

УНИВЕРСИТЕТ ГОРОДА ПЕРЕСЛАВЛЯ-ЗАЛЕССКОГО

СЛАЙДЫ К КУРСУ ЛЕКЦИЙ

“Метавычисления и их применение. Часть 1”

Преподаватель: д.ф.-м.н., Абрамов С.М.

Зав. кафедрой: д.ф.-м.н., Абрамов С.М.

http://www.botik.ru/~xsg/tsg_book/

ftp://ftp.botik.ru/rented/xsg/www/tsg_book/

2000

Введение

Исторические корни и основные идеи

- В.Ф.Турчин. Краткая научная биографическая справка.
- MST: общая теория эволюции В.Ф.Турчина. Квант эволюции, общие черты метасистем и метасистемных переходов, препосылки неограниченной эволюции, системы с неограниченными возможностями развиваться и системы, обреченные жить без развития...
- Основная идея: чтобы обеспечить эволюцию в мире программ используем теорию эволюции В.Ф.Турчина в программировании. Результат: **метавычисления**.
- Общие черты метапрограмм.
- Простейшие примеры метасистемных переходов (проекция Футамуры-Турчина).
- Более сложные метасистемы, многоуровневые метасистемы, более сложные примеры метасистемных переходов (проекция для нестандартных семантик).

Метавычисления: раздел программирования, посвященный разработке методов анализа и преобразования программ за счет реализации конструктивных метасистем (метапрограмм) над программами. Программы в метавычислениях рассматриваются как *объект* анализа и/или преобразования.

Базовые идеи:

- применение теории метасистем и метасистемных переходов;
- *процесс-ориентированный подход* к построению методов анализа и преобразования программ—разработка метапрограмм M , которые “наблюдают” за *процессами* вычисления исходной программы p (на одиночных данных d , на классах данных C) и управляют этими процессами.
- фиксация языка реализации R —на нем должны быть написаны все программы p , к которым будут применяться метапрограммы;
- если некоторая метапрограмма M написана на R , то к ней применима она сама (самоприменимость) или другие метапрограммы—метасистемная лестница.

Цель: реализация средств анализа программ и средств выполнения преобразований программ: (1) эквивалентных преобразований/оптимизаций и (2) построения новых программ, функции которых сложным образом определяются через функции исходных программ.

Актуальность проблемы

В.Ф.Турчин “Феномен науки”:

- *Метод современной науки по своему существу есть ничто иное, как построение формальных лингвистических моделей изучаемых явлений.*
- *Очередной шаг развития лингвистического моделирования, то есть очередной метасистемный переход в эволюции науки—использование компьютеров в процессе построения формальных лингвистических моделей. И это может быть сделано при помощи использования методов метавычислений.*

Сегодняшние результаты убеждают, что данная роль метавычислений действительно может быть осуществлена на практике при условии

⇒ **дальнейшего развития** методов метавычислений и методов их применения.

В области программирования речь идет о создании мощных средств автоматического анализа и преобразований программ и методов применения этих средств.

⇒ **Автоматизация программирования, обеспечение надежности ПО.**

Стремление к скорейшему завершению создания теории суперкомпиляции и практической реализации суперкомпилятора привело к тому, что некоторые понятия и методы метавычислений, некоторые вопросы применения метавычислений в программировании были недостаточно полно развиты, исследованы, обоснованы и изложены.

⇒ До сих пор **актуальная задача—закрыть данный пробел.**

Ближкие технологии—частичное выполнение, дефорестация, расширенная дефорестация, обобщенные частичные вычисления—*упрощения суперкомпиляции*.

⇒ Исследователи, ранее занимающиеся данными направлениями, **проявляют все больший интерес к метавычислениям.**

Структура курса



[1] С.Абрамов "Метавычисления и их применения"

[2] Л.Позлевич "Метавычисления и их применения. Суперкомпиляция"

Глава 1. Язык реализации: TSG

- Предметная область D:

$$D = (\text{ATOM } \text{Atoms}) \mid (\text{CONS } D \ D)$$

- Синтаксис TSG: плоский функциональный язык, типизированные программные переменные, два вида условий:
 - $(\text{EQA? } a\text{-exp}_1 \ a\text{-exp}_2);$
 - $(\text{CONS? } e\text{-exp } e.i \ e.j \ a.k).$
- Пример программы (проверка вхождения одной строки в другую);
- Семантика TSG—предъявлен *Gofer*-текст интерпретатора `int`.
- *Состояние* s_i вычисления программы p над данными d : $(term, env);$
- *Процесс* $\text{process}(p, d)$ вычисления программы p над данными d —последовательность переходов от одного состояния вычисления программы p к другому;

$$p \ d \xRightarrow{*} r \quad \left| \quad \begin{array}{ccccccc} & & p \ d & & & & r \\ & & \Downarrow & & & & \Uparrow \\ \text{process}(p, d) = & s_0 & \Rightarrow & s_1 & \Rightarrow & \dots & \Rightarrow s_n \end{array} \right.$$

- Трасса $\text{tr}(p, d)$ вычисления программы p над данными d —последовательность значений (для TSG это логические значения **TRUE** и **FALSE**), описывающих, какие варианты продолжения (какие ветви в конструкциях ветвления) избирались в процессе вычисления p над d в те моменты вычисления, когда в принципе были возможны различные варианты продолжения вычисления.

$$\text{tr}(p, d) = [b_1, b_2, \dots], \text{ где } b_i \in \{\text{TRUE}, \text{FALSE}\}$$

Фиксация языка реализации **для** конкретности изложения, точности построений и обоснований. Все построения и рассуждения могут быть повторены и для любого другого языка реализации со свойствами: *алгоритмическая полнота* и предметная область языка построена при помощи некоторого алфавита атомов и конечного набора *жестких* конструкторов.

1.1 Синтаксис программ на языке TSG

$prog ::= [def_1, \dots, def_n]$ -- $n \geq 1$
 $def ::= (DEFINE\ fname\ [parm_1, \dots, parm_n]\ term)$ -- $n \geq 0$
 $term ::= (ALT\ cond\ term_1\ term_2)$
 $\quad \quad | (CALL\ fname\ [arg_1, \dots, arg_n])$ -- $n \geq 0$
 $\quad \quad | e-exp$
 $cond ::= (EQA'\ a-exp_1\ a-exp_2)$
 $\quad \quad | (CONS'\ e-exp\ e-var_1\ e-var_2\ a-var)$
 $a-exp ::= a-val\ | a-var$
 $e-exp ::= a-val\ | a-var\ | e-var\ | (CONS\ e-exp_1\ e-exp_2)$
 $parm ::= a-var\ | e-var$
 $arg ::= a-exp\ | e-exp$
 $a-var ::= (PVA\ name)$
 $e-var ::= (PVE\ name)$

1.2 Программа проверки вхождения одной строки в другую

```

match = [ (DEFINE "Match" [e.substr,e.string]
            (CALL "CheckPos" [e.substr,e.string,e.substr,e.string])),
          (DEFINE "CheckPos" [e.subs,e.str,e.substring,e.string]
            (ALT (CONS' e.subs e.subs-head e.subs-tail a._)
                  (ALT (CONS' e.subs-head e._ e._ a.subs-head)
                        'FAILURE
                        (ALT (CONS' e.str e.str-head e.str-tail a._)
                              (ALT (CONS' e.str-head e._ e._ a.str-head)
                                    'FAILURE
                                    (ALT (EQA' a.subs-head a.str-head)
                                          (CALL "CheckPos" [e.subs-tail,e.str-tail,
                                                            e.substring,e.string])
                                          (CALL "NextPos" [e.substring,e.string]))))
                              'FAILURE)))
            'SUCCESS)),
          (DEFINE "NextPos" [e.substring,e.string]
            (ALT (CONS' e.string e._ e.string-tail a._)
                  (CALL "Match" [e.substring,e.string-tail])
                  'FAILURE)) ]

```

1.3 Вспомогательные функции для int

- `getDef fname prog` — в программе `prog` находит определение функции с именем `fname`.
- `mkEnv vars vals` — построение среды по списку переменных `vars` и списку значений `vals`. Пример:

```
mkEnv [a.x, a.y, e.z, e.u] ['A, 'B, (CONS 'D 'E), 'F ] ==
[ a.x:='A, a.y:='B, e.z:=(CONS 'D 'E), e.u:='F ]
```

- `t /. env` — замена в `t` вхождений переменных на их значения из `env`. Пример:

```
(CONS (CONS a.x e.z) a.y) /. [a.x:='A, a.y:='B, e.z:='C] ==
(CONS (CONS 'A 'C) 'B)
```

- `env +. env'` — обновление среды `env` средой `env'`. Пример:

```
[a.x:='A, a.y:='B, e.z:='C] +. [e.z:=(CONS 'D 'E), e.u:='F] ==
[e.z:=(CONS 'D 'E), e.u:='F, a.x:='A, a.y:='B]
```

1.4 Интерпретатор языка TSG (1/2)

```
int :: ProgR -> [EVal] -> EVal
```

```
int    p d  = eval s p
        where (DEFINE f prms _) : p' = p
              e = mkEnv prms d
              s = ((CALL f prms), e)
```

```
eval :: State -> ProgR -> EVal
```

```
eval  s@((CALL f args), e) p = eval s' p
        where DEFINE _ prms t' = getDef f p
              e' = mkEnv prms (args/.e)
              s' = (t',e')
```

```
eval  s@((ALT c t1 t2), e) p = case cond c e of
                                TRUE  ue -> eval (t1,e+.ue) p
                                FALSE ue -> eval (t2,e+.ue) p
```

```
eval  s@(exp,e)                p = exp/.e
```

1.5 Интерпретатор языка TSG (2/2 Функция cond)

```
data CondRes = TRUE Env | FALSE Env
```

```
cond :: Cond -> Env -> CondRes
```

```
cond (EQA' x y)      e = let x' = x/.e; y' = y/.e in
                      case (x', y') of
                        (ATOM a, ATOM b) | a==b -> TRUE [ ]
                        (ATOM a, ATOM b)      -> FALSE[ ]
```

```
cond (CONS' x vh vt va) e = let x' = x/.e in
    case x' of
        CONS h t      ->TRUE [vh:=h,vt:=t]
        ATOM a         ->FALSE[va:=x']
```

Глава 2. Представление множеств

“Традиционное”

$$2n + 1, \quad \text{где } n \geq 0$$

$$(5 \cos \phi, 5 \sin \phi), \quad \text{где } 0 \leq \phi \leq \frac{\pi}{2}$$

Для языка TSG

$$(\text{CONS } \mathcal{A}.1 \ \mathcal{E}.2, \text{ RESTR}[\mathcal{A}.1:\neq:'A'])$$

$$(\text{CONS } \mathcal{A}.1 \ \mathcal{A}.2, \text{ RESTR}[\mathcal{A}.1:\neq:\mathcal{A}.2])$$

Вводятся средства конструктивного представления множеств для языка TSG, исследуются и обосновываются их свойства.

- Конфигурационные переменные (*c-переменные*), двух типов (для языка TSG): $\mathcal{A}.index$, $\mathcal{E}.index$.
- Рестрикции—системы неравенств, задающих дополнительные ограничения на допустимые значения са-переменных.
- SR-выражение $(\mathbf{cx}, \mathbf{rs})$ представляет множество $\langle \mathbf{cx}, \mathbf{rs} \rangle$:

$$\langle \mathbf{cx}, \mathbf{rs} \rangle = \{ \mathbf{cx} /. \mathbf{s} \mid \mathbf{s} \in \text{FVS}(\mathbf{cx}, \mathbf{rs}) \}$$

где \mathbf{s} —подстановка, $(/.)$ —операция применения подстановки \mathbf{s} к \mathbf{cx} , $\text{FVS}(\mathbf{cx}, \mathbf{rs})$ —множество полных допустимых для $(\mathbf{cx}, \mathbf{rs})$ подстановок.

- Частные случаи SR-выражений: *классы* и *L-классы*—представляют множества данных языка TSG; *конфигурации*—представляют множества состояний вычисления TSG-программ над данными и т.п.

- *Подстановки*—для получения элементов представленного множества и для сужения множеств.
- *Сужения*—либо подстановка, либо рестрикция (дополнение к имеющимся ограничениям на значения с-переменных).
- *Суперпозиция* для подстановок (сужений).
- *Разбиение*—пара (c_1, c_2) элементарных сужений:
 $\langle sr/.c_i \rangle \subseteq \langle sr \rangle$, $\langle sr/.c_1 \rangle \cup \langle sr/.c_2 \rangle = \langle sr \rangle$, $\langle sr/.c_1 \rangle \cap \langle sr/.c_2 \rangle = \emptyset$.
- Операция *unify отождествления двух списков с-выражений*;
- *Отношение* (\preceq) *частичного порядка* на множестве SR-выражений: $sr_1 \preceq sr_2$, если sr_1 является результатом выполнения некоторого числа сужений sr_2 , при этом выполнено $\langle sr_1 \rangle \subseteq \langle sr_2 \rangle$.
- *Каноническая форма* (*canon C*) *записи класса C* и его свойства:
 - (a) $\langle \text{canon } C \rangle = \langle C \rangle$;
 - (b) множество $\{ \text{canon } C^* \mid C \preceq C^* \}$ канонических форм надклассов класса C —конечно для любого класса C .

2.1 Конфигурационные переменные, с-выражения

$ca-var ::= (CVA \ c-var-name)$

$ce-var ::= (CVE \ c-var-name)$

$c-var-name ::= integer$

$ca-expr ::= a-val$
 $\quad \quad \quad | \ ca-var$

$ce-expr ::= ca-expr$
 $\quad \quad \quad | \ ce-var$
 $\quad \quad \quad | \ (CONS \ ce-expr_1 \ ce-expr_2)$

$\mathcal{A}.index$ —сокращение для $(CVA \ index)$, $\mathcal{E}.index$ —сокращение для $(CVE \ index)$.

