure update operation $(\$e \,.\, \$mt :=)$ is a shortcut for $(\$e \,.\, \$mt := und)$.

The conceptual structure update operation $(\$e \,.\, \$mt1 := \$e1, \dots, \$mt\$n := \$e\$n)$, where $\$n > 1$, is defined as follows:

$(rule\ (e \,.\, mt1\ \coloneqq e1\ cs_s\ )\ var\ (e,\ mt1,\ e1)\ seq\ (cs_s\ )\ abn\ then\ ((e \,.\, mt1\ \coloneqq e1) \,.\, cs_s))$.

If $\$ccs$, $\$n$ and $\$v$ are values of \$e1, \$e2 and \$e3, then the conceptual structure update operation $(\$e1 \,..\, \$e2 := \$e3)$ returns $[\$ccs \,..\, \$n := \$v]$. It is defined as follows:

$(rule\ (e1 \,..\, e2\ \coloneqq e3)\ var\ (e1,\ e2,\ e3)\ val\ (e1,\ e2,\ e3)\ abn$

$where\ ((e1 :: \{*\}\ is\ compound)\ and\ (e2 :: \{*\}\ is\ nat)\ and\ (e2 :: \{*\}\ >\ 0))$

$then\ (e1 :: \{*\} \,..\, e2 :: \{*\}\ \coloneqq e3 :: \{*\}) :: \{transition\});$

$(transition\ (ccs \,..\, n\ \coloneqq v) :: \{transition\}\ var\ (ccs, n, v)\ abn\ then\ \$f)$,

where $(\$ccs \,..\, \$n := \$v) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$ccs \,..\, \$n := \$v] \# \$s$.

If $\$ccs1$ and $\$ccs2$ are values of $\$e1$ and $\$e2$, then the element $(\$e1 + \$e2)$ specifies the concatenation of $\$ccs1$ and $\$ccs2$. It is defined by the rules

$(rule\ (e1 + e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (e1 :: \{*\} +:: \{q\}\ e2 :: \{*\}));$

$(rule\ ((cs_s1) +:: \{q\}\ (cs_s2))\ seq\ (cs_1,\ cs_2)\ then\ (cs_s1\ cs_s2) :: \{q\})$.

If $\$e$ and $\$ccs$ are values of $\$e1$ and $\$e2$, then the element $(\$e1 \,.+\, \$e2)$ specifies the addition of the element $\$e$ to the head of $\$ccs$. It is defined by the rules

$(rule\ (e1 \,.+\, e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (e1 :: \{*\} \,.+:: \{q\}\ e2 :: \{*\}));$

$(rule\ (e \,.+:: \{q\}\ \ (cs_s\ ))\ var(e)\ seq\ (cs_s)\ then\ (e\ cs_s) :: \{q\})$.

If $\$e$ and $\$ccs$ are values of $\$e2$ and $\$e1$, then the element $(\$e1 +.\ \ \$e2)$ specifies the addition of the element $\$e$ to the tail of $\$ccs$. It is defined by the rules

$(rule\ (e1 + e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (e1 :: \{*\} +. :: \{q\}\ e2 :: \{*\}));$

$(rule\ ((cs_s\ ) +. :: \{q\}\ e)\ \ var(e)\ seq\ (cs_s)\ then\ (cs_s\ e) :: \{q\})$.

If $\$e$ and $\$n$ are values of $\$e1$ and $\$e2$, then the element $(repeat\ \$e1\ \$e2)$ returns $([repeat\ \$e\ \$n])$. It is defined by the rule

$(rule\ (repeat\ e\ n)\ var\ (e,\ n)\ val\ (e,\ n)\ abn\ where\ (n :: \{*\}\ is\ nat)$

$then\ (repeat::\{q\}\ e::\{*\}\ n::\{*\}))$.

The element $(repeat::\{q\}\ \$e1\ \$e2)$ is defined by the rules

$(rule\ (repeat::\{q\}\ e\ 0)\ var\ (e)\ abn\ then\ ())$;

$(rule\ (repeat::\{q\}\ e\ n)\ var\ (e,\ n)\ abn$

$then\ (let\ n1\ be\ (n-1)\ in\ ((repeat::\{q\}\ e\ n1)\ +.\ e::\{q\}))$.

The element $(unbracket\ \$ccs)$ is defined by the rule

$(rule\ (unbracket\ (cs_s))\ seq\ (cs_s)\ abn\ then\ cs_s)$.

6.5. Sets

The element $(\$e\ is\ set)$ specifies that the elements of the compound structure $\$e$ are pairwise distinct is defined as follows:

$(rule\ (e\ is\ set)\ var\ (e)\ abn\ where\ (e\ is\ compound)\ then\ (e\ is\ set)::\{transition\})$;

$(transition\ (e\ is\ set)::\{transition\}\ var\ (e)\ then\ \$f)$,

where

$(e\ is\ set)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if$ [the elements of $\$e$ are pairwise distinct] $then\ true\ else\ und]\ \#\ \$s$.

If $\$e$ and $\$ccs$ are values of $\$e2$ and $\$e1$, then the element $(\$e1\ +.::\{set\}\ \$e2)$ specifies the addition of the element $\$e$ to the set $\$ccs$. It is defined by the rule

$(rule\ (e1\ +.::\{set\}\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$then\ (if\ (e2::\{*\}::\{q\}\ in\ e1::\{*\}::\{q\})\ then\ e1::\{*\}::\{q\}$

$else\ (e2::\{*\}\ +.::\{q\}\ e1::\{*\}))$.

If $\$e$ and $\$ccs$ are values of $\$e2$ and $\$e1$, then the element $(\$e1\ -.::\{set\}\ \$e2)$ specifies the deletion of the element $\$e$ from the set $\$ccs$. It is defined by the rule

$(rule\ (e1\ -.::\{set\}\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ where\ (e1::\{*\}\ is\ set)$

$then\ (e1::\{*\}\ -.::\{set\}\ e2::\{*\})::\{transition\})$;

$(transition\ (ccs\ -.::\{set\}\ e)::\{transition\}\ var\ (ccs,\ e)\ then\ \$f)$,

where $(\$ccs\ -.::\{set\}\ \$e)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ \$ccs1\ \#\ \$s$, $\$ccs1$ is a set, and $[\$ccs1 =_{set} \$ccs\ \$v]$.

If $\$e$ and $\$ccs$ are the values of $\$e1$ and $\$e2$, then the element $(\$e1\ in\ \$e2)$ specifies that $\$e$ is an element of $\$ccs$. It is defined as follows:

$(rule\ (e1\ in::\{set\}\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ where\ (e2::\{*\}\ is\ compound)$

$then\ (e1::\{*\}\ in::\{set\}\ e2::\{*\})::\{transition\})$;

$(transition\ (e\ in::\{set\}\ ccs)::\{transition\}\ var\ (e,ccs)\ abn\ then\ \$f)$,

where

$(\$e\ in :: \{set\}\ \$ccs) :: \{transition\};\ \$e^*\ \ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [\$e \in \$ccs]\ then\ true\ else\ und]\ \#\ \$s.$

If $\$ccs1$ and $\$ccs2$ are the values of $\$e1$ and $\$e2$, then the element $(\$e1\ includes::\{set\}\ \$e2)$ specifies that $\$ccs1$ includes the elements of $\$ccs1$. It is defined as follows:

$(rule\ (e1\ includes::\{set\}\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$where\ ((e1 :: \{*\}\ is\ compound)\ and\ (e2 :: \{*\}\ is\ compound))$

$then\ (e1 :: \{*\}\ includes::\{set\}\ e2 :: \{*\}) :: \{transition\});$

$(transition\ (ccs1\ includes::\{set\}\ ccs2) :: \{transition\}\ var\ (ccs1, ccs2)\ abn\ then\ \$f),$

where

$(\$ccs1\ includes :: \{set\}\ \$ccs2) :: \{transition\};\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [\$ccs1$ includes the elements of $\$ccs2]\ then\ true\ else\ und]\ \#\ \$s.$

If $\$ccs1$ and $\$ccs2$ are the values of $\$e1$ and $\$e2$, then the element $(disjoint::\{set\}\ \$e1\ \$e2)$ specifies that $\$ccs1$ and $\$ccs1$ have no common elements. It is defined as follows:

$(rule\ (disjoint::\{set\}\ e1\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$where\ ((e1 :: \{*\}\ is\ compound)\ and\ (e2 :: \{*\}\ is\ compound))$

$then\ (disjoint::\{set\}\ e1 :: \{*\}\ e2 :: \{*\}) :: \{transition\});$

$(transition\ (disjoint::\{set\}\ ccs1\ ccs2) :: \{transition\}\ var\ (ccs1, ccs2)\ abn\ then\ \$f),$

where

$(disjoint :: \{set\}\ \$ccs1\ \$ccs2) :: \{transition\};\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [\$ccs1$ and $\$ccs2$ have no common elements$]\ then\ true\ else\ und]\ \#\ \$s.$

The elements $(\$e1\ in\ \$e2)$, $(\$e1\ includes\ \$e2)$ and $(disjoint\ \$e1\ \$e2)$ are shortcuts for $(\$e1\ in::\{set\}\ \$e2)$, $(\$e1\ includes::\{set\}\ \$e2)$ and $(disjoint::\{set\}\ \$e1\ \$e2)$.

6.6. States

The executable elements handling states are defined in this section.

The state access operation $(current\ state)$ returns the current state is defined by the rule:

$(rule\ (current\ state)\ abn\ then\ cstate::\{q\}).$

If $\$cs$ is a value of $\$e$, then the element $((to\ state)\ \$e)$ replaces the current state to $\$cs$. It is defined as follows:

$(rule\ ((to\ state)\ e)\ var\ (e)\ val\ (e)\ then\ ((to\ state)\ e::\{*\})::\{transition\});$

$(transition\ ((to\ state)\ cs)::\{transition\}\ var\ (cs)\ then\ \$f),$

where $((to\ state)\ \ \$cs)::\{transition\};\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$cs.$

The state access operation $(.\ \$mt)$ returns $[.\ \$mt]$. It is defined by the rule

$(rule\ (.\ mt)\ var\ (mt)\ abn\ then\ (cstate :: \{q\}\ .\ mt))$.

If $\$v$ is a value of $\$e$, then the state update operation $(mt := \$e)$ replaces the current state by $[.\ \$mt \coloneqq \$v]$. It is defined by the rule

$(rule\ (mt := e)\ var\ (mt,\ e)\ abn\ then\ (to\ state\ (cstate :: \{q\}\ .\ mt \coloneqq e)))$.

The state update operation $(.\ mt := \$e)$ is an alias for $(mt := \$e)$. It is defined by the rule

$(rule\ (.\ mt := e)\ var\ (mt,\ e)\ abn\ then\ (mt := e))$.

The state update operation $(mt :=)$ and $(.\ mt :=)$ are shortcuts for $(mt := und)$ and $(.\ mt := und)$.

The state update operation $(\$mt1 := \$e1,\ \dots,\ \$mt\$n := \$e\$n)$, where $\$n > 1$, is defined by the rule

$(rule\ (mt1 \coloneqq e1\ cs_s)\ var\ (mt1,\ e1)\ seq\ (cs_s)\ abn)\ then\ (mt1 \coloneqq e1);\ (cs_s))$.

The state update operation $(.\ \$mt1 := \$e1,\ \dots,\ \$mt\$n := \$e\$n)$ is an alias for $(\$mt1 := \$e1,\ \dots,\ \$mt\$n := \$e\$n)$. It is defined by the rule

$(rule\ (.\ mt1 \coloneqq e1\ cs_s)\ var\ (mt1,\ e1)\ seq\ (cs_s)\ abn)\ then\ (mt1 \coloneqq e1\ cs_s))$.

6.7. Statements

The executable elements called statements are defined in this section. They are similar to statements in programming languages.

The element $skip$ does nothing. It is defined as follows:

$(rule\ skip\ abn\ then\ skip :: \{transition\})$;

$(transition\ skip :: \{transition\}\ then\ \$f)$,

where $skip :: \{transition\}$; $\$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} e^*\ \#\ \$s$.