2.2 С-связи, с-среды, с-состояния

состояние: $(t, \ [var_1 := \underline{val_1} \ , \ \dots, \ var_n := \underline{val_n} \])$

обобщенное состояние: $(t, \ [var_1 := \underline{cexp_1} \ , \ \dots, \ var_n := \underline{cexp_n} \])$

2.3 Неравенства, рестрикции с-переменных

```
restr ::= INCONSISTENT
      | RESTR [ ineq1, ..., ineqn ] -- n ≥ 0
```

```
ineq ::= ca-expr :≠: ca-expr
```

Частные случаи неравенств: *противоречие*, *тавтология*.

```
isContradiction, isTautology :: InEq -> Bool
isContradiction (left :=/=: right) = (left == right)
```

```
isTautology (ATOM a :=/=: ATOM b) = (a /= b)
isTautology (left :=/=: right) = False
```

```
cleanRestr :: Restr -> Restr
cleanRestr INCONSISTENT = INCONSISTENT
cleanRestr (RESTR ineqs) = INCONSISTENT, if any isContradiction ineqs
cleanRestr (RESTR ineqs) = RESTR (nub(filter (not.isTautology) ineqs))
```

2.4 С-конструкции

'atom		} SR-база сх
cvar	ce — с-выражение	
(CONS ce ₁ ce ₂)		
[ce ₁ , ..., ce _n]	ces — список с-выражений	
pvar:=ce	cbind — с-связь	
[cbind ₁ , ..., cbind _n]	cenv — с-среда	
(term, cenv)	cst — с-состояние	
<hr/>		
ce ₁ :≠:ce ₂	ineq — неравенство	
[ineq ₁ , ..., ineq _n]	ineqs — список неравенств	
RESTR ineqs INCONSISTENT	rs — рестрикция	
<hr/>		
(сх, rs)	SR-выражение	

2.5 cvars

Для любой с-конструкции `cx` список с-переменных, входящих в `cx`, мы будем обозначать `cvars cx`. Непосредственно из определения синтаксиса SR-баз следует, что если в SR-базе нет ни одной с-переменной, то она является соответствующим объектом языка TSG.

```
class    CVARS a where cvars :: a -> [CExp]

instance CVARS CExp where
  cvars (ATOM _ )    = []
  cvars cvar@(CVA _) = [cvar]
  cvars cvar@(CVE _) = [cvar]
  cvars (CONS h t)   = nub ((cvars h)++(cvars t))

instance CVARS CBind where
  cvars (pvar := cx) = cvars cx

instance CVARS InEq where
  cvars (ax :=/= bx) = nub ((cvars ax)++(cvars bx))
```

```

instance CVARS Restr where
  cvars INCONSISTENT = []
  cvars (RESTR ineqs) = cvars ineqs

instance (CVARS a, CVARS b) => CVARS (a, b) where
  cvars (ax, bx) = nub ((cvars ax)++(cvars bx))

instance CVARS c => CVARS [c] where
  cvars cxs = nub (concat (map (cvars) cxs))

```

2.6 Подстановки

$subst ::= [c-sbind_1, \dots, c-sbind_n] \quad -- \quad n \geq 0$

$c-sbind ::= ca-var \rightarrow ca-expr$
 $\quad \quad \quad | \quad ce-var \rightarrow ce-expr$

Список всех с-переменных, которым $subst$ ставит в соответствие некоторое значение:
 $\text{dom } subst$

```

dom :: Subst -> [CExp]
dom subst = [ cvar | (cvar :-> _ ) <- subst ]

```

2.7 Применение подстановки

```
instance APPLY CExp Subst where
  (ATOM a) /.s = ATOM a
  (CONS h t)/.s = CONS (h/.s) (t/.s)
  cvar      /.s = cvar,    if cvar 'notElem' dom s
  cvar      /.s = head[ cexp | (cv:->cexp) <- s, cv==cvar ]
```

```
instance APPLY InEq Subst where
  (l:=/:r) /. subst = (l/.subst) :=/: (r/.subst)
```

```
instance APPLY CBind Subst where
  (pvar := cexpr) /. subst = pvar := (cexpr/.subst)
```

```
instance APPLY a subst => APPLY [a] subst where
  cxs /. subst = map (/.subst) cxs
```

```
instance (APPLY a subst, APPLY b subst) => APPLY (a,b) subst where
  (ax,bx) /. subst = ( (ax/.subst) , (bx/.subst) )
```

```
instance APPLY Restr Subst where
  INCONSISTENT /. subst= INCONSISTENT
  (RESTR ineqs) /. subst= cleanRestr(RESTR(ineqs/.subst))
```

Пример 1

Рассмотрим применения подстановок к рестрикции. Пусть:

```
ineqs = [ A.2:≠:A.1, A.2:≠:'C, A.3:≠:'A ]
rs = RESTR ineqs
subst1 = [ A.1:->'A, A.2:->A.3 ]
subst2 = [ A.1:->'B, A.3:->'B ]
subst3 = [ A.2:->A.1, A.3:->'A ]
```

Тогда, будем иметь следующие результаты применения подстановок:

```
inqs /. subst1 = [A.3:≠:'A, A.3:≠:'C, A.3:≠:'A]
rs /. subst1 = RESTR [A.3:≠:'A, A.3:≠:'C]
inqs /. subst2 = [A.2:≠:'B, A.2:≠:'C, 'B:≠:'A]
rs /. subst2 = RESTR [A.2:≠:'B, A.2:≠:'C]
inqs /. subst3 = [A.1:≠:A.1, A.1:≠:'C, 'A:≠:'A]
rs /. subst3 = INCONSISTENT
```

2.8 Свойства подстановок

- подстановка сохраняет “а-” и “е-”тип с-выражения;
- подстановка сохраняет вид с-конструкции (са-выражение, се-выражение, список с-выражений длины n , с-связь, с-среда, с-состояние, неравенство, противоречие, тавтология, список неравенств длины n , рестрикция, SR-выражение.);
- полная подстановка приводит SR-базу (са-выражение, се-выражение, список с-выражений длины n , с-связь, с-среда, с-состояние) к значению соответствующего вида (а-значение, е-значение, список значений длины n , связь, среда, состояние);
- в силу “жесткости” конструкторов, при помощи которых из атомов и с-переменных конструируются SR-базы, выполнено:

Утверждение 1

Если cx —SR-база, s_1, s_2 —подстановки, то $cx/.s_1 == cx/.s_2$ тогда и только тогда, когда для любой с-переменной cv из cx выполнено $cv/.s_1 == cv/.s_2$:

$$(cx/.s_1 == cx/.s_2) \iff \bigwedge_{cv \in cvars \ cx} (cv/.s_1 == cv/.s_2)$$

2.9 Отождествление с-выражений

Алгоритм отождествления `unify` по двум спискам с-выражений `ces1` и `ces2` строит следующий результат:

1. `(False, [])`—если не существует подстановки `s` такой, что `ces1/.s=ces2`.
2. `(True, s)`—если существует такая подстановка. При этом, `s`—подстановка, такая, что `ces1/.s=ces2` и `dom s=cvars ces1`.

```
type UnifRes = (Bool, Subst)
```

```
fail = (False, []) :: UnifRes
```

```
unify :: CExps -> CExps -> UnifRes
```

```
unify ces1 ces2
```

```
  | (length ces1) /= (length ces2) = fail
```

```
  | otherwise                       = unify' [ ] chs
```

```
      where chs = zipWith(\a b->(a:=:b))ces1 ces2
```

```
unify' :: Clashes -> Clashes -> UnifRes
```

```
unify' rchs [] = (True, subst)
```

```
      where subst = (map (\(a:=:b)->(a:->b)) rchs)
```



```

unify' rchs chs@(ch:chs') =
  case ch of
    ATOM a      ::=ATOM b | a==b -> unify' rchs chs'
    ATOM a      ::=cex          -> fail
    cvar@(CVA _) ::=caex@(ATOM _) -> moveCl rchs chs
    cvar@(CVA _) ::=caex@(CVA _) -> moveCl rchs chs
    cvar@(CVA _) ::=cex          -> fail
    cvar@(CVE _) ::=cex          -> moveCl rchs chs
    CONS a1 b1   ::=CONS a2 b2   -> unify' rchs (p++chs')
                                   where p=[a1::a2,b1::b2]
    CONS a1 b1   ::=cex          -> fail

```

```

moveCl:: Clashes -> Clashes -> UnifRes

```

```

moveCl rchs chs@(ch@(cvar::cexp):chs') =
  case [ y | (x::y)<-rchs, x==cvar ] of
    [ ]          -> unify' (ch:rchs) chs'
    [y] | y==cexp -> unify' rchs      chs'
    [y] | otherwise -> fail

```

2.10 Представляемое множество

Определение 1

Пусть $se=(cx,rs)$ —SR-выражение. Множество полных допустимых для se подстановок обозначим через $FVS(se)=FVS(cx,rs)$.

Будем использовать se для представления множества синтаксических объектов языка TSG. Это множество будем обозначать $\langle se \rangle$ и $\langle cx,rs \rangle$. Определим это множество следующим образом:

$$\langle se \rangle = \langle cx,rs \rangle = \{ cx/.s \mid s \in FVS(cx,rs) \}$$

Примеры:

- $(\mathcal{E}.1, RESTR[])$
- $(\mathcal{A}.1, RESTR[])$
- $\langle (CONS \ 'A \ \mathcal{E}.1), RESTR[] \rangle$
- $\langle (CONS \ \mathcal{A}.1 \ \mathcal{E}.2), RESTR [\mathcal{A}.1:\neq: 'A] \rangle$
- Пусть $e \in Eval$, $sngl(e) = (e, RESTR [])$. Тогда, $\langle e, RESTR[] \rangle$ —одноэлементное множество: $\langle e, RESTR[] \rangle = \{e\}$.
- Пусть se —любая SR-база. Тогда, $\langle se, INCONSISTENT \rangle = \emptyset$.

2.11 Классы и L-классы

Определение 2

Пусть cx —SR-база. Будем называть cx *линейной*, если в cx нет се-переменных, имеющих повторные вхождения в cx . Линейные с-выражения будем называть *Lс-выражениями*, линейные списки с-выражений будем называть *L-списками с-выражений*.

Определение 3

Пусть ces —список с-выражений, rs —рестрикция. Тогда $C=(ces,rs)$ будем называть *классом*. Если ces —L-список с-выражений, то класс $C=(ces,rs)$ будем называть *L-классом*.

Классы используются для изображения множеств входных данных TSG-программ.

Пример 2

Рассмотрим класс:

$$C=([A.1, (CONS \mathcal{E}.2 A.3)], RESTR[A.1:\neq:A.3])$$

Тогда C представляет множество всех данных, имеющих вид

$$[a_1, (CONS e_2 a_3)],$$

где e_2 —произвольное е-значение, a_1 и a_3 —два несовпадающих а-значения.

2.12 Конфигурации

Определение 4

Будем называть *конфигурацией* SR-выражение `conf`, имеющее вид

$$\text{conf} = ((t, \text{cenv}), \text{rs}),$$

где (t, cenv) —с-состояние, rs —рестрикция, t —программный терм, cenv —с-среда.

Конфигурации представляют множества состояний вычислений. Ниже будет видно, что введенных средств **достаточно** для представления всех множеств, которые возникают при метавычислениях.

2.13 Суперпозиция подстановок

```
(.*) :: Subst -> Subst -> Subst
sa *. sb = [ cvar:->((cvar/.sa)/.sb) | cvar<-dom_sa_sb ]
      where
        dom_sa_sb= nub ((dom sa)++(dom sb))
        -- объединение без повторов dom sa и dom sb
```

Утверждение 2

Пусть $sa, sb, s=sa.*.sb$ —подстановки. Тогда для произвольной с-конструкции cx выполнено: $cx/.s=(cx/.sa)/.sb$.

2.14 Сужения

$$\begin{aligned} \text{contr} &::= S \text{ subst} \\ &\quad | R \text{ restr} \end{aligned}$$

```
instance APPLY c Subst => APPLY (c, Restr) Contr where
  (cx, rs) /. (S subst) = (cx/.subst, rs/.subst)
  (cx, rs) /. (R restr) = (cx, rs+.restr)
```

Утверждение 3

Пусть (cx, rs) —SR-выражение, c —сужение, $(cx', rs') = (cx, rs) /. c$. Тогда, $\langle cx', rs' \rangle \subseteq \langle cx, rs \rangle$.

Определение 5

Пусть cx, cx' —SR-выражения. Будем использовать обозначения $cx' \preceq cx$ и $cx \succeq cx'$, если существуют такие сужения c_1, \dots, c_n , $n \geq 0$, что

$$cx' = cx /. c_1 /. c_2 \dots /. c_n.$$

Утверждение 4

Отношение \preceq является частичным порядком на множестве SR-выражений.

Утверждение 5

Пусть cx, cx' —SR-выражения и $cx' \preceq cx$, тогда $\langle cx' \rangle \subseteq \langle cx \rangle$.

Утверждение 6

Данное $d \in D$ принадлежит множеству $\langle C \rangle$, представленному классом C тогда, и только тогда, когда $\text{sngl}(d) \preceq C$.

$\text{idC} = S [] \quad :: \text{Contr}$
 $\text{emptC} = R \text{ INCONSISTENT} \quad :: \text{Contr}$

Утверждение 7

Пусть (cx, rs) —SR-выражение. Тогда, выполнено:

$(cx, rs) /. \text{idC} = (cx, rs), \quad \langle (cx, rs) /. \text{idC} \rangle = \langle cx, rs \rangle;$
 $(cx, rs) /. \text{emptC} = (cx, \text{INCONSISTENT}), \quad \langle (cx, rs) /. \text{emptC} \rangle = \emptyset.$

Утверждение 8

Если x и x' —SR-выражения, то $x' \preceq x$, тогда и только тогда, когда существуют два сужения: $c1 = (S \ s)$ и $c2 = (R \ r)$, такие, что $x' = x /. c1 /. c2$.

Идея доказательства:

$SS \rightarrow S: \quad /. (S \ s1) /. (S \ s2) = /. (S \ s1.*.s2);$
 $RR \rightarrow R: \quad /. (R \ r1) /. (R \ r2) = /. (R \ r1+.r2);$
 $RS \rightarrow SR: \quad /. (R \ r) /. (S \ s) = /. (S \ s) /. (R \ (r /. s)).$

2.15 Каноническая форма класса

Причины неоднозначности представления классом множества данных:

1. с-переменные, входящие в класс, являются *свободными параметрами*, то есть согласованное переименование всех с-переменных по сути не изменяет класса;
2. рестрикция класса—система неравенств, а каждое неравенство—неупорядоченная пара “*левая часть*: \neq :*правая часть*”, то есть перестановка левой и правой части неравенства и переупорядочивание неравенств в рестрикции класса по сути не изменяет класса;
3. в рестрикции неравенства могут входить многократно—преобразования рестрикций, выполняемые функцией `cleanRestr`, по сути, не изменяет класса.

Определим отношение ($<$) на множестве с-выражений. Неравенства будем сравнивать по лексикографическому порядку:

$$((x_1:\neq:y_1)<(x_2:\neq:y_2)) \stackrel{\text{def}}{\iff} (x_1<x_2) \quad || \quad ((x_1==x_2)\&\&(y_1<y_2)).$$

Вид x	Вид y	Определение отношения (<)
(ATOM a)	(ATOM b)	$(x < y) \stackrel{\text{def}}{\iff} (a < b)$
(ATOM a)	(CVA j)	$y < x$
(ATOM a)	(CVE j)	$y < x$
(ATOM a)	(CONS p q)	$x < y$
(CVA i)	(ATOM b)	$x < y$
(CVA i)	(CVA j)	$(x < y) \stackrel{\text{def}}{\iff} (i < j)$
(CVA i)	(CVE j)	$(x < y) \stackrel{\text{def}}{\iff} (i < j)$
(CVA i)	(CONS p q)	$x < y$
(CVE i)	(ATOM b)	$x < y$
(CVE i)	(CVA j)	$(x < y) \stackrel{\text{def}}{\iff} (i < j)$
(CVE i)	(CVE j)	$(x < y) \stackrel{\text{def}}{\iff} (i < j)$
(CVE i)	(CONS p q)	$x < y$
(CONS r s)	(ATOM b)	$y < x$
(CONS r s)	(CVA j)	$y < x$
(CONS r s)	(CVE j)	$y < x$
(CONS r s)	(CONS p q)	$(x < y) \stackrel{\text{def}}{\iff}$ $(r < p) \mid \mid ((r == p) \&\& (s < q))$

Определение 6

Класс \mathcal{C}^* будем называть *канонической формой* класса \mathcal{C} , если:

1. \mathcal{C}^* может быть получен из класса \mathcal{C} путем переименования с-переменных и переупорядочивания неравенств в рестрикции (в соответствии с $<$);
2. $\text{cvars } \mathcal{C}^*$ имеет вид: $[\text{vt}_1 \ 1, \text{vt}_2 \ 2, \dots, \text{vt}_n \ n]$, где $\text{vt}_i \in \{\text{CVA}, \text{CVE}\}$ —конструктор с-переменной. То есть, для с-переменных в \mathcal{C}^* используются в качестве индексов все числа от 1 до n , где n —число различных с-переменных. И если первое вхождение с-переменной v встречается в \mathcal{C}^* раньше (левее) первого вхождения с-переменной v' , то должно выполняться $v < v'$.
3. Для любого неравенства $x \neq y$ в \mathcal{C}^* выполнено $x < y$ и все неравенства в рестрикции отсортированы по отношению ($<$).

Утверждение 9

1. Если \mathcal{C}^* является канонической формой \mathcal{C} , то $\langle \mathcal{C}^* \rangle = \langle \mathcal{C} \rangle$.
2. Существует алгоритм `canon`, который для любого класса \mathcal{C} за конечное число шагов строит класс $(\text{canon } \mathcal{C}^*)$, являющийся канонической формой \mathcal{C} .

2.16 Подклассы и надклассы

Определение 7

Пусть $\mathcal{C}_1, \mathcal{C}_2$ —классы. Тогда,

- класс \mathcal{C}_2 будем называть *подклассом* класса \mathcal{C}_1 , а класс \mathcal{C}_1 будем называть *надклассом* класса \mathcal{C}_2 , если $\mathcal{C}_2 \preceq \mathcal{C}_1$;
- класс \mathcal{C}_2 будем называть *собственным подклассом* класса \mathcal{C}_1 , а класс \mathcal{C}_1 будем называть *собственным надклассом* класса \mathcal{C}_2 , и будем обозначать $\mathcal{C}_2 \prec \mathcal{C}_1$, если $\mathcal{C}_2 \preceq \mathcal{C}_1$ и канонические формы для классов \mathcal{C}_1 и \mathcal{C}_2 не совпадают:
 $\text{canon } \mathcal{C}_2 \neq \text{canon } \mathcal{C}_1$.

Теорема 10

Если множество атомов AVal конечно, то множество канонических форм надклассов класса \mathcal{C}

$$\{ \text{canon } \mathcal{C}^* \mid \mathcal{C} \preceq \mathcal{C}^* \}$$

конечно для любого класса \mathcal{C} .

2.17 Разбиения

Определение 8

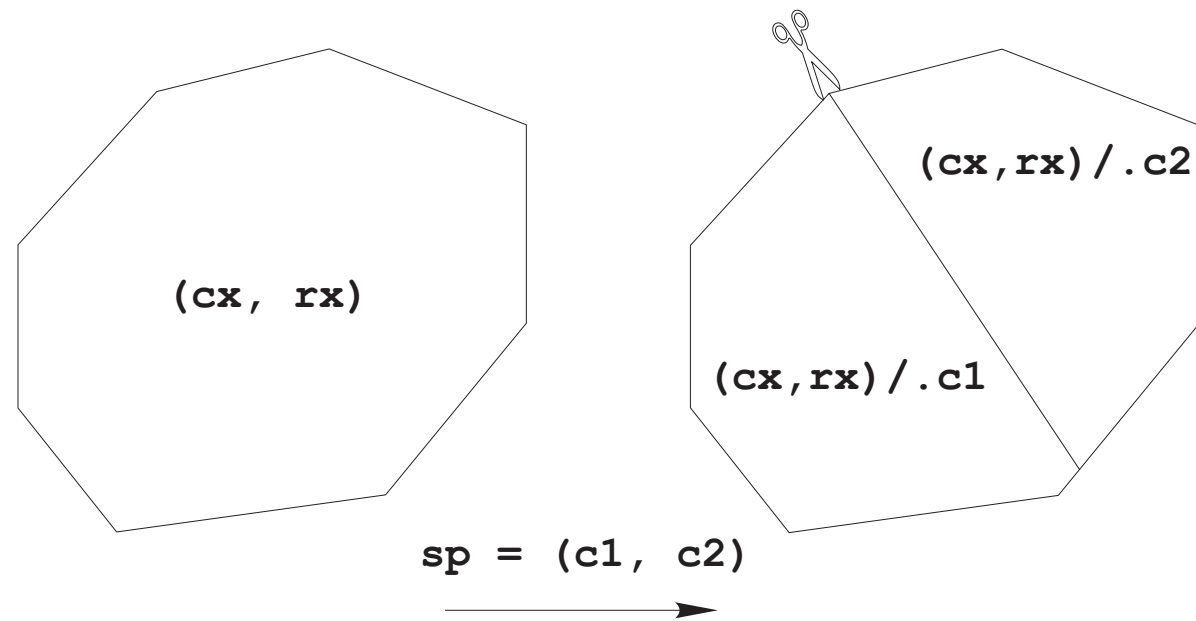
Пусть (cx, rs) —SR-выражение. Тогда пару сужений $sp=(c_1, c_2)$ назовем *разбиением* для (cx, rs) , а сами сужения c_1 и c_2 назовем *элементарными* сужениями для (cx, rs) , если пара (c_1, c_2) или пара (c_2, c_1) имеет один из следующих видов:

1. $(\text{idC} \quad , \text{emptC});$
2. $(S [\mathcal{E}.i:->(\text{CONS } \mathcal{E}.m \ \mathcal{E}.n)] , S [\mathcal{E}.i:->\mathcal{A}.p]);$
3. $(S [\mathcal{A}.j:->\mathcal{A}.k] \quad , R \text{ RESTR}[\mathcal{A}.j:\neq:\mathcal{A}.k]);$
4. $(S [\mathcal{A}.j:->'atom] \quad , R \text{ RESTR}[\mathcal{A}.j:\neq:'atom]).$

где $\mathcal{E}.i$, $\mathcal{A}.j$, $\mathcal{A}.k$ —переменные из cx , то есть они входят в $cvars \ cx$; $\mathcal{E}.m$, $\mathcal{E}.n$, $\mathcal{A}.p$ —новые c -переменные для cx , то есть индексы m , n , p не используются в качестве индексов c -переменных в cx ; $'atom$ —некоторый атом.

Теорема 11

Пусть $sp=(c_1, c_2)$ —разбиение для SR-выражения x , $x_1=x/.c_1$, $x_2=x/.c_2$. Тогда $\langle x_1 \rangle$ и $\langle x_2 \rangle$ —разбиение множества $\langle x \rangle$: $\langle x_1 \rangle \subseteq \langle x \rangle$, $\langle x_2 \rangle \subseteq \langle x \rangle$, $\langle x_1 \rangle \cup \langle x_2 \rangle = \langle x \rangle$, $\langle x_1 \rangle \cap \langle x_2 \rangle = \emptyset$.



Глава 3. Дерево процессов

Понятие *дерева процессов* и *алгоритм построения дерева процессов*—базовое понятие и базовый алгоритм метавычислений.

- *Дерево tree конфигураций*: ориентированное (возможно бесконечное) дерево, каждому узлу n которого приписана некоторая конфигурация c ; каждому ребру a , выходящему из некоторого узла n с конфигурацией c , приписано сужение cnt ; причем, если из узла n с конфигурацией c выходит $k \geq 1$ ребер a_1, \dots, a_k с сужениями $\text{cnt}_1, \dots, \text{cnt}_k$, то множества $\langle c/. \text{cnt}_i \rangle$ попарно не пересекаются, а их объединение совпадает с $\langle c \rangle$.
- Путь w в дереве конфигураций *tree* представляет множество $\langle w \rangle$ последовательностей P переходов из одного состояния s_i вычисления в языке TSG к другому: $\langle w \rangle = \{ P \mid P \text{ отождествимо с } w \}$, где “ P отождествимо с w ” (или, “ w представляет P ”) означает, что:

$$\begin{aligned} w &= c_0 \xrightarrow{\text{cnt}_0} c_1 \xrightarrow{\text{cnt}_1} \dots c_n \dots \\ P &= s_0 \Rightarrow s_1 \Rightarrow \dots s_n \dots \end{aligned}$$

- (a) P и w имеют одинаковые длины и (b) для каждого s_i выполнено: $s_i \in \langle c_i \rangle$, и если s_i не последнее в P , то $s_i \in \langle c_i/. \text{cnt}_i \rangle$.

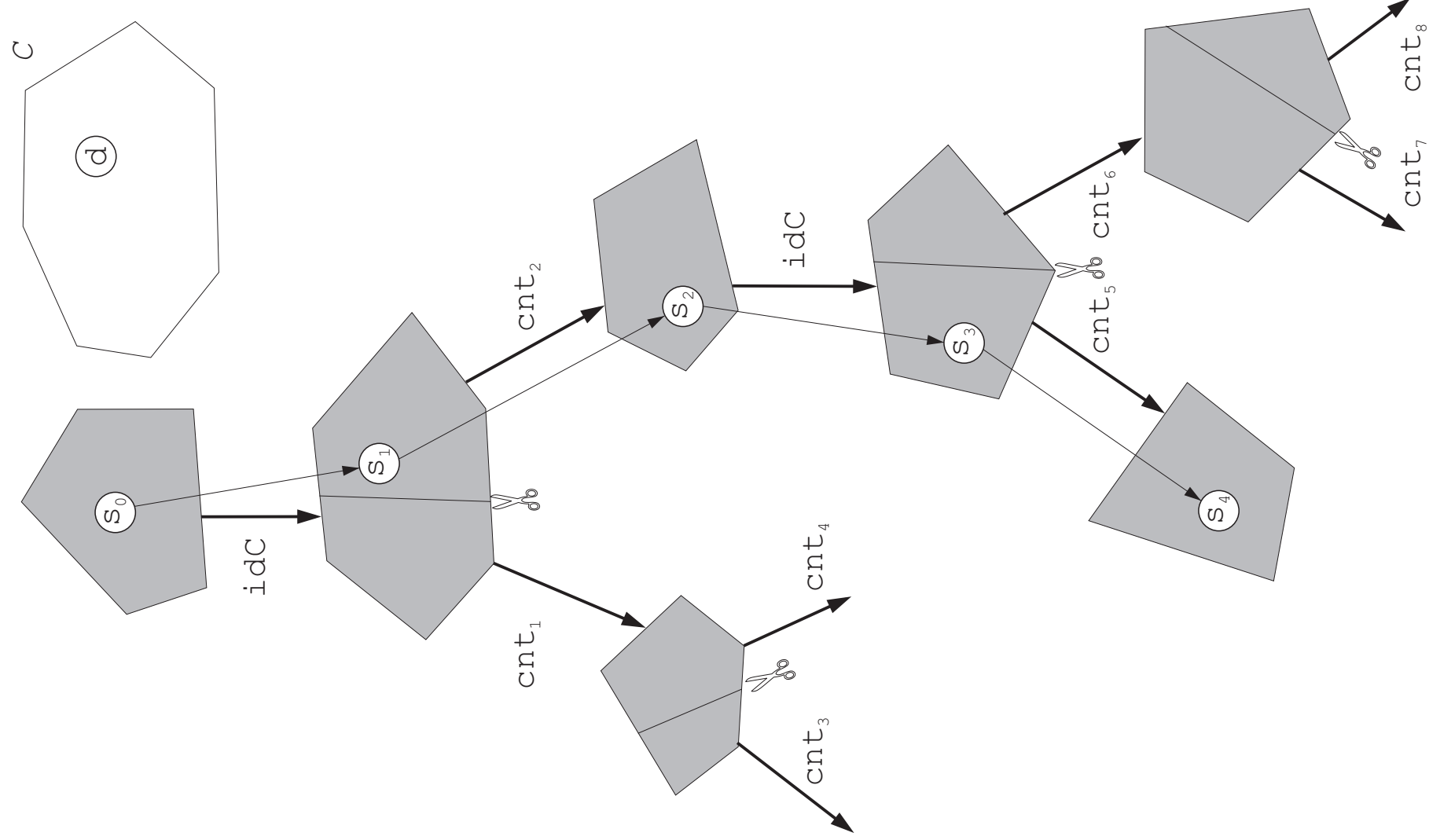
Дерево **tree** представляет множество $\langle \text{tree} \rangle$ последовательностей P —объединение всех $\langle w \rangle$, где w —путь в **tree** из корневой вершины либо (а) имеющий бесконечную длину, либо (б) завершающийся в листовой вершине дерева **tree**.