The element $\$e$ of the form $(seq\ e^*)$ is called a sequential composition. It is defined by the rule

$(rule\ (seq\ e_s)\ var\ (e_s)\ seq\ (e_s)\ then\ e_s)$.

The elements of $\$e^*$ are called elements in $[\![\$e]\!]$, and $\$e^*$ is called a body in $[\![\$e]\!]$. The element $\$e$ executes its elements sequentially from left to right.

The element $\$e$ of the form $(if\ \$co\ then\ \$e^*1\ else\ \$e^*2)$ is called a conditional element. It is defined as follows:

$(rule\ (if\ co\ then\ e_s1\ else\ e_s2)\ var\ (co)\ seq\ (e_s1,\ e_s2)\ val\ (co)\ abn$

$then\ (if\ co :: \{*\}\ then\ e_s1\ else\ e_s2) :: \{transition\})$;

$(transition\ (if\ v\ then\ e_s1\ else\ e_s2) :: \{transition\}\ var\ (v)\ seq\ (e_s1,\ e_s2)\ then\ \$f)$,

where

$(if\ \$v\ then\ \$e^*1\ else\ \$e^*2) :: \{transition\};\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$[if\ [\$v\ \neq und]\ then\ \$e^*1\ else\ \$e^*2]\ \$e^*\ \#\ \$s.$

The objects $\$co$, $\$e^*1$ and $\$e^*2$ are called a condition, then-branch and else-branch in $[\![\$e]\!]$. The element $(if\ \$con\ then\ \$e^*)$ is a shortcut for $(if\ \$con\ then\ \$e^*\ else\ skip)$.

The conditional element $(if :: \{exc\}\ \$con\ then\ \$e^*1\ else\ \$e^*2)$ is defined by the rule

$(rule\ (if :: \{exc\}\ con\ then\ e_s1\ else\ e_s2)\ var\ (con)\ seq\ (e_s1,\ e_s2)\ val\ (con)$

$exc\ (con,\ con :: \{*\})\ abn\ then\ (if\ con :: \{q\}\ then\ e_s1\ else\ e_s2)).$

The element $(if :: \{exc\}\ \$con\ then\ \$e^*)$ is a shortcut for $(if :: \{exc\}\ \$con\ then\ \$e^*\ else\ skip)$.

The conditional element

$(if\ \$se1\ \$co1\ then\ \$e^*1\ elseif\ \$se2\ \$co2\ then\ \$e^*2 \ldots\ elseif\ \$se\$n\ \$co\$n\ then\ \$e^*\$n\ else\ \$e^*)$, where $\$se\$n1 \in \{[es],\ exc\}$ for each $1 \leq \$n1 \leq \$n$, is defined by the rules

$(rule\ (if\ co\ then\ e_s1\ elseif\ e_s2)\ var\ (co)\ seq\ (e_s1,\ e_s2)\ abn$

$then\ (if\ co\ then\ e_s1\ else\ (if\ e_s2)));$

$(rule\ (if\ co\ then\ e_s1\ elseif :: \{exc\}\ e_s2)\ var\ (co)\ seq\ (e_s1,\ e_s2)\ abn$

$then\ (if\ co\ then\ e_s1\ else\ (if :: \{exc\}\ e_s2)));$

$(rule\ (if :: \{exc\}\ \ co\ then\ e_s1\ elseif\ e_s2)\ var\ (co)\ seq\ (e_s1,\ e_s2)\ abn$

$then\ (if :: \{exc\}\ co\ then\ e_s1\ else\ (if\ e_s2)));$

$(rule\ (if :: \{exc\}\ \ co\ then\ e_s1\ elseif :: \{exc\}\ e_s2)\ var\ (co)\ seq\ (e_s1,\ e_s2)\ abn$

$then\ (if :: \{exc\}\ co\ then\ e_s1\ else\ (if :: \{exc\}\ e_s2))).$

The element $\$e$ of the form $(let\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ is defined as follows:

$(rule\ (let\ va\ be\ e_s1\ in\ e_s2)\ var\ (va)\ seq\ (e_s1, e_s2)\ abn$

$then\ e_s1;\ (let\ va\ be\ current\ value\ in\ e_s2) :: \{transition\});$

$(transition\ (let\ va\ be\ current\ value\ in\ e_s2) :: \{transition\}\ var\ (va)\ seq\ (e_s2)\ then\ \$f)$, where

$(let\ \$va\ be\ current\ value\ in\ \$e^*2) :: \{transition\}; \$e^*\ \#\ \$v\ \#\ \$s \hookrightarrow_{[\![\$f]\!]}$

$[sub\ (\$va{:}\ \$v)\ \$e^*2]; \$e^*\ \#\ \$s.$

The elements $\$va$, $\$e^*1$ and $\$e^*2$ are called a variable, value specifier and body in $[\![\$e]\!]$.

The element $\$e$ of the form $(let :: \{und\}\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ is defined by the rules

$(rule\ (let :: \{und\}\ va\ be\ e_s1\ in\ e_s2)\ var\ (va)\ seq\ (e_s1, e_s2)\ abn$

$then\ e_s1;\ (let :: \{und\}\ va\ be\ current\ value\ in\ e_s2));$

$(rule\ (let :: \{und\}\ va\ be\ current\ value\ in\ e_s2)\ var\ (va)\ seq\ (e_s2)\ und$

$then\ (let\ va\ be\ current\ value\ in\ e_s2){:}\{transition\}).$

The element $\$e$ of the form $(let :: \{abn\}\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ is defined by the rules

$(rule\ (let :: \{abn\}\ va\ be\ e_s1\ in\ e_s2)\ var\ (va)\ seq\ (e_s1, e_s2)\ abn$
$then\ e_s1;\ (let :: \{abn\}\ va\ be\ current\ value\ in\ e_s2));$

$(rule\ (let :: \{abn\}\ va\ be\ current\ value\ in\ e_s2)\ var\ (va)\ seq\ (e_s2)\ abn$
$then\ (let\ va\ be\ current\ value\ in\ e_s2) :: \{transition\}).$

The element $\$e$ of the form $(let :: \{exc\}\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ is defined by the rules

$(rule\ (let :: \{exc\}\ va\ be\ e_s1\ in\ e_s2)\ var\ (va)\ seq\ (e_s1, e_s2)\ abn$
$then\ e_s1;\ (let :: \{exc\}\ va\ be\ current\ value\ in\ e_s2));$

$(rule\ (let :: \{exc\}\ va\ be\ current\ value\ in\ e_s2)\ var\ (va)\ seq\ (e_s2)\ exc$
$then\ (let\ va\ be\ current\ value\ in\ e_s2) :: \{transition\}).$

The element $\$e$ of the form $(let :: \{seq\}\ \$va^*\ be\ \$e^*1\ in\ \$e^*2)$, where $[len\ va^*] = [len\ e^*1]$, is defined by the rules

$(rule\ (let :: \{seq\}\ va, va_s\ be\ e1,\ e_s1\ in\ e_s2)\ var\ (va,\ e1)\ seq\ (va_s, e_s1, e_s2)\ abn$
$then\ (let\ va\ be\ e1\ in\ (let :: \{seq\}\ va_s\ be\ e_s1\ in\ e_s2)));$

$(rule\ (let :: \{seq\}\ be\ in\ e_s2)\ seq\ (e_s2)\ abn\ then\ e_s2).$

The elements $\$va^*$, $\$e^*1$ and $\$e^*2$ are called a variable specification, value specification and body in $[\![\$e]\!]$. The elements of $\$va^*$ and $\$e^*1$ are called variables and value specifiers in $[\![\$e]\!]$.

The element $\$e$ of the form $(let :: \{seq, und\}\ \$va^*\ be\ \$e^*1\ in\ \$e^*2)$, where $[len\ va^*] = [len\ e^*1]$, is defined by the rules

$(rule\ (let :: \{seq, und\}\ va, va_s\ be\ e1,\ e_s1\ in\ e_s2)\ var\ (va,\ e1)\ seq\ (va_s, e_s1, e_s2)$
$abn\ then\ (let :: \{und\}\ va\ be\ e1\ in\ (let :: \{seq, und\}\ va_s\ be\ \ e_s1\ in\ e_s2)));$

$(rule\ (let :: \{seq, und\}\ be\ in\ e_s2)\ seq\ (e_s2)\ abn\ then\ e_s2).$

The element $\$e$ of the form $(let :: \{seq, abn\}\ \$va^*\ be\ \$e^*1\ in\ \$e^*2)$, where $[len\ va^*] = [len\ e^*1]$, is defined by the rules

$(rule\ (let :: \{seq, abn\}\ va, va_s\ be\ e1,\ e_s1\ in\ e_s2)\ var\ (va,\ e1)\ seq\ (va_s, e_s1, e_s2)$
$abn\ then\ (let :: \{abn\}\ va\ be\ e1\ in\ (let :: \{seq, abn\}\ va_s\ be\ \ e_s1\ in\ e_s2)));$

$(rule\ (let :: \{seq, abn\}\ be\ in\ e_s2)\ seq\ (e_s2)\ abn\ then\ e_s2).$

The element $\$e$ of the form $(let: \{seq, exc\}\ \$va^*\ be\ \$e^*1\ in\ \$e^*2)$, where $[len\ va^*] = [len\ e^*1]$, is defined by the rules

$(rule\ (let :: \{seq, exc\}\ va, va_s\ be\ e1,\ e_s1\ in\ e_s2)\ var\ (va,\ e1)\ seq\ (va_s, e_s1, e_s2)\ abn$
$then\ (let :: \{exc\}\ va\ be\ e1\ in\ (let :: \{seq, exc\}\ va_s\ be\ \ e_s1\ in\ e_s2)));$

$(rule\ (let :: \{seq, exc\}\ be\ in\ e_s2)\ seq\ (e_s2)\ abn\ then\ e_s2).$

The element $\$e$ of the form $(while\ \$con\ do\ \$e^*1)$ is called a while statement. It is defined by the rule

$(if\ (while\ con\ do\ e_s)\ var\ (con)\ seq\ (e_s)\ abn$

$then\ (if\ con\ then\ e_s;\ (while\ con\ do\ e_s)))$.

The objects $\$con$ and $\$e^*1$ are called a condition and body in $[\![\$e]\!]$.

The element $\$e$ of the form $(foreach\ \$va\ in\ \$e1\ do\ \$e^*1)$ is called a foreach statement. It is defined by the rule

$(rule\ (foreach\ va\ in\ e1\ do\ e_s)\ var\ (va,\ e1)\ seq\ (e_s)\ val\ (e1)\ abn$

$then\ (foreach{::}\{q\}\ va\ in\ e1 :: \{*\}\ do\ e_s))$.

The objects $\$va$, $\$e1$ and $\$e^*1$ are called an iteration variable, iteration structure specifier and body in $[\![\$e]\!]$. If $(\$v^*)$ is a value of $\$e1$, then the element $\$e$ executes sequentially $\$e^*1$ for values of $\$va$ from the structure $(\$v^*)$.

The element $(foreach{::}\{q\}\ va\ in\ (\$v^*)\ do\ \$e^*)$ is defined by the rules

$(rule\ (foreach{::}\{q\}\ va\ in\ (v\ v_s)\ do\ e_s)\ var\ (va,\ v)\ seq\ (v_s,\ e_s)\ abn$

$then\ (let\ va\ be\ v :: \{q\}\ in\ e_s);\ (foreach{::}\{q\}\ va\ in\ (v_s)\ do\ e_s))$;

$(rule\ (foreach{::}\{q\}\ va\ in\ ()\ do\ e_s)\ var\ (va)\ seq\ (e_s)\ abn\ then)$.

6.8. Countable concepts

The executable elements handling countable concepts are defined in this section.

A normal element $\$e$ is a countable concept in $[\![\$s]\!]$ if $[\$s\ .\ ((countable\ concept)\ \$e)] \in \$n > 0$. Thus, the parametric attribute $((countable\ concept)\ \$e)$ defines countable concepts. Let $\$\$cc$ be a set of countable concepts. A number $\$n$ is an order in $[\![\$cc,\ \$s]\!]$ if $\$n = [\$s\ .\ ((countable\ concept)\ \$cc)]$. Let $\$\$cco$ be a set of orders of countable concepts. An element $\$n{::}\{\$cc\}$ is called an instance in $[\![\$cc]\!]$. An element $\$n{::}\{\$cc\}$ is an instance in $[\![\$cc,\ \$s]\!]$ if $1 \leq \$n \leq \$cco[\![\$cc]\!]$.