- Множество $\mathcal{P}(p, \mathcal{C}) \stackrel{\text{def}}{=} \{ \text{process}(p, d) \mid d \in \langle \mathcal{C} \rangle \}$ процессов вычисления p на данных из некоторого класса \mathcal{C} (на обобщенном данном программы p).
- *Дерево процессов вычисления p на данных из $\langle \mathcal{C} \rangle$* —дерево конфигураций такое, что $\mathcal{P}(p, \mathcal{C}) \subseteq \langle \text{tree} \rangle$. Если при этом для каждой вершины дерева существует $d \in \mathcal{C}$, такой, что эта вершина *используется* в представлении $\text{process}(p, d)$, то **tree**—*перфектное дерево*.

Приводятся и обосновываются:

- алгоритм **ptr** построения дерева **tree=ptr** p \mathcal{C} процессов вычисления p на данных из $\langle \mathcal{C} \rangle$;
- алгоритм **xptr** построения перфектного дерева **ptree=xptr** p \mathcal{C} процессов вычисления p на данных из $\langle \mathcal{C} \rangle$.

Дерево процессов



3.1 Синтаксис представления дерева процессов

$$\begin{aligned} tree & ::= (\text{LEAF } conf) \\ & \quad | (\text{NODE } conf [branch_1, \dots, branch_n]) \quad -- \quad n \geq 0 \end{aligned}$$

$$branch ::= (contr, tree)$$

3.2 Вспомогательные функции в алгоритме ptr

$$\text{mkCAVar}, \text{mkCEVar} :: \text{FreeIndx} \rightarrow (\text{CVar}, \text{FreeIndx})$$

$$\text{mkCEVar } i = ((\text{CVE } i), i+1)$$

$$\text{mkCAVar } i = ((\text{CVA } i), i+1)$$

$$\text{splitA} :: \text{CVar} \rightarrow \text{CExp} \rightarrow \text{Split}$$

$$\text{splitA } cv@(\text{CVA } _) \text{ caexp} = (S[cv:->\text{caexp}], R(\text{RESTR}[cv:=/:caexp]))$$

$$\text{splitE} :: \text{CVar} \rightarrow \text{FreeIndx} \rightarrow (\text{Split}, \text{FreeIndx})$$

$$\begin{aligned} \text{splitE } cv@(\text{CVE } _) \text{ i} = & ((S[cv:->(\text{CONS } cvh \text{ cvt})], \\ & S[cv:->cva]), i') \end{aligned}$$

$$\begin{aligned} \text{where } (cvh, i1) = & \text{mkCEVar } i \\ & (cvt, i2) = \text{mkCEVar } i1 \\ & (cva, i') = \text{mkCAVar } i2 \end{aligned}$$


```

freeindx :: FreeIndx -> Class -> FreeIndx
freeindx i c = 1 + maximum(i:(map index (cvars c)))
      where index :: CVar -> Int
            index (CVA i) = i
            index (CVE i) = i

```

3.3 Алгоритм ptr

```

ptr :: ProgR -> Class -> Tree
ptr  p cl@(ces, r) = evalT c p i
      where
        (DEFINE f prms _) : p' = p
        ce = mkEnv prms ces
        c  = ((CALL f prms, ce), r)
        i  = freeindx 0 cl

```

3.4 Алгоритм ptr (функция evalT)

evalT :: Conf -> ProgR -> FreeIndx -> Tree

```
evalT c@(( CALL f args , ce), r) p i =
    NODE c [ (idC, evalT c' p i) ]
    where
        DEFINE _ prms t' = getDef f p
        ce' = mkEnv prms (args/.ce)
        c'  = ((t',ce'),r)
```

```
evalT c@(( ALT cnd t1 t2 , ce), r) p i =
    NODE c [(cnt1,evalT c1' p i'),(cnt2,evalT c2' p i')]
    where
        ((cnt1,cnt2), uce1, uce2, i') = ccond cnd ce i
        ((_ ,ce1),r1) = c/.cnt1
        c1' = ((t1, ce1+.uce1), r1)
        ((_ ,ce2),r2) = c/.cnt2
        c2' = ((t2, ce2+.uce2), r2)
```

```
evalT c@((exp,ce),r) p i = LEAF c
```

3.5 Алгоритм ptr (функция ccond)

`ccond :: Cond -> CEnv -> FreeIndx -> (Split, CEnv, CEnv, FreeIndx)`

```

ccond (EQA' x y)      ce i =
  let x' = x/.ce; y' = y/.ce in
  case (x', y') of
    (a,      b      ) | a==b -> ((idC ,emptC), [], [], i)
    (ATOM a,ATOM b)      -> ((emptC, idC), [], [], i)
    (CVA _, a      )      -> ( splitA x' a , [], [], i)
    (a,      CVA _ )      -> ( splitA y' a , [], [], i)

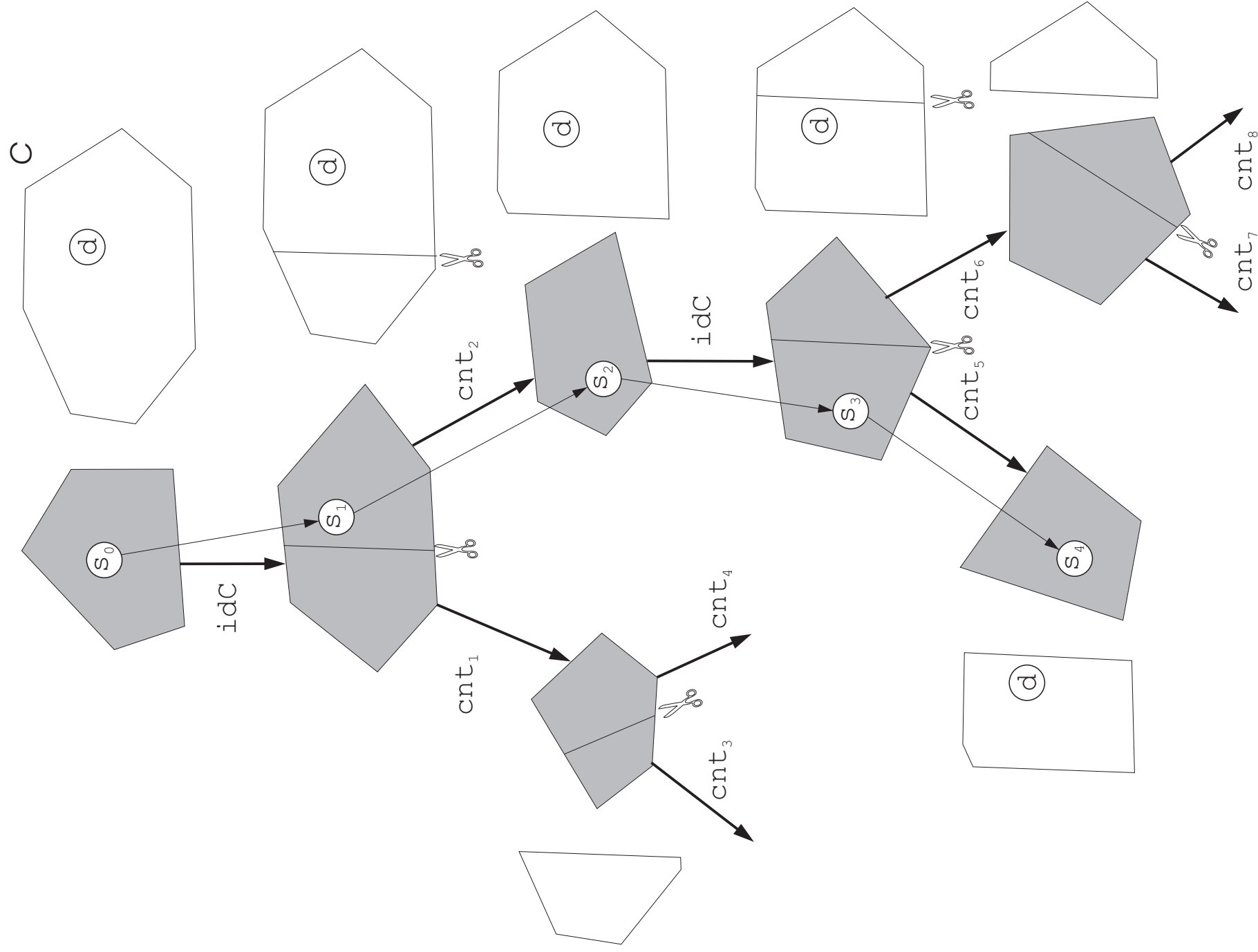
```

```

ccond (CONS' x vh vt va) ce i =
  let x' = x/.ce in
  case x' of
    CONS h t -> ((idC,emptC), [vh:=h,vt:=t], [], i )
    ATOM a   -> ((emptC,idC), [], [va:=x'], i )
    CVA _    -> ((emptC,idC), [], [va:=x'], i )
    CVE _    -> ( split, [vh:=ch,vt:=ct], [va:=ca], i')
    where
      (split,i') = splitE x' i
      (S[_:->(CONS ch ct)], S[_:->ca]) = split

```


Дерево процессов, множества достижимости



Глава 4. Универсальный решающий алгоритм

УРА—один из методов метавычислений, позволяющий выполнять инверсное вычисление программ.

- Проблема *инверсного вычисления программы* — построение представления множества:

$$\langle x \rangle_{p^{-1}y} \stackrel{\text{def}}{=} \{ d \mid d \in \langle x \rangle, p \, d \stackrel{*}{\Rightarrow} y \} = \langle x \rangle \cap (p^{-1} y).$$

- Актуальность проблемы: (а) вычисление функции, обратной к функции заданной программы, (б) построение множества истинности программы-предиката, (с) обработка сложных запросов к базе знаний, (d) решение уравнений, без разработки специальных методов и алгоритмов их решений.
- Алгоритм **tab** приведения функции программы к табличной форме

$$\text{tab } p \, x \stackrel{*}{\Rightarrow} L = [(x_1, \text{сех}_1), (x_2, \text{сех}_2), \dots]$$

Для любого $d \in \langle x \rangle$ выполнено:

$$p \, d = \begin{cases} \text{сех}_i / .s & \text{если } d \in \langle x_i \rangle, s \text{—подстановка такая, что} \\ & x_i / .s = (d, \text{RESTR}[]) \\ \text{неопределено} & \text{если не существует } i \text{ такого, что } d \in \langle x_i \rangle \end{cases}$$

- Две модификации **ura'** и **ura** универсального решающего алгоритма:

$$\text{ura}' \quad p \ x \ y \xRightarrow{*} L' = [x_1, x_2, \dots]$$

$$\text{ura} \quad p \ x \ y \xRightarrow{*} L = [(s_1, r_1), (s_2, r_2), \dots]$$

- **ura'** перечисляет подклассы x_i класса x , на которых вычисление p приводит к результату y : $p \langle x_i \rangle = \{y\}$;
- **ura** перечисляет значения с-переменных из x , при которых вычисление p приводит к результату y : $p \langle x / .(S \ s_i) / .(R \ r_i) \rangle = \{y\}$.

(a) “Естественность” перечисления,

(b) в некоторых случаях **ura'** позволяет построить *конечное* представление множества $\langle x \rangle_{p^{-1}y}$ —список L' может быть пустым или конечным.

- Примеры инверсного вычисления для языка TSG.
- Обзор иных подходов к инверсии программ.

4.1 Приведение программы к табличной форме

```

type TLevel = [(Class,Tree)]           type Tab = [(Class, CExp)]

tab :: ProgR -> Class -> Tab
tab p x = tab' [ (x,tree) ] [ ]
           where tree = xptr p x

tab' :: TLevel -> TLevel -> Tab
tab' ((xi, LEAF c):lv1) lv2 = (xi,cex):(tab' lv1 lv2)
                               where ((exp,ce),_) = c
                                      cex = exp/.ce

tab' ((xi, NODE _ bs):lv1) lv2 = tab' lv1 lv2'
                               where lv2' = tabB xi bs lv2

tab' [ ] [ ] = [ ]

tab' [ ] lv2 = tab' lv2 [ ]

tabB :: Class -> [Branch] -> TLevel -> TLevel
tabB xi ((cnt,tree):bs) lv = tabB xi bs ((xi/.cnt,tree):lv)
tabB xi [ ] lv = lv

```


Теорема 12

Пусть p —произвольная программа на языке TSG, x —класс, s —данное для p .

Тогда алгоритм **tab** по заданным p и x строит перечисление—то есть список $L = [(x_1, cex_1), (x_2, cex_2), \dots]$, возможно бесконечной длины:

$$\text{tab } p \ x \xRightarrow{*} L$$

где x_i —класс, cex_i — s -выражение над переменными из класса x_i . При этом выполнено следующее:

1. Классы x_i —попарно непересекающиеся подклассы класса x .
2. Для любого i и любого $d \in \langle x_i \rangle$ —программа p определена на d .
3. Для любого данного $d \in \langle x \rangle$ на котором программа p определена— $p \ d \xRightarrow{*} res$,— существует i такой, что $d \in \langle x_i \rangle$, и если подстановка s такая, что

$$x_i / .s = (d, \text{RESTR}[]),$$

то выполнено $res = cex_i / .s$. Кроме того, в данном случае, префикс длины i списка L будет построен алгоритмом **tab** за конечное число шагов.

4.2 Алгоритм инверсного вычисления программ

```
ura' :: ProgR -> Class -> EVal -> Classes
```

```
ura' p x y = urac (tab p x) y
```

```
urac :: Tab -> EVal -> Classes
```

```
urac ((xi, cex):ptab') y =
    case (unify [cex] [y]) of
        (False, _) -> tail
        (True, s) -> case xi' of
            (_, INCONSISTENT) -> tail
            _ -> xi':tail
            where xi' = xi/.s
    where tail = urac ptab' y
```

```
urac [ ] y = [ ]
```

Теорема 13

Пусть p —произвольная программа на языке TSG, x —класс, s -данное для p , y —значение, $y \in \text{Eval}$.

Тогда алгоритм ura' по заданным p , x , y строит перечисление—то есть список $L' = [x'_1, x'_2, \dots]$, возможно бесконечной длины:

$$\text{ura}' \ p \ x \ y \xRightarrow{*} L'$$

При этом выполнено следующее:

1. Классы x_i —попарно непересекающиеся подклассы класса x .
2. Для любого i и любого данного $d \in \langle x'_i \rangle$ —программа p определена на d и $p \ d = y$;
3. Для любого данного $d \in \langle x \rangle$, на котором программа p определена и $p \ d = y$, существует номер i такой, что $d \in \langle x'_i \rangle$. Кроме того, в данном случае, префикс длины i списка будет построен алгоритмом ura' за конечное число шагов.

4.3 Альтернативное представление результатов инверсного вычисления

```
ura :: ProgR -> Class -> EVal -> [(Subst, Restr)]
```

```
ura p x y = map altRepr (ura' p x y)
```

```
  where
```

```
    altRepr :: Class -> (Subst, Restr)
```

```
    altRepr xi = subClassCntr x xi
```

```
subClassCntr :: Class -> Class -> (Subst, Restr)
```

```
subClassCntr x@(cesx, rx) x'@(cesx', rx') = (s, r)
```

```
  where
```

```
    (True, s) = unify cesx cesx'
```

```
    r = case rx' of
```

```
      INCONSISTENT->INCONSISTENT
```

```
      RESTR ineqs'->RESTR
```

```
          [ ineq | ineq<-ineqs', not(ineq'elem'ineqs)]
```

```
    where RESTR ineqs = rx/.s
```

4.4 Универсальный решающий алгоритм. Примеры

```

str    = CONS 'A (CONS 'B (CONS 'C 'NIL))
x      = ( [E.1,str], RESTR[] )
aaa    = CONS 'A (CONS 'A (CONS 'A 'NIL))
xa     = ( [E.1,aaa], RESTR[] )
true   = 'SUCCESS
false  = 'FAILURE

```

```

ura prog x true =
[ ([E.1 :-> A.4], RESTR[]),
  ([E.1 :-> CONS 'A A.10], RESTR[]),
  ([E.1 :-> CONS 'A (CONS 'B A.16)], RESTR[]),
  ([E.1 :-> CONS 'B A.10], RESTR[]),
  ([E.1 :-> CONS 'A (CONS 'B (CONS 'C A.22))], RESTR[]),
  ([E.1 :-> CONS 'B (CONS 'C A.16)], RESTR[]),
  ([E.1 :-> CONS 'C A.10], RESTR[])
]

```

ura prog xa true =

```
[ ([ $\mathcal{E}.1$  :->  $\mathcal{A}.4$ ], []),
  ([ $\mathcal{E}.1$  :-> CONS 'A  $\mathcal{A}.10$ ], []),
  ([ $\mathcal{E}.1$  :-> CONS 'A (CONS 'A  $\mathcal{A}.16$ )], []),
  ([ $\mathcal{E}.1$  :-> CONS 'A (CONS 'A (CONS 'A  $\mathcal{A}.22$ ))], []) ]
```

ura prog x false =

```
[([ $\mathcal{E}.1$ :->CONS (CONS  $\mathcal{E}.13$   $\mathcal{E}.14$ )  $\mathcal{E}.11$ ], []),
  ([ $\mathcal{E}.1$ :->CONS 'A (CONS (CONS  $\mathcal{E}.19$   $\mathcal{E}.20$ )  $\mathcal{E}.17$ )], []),
  ([ $\mathcal{E}.1$ :->CONS 'A (CONS 'B (CONS (CONS  $\mathcal{E}.25$   $\mathcal{E}.26$ )  $\mathcal{E}.23$ ))], []),
  ([ $\mathcal{E}.1$ :->CONS 'B (CONS (CONS  $\mathcal{E}.19$   $\mathcal{E}.20$ )  $\mathcal{E}.17$ )], []),
  ([ $\mathcal{E}.1$ :->CONS 'A (CONS 'B (CONS 'C (CONS (CONS  $\mathcal{E}.31$   $\mathcal{E}.32$ )  $\mathcal{E}.29$ )))], []),
  ([ $\mathcal{E}.1$ :->CONS 'A (CONS 'B (CONS 'C (CONS  $\mathcal{A}.33$   $\mathcal{E}.29$ )))], []),
  ([ $\mathcal{E}.1$ :->CONS 'B (CONS 'C (CONS (CONS  $\mathcal{E}.25$   $\mathcal{E}.26$ )  $\mathcal{E}.23$ ))], []),
  ([ $\mathcal{E}.1$ :->CONS 'B (CONS 'C (CONS  $\mathcal{A}.27$   $\mathcal{E}.23$ ))], []),
  ([ $\mathcal{E}.1$ :->CONS 'C (CONS (CONS  $\mathcal{E}.19$   $\mathcal{E}.20$ )  $\mathcal{E}.17$ )], []),
  ([ $\mathcal{E}.1$ :->CONS 'C (CONS  $\mathcal{A}.21$   $\mathcal{E}.17$ )], []),
  ([ $\mathcal{E}.1$ :->CONS  $\mathcal{A}.15$   $\mathcal{E}.11$ ], [ $\mathcal{A}.15 \neq$  'A,  $\mathcal{A}.15 \neq$  'B,  $\mathcal{A}.15 \neq$  'C]),
  ([ $\mathcal{E}.1$ :->CONS 'A (CONS  $\mathcal{A}.21$   $\mathcal{E}.17$ )], [ $\mathcal{A}.21 \neq$  'B]),
  ([ $\mathcal{E}.1$ :->CONS 'B (CONS  $\mathcal{A}.21$   $\mathcal{E}.17$ )], [ $\mathcal{A}.21 \neq$  'C]),
  ([ $\mathcal{E}.1$ :->CONS 'A (CONS 'B (CONS  $\mathcal{A}.27$   $\mathcal{E}.23$ ))], [ $\mathcal{A}.27 \neq$  'C']) ]
```

Глава 5. Инверсное программирование

Технология программирования, основанная на использовании инверсных вычислений.

- *Инверсное программирование*: программист вместо того чтобы программировать требуемую в задании функцию f , реализует программу p , функция которой при инверсном вычислении совпадает с функцией f .

С каждым текстом программы p можно связать не одну, а две функции программы:

- $p :: D \rightarrow \text{Eval}$ —обычная функция программы (семантика):

$$p\ d \stackrel{\text{def}}{=} \text{int}\ p\ d;$$

- $p_{inv} :: D_{inv} \rightarrow R_{inv}$ —инверсная функция (семантика) программы:

$$p_{inv}\ (x, y) \stackrel{\text{def}}{=} \text{ura}\ p\ x\ y;$$

где $D_{inv} = (\text{Class}, \text{Eval})$ —тип запросов на инверсное вычисление,
 $R_{inv} = [(\text{Subst}, \text{Restr})]$ —тип результатов инверсного вычисления.

- *Перенос инверсных вычислений с языка R на произвольный язык программирования L.*

```

inv :: ProgR -> D -> Dinv -> Rinv
inv intL p (x,y) = ura intL (p::x) y
                    where
                        (::.) :: ProgL->Class -> Class
                        p::.(ces,r) = ((p:ces),r)

```

$$\text{inv intL } p \text{ (x,y)} \xRightarrow{*} [(s_1, r'_1), (s_2, r'_2), \dots]$$

- *Сравнение логического программирования с инверсным программированием программ-предикатов:*
 - Свойства инверсного программирования программ-предикатов позволяют рассматривать язык L_{inv} , как язык логического программирования.
 - Инверсное программирование программ-предикатов предоставляет больше изобразительных средств в распоряжение программиста, нежели традиционные языки логического программирования.
 - Концепция инверсного программирования не исчерпывается инверсным программированием программ-предикатов. Ее можно использовать для гораздо более широкого класса задач.

Глава 6. Окрестностный анализ

Впервые введен в рассмотрение в 1970-ых годах В.Ф.Турчиным, при участии других членов московской Рабочей группы по языку Рефал—“неточный”, “ad hoc”. Предложенный здесь окрестностный анализ—один из методов метавычислений, инструмент общего назначения для построения формальных ответов на интуитивный вопрос:

Какая информация о тексте d была использована, и какая информация о d не была использована в некотором процессе обработки текста d ?

- Шаги формализации:

- $o(p, d) = \{ d' \mid \text{процесс } p \text{ } d' \xRightarrow{*} r \text{ такой же, как } p \text{ } d \xRightarrow{*} r \};$
- $o(p, d) = \{ d' \mid \text{tr}(p, d) = \text{tr}(p, d'), p \text{ } d = p \text{ } d' \};$
- Понятие *окрестности*: L-класс \mathcal{O} является окрестностью данных d , если $d \in \langle \mathcal{O} \rangle$.
Подокрестность $(\mathcal{O}_2^d \preceq \mathcal{O}_1^d)$ и *собственная подокрестность* $(\mathcal{O}_2^d \prec \mathcal{O}_1^d)$.
 Не существует бесконечной последовательности $\mathcal{O}_1^d \succ \mathcal{O}_2^d \succ \mathcal{O}_3^d \succ \dots$

- Построен и обоснован алгоритм окрестностного анализатора `nan`:

$$\text{nan } p \text{ } d = (\text{res}, \mathcal{O}^d), \quad \text{где } \langle \mathcal{O}^d \rangle = o(p, d).$$

- Наглядная (*эксплицированная*) форма записи окрестности \mathcal{O}^d —с-переменные в \mathcal{O}^d соответствуют фрагментам данных d , которые можно варьировать; типы с-переменных

и рестрикции в окрестности \mathcal{O}^d —ограничения на допустимые вариации данных фрагментов.

- Операции над классами и окрестностями:

1. Пересечение ($*$) и разность (\setminus) классов, L-классов, окрестностей.
2. Проверка принадлежности данных классу (алгоритм `isElem`);
3. Декомпозиции окрестности. Если данные d для программы p имеют вид $d=d_1++d_2$ и $\langle \mathcal{O}^{d_1++d_2} \rangle = o(p, d_1++d_2)$, то получаемые в результате декомпозиции:

$$\text{decompose } \mathcal{O}^{d_1++d_2} \ d_1 \ d_2 = (\mathcal{O}^{d_1}, \mathcal{O}^{d_2})$$

окрестности \mathcal{O}^{d_i} ($i = 1, 2$) определяют, какая информация о фрагменте d_i данных d_1++d_2 была использована в процессе $p(d_1++d_2) \xRightarrow{*} \text{res}$ обработки данных d_1++d_2 .

6.1 Основные понятия

Окрестностный анализ—инструмент формализации интуитивного вопроса:

Какая информация о тексте d была использована, и какая информация о d не была использована в некотором процессе p обработки текста d ?

1. Рассматриваем только *конструктивные* процессы $p \ d \xRightarrow{*} \text{res}$, где $d \in D$ и $p \in R$.
2. “Не вся информация использована”: d можно *подменить* на d' такой, что процесс обработки $p \ d' \xRightarrow{*} \text{res}$ будет таким же, как $p \ d \xRightarrow{*} \text{res}$.

Множество всех возможных таких подмен:

$$o(p, d) = \left\{ d' \mid \begin{array}{l} \text{процесс } p \ d' \xRightarrow{*} \text{res} \text{ точно такой же, как} \\ \text{процесс } p \ d \xRightarrow{*} \text{res} \end{array} \right\} \quad (6.1)$$

определяет, какая информация о тексте d была использована, и какая информация о тексте d не была использована в процессе $p \ d \xRightarrow{*} \text{res}$.

3. За счет понятия трассы множество $o(p, d)$ получает формальное определение:

$$o(p, d) = \{ d' \mid \text{tr}(p, d) = \text{tr}(p, d'), \ p \ d = p \ d' \} \quad (6.2)$$

Осталось уметь строить в виде конструктивного объекта множество $o(p, d)$ для заданных p и d .

6.2 Окрестности

Определение 9

Пусть $\mathcal{O}=(\text{ces}, \text{rs})$ —L-класс, $d \in \langle \mathcal{O} \rangle \subseteq [\text{Eval}]$. Тогда будем называть \mathcal{O} окрестностью d (окрестностью с центром d).

1. Часто удобно записывать центр d в верхнем индексе: \mathcal{O}^d ;
2. Окрестность не может быть пустой: $d \in \langle \mathcal{O}^d \rangle$;
3. \mathcal{O}_i^d используются для представления $o(p, d)$ —информационная интерпретация:
 - Если d —единственный (не единственный) элемент $\langle \mathcal{O}^d \rangle = o(p, d)$, то ...
 - Если $\langle \mathcal{O}_1^d \rangle = o(p_1, d)$, $\langle \mathcal{O}_2^d \rangle = o(p_2, d)$ и $\mathcal{O}_2^d \preceq \mathcal{O}_1^d$ (или $\mathcal{O}_2^d \prec \mathcal{O}_1^d$), то ...