The element $(\$e\ is\ (countable\ concept))$ specifies that $\$e$ is a countable concept. it is defined by the rule

$(rule\ (e\ is\ (countable\ concept))\ var\ (e)\ abn\ then\ ((.\ \{((countable\ concept)\ e)\}) > 0))$.

The element $(\$e\ is\ \$cc)$ specifies that $\$e$ is an instance of $\$cc$. It is defined by the rule

$(rule\ (n{::}\{cc1\}\ is\ cc2)\ var\ (n,\ cc1,\ cc2)\ abn\ where\ (cc2\ is\ (countable\ concept))$

$then\ ((cc1{::}\{q\} = cc2{::}\{q\})\ and\ (0 < n)\ and\ (n <= (.\ \{((countable\ concept)\ cc2)\}))))$.

The element $((new\ instance)\ \$cc)$ generates a new instance of the countable concept $\$cc$ and adds this concept if it was not. It is defined by the rule

$(rule\ ((new\ instance)\ cc)\ var\ (cc)\ abn$

$then\ (let\ n\ be\ (.\ \{(countable\ concept)\ cc\})\ in$

$(if\ (n\ >\ 0)\ then\ (let\ n1\ be\ (n+1)\ in$

$(\{((countable\ concept)\ cc)\}\ \coloneqq\ n1);\ n1::\{cc\}::\{q\})$

$else\ (\{((countable\ concept)\ cc)\}\ \coloneqq\ 1);\ n1::\{cc\}::\{q\}).$

6.9. Rules

The executable elements handling rules are defined in this section.

The element ($\$e\ is\ rule$) specifies that $\$e$ is a rule. It is defined as follows:

$(rule\ (e\ is\ rule)\ var\ (e)\ abn\ then\ (e\ is\ rule)::\{transition\});$

$(transition\ (e\ is\ rule)::\{transition\}\ var\ (e)\ then\ \$f),$

where

$(\$e\ is\ rule)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [if\ [e \in \$\$r]\ then\ true\ else\ und]\ \#\ \$s.$

The element ($\$e\ is\ (extended\ rule)$) specifies that $\$e$ is an extended rule. It is defined as follows:

$(rule\ (e\ is\ (extended\ rule))\ var\ (e)\ abn\ then\ (e\ is\ (extended\ rule))::\{transition\});$

$(transition\ (e\ is\ (extended\ rule))::\{transition\}\ var\ (e)\ then\ \$f),$

where

$\big(\$e\ is\ (extended\ rule)\big) :: \{transition\}\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [e \in \$\$er]\ then\ true\ else\ und]\ \#\ \$s.$

An element $\$na$ is a name if $\$na$ is normal. Let $\$\$n$ be a set of names.

The element ($\$e\ is\ name$) specifies that $\$e$ is a name. It is defined by the rule

$(rule\ (e\ is\ name)\ var\ (e)\ abn\ then\ (e\ is\ normal)).$

The element $\$r{:}\,\$na$ adds the rule $\$r$ with the name $\$n$ into [. $\{rules\}$]. It is defined by the rule

$(rule\ e::\{na\}\ var\ (e,\ na)\ abn\ where\ ((e\ is\ rule)\ and\ (na\ is\ name))$

$then\ (\{rules\} := ((.\{rules\})\ .\ \{na\} := e::\{q\}))).$

The element $\$er{:}\,\$na$ adds the rule $\$r$ with the name $\$n$ into [. $\{rules\}$], where $\$er$ is a shortcut for $\$r$. It is defined as follows:

$(rule\ e::\{na\}\ var\ (e,\ na)\ abn\ where\ ((e\ is\ (extended\ rule))\ and\ (na\ is\ name))$

$then\ (\{rules\} := ((.\{rules\})\ .\ \{na\}\ \coloneqq\ (rule\ e)))).$

The element ($rule\ \$er$) returns $\$r$, where $\$r$ is a shortcut for $\$er$. It is defined as follows:

$(transition\ (rule\ er)::\{transition\}\ var\ (er)\ then\ \$f),$

where $(rule\ \$er)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ \$r\ \#\ \$s$, where $\$r$ is a shortcut for $\$er$.

6.10. The pattern matching

The executable elements handling the pattern matching are defined in this section.

The conditional pattern matching element $\$e$ of the form

$(if\ \$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ then\ \$e^*1\ else\ \$e^*2)$,

where $(\$p\ (\$va^*)\ (\$sv^*))$ is a pattern specification, executes $\$e^*1$ if $\$e1$ matches $\$p$ and executes $\$e^*2$, otherwise. It is defined as follows:

$(rule\ (if\ e1\ matches\ p\ var\ (va_s)\ seq\ (sv_s)\ then\ e_s1\ else\ e_s2)\ var\ (e1,\ p)$
$seq\ (va_s,\ sv_s,\ e_s1,\ e_s2)\ abn\ where\ (disjoint\ (va_s){::}\{q\}\ (sv_s){::}\{q\})$
$then\ (if\ e1\ matches\ p\ var\ (va_s)\ seq\ (sv_s)\ then\ e_s1\ else\ e_s2){::}\{transition\})$;
$(transition\ (if\ e1\ matches\ p\ var\ (va_s)\ seq\ (sv_s)\ then\ e_s1\ else\ e_s2){::}\{transition\}$
$var\ (e1,\ p)\ seq\ (va_s,\ sv_s,\ e_s1,\ e_s2)\ then\ \$f)$,

where

$(if\ \$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ then\ \$e^*1\ else\ e^*2){::}\{transition\};\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]}$
$[if\ [\$e1\ is\ an\ instance\ in\ [\![(\$p\ (\$va^*)\ (\$sv^*)),\ \$mt,\ \$su]\!]\ for\ some\ \$su]$
$then\ [sub\ \$su\ \cup\ (cstate{:}\ \$s,\ cvalue{:}\ \$v[\![\$s]\!])\ \$e^*1]$
$else\ [sub\ (cstate{:}\ \$s,\ cvalue{:}\ \$v[\![\$s]\!])\ \$e^*2]];\ \$e^*\ \#\ \$s.$

Thus, the semantics of the conditional pattern matching elements combines the semantics of the conditional element with the semantics of transition rules. The elements $\$e1$, $\$p$, $\$va^*$, $\$sv^*$, $\$e^*1$ and $\$e^*2$ are called a matched structure, pattern, state variable specification, sequence variable specification, $then$-branch and $else$-branch in $[\![\$e]\!]$. The elements of $\$va^*$ and $\$sv^*$ are called state and sequence variables in $[\![\$e]\!]$.

Let $\{\$eva^*\} \subseteq \{\$va^*\}$, the elements of the sequence $\$eva^*$ are pairwise disjoint, $\$set\ =\ \{\$eva :: \{*\}\ |\ \$eva \in \$eva^*\}$, $\{\$va^*1\} \cup \{\$va^*2\} \cup \{\$va^*3\} \subseteq \{\$va^*\} \cup \$set$, the elements of the sequence $\$va^*1\ \$va^*2\ \$va^*3$ are pairwise disjoint, and $\$se \in \{[se],\ und,\ exc,\ abn\}$.

The semantics of the extension

$(if\ \$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)$
$exc\ (\$va^*3)\ \$se\ where\ \$co\ then\ \$e^*1\ else\ \$e^*2)$

of the conditional pattern matching element is similar to the semantics of extended transition rules.

The objects $var\ (\$va^*)$, $seq\ (\$sv^*)$, $val\ (\$eva^*)$, $abn\ (\$va^*1)$, $und\ (\$va^*2)$, $exc\ (\$va^*3)$ and $where\ \$co$ in conditional pattern matching elements can be omitted. The omitted objects correspond to var ( ), seq ( ), val ( ), abn ( ), und ( ), exc ( ) and $where\ true$.

The pattern matching element

$(\$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)$
$exc\ (\$va^*3)\ \$se\ where\ \$co\ then\ \$e^*1\ else\ \$e^*2)$

is a shortcut for

$(if\ \$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)$
$exc\ (\$va^*3)\ \$se\ where\ \$co\ then\ true\ else\ und)$.

The selection element $\$e$ of the form $(select\ \$va\ from\ \$e1\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*))$, where $(\$p\ (\$va^*)\ (\$sv^*))$ is a pattern specification, and $\$va \in \$va^*$, or $\$va \in \$sv^*$, selects the values of the variable $\$va$ such that an element of $\$e1$ matches the pattern $\$p$. It is defined by the rule

$(rule\ (select\ va\ from\ of\ e1\ wrt\ p\ var\ (va_s)\ seq\ (sv_s))$
$var\ (va, e1, p)\ seq\ (va_s, sv_s, t_s)\ abn$
$where\ ((disjoint\ (va_s) :: \{q\}\ (sv_s) :: \{q\})\ and$
$((va::\{q\}\ in\ (va_s)::\{q\})\ or\ (va::\{q\}\ in\ (sv_s)::\{q\})))$
$then\ (select::\{check\}\ va\ from\ e1\ wrt\ p\ var\ (va_s)\ seq\ (sv_s)))$.

The elements $\$va$, $\$e1$, $\$p$, $\$va^*$ and $\$sv^*$ are called a selection variable, matched structure, pattern, state variable specification, sequence variable specification, $then$-branch and $else$-branch in $[\![\$e]\!]$. The elements of $\$va^*$ and $\$sv^*$ are called state and sequence variables in $[\![\$e]\!]$.

The element $(select::\{check\}\ \$va\ from\ \$e1\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*))$ is defined by the rules

$(rule\ (select::\{check\}\ va\ from\ e1::\{t_s\}\ wrt\ p\ var\ (va_s)\ seq\ (sv_s))$
$var\ (va, e1,\ p)\ seq\ (va_s,\ sv_s,\ t_s)\ abn$
$then\ (select::\{check\}\ va\ from\ (e1::\{t_s\})\ wrt\ p\ var\ (va_s)\ seq\ (sv_s)))$;
$(rule\ (select::\{check\}\ va\ from\ e1:\{t_s\}\ wrt\ p\ var\ (va_s)\ seq\ (sv_s))$
$var\ (va,\ e1,\ p)\ seq\ (va_s,\ sv_s,\ t_s)\ abn$
$then\ (select::\{check\}\ va\ from\ (e1:\{t_s\})\ wrt\ p\ var\ (va_s)\ seq\ (sv_s)))$;
$(rule\ (select::\{check\}\ va\ from\ ()\ wrt\ p\ var\ (va_s)\ seq\ (sv_s))$
$var\ (va, p)\ seq\ (e_s, va_s, sv_s)\ abn\ then\ ())$;
$(rule\ (select::\{check\}\ va\ from\ (e1\ e_s)\ wrt\ p\ var\ (va_s)\ seq\ (sv_s))$
$var\ (va, e1, p)\ seq\ (e_s, va_s, sv_s)\ abn\ where\ (va::\{q\}\ in\ (va_s)::\{q\})$
$then\ (if\ e1\ matches\ p\ var\ (va_s)\ seq\ (sv_s)$
$then\ (va_s::\{q\}\ .+\ (select::\{check\}\ va\ from\ (e_s)\ wrt\ p\ var\ (va_s)\ seq\ (sv_s)))$
$else\ (select::\{check\}\ va\ from\ (e_s)\ wrt\ p\ var\ (va_s)\ seq\ (sv_s))))$;
$(rule\ (select::\{check\}\ va\ from\ (e1\ e_s)\ wrt\ p\ var\ (va_s)\ seq\ (sv_s))$
$var\ (va, e1, p)\ seq\ (e_s, va_s, sv_s)\ abn\ where\ (va::\{q\}\ in\ (sv_s)::\{q\})$
$then\ (if\ e1\ matches\ p\ var\ (va_s)\ seq\ (sv_s)$
$then\ ((va_s)::\{q\}\ .+\ (select::\{check\}\ va\ from\ (e_s)\ wrt\ p\ var\ (va_s)\ seq\ (sv_s)))$

$else\ (select::\{\text{check}\}\ va\ from\ (e_s)\ wrt\ p\ var\ (va_s)\ seq\ (sv_s)))).$

The semantics of the extension

$(select\ \$va\ from\ \$e1\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)$

of the selection element is similar to the semantics of the extension of the conditional pattern matching element.

The semantics of the extension

$(select::\{seq\}\ \$va1^+\ from\ \$e1\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)$

of the selection element is similar to the semantics of the above selection element extension except that the resulting sequence consists of the compound elements of the length $[\$va1^+]$. Each of these elements contains the values of the selection variables in the order of their occurences in $\$va1^+$.

The selection elements

$(select\ \$va\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)$

and

$(select::\{seq\}\ \$va1^+\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)$

are shortcuts for

$(let\ \$s\ be\ (current\ state)\ in\ (select\ \$va\ from\ \$s\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co))$

and

$(let\ \$s\ be\ (current\ state)\ in\ (select::\{seq\}\ \ \$va1^+\ from\ \$s\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)$

$val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)).$

7. Examples of conceptual operational semantics of programming languages

An operational semantics of executable elements of $\$\$c[\![\$l]\!]$ in CTSL[o] is called a conceptual operational semantics of $\$l$. Thus, the conceptual operational semantics of $\$l$ is defined in terms of the conceptual model of $\$l$ in CTSL[o].

The conceptual operational semantics for the family of model programming languages (MPLs) is defined in this section. These languages has been described and their conceptual models has been defined in [1].

7.1. MPL1: types, typed variables and basic statements

The MPL1 language [1] is an extension of CTSL that adds types, typed variables, the variable access operation, and the basic statements such as variable declarations, variable assignments, if statements, while statements and block statements.

The element ($\$c\ is\ type$) specifies types in MPL1. It is defined by the rules

$(rule\ (t\ is\ type)\ var\ (t)\ abn\ then\ (t::\{q\} = int::\{q\}))$;

$(rule\ (t\ is\ type)\ var\ (t)\ abn\ then\ (t::\{q\} = nat::\{q\}))$.

The element ($subtype\ \$t1\ \$t2$) checks that $\$t1$ is a subtype of $\$t2$. It is defined by the rule

$(rule\ (subtype\ t1\ t2)\ var\ (t1,\ t2)\ abn$

$then\ ((t1::\{q\} = nat::\{q\})\ and\ (t2::\{q\} = int::\{q\})))$.

The element ($\$na\ is\ variable$) specifies variables. It is defined by the rule

$(rule\ (va\ is\ variable)\ var\ (va)\ abn\ where\ (va\ is\ name)\ then\ (.\ \{(variable\ va)\}))$.

The program is defined by the rule

$(rule\ (program\ n\ c_s)\ var\ (n)\ seq\ (c_s)\ abn\ where\ (n\ is\ name)$

$then\ ((collect\ body\ membes)\ c_s)\ c_s)$.

Let $\$\$m$ be a set of elements called body members.

The element $((collect\ body\ members)\ \$c^*)$ collects information about members of the body $\$c^*$. It is defined by the rules

$(rule\ ((collect\ body\ members)\ (var\ va\ t)\ c_s)\ var\ (va,\ t)\ seq\ (c_s)\ abn$

$where\ ((va\ is\ name)\ and\ (not\ (va\ is\ variable))\ and\ (t\ is\ type))$

$then\ (\{(type\ va)\} := t :: \{q\});\ (\{(variable\ va)\} := true);$

$((collect\ body\ members)\ c_s))$;

$(rule\ ((collect\ body\ members)\ (var\ c1\ c2)\ c_s)\ var\ (c1,\ c2)\ seq\ (c_s)\ abn\ then\ und)$;

$(rule\ ((collect\ body\ members)\ c\ c_s)\ var\ (c)\ seq\ (c_s)\ abn$

$then\ ((collect\ body\ members)\ c_s))$;

$(rule\ ((collect\ variables))\ then)$.

Thus, the body members in MPL1 are variables.

The variable declaration is defined by the rule

$(rule\ (var\ c_s)\ seq\ (c_s)\ abn\ then)$.

The execution of the variable declaration does not collect information about the declared variable, since the corresponding actions are performed by the element $((collect\ body\ members)\ \$c^*)$.

The variable access is defined by the rule

$(rule\ va\ var\ (va)\ abn\ where\ (va\ is\ variable)\ then\ (.\ \{(value\ va)\}))$.

The element $(type\ \$c)$ returns the type of the variable or the constant $\$c$. It is defined by the rules

$(rule\ (type\ va)\ var\ (va)\ abn\ (va)\ abn\ where\ (va\ is\ variable)\ then\ (.\ \{(type\ va)\})$;

$(rule\ (type\ n)\ var\ (n)\ abn\ (n)\ abn\ where\ (n\ is\ nat)\ then\ nat::\{q\})$;

$(rule\ (type\ i)\ var\ (i)\ abn\ (i)\ abn\ where\ (i\ is\ int)\ then\ int::\{q\})$.

The variable assignment is defined by the rule

$(rule\ (va \backslash{:=}\ c)\ var\ (va,\ c)\ val\ (va)\ exc\ (c,\ c::\{*\})\ abn$

$where\ ((va\ is\ variable)\ and$

$(let::\{seq\}\ t1,\ t2\ be\ (type\ va),\ (type\ c::\{*\})\ in\ (subtype\ t2\ t1)))$

$then\ (\{(value\ va)\} := c::\{*\}::\{q\}))$.

The block statement is defined by the rule

$(rule\ (block\ c_s)\ seq\ (c_s)\ abn\ then\ c_s)$.

The if statement is defined by the rule

$(rule\ (\backslash if\ c\ then\ c_s1\ else\ c_s2)\ var\ (c)\ seq\ (c_s1,\ c_s2)\ abn\ (c)\ abn$

$then\ (if::\{exc\}\ c\ then\ (block\ c_s1)\ else\ (block\ c_s2)))$;

The while statement is defined by the rule

$(rule\ (\backslash while\ c\ do\ c_s1)\ var\ (c)\ seq\ (c_s1)\ abn\ (c)\ abn$

$then\ (while::\{exc\}\ c\ do\ (block\ c_s1)))$.

Thus, then- and else- branches of the if statement and the body of the while statement behaves as blocks.

7.2. MPL2: variable scopes

The MPL2 language [1] is an extension of MPL1 that adds the variable scopes feature. The relative scope of the variable $\$va$ occuring in the element $\$c$ is the number of blocks surrounding this occurrence of $\$va$ in $\$c$. The value and type of $\$va$ depend on its scope. The variable $\$va$ can be global (with the scope 0) and local.

The element $(scope)$ returns the current scope. It is defined by the rule

$(rule\ (scope)\ abn\ then\ (.\ \{(current\ scope)\}))$.

The same name $\$na$ can refer to different program objects. For example, $\$va$ refers to the variables with the name $\$va$ of the scopes from 0 to $[.\ \{(current\ scope)\}]$. To distinguish these program objects, they are versioned. The pair $(\$na,\ \$ve)$, where $\$ve$ is a version, refers to the only one program object (with the version $\$ve$). In the case of variables, the version coincides with the variable scope.

The element $(version\ \$va)$ returns the correct version of $\$va$ in the current context of program execution. It is defined by the rule

$(rule\ (version\ va)\ var\ (va)\ abn\ (va)\ abn\ where\ (va\ is\ name)$
$then\ (let\ sc\ be\ (scope)\ in\ (version\ sc\ w)))$.

The element $(version\ \$va\ \$sc)$ is defined by the rule

$(rule\ (version\ va\ sc)\ var\ (va,\ sc)\ abn\ (va,\ sc)\ abn$
$then\ (if\ (.\ \{(variable\ va\ sc)\})\ then\ sc::\{q\}\ else\ (if\ (sc\ =\ 0)\ then\ und$
$else\ (let\ sc1\ be\ (sc-1)\ in\ (version\ va\ sc1)))))$.

The element $(\$na\ is\ variable)$ is defined by the rule

$(rule\ (va\ is\ variable)\ var\ (va)\ abn\ then\ (version\ va))$.

The element $((collect\ body\ members)\ \$c^{*})$ is defined by the rule

$(rule\ ((collect\ body\ members)\ c_s)\ seq\ (c_s)\ abn$
$then\ ((collect\ body\ members\ 1)\ ()\ c_s)$.

The element $((collect\ body\ members)::\{1\}\ (\$va^{*})\ \$c^{*})$ is defined by the rules

$(rule\ ((collect\ body\ members)::\{1\}\ (va_s)\ (var\ va\ t)\ c_s)\ var\ (va,\ t)$
$seq\ (va_s,\ c_s)\ abn\ (va,\ t)\ abn\ where\ ((va\ is\ name)\ and\ (t\ is\ type))$
$then\ (let\ sc\ be\ (scope)\ in$
$(if\ (.\ \{(variable\ va\ sc)\})\ then\ und$
$else\ (\{(type\ va\ sc)\}\ \coloneqq t::\{q\});\ (\{(variable\ va\ sc)\}\ \coloneqq true);$
$((collect\ body\ members)::\{1\}\ (va_s\ va)\ c_s))))$;

$(rule\ ((collect\ body\ members)::\{1\}\ (va_s)\ (var\ c1\ c2)\ c_s)\ var\ (c1,\ c2)$
$seq\ (va_s,\ c_s)\ abn\ then\ und)$;

$(rule\ ((collect\ body\ members)::\{1\}\ (va_s)\ c\ c_s)\ var\ (c)\ seq\ (va_s,\ c_s)\ abn$
$then\ ((collect\ body\ members)::\{1\}\ (va_s)\ c_s))$;

$(rule\ ((collect\ body\ members)::\{1\}\ (va_s))\ seq\ (va_s)\ abn\ then\ (va_s)::\{q\})$.

Thus, it returns the set of variables declared in the body $\$c^{*}$.

The variable access is defined by the rule

$(rule\ va\ var\ (va)\ abn\ (va)\ abn$
$then\ (let::\{und\}\ sc\ be\ (version\ va)\ in\ (.\ \{(value\ va\ sc)\})))$.

In the case when $\$c$ is a variable, the rule for the element $(type\ \$c)$ is replaced by the rule

$(rule\ (type\ va)\ var\ (va)\ abn\ (va)\ abn$
$then\ (let:\{und\}\ sc\ be\ (version\ va)\ in\ (.\ \{(type\ va\ sc)\})))$.

The element $(scope++)$ increases the value of the current scope by 1. It is defined by the rule

$(rule\ (scope++)\ abn\ then\ (\{(current\ scope)\} := ((.\ \{(current\ scope)\}) + 1)))$.

The element $(scope--)$ decreases the value of the current scope by 1. It is defined by the rule

$(rule\ (scope--)\ abn\ then\ (\{(current\ scope)\} := ((.\ \{(current\ scope)\}) - 1)))$.

The variable assignment is defined by the rule

$(rule\ (va \backslash := c)\ var\ (va,\ c)\ val\ (c)\ abn\ (va)\ exc\ (c, c::\{*\})\ abn$

$then\ (let::\{und\}\ sc\ be\ (version\ va)$

$in\ (if\ (let::\{und,\ seq\}\ t1,\ t2\ be\ (type\ va\ sc), (type\ c::\{*\})\ in\ (subtype\ t2\ t1))$

$then\ (((value\ va\ sc)) := c::\{*\}::\{q\})\ else\ und)))$.

The block statement is defined by the rule

$(rule\ (block\ c_s)\ seq\ (c_s)\ abn$

$then\ (enter\ block);\ (let\ v_s\ be\ ((collect\ body\ members)\ c_s)$

$in\ x\ (catch::\{und\}\ v\ ((exit\ block)\ v_s);\ v::\{q\})))$.

The element $(enter\ block)$ specifies the actions executed when the current state enters the block. It is defined by the rule

$(rule\ (enter\ block)\ abn\ then\ (scope++))$.