Теорема 14

Для любого данного $d \in D$ множество канонических форм окрестностей d конечно и не существует бесконечной последовательности $\mathcal{O}_1^d \succ \mathcal{O}_2^d \succ \mathcal{O}_3^d \succ \dots$ окрестностей данных d (невозможно бесконечно уточнять информацию о d).

6.3.1 Окрестностный анализатор

Идея: “параллельные и синхронные” *эталонное и обобщенное вычисления*:

```

nan :: ProgR -> [EVal] -> (EVal, Class)
nan p d = eval' (CALL f prms) e ce (RESTR[]) ces p i
      where (DEFINE f prms _) : p' = p
            (ces, i) = mkCExps prms 1
            e         = mkEnv prms d
            ce        = mkEnv prms ces

eval' :: Term->Env->CEnv->Restr->CExps->ProgR->FreeIndx ->
                                             (EVal, Class)

eval' (CALL f args) e ce r ces p i =
  eval' t e' ce' r ces p i
  where
    DEFINE _ prms t=getDef f p
    e'   = mkEnv prms (args/.e)
    ce'  = mkEnv prms (args/.ce)

```

```

eval' (ALT c t1 t2) e ce r ces p i =
  case cond c e of
    TRUE  ue  -> eval' t1 (e+.ue) ce1' r1 ces1 p i'
    FALSE ue  -> eval' t2 (e+.ue) ce2' r2 ces2 p i'
  where
    ((cnt1,cnt2), uce1, uce2, i') = ccond c ce i
    ((ce1,ces1),r1) =((ce,ces),r)/.cnt1
    ce1' =ce1+.uce1
    ((ce2,ces2),r2) =((ce,ces),r)/.cnt2
    ce2' =ce2+.uce2

```

```

eval' exp e ce r ces p i =
  (res, (ces, r)/.subst)
  where
    res  =exp/.e
    (True,subst)=unify [exp/.ce] [res]

```

Теорема 15

Пусть p — произвольная программа на TSG, d — произвольные входные данные для p , процесс вычисления p на d завершается за конечное число n шагов. Тогда алгоритм **nan** на данных p d завершается (за n выполнений функции **eval'**) со следующим результатом:

$$\text{nan } p \ d = (\text{res}, \mathcal{O}^d)$$

где **res**—результат вычисления p на d : $p \ d \xRightarrow{*} \text{res}$, \mathcal{O}^d —окрестность d такая, что:

$$\langle \mathcal{O}^d \rangle = o(p, d) = \{ d' \mid \text{tr}(p, d) = \text{tr}(p, d'), \ p \ d = p \ d' \}$$

Утверждение 16

Пусть $p \in R$, d —произвольные входные данные для p , программа p завершается на данных d , $\text{nan } p \ d = (\text{res}, \mathcal{O}^d)$ и $d' \in \langle \mathcal{O}^d \rangle$.

Тогда программа p завершается на данных d' с теми же результатом и трассой, что и на данных d , $\text{nan } p \ d = (\text{res}, \mathcal{O}^{d'})$ и окрестности \mathcal{O}^d и $\mathcal{O}^{d'}$ текстуально совпадают: $\mathcal{O}^d = \mathcal{O}^{d'}$.

6.4 Интуитивный смысл окрестностей. Примеры

`nan` строит представление \mathcal{O}^d для $o(p, d)$. Значит \mathcal{O}^d описывает “какая информация о данных d была использована в процессе $p \ d \xRightarrow{*} res$ ”. Насколько данное описание удобно для человека? (На примере программы `pmatch` проверки ...):

```
nan pmatch d1 = ('SUCCESS,  $\mathcal{O}^{d1}$ )
  d1 = [CONS 'A 'NIL, CONS 'A 'NIL]
 $\mathcal{O}^{d1}$  = ([CONS  $\mathcal{A}.14$   $\mathcal{A}.17$ , CONS  $\mathcal{A}.14$   $\mathcal{E}.10$ ], RESTR[])
```

Определение. Позволим изображать:

- са-переменную в виде любого *надчеркнутого* атома, над которым надписан индекс,
- се-переменную в виде произвольного *дважды надчеркнутого* е-значения, над которым надписан индекс.

Эксплицированная форма записи окрестности $\mathcal{O}^d = (ces, r)$: в `ces` все с-переменные v запишем в указанной выше форме, взяв в качестве *надчеркиваемого* значение $v/.s$ с-переменной, полученное при отождествлении `ces` с d : $(True, s) = unify\ ces\ d$. Терминология: надчеркнутые фрагменты d *покрыты* с-переменными в \mathcal{O}^d .

```
 $\mathcal{O}^{d1}$  = ([CONS  $\overline{\overline{\mathcal{A}^{14}}}$   $\overline{\overline{\mathcal{A}^{17}}}$ , CONS  $\overline{\overline{\mathcal{A}^{14}}}$   $\overline{\overline{\mathcal{E}^{10}}}$ ], RESTR[])
```

При использовании эксплицированной формы записи окрестности нет необходимости особо указывать центр окрестности:

$$\begin{aligned}
\mathcal{O}^{d2} &= ([\text{CONS } \overset{8}{\text{'A}} \overset{4}{\text{'NIL}}, \text{CONS } \overset{14}{\text{'B}} \overset{17}{\text{'NIL}}], \text{RESTR}[\mathcal{A}.8:\neq:\mathcal{A}.14]) \\
\mathcal{O}^{d3} &= ([\text{CONS } \overset{8}{\text{'X}} \overset{4}{\text{'NIL}}, \text{CONS } \overset{14}{\text{'Y}} \overset{17}{\text{'NIL}}], \text{RESTR}[\mathcal{A}.8:\neq:\mathcal{A}.14]) \\
\mathcal{O}^{d4} &= ([\text{CONS } \overset{32}{\text{'A}} (\text{CONS } \overset{44}{\text{'B}} \overset{47}{\text{'NIL}}), \\
&\quad \text{CONS } \overset{14}{\text{'X}} (\text{CONS } \overset{20}{\text{'Y}} (\text{CONS } \overset{26}{\text{'Z}} (\text{CONS } \overset{32}{\text{'A}} (\text{CONS } \overset{44}{\text{'B}} \overset{40}{\text{'NIL'}})))) \\
&\quad], \text{RESTR}[\mathcal{A}.32:\neq:\mathcal{A}.14, \mathcal{A}.32:\neq:\mathcal{A}.20, \mathcal{A}.32:\neq:\mathcal{A}.26]) \\
\mathcal{O}^{d5} &= ([\text{CONS } \overset{26}{\text{'Z}} (\text{CONS } \overset{38}{\text{'A}} \overset{41}{\text{'NIL}}), \\
&\quad \text{CONS } \overset{14}{\text{'X}} (\text{CONS } \overset{20}{\text{'Y}} (\text{CONS } \overset{26}{\text{'Z}} (\text{CONS } \overset{38}{\text{'A}} \overset{34}{\text{'B'}} \overset{34}{\text{'NIL'}})))) \\
&\quad], \text{RESTR}[\mathcal{A}.26:\neq:\mathcal{A}.14, \mathcal{A}.26:\neq:\mathcal{A}.20])
\end{aligned}$$

Выводы: окрестности в наглядной форме определяют:

- какая информация о данных была использована в процессе обработки, а какая—нет;
- какие фрагменты в данных и как можно варьировать, сохраняя неизменным процесс $p \ d \xRightarrow{*} \text{res}$ и его результат.

6.5 Операции над классами и окрестностями

Конечное объединение классов $\mathcal{C}_1, \dots, \mathcal{C}_n$. Обозначим:

1. $[\mathcal{C}_1 + \dots + \mathcal{C}_n]$ — список классов $[\mathcal{C}_1, \dots, \mathcal{C}_n]$;
2. $\langle \mathcal{C}_1 + \dots + \mathcal{C}_n \rangle$ — множество $\langle \mathcal{C}_1 \rangle \cup \dots \cup \langle \mathcal{C}_n \rangle$.

Теорема 17 (unifC) Пусть $\mathcal{C}_0^a = (\text{aces}, \text{ar})$ — L-класс, $\mathcal{C}_0^b = (\text{bces}', \text{br}')$ — класс, s -переменные из \mathcal{C}^a не входят в \mathcal{C}^b , i — свободный индекс. Тогда, за конечное число шагов алгоритм unifC:

$$\text{unifC } \mathcal{C}^a \mathcal{C}^b i \xRightarrow{*} (\text{su}, [\mathcal{C}_0^b, \mathcal{C}_1^b, \dots, \mathcal{C}_n^b], i')$$

строит подстановку su для \mathcal{C}^a и подклассы $\mathcal{C}_j^b \preceq \mathcal{C}^b$ ($j = 0, \dots, n, n \geq 1$), такие, что:

1. $\langle \mathcal{C}^b \rangle = \langle \mathcal{C}_0^b + \mathcal{C}_1^b + \dots + \mathcal{C}_n^b \rangle$, $\langle \mathcal{C}_j^b \rangle \cap \langle \mathcal{C}_k^b \rangle = \emptyset$, где $0 \leq j < k \leq n$;
2. $\langle \mathcal{C}^b \rangle \cap \langle \mathcal{C}^a \rangle = \langle \mathcal{C}_0^b \rangle$, $\langle \mathcal{C}^b \rangle \setminus \langle \mathcal{C}^a \rangle = \langle \mathcal{C}_1^b + \dots + \mathcal{C}_n^b \rangle$;
3. (a) Если $\langle \mathcal{C}^b \rangle \cap \langle \mathcal{C}^a \rangle = \emptyset$,
то $\mathcal{C}_0^b = \mathcal{C}^b / .\text{emptC}$, $\mathcal{C}_1^b = \mathcal{C}^b / .\text{idC} = \mathcal{C}^b$, \mathcal{C}_0^b имеет вид $(\text{ces}_0, \text{INCONSISTENT})$,
 $\langle \mathcal{C}^b \rangle \cap \langle \mathcal{C}^a \rangle = \emptyset = \langle \mathcal{C}_0^b \rangle$, $\langle \mathcal{C}^b \rangle \setminus \langle \mathcal{C}^a \rangle = \langle \mathcal{C}_1^b \rangle = \langle \mathcal{C}^b \rangle$, и $\text{su} = []$;
- (b) Если $\langle \mathcal{C}^b \rangle \cap \langle \mathcal{C}^a \rangle \neq \emptyset$,
то $\mathcal{C}_0^b \preceq \mathcal{C}^a$ (общий подкласс \mathcal{C}^a и \mathcal{C}^b), $\text{bces}' = \text{aces} / .\text{su}$ и каждое неравенство из $\text{ar} / .\text{su}$ входит в br' .

$$\begin{array}{ccccc}
 \mathcal{C}^b & & \mathcal{C}_2^b & \dots & \\
 & & \mathcal{C}_1^b & & \mathcal{C}_n^b \\
 & & & \mathcal{C}_0^b & \\
 \mathcal{C}^a & & & & \mathcal{C}^a \\
 & \text{unifC} & \mathcal{C}^a & \mathcal{C}^b &
 \end{array}$$

1. Обозначим: $\mathcal{C}^b * \mathcal{C}^a = \mathcal{C}_0^b$, $\mathcal{C}^b \setminus \mathcal{C}^a = [\mathcal{C}_1^b + \dots + \mathcal{C}_n^b]$.
2. Если $\mathcal{C}^a, \mathcal{C}^b$ —L-классы, то $\mathcal{C}_0^b, \mathcal{C}_1^b, \dots, \mathcal{C}_n^b$ —L-классы.
3. Объединение, пересечение и разность множеств, представленных конечным объединением L-классов, представимы конечным объединений L-классов. Конечное пересечение множеств, представленных L-классами, представимо L-классом.
4. Если \mathcal{O}_1^d и \mathcal{O}_2^d окрестности d , то $\mathcal{O}_1^d * \mathcal{O}_2^d$ —окрестность d (*интерпретация...*).

6.6 Проверка принадлежности данных классу

Утверждение 18

Для любого данного $d \in D = [EVal]$ и класса $C = (ces, r)$ приведенный ниже алгоритм `isElem` за конечное число шагов определяет, является ли d элементом множества $\langle C \rangle$ или нет.

```
isElem :: [EVal] -> Class -> Bool
isElem d (ces, r) = case (b, r') of
    (True, RESTR[]) -> True
    (_, _) -> False
  where (b, s) = unify ces d
        r'      = r /. s
```

6.7 Декомпозиция окрестности

Данные $d \in [EVal]$ имеют вид $d = d_1 ++ d_2$, $\mathcal{O}^{d_1 ++ d_2}$ — окрестность $d_1 ++ d_2$, представляющая $o(p, d_1 ++ d_2)$. Нас интересует:

1. Представление \mathcal{O}^{d_1} множества всех вариаций d' фрагмента d_1 в $d_1 ++ d_2$, сохраняющих неизменным процесс и результат $p(d_1 ++ d_2) \xRightarrow{*} res$ (интерпретация...).
2. Представление \mathcal{O}^{d_2} множества всех вариаций d' фрагмента d_2 в $d_1 ++ d_2$, сохраняющих неизменным процесс и результат $p(d_1 ++ d_2) \xRightarrow{*} res$ (интерпретация...).