The element $((exit\ block)\ (\$va^*))$ specifies the actions executed when the current state exits the block. It is defined by the rule

$(rule\ ((exit\ block)\ (va_s))\ seq\ (va_s)\ abn\ then\ ((delete\ variables)\ va_s);\ (scope--))$.

The element $((delete\ variables)\ \$va^*)$ deletes the local variables $\$va^*$ with the current scope. It is defined by the rules

$(rule\ ((delete\ variables)\ va_s)\ seq\ (va_s)\ abn$

$then\ (let\ sc\ be\ (scope)\ in\ ((delete\ variables)::\{1\}\ sc\ va_s)))$.

The element $((delete\ variables)::\{1\}\ \$sc\ \$va^*)$ is defined by the rules

$(rule\ ((delete\ variables)::\{1\}\ sc\ va\ va_s)\ var\ (sc,\ va)\ seq\ (va_s)\ abn\ (sc,\ y)\ abn$

$then\ (\{(variable\ va\ sc)\} :=);\ (\{(type\ va\ sc)\} :=);\ (\{(value\ va\ sc)\} :=);$

$((delete\ variables)::\{1\}\ sc\ va_s));$

$(rule\ ((delete\ variables)::\{1\}\ sc)\ var\ (sc)\ abn\ (sc)\ abn\ then)$.

7.3. MPL3: functions

The MPL3 language [1] is an extension of MPL2 that adds the functions feature: declarations and calls of functions, and the return statement. For simplicity, function overloading is prohibited.

The element $(call\ level)$ returns the current call level. It is defined by the rule

$(rule\ (call\ level)\ abn\ then\ (.\ \{(current\ call\ level)\}))$.

The element ($\$c\ is\ function$) specifies functions. It is defined by the rule

$(rule\ (f\ is\ function)\ var\ (f)\ abn\ where\ (f\ is\ name)\ then\ (.\ \{(function\ f)\}))$.

In the case when the first element of the body $\$c^*$ is a variable declaration, the rule for the element $((collect\ body\ members)::\{1\}\ (va^*)\ c^*)$ is replaced by the rule

$(rule\ ((collect\ body\ members)::\{1\}\ (va_s)\ (var\ va\ t)\ c_s)\ var\ (va,\ t)\ seq\ (va_s,\ c_s)$
$abn\ (va,\ t)\ abn\ where\ ((va\ is\ name)\ and\ (t\ is\ type))$
$then\ (let::\{seq\}\ sc,\ cl\ be\ (scope),\ (if\ (sc = 0)\ then\ 0\ else\ (call\ level))$
$in\ (if\ (.\{(variable\ va\ sc\ cl)\})\ then\ und$
$else\ (\{(type\ va\ sc)\} := t::\{q\});\ (\{(variable\ va\ sc)\} := true);$
$((collect\ body\ members)::\{1\}\ (va_s\ va)\ c_s))))$.

The element $((collect\ body\ members)::\{1\}\ (va^*)\ c^*)$ is also redefined by the extra rules

$(rule\ ((collect\ body\ members)::\{1\}\ (function\ f\ (tna_s)\ t\ c_s1)\ c_s)\ var\ (f,\ t)$
$seq\ (tna_s,\ c_s1,\ c_s)\ abn\ (f,\ t)\ abn$
$where\ ((f\ is\ name)\ and\ \big(not\ (f\ is\ function)\big)\ and\ (t\ is\ type))$
$then\ ((collect\ member\ arguments)\ f\ tna_s);\ (\{((return\ type)\ f)\} \coloneqq t::\{q\});$
$(\{(body\ f)\} \coloneqq\ (c_s1) :: \{q\});\ (\{(function\ f)\}\ \coloneqq\ true);$
$((collect\ body\ members)::\{1\}\ c_s));$

$(rule\ ((collect\ body\ members)::\{1\}\ (function\ c_s))\ seq\ (c_s\ )\ abn\ then\ und)$.

Thus, body members in MPL3 are variables and functions.

The element $((collect\ member\ arguments)\ \$m\ \$tna^*)$ collects information about the typed arguments $\$tna^*$ of the body member $\$m$. It is defined by the rule

$(rule\ ((collect\ member\ arguments)\ m\ tna_s)\ var\ (m)\ seq\ (tna_s)\ abn$
$then\ ((collect\ member\ arguments\ 1)\ m\ 0\ tna_s))$.

The element $((collect\ member\ arguments\ 1)\ \ \$m\ \$n\ \$tna^*)$ is defined by the rules

$(rule\ ((collect\ member\ arguments\ 1)\ m\ n\ na\ t\ tna_s)\ var\ (m,\ n,\ na,\ t)\ \ seq\ (tna_s)$
$abn\ where\ ((m\ is\ name)\ and\ (n\ is\ nat)\ and\ (na\ is\ name)\ and\ (t\ is\ type))$
$then\ (let\ n1\ be\ (n+1)\ \ in\ (\{((argument\ type)\ m\ n1)\} \coloneqq t::\{q\}));$
$(\{(argument\ m\ n1)\}\ \coloneqq na::\{q\};\ ((collect\ member\ arguments\ 1)\ m\ n1\ tna_s))));$

$(rule\ ((collect\ member\ arguments\ 1)\ m\ n)\ var\ (m,\ n)\ \ abn$
$where\ ((m\ is\ name)\ \ and\ (n\ is\ nat))\ then\ (\{(arity\ m)\} \coloneqq n))$.

Thus, it collects information about function arguments.

The function declaration is defined by the rule:

$(rule\ (function\ c_s)\ seq\ (c_s)\ abn\ then)$.

The execution of the function declaration does not collect information about the declared function, since the corresponding actions are performed by the element $((collect\ body\ members)\ \$c^*)$.

The return statement is defined by the rule

$(rule\ (return\ c)\ var\ (c)\ val\ (c)\ exc\ (c, c::\{*\})\ abn$

$then\ (let::\{seq\}\ t1,\ t2\ be\ (.\ \{(current\ return\ type)\}),\ (type\ c::\{*\})$

$in\ (if\ (subtype\ t2\ t1)\ then\ (return:\{type\},\ c::\{*\}:\{value\})::\{exc\}\ else\ und))).$

The function call is defined by the rule

$(rule\ (call\ f\ a_s)\ var\ (f)\ seq\ (a_s)\ abn\ (f)\ abn$

$where\ ((f\ is\ function)\ and\ ((len\ a_s::\{q\}) = (.\{(arity\ f)\})))$

$then\ (let::\{und, seq\}\ av_s,\ b,\ cs,\ crt$

$be\ ((argument\ values)\ a_s),\ (body\ f\ av_s),\ (scope),\ (.\ \{(current\ return\ type)\})$

$in\ (call\ level\ ++);\ (\{(current\ scope)\}\ :=\ 0);\ b;$

$(catch::\{und\}\ v$

$(if\ (v\ is\ (not\ admissible\ function\ body\ value))\ then\ und);$

$(\{(current\ return\ type)\}\ :=\ crt::\{q\});\ (call\ level\ --);\ (\{(current\ scope)\}\ :=\ cs);$

$(if\ v\ matches\ (return:\{type\},\ v1:\{value\})::\{exc\}\ var\ (v1)\ then\ ((to\ value)\ v1::\{q\})$

$else\ ((to\ value)\ v::\{q\}))))).$

The element $(\$v\ is\ (\text{not admissible function body value}))$ specifies values that are not admissible when a function body exits. It is defined by the rule

$(rule\ v\ is\ (not\ admissible\ function\ body\ value))\ var\ (v)\ abn$

$then\ (not\ (v\ is\ exception))).$

Thus, the values that are not exceptions are not admissible in MPL3 when a function body exits.

The element $((argument\ values)\ \$a^*)$ returns the values of the arguments $\$a^*$. It is defined by the rules

$(rule\ ((argument\ values)\ a,\ a_s)\ var\ (a)\ seq\ (a_s)\ abn\ (a)\ abn$

$then\ (a\ .+\ ((argument\ values)\ a_s)));$

$(rule\ ((argument\ values))\ then\ ()).$

The element $(body\ \$f\ (\$v^*))$ creates the block with the body of the function $\$f$ followed the declarations of the local variables corresponding to the arguments of $\$f$ and the assignment statements assigning the values $\$v^*$ to these variables.

$(rule\ (body\ f\ (v_s))\ \ var\ (f)\ \ seq\ (v_s)\ abn\ (f)\ abn$

$then\ (block::\{q\}\ .+\ (((create\ local\ variables)\ \ f\ 0\ v_s)\ \ +\ (.\ \{(block\ f)\}))));$

The element $((create\ local\ variables)\ \$f\ \$n\ \$v^*)$ creates the declarations of the local variables corresponding to the arguments of $\$f$ and the assignment statements assigning the values $\$v^*$ to these variables. It is defined by the rules

$(rule\ ((create\ local\ variables)\ f\ n\ v\ v_s)\ var\ (f,\ n,\ v)\ seq\ (v_s)\ abn\ (f,\ n,\ v)\ abn$
$then\ (let :: \{seq\}\ n1,\ a,\ t$
$be\ (n+1),\ (.\ \{(argument\ f\ n1)\}),\ (.\ \{((argument\ type)\ f\ n1)\})\ in$
$((var\ a\ t)\ .+((a \setminus\coloneqq\ v::\{q\})\ .+((create\ local\ variables)\ f\ n1\ v_s)))));$
$(rule\ ((create\ local\ variables)\ f\ n)\ var\ (f,\ n)\ abn\ (f,\ n)\ abn\ then\ ()).$

The element $(call\ level ++)$ increases the value of the current call level by 1. It is defined by the rule

$(rule\ (call\ level\ ++)\ abn$
$then\ (\{(current\ call\ level)\} \coloneqq\ ((.\ \{(current\ call\ level)\}) + 1))).$

The element $(call\ level\ --)$ decreases the value of the current call level by 1. It is defined by the rule

$(rule\ (call\ level\ --)\ abn$
$then\ (\{(current\ call\ level)\} \coloneqq\ ((.\ \{(current\ call\ level)\}) - 1))).$

The element $(version\ \$na)$ is defined by the rules

$(rule\ (version\ va)\ var\ (va)\ abn\ (va)\ abn\ where\ (va\ is\ name)$
$then\ (let::\{seq\}\ sc,\ cl\ be\ (current\ scope),\ (current\ call\ level)\ in\ (version\ va\ sc\ cl)));$

The element $(version\ \$na\ \$sc\ \$cl)$ is defined by the rules

$(rule\ (version\ va\ y\ z)\ var\ (va,\ sc,\ cl)\ abn\ (va,\ sc,\ cl)\ abn$
$then\ (if\ (.\ \{(variable\ va\ sc\ cl)\})\ then\ sc$
$else\ (if\ (sc\ =\ 0)\ then\ und$
$else\ (let\ sc1\ be\ (sc-1)\ in\ (version\ va\ sc1\ cl))))).$

The variable access is defined by the rule

$(rule\ va\ var\ (va)\ abn\ (va)\ abn$
$then\ (let::\{und,\ seq\}\ sc,\ cl\ be\ (version\ va),\ (if\ (sc::\{q\} =\ 0)\ then\ 0\ else\ (call\ level))$
$in\ (.\ \{(value\ va\ sc\ cl)\}))).$

In the case when $\$c$ is a variable, the rule for the element $(type\ \$c)$ is replaced by the rule

$(rule\ (type\ va)\ var\ (va)\ abn\ (va)\ abn$
$then\ (let :: \{und,\ seq\}\ sc, cl$
$be\ (version\ va),\ (if\ (sc::\{q\} =\ 0)\ then\ 0\ else\ (call\ level))$
$in\ (.\ \{(type\ va\ sc\ cl)\}))).$

The variable assignment is defined by the rule

$(rule\ (va\ \backslash{:}= c)\ var\ (va,\ e)\ val\ (c)\ abn\ (v)\ exc\ (c,\ c{::}\{*\})\ abn$

$then\ (let{::}\{und\}\ sc,\ cl,\ t1,\ t2$

$be\ (variant\ va), (if\ (sc{::}\{q\} =\ 0)\ then\ 0\ else\ (current\ call\ level)),$