Декомпозиция окрестности

$$\mathcal{O}^{d_1} \quad d_1 \quad c_1^{\mathcal{O}} \quad c_1 \quad d_1++d_2 \quad \mathcal{O}^{d_1++d_2}$$

$$c_2^{\mathcal{O}}$$

$$c_2$$

$$d_2$$

$$\mathcal{O}^{d_2}$$

6.8 Алгоритм декомпозиции окрестности

```

decompose::Class->[EVal]->[EVal] -> (Class,Class)
decompose o12 d1 d2 = (o1, o2)
  where
    l1 = length d1; l2 = length d2
    i = freeindx 0 o12
    (ces1,i1) = mkCEVs [ ] l1 i
    (ces2,i2) = mkCEVs [ ] l2 i1
    c1 = (ces1++d2, RESTR[])
    c2 = (d1++ces2, RESTR[])
    (_,((ces'1,r1):_)) = unifC c1 o12      -- c1*o12
    (_,((ces'2,r2):_)) = unifC c2 o12      -- c2*o12
    o1 = ( (take l1 ces'1), r1 )            -- взяли первые L1
    o2 = ( (drop l1 ces'2), r2 )           -- откинули первые L1

-- генератор списка уникальных ce-переменных заданной длины
mkCEVs::CVars->Int->FreeIndx -> (CVars,FreeIndx)
mkCEVs cvs 0      i = (cvs,i)
mkCEVs cvs (n+1) i = mkCEVs (cv:cvs) n i'
                        where (cv,i') = mkCEVar i

```

6.9 Примеры пересечения и декомпозиции окрестностей (1/2)

Рассмотрим эксплицированные канонические формы результатов реального счета `nan` для `d4 = [s4,str]` и `d5 = [s5,str]` и последующих операций:

```
s4 = (CONS 'A (CONS 'B 'NIL))
s4 = (CONS 'Z (CONS 'A 'NIL))
str = (CONS 'Z (CONS 'Z (CONS 'Z (CONS 'Z (CONS 'Z 'NIL)))))
```

```
(res4, O[s4,str]) = nan pmatch O[s4,str] [s4] [str]
```

```
(res5, O[s5,str]) = nan pmatch O[s5,str] [s4] [str]
```

```
(Os4, O1str) = decompose O[s4,str] [s4] [str]
```

```
(Os5, O2str) = decompose O[s5,str] [s5] [str]
```

$$O_{1,2}^{\text{str}} = O_1^{\text{str}} * O_2^{\text{str}}$$

Примеры пересечения и декомпозиции окрестностей (2/2)

$$\mathcal{O}^{[s4, \text{str}]} = ([\text{CONS } \overset{1}{\text{'A}} (\text{CONS } \overset{2}{\text{'B}} \overset{3}{\text{'NIL}}), \\ \text{CONS } \overset{4}{\text{'X}} (\text{CONS } \overset{5}{\text{'Y}} (\text{CONS } \overset{6}{\text{'Z}} (\text{CONS } \overset{1}{\text{'A}} (\text{CONS } \overset{2}{\text{'B}} \overset{7}{\text{'NIL'}})))) \\], \text{RESTR}[\mathcal{A}.1:\neq:\mathcal{A}.4, \mathcal{A}.1:\neq:\mathcal{A}.5, \mathcal{A}.1:\neq:\mathcal{A}.6])$$

$$\mathcal{O}^{[s5, \text{str}]} = ([\text{CONS } \overset{1}{\text{'Z}} (\text{CONS } \overset{2}{\text{'A}} \overset{3}{\text{'NIL}}), \\ \text{CONS } \overset{4}{\text{'X}} (\text{CONS } \overset{5}{\text{'Y}} (\text{CONS } \overset{1}{\text{'Z}} (\text{CONS } \overset{2}{\text{'A}} (\text{CONS } \overset{6}{\text{'B}} \overset{7}{\text{'NIL'}})))) \\], \text{RESTR}[\mathcal{A}.1:\neq:\mathcal{A}.4, \mathcal{A}.1:\neq:\mathcal{A}.5])$$

$$\mathcal{O}^{s4} = ([\text{CONS } \text{'A'} (\text{CONS } \text{'B'} \overset{1}{\text{'NIL'}})], \text{RESTR } [])$$

$$\mathcal{O}^{s5} = ([\text{CONS } \text{'Z'} (\text{CONS } \text{'A'} \overset{1}{\text{'NIL'}})], \text{RESTR } [])$$

$$\mathcal{O}_1^{\text{str}} = ([\text{CONS } \overset{1}{\text{'X'}} (\text{CONS } \overset{2}{\text{'Y'}} (\text{CONS } \overset{3}{\text{'Z'}} (\text{CONS } \text{'A'} (\text{CONS } \text{'B'} \overset{4}{\text{'NIL'}}))))], \\ \text{RESTR } [\mathcal{A}.1:\neq:\text{'A'}, \mathcal{A}.2:\neq:\text{'A'}, \mathcal{A}.3:\neq:\text{'A'}])$$

$$\mathcal{O}_2^{\text{str}} = ([\text{CONS } \overset{1}{\text{'X'}} (\text{CONS } \overset{2}{\text{'Y'}} (\text{CONS } \text{'Z'} (\text{CONS } \text{'A'} (\text{CONS } \text{'B'} \overset{3}{\text{'NIL'}}))))], \\ \text{RESTR } [\mathcal{A}.1:\neq:\text{'Z'}, \mathcal{A}.2:\neq:\text{'Z'}])$$

$$\mathcal{O}_{1,2}^{\text{str}} = ([\text{CONS } \overset{1}{\text{'X'}} (\text{CONS } \overset{2}{\text{'Y'}} (\text{CONS } \text{'Z'} (\text{CONS } \text{'A'} (\text{CONS } \text{'B'} \overset{42}{\text{'NIL'}}))))], \\ \text{RESTR } [\mathcal{A}.1:\neq:\text{'A'}, \mathcal{A}.1:\neq:\text{'Z'}, \mathcal{A}.2:\neq:\text{'A'}, \mathcal{A}.2:\neq:\text{'Z'}])$$

Глава 7. Окрестностное тестирование программ

7.1 Тестирование. Основные понятия

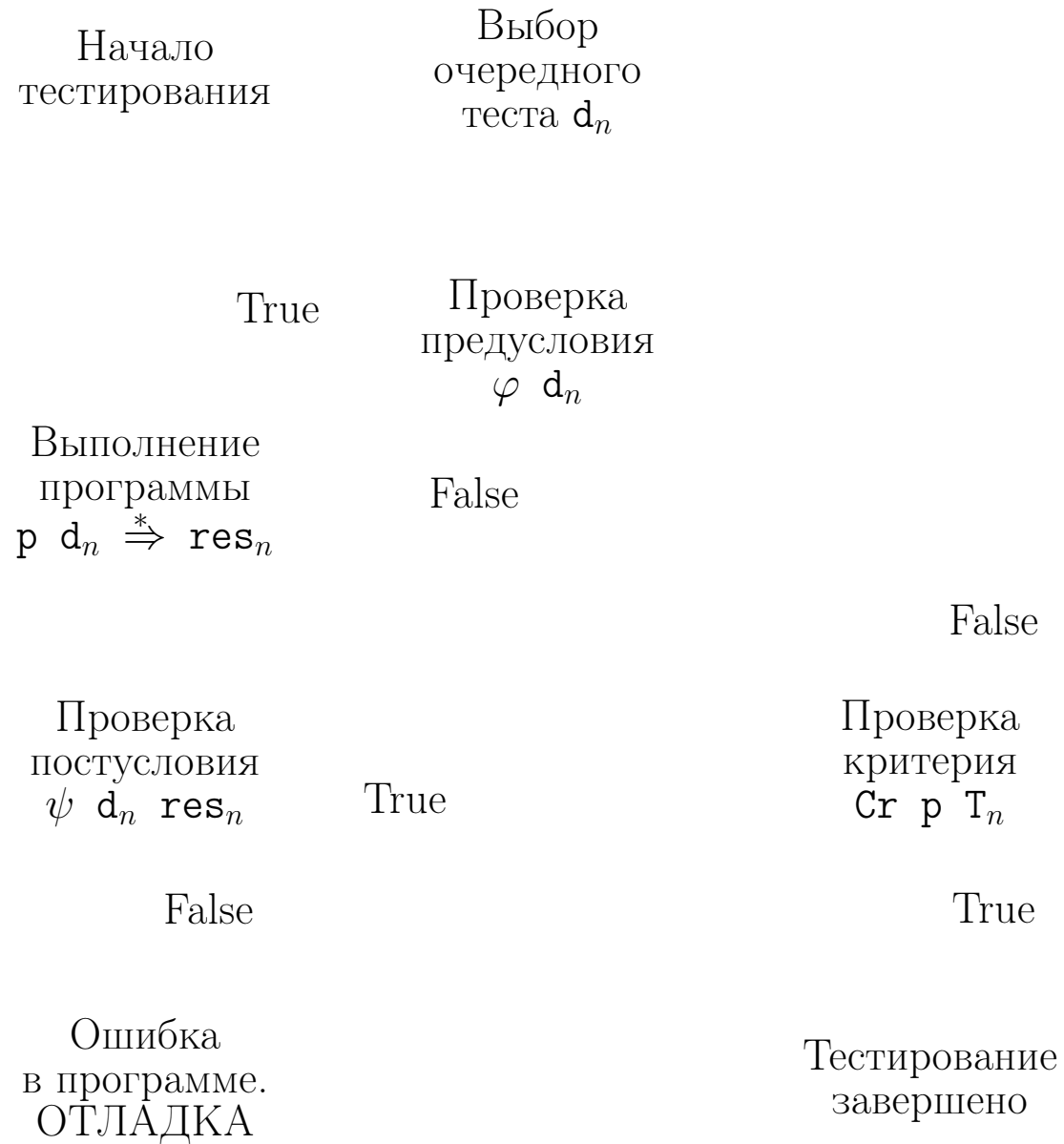
- Программа p на языке L определяет *функцию программы*: $p :: D \rightarrow D$. Возможно расхождение между *желаемой* и *реальной* функцией программы.
- *Спецификация* описывает требования на допустимые входные данные для p и желаемые соотношения между входными данными и результатом: $ps=(\varphi, \psi)$, где $\varphi :: D \rightarrow \text{Bool}$ —предусловие, $\psi :: D \rightarrow D \rightarrow \text{Bool}$ —постусловие.
- p *частично корректна* относительно φ и ψ :

$$\forall d : ((\varphi \ d) \& (\text{терминируется } p \ d \xRightarrow{*} \text{res})) \Rightarrow (\psi \ d \ \text{res}).$$
- p *корректна* относительно φ и ψ :

$$\forall d : (\varphi \ d) \Rightarrow ((\text{терминируется } p \ d \xRightarrow{*} \text{res}) \& (\psi \ d \ \text{res})).$$
- Как правило в *теории тестирования* рассматривают частичную корректность программ, в *теории верификации*—оба вида корректности.
- Когда спецификация задана формально, можно попытаться *верифицировать* ее—доказать корректность как формальную теорему. На практике доказывать программы бывает трудно, в этом случае прибегают к тестированию.

7.2 Процесс тестирования программы

- *Тестирование программы p*: серия *тестовых прогонов* программы над *тестами* $d_i \in D$, удовлетворяющими предусловию. После каждого *тестового прогона* $p \ d_i \xRightarrow{*} res_i$ проверяют истинность постусловия.
- Пусть $T_n = [d_1, \dots, d_n]$ набор тестов. Используют терминологию:
 - Тест d_i пройден программой p неуспешно (d_i —неуспешный тест для программы p , d_i вскрывает ошибку в p), если \dots (Аналогично для T_n).
 - Тест d_i пройден программой p успешно (d_i —успешный тест для программы p), если \dots (Аналогично для T_n).
- *Критерия выбора тестов Cr* определяет, когда конечный набор тестов T_n *можно считать достаточным* для тестирования p . Пока ограничимся рассмотрением критериев Cr такого класса: $Cr :: P \rightarrow TS \rightarrow Bool$, где $TS = [D]$.



7.3 Структурные критерии выбора тестов

Среди критериев выбора тестов, использующих только информацию о наборе тестов и тестируемой программе, наиболее известными являются *структурные критерии*:

- *Тестирование команд*. Набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе в совокупности было обеспечено прохождение каждой достижимой команды данной программы.
- *Тестирование ветвей*. Набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе в совокупности было обеспечено прохождение каждого реализуемого варианта ветвления для каждой команды ветвления в данной программе.
- *Тестирование путей*. Набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе в совокупности было обеспечено прохождение каждого реализуемого пути в *графе управляющей структуры—блок-схеме*—данной программы, с числом итераций циклов 0, 1 и 2.

В силу неразрешимости проблемы распознавания достижимости команды в программе структурные критерии невычислимы.

7.4 Свойства критериев выбора тестов

В теории тестирования изучают следующие свойства критериев выбора тестов.

- *Полнота*. Критерий называют полным, если в случае наличия ошибки в программе существует набор тестов, удовлетворяющий данному критерию и раскрывающий ошибку.
- *Непротиворечивость (надежность)*. Любые два набора тестов, удовлетворяющие критерию, должны быть одновременно успешными или неуспешными для данной программы.
- *Простота проверки*—критерий должен “легко” проверяться, например, он должен быть *вычислимым*.

Данные свойства недостижимы на практике:

1. структурные критерии не являются ни надежными, ни вычислимыми;
2. известно, что *не существует* полного непротиворечивого критерия, зависящего от набора тестов и программы;
3. известно, что *не существует* вычислимого полного непротиворечивого критерия, зависящего от набора тестов, программы и спецификаций.

Подмена свойств критериев “разрешимыми приближениями”

- *Исследование полноты и надежности критериев на ограниченном классе программ и/или ограниченном классе ошибок.*

Как правило, такому исследованию поддаются очень узкие классы программ и/или ошибок, которые трудно признать имеющими отношение к реальной практике программирования.

- *Статистические исследования “реальной” полноты и надежности критериев.*

Например, на выбранных из практики реальных программах с реальными ошибками.

- *Введение числовой эвристической оценки полноты тестирования вместо проверки невычислимого критерия.*

Например, в системах тестирования, основанных на структурных критериях, оператору может выдаваться числовой коэффициент—“процент проверенной части” программы.