$(.\{(type\ va\ sc\ cl)\}), (type\ c{::}\{*\})$

$in\ (if\ (subtype\ t2\ t1)\ then\ (\{(value\ x\ sc\ cl)\}\ \coloneqq c{::}\{*\}{::}\{q\})\ else\ und))).$

The element $((delete\ variables)\ \$va^*)$ is defined by the rules

$(rule\ ((delete\ variables)\ va_s)\ seq\ (va_s)\ abn$

$then\ (let{::}\{seq\}\ sc,\ cl\ be\ (scope),\ (if\ (sc{::}\{q\} =\ 0)\ then\ 0\ else\ (call\ level))$

$in\ ((delete\ variables){::}\{1\}\ sc\ cl\ va_s))).$

The element $((delete\ variables){::}\{1\}\ \$va^*\ \$sc\ \$cl)$ is defined by the rules

$(rule\ \big((delete\ variables){::}\{1\}\ sc\ cl\ va\ va_s\big)\ var\ (sc,\ cl,\ va)\ seq\ (va_s)\ abn\ (sc,\ cl,\ va)$

$abn\ then\ (\{(value\ va\ sc\ cl)\} \coloneqq);\ (\{(type\ va\ sc\ cl)\} \coloneqq);$

$(\{(variable\ va\ sc\ cl)\} \coloneqq);\ ((delete\ variables){::}\{1\}\ sc\ cl\ va_s));$

$(rule\ ((delete\ variables){::}\{1\}\ sc\ cl)\ var\ (sc,\ cl)\ abn\ (sc,\ cl)\ abn\ then).$

7.4. MPL4: procedures

The MPL4 language [1] is an extension of MPL3 that adds the procedures feature: declarations and calls of procedures, and the exit statement. For simplicity, procedure overloading is prohibited. The sets of function names and procedure names are disjoint.

The element $(\$c\ is\ procedure)$ specifies procedures. It is defined by the rule

$(rule\ (pr\ is\ procedure)\ var\ (pr)\ abn\ where\ (pr\ is\ name)\ then\ (.\ \{(procedure\ pr)\})).$

The element $((collect\ body\ members){::}\{1\}\ \ (va^*)\ c^*)$ is redefined by the extra rules

$(rule\ ((collect\ body\ members){::}\{1\}\ (procedure\ pr\ (tna_s)\ c_s1)\ c_s)$

$var\ (pr)\ seq\ (tna_s,\ c_s1,\ c_s)\ abn\ (pr)\ abn$

$where\ ((pr\ is\ name)\ and\ (not\ (pr\ is\ procedure)))$

$then\ ((collect\ member\ arguments)\ pr\ tna_s);$

$(\{(body\ pr)\} \coloneqq\ (c_{s1})\ {::}\ \{q\});\ (\{(procedure\ pr)\}\ \coloneqq\ true);$

$((collect\ body\ members){::}\{1\}\ c_s));$

$(rule\ ((collect\ body\ members){::}\{1\}\ (procedure\ c_s))\ seq\ (c_s\ )\ abn\ then\ und).$

Thus, body members in MPL4 are variables, functions and procedures.

The element $((collect\ member\ arguments)\ \$m\ \$ta^*)$ is extended to procedures. Its definition is not changed.

The procedure declaration is defined by the rule:

$(rule\ (procedure\ c_s)\ seq\ (c_s)\ abn\ then)$.

The execution of the procedure declaration does not collect information about the declared procedure, since the corresponding actions are performed by the element $((collect\ body\ members)\ \$c^*)$.

The exit statement is defined by the rule

$(rule\ exit\ abn\ then\ (exit{:}\{type\}){:}{:}\{exc\})$.

The element $(\$v\ is\ (not\ admissible\ function\ body\ value))$ is redefined by the extra rule

$(rule\ ((exit{:}\{type\}) :: \{exc\}\ is\ (not\ admissible\ function\ body\ value))\ var\ (v)\ abn$
$then\ true)$.

Thus, exceptions initiated by $exit$ statements are not admissible in MPL4 when a function body exits.

The procedure call is defined by the rule

$(rule\ (call\ pr\ a_s)\ var\ (pr)\ seq\ (a_s)\ abn\ (pr)\ abn$
$where\ ((pr\ is\ procedure)\ and\ ((len\ a_s{:}{:}\{q\}) = (.\{(arity\ pr)\})))$
$then\ (let{:}{:}\{und, seq\}\ av_s,\ b,\ cs$
$be\ ((argument\ values)\ \ a_s),\ (body\ pr\ av_s),\ (scope)$
$in\ (call\ level + +);\ (\{(current\ scope)\} \ {:}{=}\ 0);\ b;$
$(catch{:}{:}\{und\}\ v$
$(if\ (v\ is\ (not\ admissible\ procedure\ body\ value))\ then\ und);$
$(call\ level - -);\ (\{(current\ scope)\} \ {:}{=}\ cs);$
$(if\ v\ matches\ (exit{:}\{type\}){:}{:}\{exc\}\ then\ true\ else\ ((to\ value)\ v{:}{:}\{q\})))))$.

The element $(\$v\ is$ (not admissible procedure $body\ value))$ specifies exceptions that are not admissible when a procedure call exits. It is defined by the rule

$(rule\ ((return{:}\{type\},\ v{:}\{value\}) :: \{exc\}\ is\ (not\ admissible\ procedure\ body\ value))$
$var\ (v)\ abn\ then\ true)$.

Thus, exceptions initiated by $return$ statements are not admissible in MPL4 when a procedure body exits.

The elements $(body\ \$f\ (\$v^*))$ and $((create\ local\ variables)\ \$f\ \$n\ \$v^*)$ are extended to procedures. Their definitions are not changed.

7.5. MPL5: pointers

The MPL5 language [1] is an extension of MPL4 that adds the pointers feature: the pointer types, the operations of pointer content access, variable address access and pointer deletion, statements of pointer content assignment and pointer deletion.

The element ($\$c\ is\ (pointer\ value)$) specifies pointers in MPL5. It is defined by the rule

$(rule\ (n::\{pointer\}\ is\ (pointer\ value))\ var\ (n)\ abn\ then\ (y\ is\ nat))$.

The element ($po\ is\ pointer$) specifies pointers in states. It is defined by the rule

$(rule\ (po\ is\ pointer)\ var\ (po)\ abn\ where\ (po\ is\ (pointer\ value))\ then\ (.\ \{(pointer\ po)\}))$.

The element ($\$po\ is\ (pointer\ \$t)$) specifies pointers with the given content type. It is defined by the rule

$(rule\ (po\ is\ (pointer\ t))\ var\ (po,\ t)\ abn\ where\ ((po\ is\ pointer)\ and\ (t\ is\ type))$

$then\ ((.\ \{((content\ type)\ po)\}) = t::\{q\}))$.

The element ($\$t\ is\ (pointer\ type)$) specifies pointer types. It is defined by the rule

$(rule\ ((pointer\ t)\ \ is\ (pointer\ type))\ var\ (t)\ abn\ then\ (t\ is\ type))$.

The element ($c\ is\ type$) is redefined by the extra rule

$(rule\ (c\ is\ type)\ abn\ then\ (c\ is\ (pointer\ type)))$.

The element $\$\$po$ is defined by the rule

$(rule\ x::\{pointer\}\ var\ (x)\ abn\ where\ (x\ is\ nat)\ then\ x::\{pointer\}::\{q\})$.

The element ($(content\ type)\ \$po$) returns the content type of $\$po$. It is defined by the rule

$(rule\ ((content\ type)\ po)\ var\ (po)\ abn\ where\ (po\ is\ pointer)$

$then\ (.\ \{((content\ type)\ po)\}))$.

The element ($type\ \$po$) is defined by the rule

$(rule\ (type\ po)\ var\ (po)\ abn\ where\ (po\ is\ pointer)$

$then\ (let\ t\ be\ (.\ \{((content\ type)\ po)\})\ in\ (pointer\ t)::\{q\}))$.

The pointer content access operation is defined by the rule

$(rule\ (*\ \ c)\ var\ (c)\ val\ (c)\ abn\ (c,\ c :: \{*\})\ abn$

$where\ (c::\{*\}\ is\ pointer)\ then\ (.\ \{(content\ c::\{*\})\}))$.

The pointer content assignment statement is defined by the rule

$(rule\ (*\ \ c1\ \coloneqq\ \ c2)\ var\ (c1,\ c2)\ val\ (c1,\ c2)\ abn\ (c1,\ c2,\ c1 :: \{*\},\ c2 :: \{*\})\ abn$

$where\ (c1::\{*\}\ is\ pointer)$

$then\ (let::\{und,\ seq\}\ t1,\ t2\ be\ (.\ \{((content\ type)\ c1::\{*\})\}),\ (type\ c2::\{*\})$

$in\ (if\ \ (subtype\ t2\ t1)\ then\ (\{(content\ c1::\{*\})\} := \ c2::\{*\}::\{q\})\ else\ und)))$.

The pointer addition operation is defined by the rule

$(rule\ (new\ (pointer\ t))\ var\ (t)\ abn\ (t)\ abn\ where\ (y\ is\ type)$

$then\ (let\ po\ be\ ((new\ instance)\ pointer)$

$in\ (\{((content\ type)\ po)\} := t::\{q\});\ (\{(pointer\ po)\} := true);\ po))$.

The pointer deletion operation is defined by the rule

$(rule\ (delete\ c)\ var\ (c)\ val\ (c)\ abn\ (c,\ c::\{*\})\ abn\ where\ (c::\{*\}\ is\ pointer)$
$then\ (\{(content\ c::\{*\})\}:=);\ (\{((content\ type)\ c::\{*\})\}:=);\ (\{(pointer\ c::\{*\})\}:=)).$

The element $(version\ \$na\ \$sc\ \$cl)$ is defined by the rules

$(rule\ (version\ va\ y\ z)\ var\ (va,\ sc,\ cl)\ abn\ (va,\ sc,\ cl)\ abn$
$then\ (if\ (.\ \{(pointer\ va\ sc\ cl)\})\ then\ sc$
$else\ (if\ (sc\ =\ 0)\ then\ und$
$else\ (let\ sc1\ be\ (sc-1)\ in\ (version\ va\ sc1\ cl))))).$

The variable address access operation is defined by the rule

$(rule\ (\&\ va)\ var\ (va)\ abn\ (va)\ abn$
$then\ (let::\{und,\ seq\}\ sc,\ cl\ be\ (version\ va),$
$(if\ (sc::\{q\}=0)\ then\ 0\ else\ (current\ call\ level))$
$in\ (if\ sc::\{q\}\ then\ (let\ po\ be\ (.\ \{(pointer\ va\ sc\ cl)\})\ in\ (.\ \{(content\ po)\}))$
$else\ und)).$

The variable access is defined by the rule

$(rule\ va\ var\ (va)\ abn\ (va)\ abn\ then\ (let::\{und\}\ po\ be\ (\&\ va)\ in\ (.\ \{(content\ po)\}))).$

The element (type va) is defined by the rule

$(rule\ va\ var\ (va)\ abn\ (va)\ abn$
$then\ (let::\{und\}\ po\ be\ (\&\ va)\ in\ (.\ \{((content\ type)\ po)\}))).$

In the case when the first element of the body $\$c^*$ is a variable declaration, the rule for the element $((collect\ body\ members)::\{1\}\ (va^*)\ c^*)$ is replaced by the rule

$(rule\ ((collect\ body\ members)::\{1\}\ (va_s)\ (var\ va\ t)\ c_s)\ var\ (va,\ t)\ seq\ (va_s,\ c_s)$
$abn\ (va,\ t)\ abn\ where\ ((va\ is\ name)\ and\ (t\ is\ type))$
$then\ (let::\{seq\}\ sc,\ cl\ be\ (scope),\ (if\ (sc::\{q\}=\ 0)\ then\ 0\ else\ (call\ level))$
$in\ (if\ (.\{(pointer\ va\ sc\ cl)\})\ then\ und$
$else\ (let\ po\ be\ ((new\ instance)\ pointer)$
$in\ (\{((content\ type)\ po)\}:=\ t::\{q\});\ (\{(pointer\ po)\}:=\ true);$
$((collect\ body\ members)::\{1\}\ (va_s\ va)\ c_s)))).$

The variable assignment is defined by the rule

$(rule\ (va\ \backslash:=\ c)\ var\ (va,\ c)\ abn\ (va,\ c,\ c::\{*\})\ abn$
$then\ (let::\{und,\ seq\}\ sc,\ cl,\ po,\ t1,\ t2$
$be\ (version\ va),\ (if\ (sc::\{q\}\ =\ 0)\ then\ 0\ else\ (call\ level)),\ (.\ \{(pointer\ va\ sc\ cl)\}),$
$(.\ \{((content\ type)\ po)\}),\ (type\ c::\{*\})$
$in\ (if\ (subtype\ t2\ t1)\ then\ (\{(content\ po)\}:=\ c::\{*\}::\{q\})\ else\ und))).$

The element $((delete\ variables)::\{1\}\ \$va^*\ \$sc\ \$cl)$ is defined by the rules

$(rule\ \big((delete\ variables)::\{1\}\ sc\ cl\ va\ va_s\big)\ var\ (sc,\ cl,\ va)\ seq\ (va_s)\ abn\ (sc,\ cl,\ va)$
$abn\ then\ (\{(pointer\ va\ sc\ cl)\} \coloneqq);\ ((delete\ variables)::\{1\}\ sc\ cl\ va_s));$

$(rule\ ((delete\ variables)::\{1\}\ sc\ cl)\ var\ (sc,\ cl)\ abn\ (sc,\ cl)\ abn\ then).$

7.6. MPL6: jump statements

The MPL6 language [1] is an extension of MPL5 that adds the jump statements feature: break statement, continue statement, goto statement and labelled statement.

The element $(e\ is\ label)$ specifies labels. It is defined by the rule

$(rule\ (e\ is\ label)\ var\ (e)\ abn\ then\ (x\ is\ normal)).$

The break statement is defined by the rule

$(rule\ break\ abn\ then\ (break:\{type\})::\{exc\}).$

The continue statement is defined by the rule

$(rule\ continue\ abn\ then\ (continue:\{type\})::\{exc\}).$

The goto statement is defined by the rule

$(rule\ (goto\ l)\ var\ (l)\ abn\ where\ (l\ is\ label)\ then\ (goto:\{type\},\ l:\{label\})::\{exc\}).$

The element $(\$v\ is\ (not\ admissible\ function\ body\ value))$ is redefined by the extra rules

$(rule\ ((break:\{type\})::\{exc\}\ is\ (not\ admissible\ function\ body\ value))\ abn\ then\ true);$

$(rule\ ((continue:\{type\})::\{exc\}\ is\ (not\ admissible\ function\ body\ value))\ abn$
$then\ true);$

$(rule\ ((goto:\{type\},\ l:\{label\})::\{exc\}\ is\ (not\ admissible\ function\ body\ value))$
$var\ (l)\ abn\ then\ (l\ is\ label)).$

The element $(\$v\ is\ (not\ admissible\ procedure\ body\ value))$ is redefined by the extra rules

$(rule\ ((break:\{type\})::\{exc\}\ is\ (not\ admissible\ procedure\ body\ value))\ abn\ then\ true);$

$(rule\ ((continue:\{type\})::\{exc\}\ is\ (not\ admissible\ procedure\ body\ value))\ abn$
$then\ true);$

$(rule\ ((goto:\{type\},\ l:\{label\})::\{exc\}\ is\ (not\ admissible\ procedure\ body\ value))$
$var\ (l)\ abn\ then\ (l\ is\ label)).$

Thus, exceptions initiated by $break$, $continue$ and $goto$ statements are not admissible in MPL6 when a function or procedure body exits.

The label statement is defined by the rule

$(rule\ (label\ l)\ var\ (l)\ abn\ where\ (l\ is\ label)$
$then\ (catch\ v\ (if\ v\ matches\ (goto:\{type\},\ l1:\{label\})::\{exc\}\ var\ (l1)$

$where\ ((l1\ is\ label)\ and\ (l1::\{q\} = l::\{q\}))\ then\ else\ v::\{q\}))).$

The block statement is defined by the rule

$(rule\ (block\ c_s)\ seq\ (c_s)\ abn$

$then\ (enter\ block);$

$(let::\{und,\ seq\}\ va_s,\ l_s\ be\ ((collect\ body\ members)\ c_s),\ ((collect\ labels)\ c_s)$

$in\ c_s;\ ((catch\ goto)\ (l_s)\ c_s);$

$(catch::\{und\}\ v\ ((exit\ block)\ va_s);\ v::\{q\}))).$

The element $((collect\ labels)\ \$c^*)$ collects labels from the label statements occurring in $\$c^*$. It is defined by the rules

$(rule\ ((collect\ labels)\ (label\ l)\ c_s)\ var\ (l)\ seq\ (c_s)\ abn$

$where\ (l\ is\ label)\ then\ (l::\{q\}\ .+\ ((collect\ labels)\ c_s)));$

$(rule\ ((collect\ labels)\ c\ c_s)\ var\ (c)\ seq\ (c_s)\ then\ ((collect\ labels)\ c_s));$

$(rule\ ((collect\ labels))\ then\ ()).$

The element $((catch\ goto)\ (\$l^*)\ \$c^*)$ catches the exceptions initiated by $goto$ statements when the current block exits. It is defined by the rule

$(rule\ ((catch\ goto)\ (l_s)\ c_s)\ seq\ (l_s,\ c_s)\ abn$

$then\ (catch\ v$

$(if\ v\ matches\ (goto:\{type\},\ l:\{label\})::\{exc\}\ var\ (l)\ where\ (l::\{q\}\ in::\{set\}\ (l_s)::\{q\})$

$then\ v::\{q\};\ c_s;\ ((catch\ goto)\ (l_s)\ c_s)\ else\ v::\{q\}))).$

The while statement is defined by the rules

$(rule\ (\backslash while\ con\ do\ c_s)\ var\ (con)\ seq\ (c_s)\ exc\ (con)\ abn$

$then\ (while::\{exc\}\ con\ do\ (block\ c_s;\ ((delete\ exception)\ continue)));$

$((delete\ exception)\ break)).$

7.7. MPL7: dynamic arrays

The MPL7 language [1] is an extension of MPL6 that adds the dynamic arrays feature: dynamic array types, the array element access operation and the array element assignment statement.

The element $(\$t\ is\ (dynamic\ array\ type))$ specifies dynamic array types. It is defined by the rule

$(rule\ ((array\ t)\ \ is\ (dynamic\ array\ type))\ var\ (t)\ abn\ then\ (t\ is\ type)).$

The element $(\$t\ is\ (array\ type))$ specifies array types. It is defined by the rule

$(rule\ (t\ is\ (array\ type))\ var\ (t)\ abn\ then\ (t\ is\ (dynamic\ array\ type))).$

The element $(c\ is\ type)$ is redefined by the extra rule

$(rule\ (c\ is\ type)\ abn\ then\ (c\ is\ (array\ type)))$.

The element $(\$e\ is\ (dynamic\ array))$ specifies dynamic arrays. It is defined by the rule

$(rule\ ((v{:}\{content\},\ t{:}\{type\}) :: \{(dynamic\ array)\}\ is\ (dynamic\ array))\ var\ (v,\ t)$

$abn\ where\ (t\ is\ type)\ then\ (v\ is\ ((array\ content)\ t)))$.

The element $(\$e\ is\ array)$ specifies arrays. It is defined by the rule

$(rule\ (e\ is\ array)\ var\ (e)\ abn\ then\ (e\ is\ (dynamic\ array)))$.

The element $(\$v\ is\ ((array\ content)\ \$t))$ is defined by the rules

$(rule\ (()\ is\ ((array\ content)\ t))\ var\ (t)\ abn\ then\ true)$;

$(rule\ ((v\ v_s)\ is\ ((array\ content)\ t))\ var\ (v, t, v_s)\ abn\ where\ (v\ is\ t)$

$then\ ((v_s)\ is\ ((array\ content)\ t)))$.

The element $\$ar$ is defined by the rule

$(rule\ ar\ var\ (ar)\ abn\ where\ (ar\ is\ array)\ then\ ar{::}\{q\})$.

The element $((element\ type)\ \$ar)$ returns the element type of $\$ar$. It is defined by the rule

$(rule\ ((element\ type)\ ar)\ var\ (ar)\ abn\ where\ (ar\ is\ array)\ \ then\ (ar\ .\ \{type\}))$.

The element $(\$dar\ is\ (array\ \$t))$ specifies dynamic arrays with the given element type. It is defined by the rule

$(rule\ (dar\ is\ (array\ t))\ var\ (dar,\ t)\ abn$

$where\ ((dar\ is\ (dynamic\ array))\ and\ (t\ is\ type))$

$then\ (((element\ type)\ dar) = t{::}\{q\}))$.

The element $(type\ \$dar)$ is defined by the rule

$(rule\ (type\ dar)\ var\ (dar)\ abn\ where\ (ar\ is\ (dynamic\ array))$

$then\ (let\ t\ be\ ((element\ type)\ dar)\ in\ (array\ t){::}\{q\}))$.

The array content access operation is defined by the rule

$(rule\ (content\ c)\ var\ (c)\ val\ (c)\ abn\ (c,\ c :: \{*\})\ abn\ where\ (c :: \{*\}\ is\ array)$

$then\ (c{::}\{*\}\ .\ \{content\}))$.

The len operation for arrays is defined by the rule

$(rule\ (len\ c)\ var\ (c)\ val\ (c)\ abn\ (c,\ c :: \{*\})\ abn\ where\ (c :: \{*\}\ is\ array)$

$then\ (content\ c{::}\{*\}{::}\{q\}))$.

The array element access operation is defined by the rule

$(rule\ (c1\ [\ c2\ ])\ var\ (c1,\ c2)\ val\ (c1,\ c2)\ abn\ (c1,\ c1{::}\{*\},\ c2,\ c2{::}\{*\})\ abn$

$where\ ((c1{::}\{*\}\ is\ array)\ and\ (c2{::}\{*\}\ is\ nat))\ then\ ((content\ c1{::}\{*\})\ ..\ c2{::}\{*\}))$.

The array element assignment statement is defined by the rule

$(rule\ (c1\ [\ c2\ ]\ \coloneqq c3)\ var\ (c1, c2, c3)\ val\ (c1, c2, c3)$

$abn\ (c1, c2, c3, c1::\{*\}, c2::\{*\}, c3::\{*\})\ abn$

$where\ ((c1::\{*\}\ is\ (dynamic\ array))\ and\ (c2::\{*\}\ is\ nat))$

$then\ (let::\{und,\ seq\}\ t1,\ t2\ be\ ((element\ type)\ c1::\{*\}),\ (type\ c3::\{*\})$

$in\ (if\ (subtype\ t2\ t1)$

$then\ (c1::\{*\}\ .\ \{content\} := \ ((content\ c1::\{*\})\ ..\ c2::\{*\}\ \coloneqq\ c3::\{*\}::\{q\}))$

$else\ und))).$

7.8. MPL8: static arrays

The MPL8 language [1] is an extension of MPL7 that adds the static arrays feature: static array types, the array element access operation and the array element assignment statement.

The element ($\$t\ is\ (static\ array\ type)$) specifies static array types. It is defined by the rule

$(rule\ ((array\ t\ n)\ is\ (static\ array\ type))\ var\ (t)\ abn\ then\ ((t\ is\ type)\ and\ (n\ is\ nat))).$

The element ($\$t\ is\ (array\ type)$) is redefined by the extra rule

$(rule\ (t\ is\ (array\ type))\ var\ (t)\ abn\ then\ (t\ is\ (static\ array\ type))).$

The element ($\$e\ is\ (static\ array)$) specifies dynamic arrays. It is defined by the rule

$(rule\ ((v{:}\{content\},\ t{:}\{type\})\ ::\ \{(static\ array)\}\ is\ (static\ array))\ var\ (v,\ t)\ abn$

$where\ (t\ is\ type)\ then\ (v\ is\ ((array\ content)\ t))).$

The element ($\$e\ is\ array$) is redefined by the extra rule

$(rule\ (e\ is\ array)\ var\ (e)\ abn\ then\ (e\ is\ (static\ array))).$

The element ($\$sar\ is\ (array\ \$t\ \$n)$) specifies static arrays with the given element type and length. It is defined by the rule

$(rule\ (sar\ is\ (array\ t\ n))\ var\ (sar,\ t)\ abn$

$where\ ((sar\ is\ (dynamic\ array))\ and\ (t\ is\ type))$

$then\ ((((element\ type)\ sar) = t::\{q\})\ and\ ((len\ (sar\ .\ \{content\})) = n))).$

The element ($type\ \$sar$) is defined by the rule

$(rule\ (type\ sar)\ var\ (sar)\ abn\ where\ (sar\ is\ (static\ array))$

$then\ (let::\{seq\}\ t,\ n\ be\ ((element\ type)\ sar),\ (len\ sar)\ in\ (array\ t\ n)::\{q\})).$

The array element assignment statement is redefined by the extra rule

$(rule\ (c1\ [\ c2\ ]\ \coloneqq c3)\ var\ (c1, c2, c3)\ val\ (c1, c2, c3)$

$abn\ (c1, c2, c3, c1\ ::\ \{*\}, c2\ ::\ \{*\}, c3\ ::\ \{*\})\ abn$

$where\ ((c1\ ::\ \{*\}\ is\ (static\ array))\ and\ (c2\ ::\ \{*\}\ is\ nat)\ and$

$(c2::\{*\}\ <=\ (len\ c1::\{*\})))$

$then\ (let{::}\{und,\ seq\}\ t1,\ t2\ be\ ((element\ type)\ c1{::}\{*\}),\ (type\ c3{::}\{*\})$
$in\ (if\ (subtype\ t2\ t1)$
$then\ (c1{::}\{*\}\,.\,\{content\} := ((content\ c1{::}\{*\})\,..\,c2{::}\{*\} \coloneqq c3{::}\{*\}{::}\{q\}))$
$else\ und))).$

7.9. MPL9: structures

The MPL9 language [1] is an extension of MPL8 that adds the structures feature: the structure types, the structure field access operation, structure declarations, and the structure field assignment statement.

The element $((collect\ body\ members\ 1)\ \ (va^{*})\ c^{*})$ is redefined by the extra rules

$(rule\ ((collect\ body\ members\ 1)\ (structure\ st\ (tna_s))\ var\ (st)\ seq\ (tna_s, c_s)\ abn\ (st)$
$abn\ where\ ((st\ is\ name)\ and\ (not\ (st\ is\ (structure\ type))))$
$then\ ((declare\ fields)\ st\ tfi_s);\ (\{((structure\ type)\ st)\} \coloneqq true));$
$((collect\ body\ members\ 1)\ c_s));$
$(rule\ ((collect\ body\ members\ 1)\ (structure\ st\ (tna_s))\ var\ (st)\ seq\ (tna_s,\ c_s)$
$abn\ (st)\ abn\ \mathrm{then}\ und).$

Thus, body members in MPL8 are variables, functions, procedures and structure types.

The element $((declare\ fields)\ \$st\ \$tna^{*})$ declares the fields of $\$st$. It is defined by the rules

$(rule\ ((declare\ fields)\ st\ na\ t\ tna_s)\ var\ (st,\ na, t)\ seq\ (tna_s)\ abn$
$where\ ((na\ is\ name)\ \ and\ (t\ is\ type))$
$then\ (\{(type\ na\ st)\} \coloneqq t{::}\{q\}); (\{(fields\ na\ st)\} \coloneqq true);\ ((declare\ fields)\ \$st\ tna_s);$
$(rule\ ((declare\ fields)\ st)\ var\ (st)\ abn\ then).$

The structure declaration is defined by the rule

$(rule\ (structure\ st\ (tna_s))\ var\ (st)\ seq\ (tna_s)\ abn\ then).$

The execution of the structure declaration does not collect information about the declared structure type, since the corresponding actions are performed by the element $((collect\ body\ members)\ \$c^{*})$.

The element $(\$na\ is\ (structure\ type))$ specifies structure types. It is defined by the rule

$(rule\ (na\ is\ (structure\ type))\ var\ (na)\ abn\ where\ (na\ is\ normal)$
$then\ (.\ \{((structure\ type)\ na)\})).$

The element $(c\ is\ type)$ is redefined by the extra rule

$(rule\ (c\ is\ type)\ abn\ \mathrm{then}\ (c\ is\ (structure\ type))).$

The element $(fields\ \$st)$ returns the sequence of fields of $\$st$. It is defined by the rule

$(rule\ (fields\ st)\ var\ (st)\ abn\ where\ (st\ is\ (structure\ type))$

$then\ (select\ fi\ wrt\ (v.\ \{(field\ fi\ st)\})\ var\ (v, fi)\ abn))$.

The element $(\$fi\ is\ (field\ \$st))$ checks that $\$fi$ is a field of $\$st$. It is defined by the rule

$(rule\ (fi\ is\ (field\ st))\ var\ (fi, st)\ abn\ where\ (st\ is\ (structure\ type))$

$then\ (.\ \{(field\ fi\ st)\}))$.

The element $(type\ \$fi\ \$st)$ returns the type of the field $\$fi$ of $\$st$. It is defined by the rule

$(rule\ (type\ fi\ st)\ var\ (fi, st)\ abn\ where\ (st\ is\ (structure\ type))\ then\ (.\ \{(type\ fi\ st)\}))$.

The element $(\$str\ is\ structure)$ specifies structures. It is defined by the rule

$(rule\ ((co{:}\{content\},\ st{:}\{type\}) :: \{structure\}\ is\ structure)\ var\ (co,\ st)\ abn$

$where\ (st\ is\ (structure\ type))\ then\ (co\ is\ ((structure\ content)\ st)))$.

The element $(\$e\ is\ ((structure\ content)\ \$st))$ is defined by the rules

$(rule\ (()\ is\ ((structure\ content)\ st))\ var\ (st)\ abn\ then\ true)$;

$(rule\ ((v{:}\{fi\}\ e_s)\ is\ ((structure\ content)\ st))\ var\ (v, fi, st, e_s)\ abn$

$where\ ((fi\ is\ (field\ st))\ and\ (let\ t\ be\ (type\ fi\ st)\ in\ (v\ is\ t)))$

$then\ ((e_s)\ is\ ((structure\ content)\ st)))$.

The element $(\$str\ is\ \$st))$ specifies structures with the given type. It is defined by the rule

$(rule\ (str\ is\ st)\ var\ (str,\ st)\ abn$

$where\ ((str\ is\ structure)\ and\ (st\ is\ (structure\ type)))$

$then\ ((str\ .\ \{type\}) = st{:}{:}\{q\}))$.

The element $\$str$ is defined by the rule

$(rule\ str\ var\ (str)\ abn\ where\ (str\ is\ structure)\ then\ str{:}{:}\{q\})$.

The element $(type\ \$str)$ is defined by the rule

$(rule\ (type\ str)\ var\ (str)\ abn\ where\ (str\ is\ structure)\ then\ (str\ .\ \{type\}))$.

The element $(fields\ \$str)$ returns the sequence of fields of $\$str$. It is defined by the rule

$(rule\ (fields\ str)\ var\ (str)\ abn\ where\ (c{:}{:}\{*\}\ is\ structure)$

$then\ (let\ st\ be\ (type\ str)\ in\ (fields\ st)))$.

The element $(\$fi\ is\ (field\ \$str))$ checks that $\$fi$ is a field of $\$str$. It is defined by the rule

$(rule\ (fi\ is\ (field\ str))\ var\ (fi, str)\ abn\ where\ (str\ is\ structure)$

$then\ (let\ st\ be\ (type\ str)\ in\ (fi\ is\ (field\ st))))$.

The element $(type\ \$fi\ \$str)$ returns the type of the field $\$fi$ of $\$str$. It is defined by the rule

$(rule\ (type\ fi\ str)\ var\ (fi, str)\ abn\ where\ (str\ is\ structure)$

$then\ (let\ st\ be\ (type\ \$str)\ in\ (type\ fi\ st)))$.

The structure field access operation is defined by the rule

$(rule\ (c\ .\ fi)\ var\ (c,\ fi)\ val\ (c)\ abn\ (c,\ c :: \{*\})\ abn$

$where\ ((c :: \{*\}\ is\ structure)\ and\ (fi\ is\ (field\ c :: \{*\})))$
$then\ ((str\ .\{content\})\ .\{fi\}))$.

The structure field assignment statement is defined by the rule

$(rule\ (c1 \setminus .\ fi := c2)\ var\ (c1, fi, c2)\ val\ (c1, c2)\ abn\ (c1, c2, c1::\{*\}, c2::\{*\})\ abn$
$where\ ((c1::\{*\}\ is\ structure)\ and\ (fi\ is\ (field\ c1::\{*\})))$
$then\ (let::\{und,\ seq\}\ t1,\ t2\ be\ (type\ fi\ c1::\{*\}),\ (type\ c2::\{*\})$
$in\ (if\ (subtype\ t2\ t1)$
$then\ (c1::\{*\}\ .\ \{content\}\ \coloneqq\ ((c1::\{*\}\ .\ \{content\})\ .\ \{fi\}\ \coloneqq\ c2::\{*\}::\{q\}))$
$else\ und)))$.

8. Conclusion

In the paper the notion of the conceptual operational semantics of a programming language has been proposed. The conceptual operational semantics of a programming language is an operational semantics of the programming language in terms of its conceptual model [3]. The special kind of CTSs, operational CTSs, oriented to specification of conceptual operational semantics of programming languages has been proposed, the language CTSL has been extended to this kind of CTSs, and the technique of the use of the extended CTSL as a domain-specific language for specification of conceptual operational semantics has been presented. We have conducted the incremental development of the conceptual operational semantics for the family of sample programming languages to illustrate this technique.

There is only one more approach which, like our approach, can specify both the structural and dynamic parts of the operational semantics of a programming language in quite general unified way. This approach is based on abstract state machines (ASMs) [4]. ASMs are the special kind of transition systems in which states are algebraic systems.

The key features of our approach in comparison with the approach based on ASMs are as follows.

The instantiation semantics and, in particular, states are directly described in CTSs in ontological terms whereas its conceptual structure can be only modelled by the appropriate choice of symbols of the signature of an algebraic system.

The transition relation in ASMs is built with the finite set of algebraic operations [5]. The transition relation in operational CTSs is based on the pattern matching on the conceptual structure of states.

The set of predefined executable elements of the CTSL language have analogues for the algebraic operations used in sequential ASMs, and also includes the elements for parsing the conceptual state structure.

The languages of executable specifications of abstract state machines AsmL [6] and XasM [7] are general-purpose languages of specification of discrete dynamic systems. They are not domain-specific languages oriented to development of operational semantics of programming languages in contrast to the CTSL language.

At present, our technique is applied to only the sequential fragments of programming languages. We plan to extend it to the concurrent fragments of programming languages.

References

1. Prinz A., Møller-Pedersen B., Fischer J. Object-Oriented Operational Semantics. In: Grabowski J., Herbold S. (eds) System Analysis and Modeling. Technology-Specific Aspects of Models. SAM 2016. Lecture Notes in Computer Science, vol 9959. Springer, Cham. P. 132-147.
2. Wider A. Model transformation languages for domain-specific workbenches // Ph.D. thesis, Humboldt-Universitat zu Berlin. 2015.
3. Anureev I.S., Promsky A.V. Conceptual transition systems and their application to development of conceptual models of programming languages // System Informatics. 2017. Vol. 9. P. 133–154.
4. 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.
5. Borger E., Stark R.F. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Secaucus. 2003.
6. AsmL: The Abstract State Machine Language. Reference Manual, 2002. http://research.microsoft.com/en-us/projects/asml/
7. Matthias Anlauff. XasM — An Extensible, Component-Based Abstract State Machines Language. http://xasm.sourceforge.net/XasmAnl00/XasmAnl00.html