7.5 Постановка проблемы. Критика структурных критериев

Идея, структурных критериев: изучение лабиринта—побывать во всех комнатах, пройти все варианты ветвлений и все пути, чтобы *получить полную информацию* о лабиринте. Можно предположить, что структурные критерии есть различные формализации *неформального* критерия выбора тестов:

Набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе *в совокупности была использована вся информация о графе управляющей структуры—блок-схеме—данной программы.* (7.1)

Различные подходы к формализации фразы—“использована вся информация о графе. . .”—приводят к различным структурным критериям. **Недостатки:**

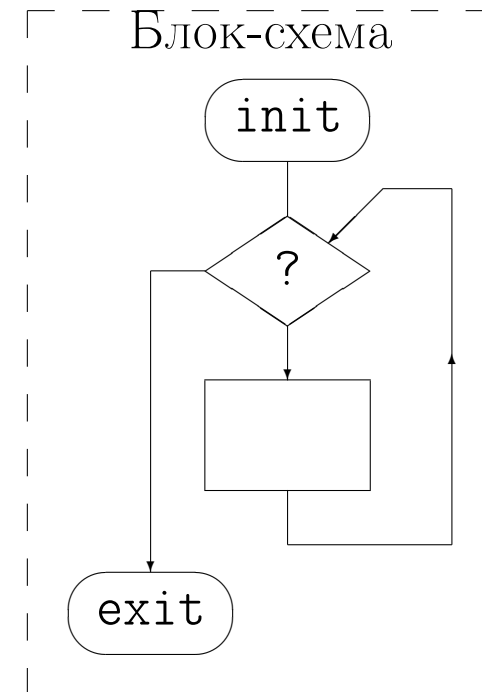
- Явно ориентированы на классические императивные языки.
- Цель тестирования—проверка *функции программы* на совпадение с желаемой функцией, а полнота проверки определяется по полноте использования информации об *управляющей структуре* программы. Но разве функция программы полностью определяется только управляющей структурой программы? *Мысленный эксперимент. . .*
- Структурные критерии будут особенно ненадежны для программ, функция которых “в малой степени” определяется управляющей структурой программы.

Конечно-автоматное преобразование строки

```

function F(x: string): string;
type  state = (q0, q1,... ); { множество состояний автомата }
const NewState:  array[state, char] of state = (...constants...);
      OutputChar: array[state, char] of char  = (...constants...);
var   i: integer;
      y: string; { результат }
      q: state;  { тек.состояние автомата}
begin
  q := q0;
  y := '';
  for i:=1 to length(x) do begin
    y := y + OutputChar[q,x[i]];
    q :=      NewState[q,x[i]];
  end;
  F := y;
end;

```



7.6 Постановка проблемы. Неформальный предельный критерий

- *Анализ только управляющей структуры программы—одна из возможных причин ненадежности структурных критериев.*
- *Апелляция к понятию “управляющая структура” ограничивает область применения критериев (только классические императивные языки).*

Как исправить? Для оценки полноты тестирования надо анализировать *то, что* полностью определяет функцию программы: *текст программы целиком!*

Набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе *в совокупности было обеспечено использование всей информации о полном тексте* данной программы. (7.2)

Для *формализации* критерия 7.2 будет использован *окрестностный анализ*.

7.7 Метасистемные уровни в окрестностном тестировании

1. “Что” использует информацию о тексте программы p в процессе тестовых прогонов? —переход от

$$p \ d_i \xrightarrow[L]{*} res_i$$

к

$$intL(p:d_i) \xrightarrow[R]{*} res_i.$$

2. Следующий шаг формализации—определение понятия “использована информация”, введение метасистемы, наблюдающей, как интерпретатор $intL$ выполняет вычисление $p \ d_i$ и при этом *потребляет* информацию о p . Это окрестностный анализатор

$$nan \ intL \ (p:d_i) \xRightarrow{*} (res_i, \mathcal{O}_i^{(p:d_i)})$$

Метауровни:

1. $(p:d)$ —объекты, данные для $intL$;
2. $intL$ —система, выполняет вычисление $p \ d_i \xRightarrow{*} res_i$;
3. nan —метасистема, наблюдает, как работает система $intL$ на данных $(p:d)$, определяет, какая информация о данных $(p:d)$ при этом используется системой $intL$.

7.8 Шаг окрестностного тестирования

Окрестностное тестирование—последовательность $(i = 1, 2, \dots)$ вычислений:

$$\begin{aligned} \text{nan intL } (p:d_i) &\stackrel{*}{\Rightarrow} (\text{res}_i, \mathcal{O}_i^{(p:d_i)}) \\ \text{decompose } \mathcal{O}_i^{(p:d_i)} [p] \ d_i &\stackrel{*}{\Rightarrow} (\mathcal{O}_i^p, \mathcal{O}_i^{d_i}) \\ \overline{\mathcal{O}}_i^p &= \mathcal{O}_1^p * \mathcal{O}_2^p * \dots * \mathcal{O}_i^p \end{aligned}$$

со следующими результатами на каждом шаге i :

$$\begin{aligned} \text{res}_i &\text{—результат R-вычисления intL } (p:d_i) \text{ и L-вычисления } p \ d_i \stackrel{*}{\underset{L}{\Rightarrow}} \text{res}_i; \\ \mathcal{O}_i^{(p:d_i)} &\text{—окрестность } (p:d_i), \text{ (информационная интерпретация...)}, \\ \mathcal{O}_i^p \text{ и } \overline{\mathcal{O}}_i^p &\text{—окрестности } p, \text{ (информационная интерпретация...)}, \\ \mathcal{O}_i^{d_i} &\text{—окрестность } d_i, \text{ (информационная интерпретация...)} \end{aligned}$$

Заметим, что по определению $\overline{\mathcal{O}}_k^p$ выполнено:

$$\begin{aligned} \overline{\mathcal{O}}_1^p &= \mathcal{O}_1^p, \quad \overline{\mathcal{O}}_{k+1}^p = \overline{\mathcal{O}}_k^p * \mathcal{O}_{k+1}^p, \quad (k = 1, 2, \dots), \\ \overline{\mathcal{O}}_1^p &\succeq \overline{\mathcal{O}}_2^p \succeq \dots \succeq \overline{\mathcal{O}}_k^p \succeq \overline{\mathcal{O}}_{k+1}^p \succeq \dots \quad \text{в данной последовательности окрестностей не} \\ &\quad \text{может быть бесконечного числа отношений } (\succ). \end{aligned}$$

7.9 Окрестностный критерий выбора тестов

Определение. Окрестность $\overline{\mathcal{O}}_k^p$ *неподвижная* если независимо от выбора теста d_{k+1} выполнено: $\overline{\mathcal{O}}_{k+1}^p = \overline{\mathcal{O}}_k^p$.

Интерпретация. Если $\overline{\mathcal{O}}_k^p$ — неподвижная, то для любого d_{k+1} никакая новая информация о p не может быть использована в процессе вычислений p на тесте d_{k+1} .

Другими словами, *вся информация о p , которая необходима для проведения вычисления программы p на любых данных*, была использована в тестовых прогонах программы p на тестах d_1, \dots, d_k .

То есть, неподвижность $\overline{\mathcal{O}}_k^p$ означает, что в тестовых прогонах p на тестах d_1, \dots, d_k, d_{k+1} была использована вся доступная информация о p .

Определение. *Окрестностный критерий тестирования:* Набор (конечный список) тестов $T_k = [d_1, \dots, d_k]$ будем считать достаточным для тестирования программы p , если $\overline{\mathcal{O}}_k^p$ является неподвижной окрестностью p .

7.10 Свойства окрестностного тестирования

Теорема 19 Для любой программы p существует конечный набор тестов T , удовлетворяющий окрестностному критерию тестирования.

Идея доказательства: рассмотрим любое перечисление множества $\text{dom}(p)$ данных, на которых p терминируется...

Теорема 20 *Окрестностный критерий тестирования программ—полный*: в случае наличия ошибки в программе p существует конечный набор тестов T' , раскрывающий ошибку и удовлетворяющий окрестностному критерию.

- Известно: не существует полного надежного (непротиворечивого) критерия, зависящего от набора тестов и программы.
- Далее будет показано, что критерий окрестностного тестирования программы p с заданной спецификацией ps не только полный, но и непротиворечивый.

Следовательно, *критерий окрестностного тестирования алгоритмически неразрешим*.

- В общем случае окрестностный критерий тестирования должен проверяться экспертно.

В некоторых случаях неподвижность $\overline{\mathcal{O}}_n^p$ может быть очевидна.

В остальных случаях полагаемся на то, что предлагаемые экспертам окрестности $\overline{\mathcal{O}}_n^p$ имеют достаточно наглядную форму.

Еще возможно ввести эвристическую числовую оценку близости окрестности $\overline{\mathcal{O}}_n^p$ к неподвижной окрестности *(на следующем слайде)*.

- Еще одно свойство: для окрестностного тестирования несущественны *характеристики языка* L .

слайд про эвристическую числовую оценку

7.11 Традиционные проблемы тестирования. Выбор очередного теста

Пусть успешно выполнено тестирование p для набора $T_n = [d_1, \dots, d_n]$, и критерий для T_n не выполнен. Как выбрать следующий тест d_{n+1} ? Естественно, желаем продвижения в сторону выполнения критерия...

- Для структурных критериев—проблема алгоритмически неразрешима.
- Если взять $d_{n+1} \in \langle \mathcal{O}_i^{d_i} \rangle$, где $i \in [1..n]$, то получим текстуальное раженство результатов, трасс, окрестностей.

$$(p:d_{n+1}) \in \mathcal{O}_i^{(p:d_i)} \Rightarrow \text{tr}(\text{int}, (p:d_{n+1})) = \text{tr}(\text{int}, (p:d_i)),$$

$$p \ d_{n+1} = p \ d_i = \text{res}_i, \quad \mathcal{O}_{n+1}^{(p:d_{n+1})} = \mathcal{O}_i^{(p:d_i)}, \quad \mathcal{O}_{n+1}^p = \mathcal{O}_i^p, \quad \overline{\mathcal{O}}_{n+1}^p = \overline{\mathcal{O}}_n^p.$$

В вычислении p на d_{n+1} не будет использовано никакой новой информации о p .

- **Вывод:** d_{n+1} необходимо выбирать из следующего множества:

$$\langle D_p \rangle \setminus (\langle \mathcal{O}_1^{d_1} \rangle \cup \dots \cup \langle \mathcal{O}_n^{d_n} \rangle) = \langle Q_n \rangle, \text{ где}$$

$$Q_n = D_p \setminus [\mathcal{O}_1^{d_1} + \dots + \mathcal{O}_n^{d_n}] = [C_1^Q + \dots + C_{k_n}^Q].$$

Разумно взять k_n новых тестов, по одному из каждого C_i^Q , $i = 1, \dots, k_n$.

Утверждение 21 Если $\langle Q_n \rangle = \emptyset$, то $\overline{\mathcal{O}}_n^p$ неподвижная окрестность.

слайд про выбор тестов

7.12 Традиционные проблемы тестирования. Тестирование и отладка. Локализация ошибки

Предположим, что успешно пройден набор тестов T_n и неуспешно—тест d_{n+1} . Программа p содержит ошибку. Где прежде всего следует ее искать?

- *Предположим, что мы хотим сохранить не только результаты вычислений $p \ d_i = \text{res}_i$, но и трассы $\text{tr}(\text{int}, (p : d_i))$ успешных тестовых прогонов на данных d_1, \dots, d_n . В таком предположении:*
 - Исправленная программа p' должна удовлетворять требованию $p' \in \overline{\mathcal{O}}_n^p$. Это означает, что исправления должны вноситься в надчеркнутые фрагменты в эксплицированной записи $\overline{\mathcal{O}}_n^p$.
 - С другой стороны, результат $p' \ d_{n+1}$ должен отличаться от неверного результата $p \ d_{n+1}$. Таким образом, $p' \notin \langle \mathcal{O}_{n+1}^p \rangle$, т.е. $p' \in \langle \overline{\mathcal{O}}_n^p \rangle \setminus \langle \mathcal{O}_{n+1}^p \rangle$. Это еще больше локализует места возможных исправлений.
- В случае, если нет уверенности в необходимости сохранения трасс успешных прогонов на наборе T_n , надо данную рекомендацию о местах внесения исправлений рассматривать как *эвристику* и начинать поиск ошибки именно с этих мест.

7.13 Традиционные проблемы тестирования. Продолжение тестирования после исправления ошибки

Предположим, что ошибка найдена, программа p исправлена, p' — новый текст программы. Нужно ли повторять все вычисления p' на *всех* тестах d_1, \dots, d_n ? Или можно гарантировать, что на части тестов из T_n программа p' будет успешно работать? Как отобрать из T_n эти тесты?

- Если выполнено условие $p' \in \langle \overline{\mathcal{O}}_i^p \rangle$, где $i \in [1..n]$, то нет необходимости выполнять вычисление p' на тесте d_i . Действительно:

$$\begin{aligned} p' \in \langle \overline{\mathcal{O}}_i^p \rangle &\Rightarrow (p' : d_i) \in \langle \mathcal{O}_i^{(p:d_i)} \rangle \Rightarrow \\ \text{tr}(\text{intL}, (p' : d_i)) &= \text{tr}(\text{intL}, (p : d_i)), \quad p' d_i = p d_i = \text{res}_i, \\ \mathcal{O}_i^{(p' : d_i)} &= \mathcal{O}_i^{(p : d_i)}, \quad \mathcal{O}_i^{p'} = \mathcal{O}_i^p. \end{aligned}$$

Таким образом, необходимо только пересчитать $\mathcal{O}_i^{d_i}$:

$$\text{decompose } \mathcal{O}_i^{(p' : d_i)} [p'] d_i = (\mathcal{O}_i^{p'}, \mathcal{O}_i^{d_i})$$

- В тех случаях, когда программа p достаточно сложная, указанная проверка $p' \in \langle \overline{\mathcal{O}}_i^p \rangle$ и пересчет $\mathcal{O}_i^{d_i}$ займет значительно меньшее время, чем выполнение вычисления p' на тесте d_i .

7.14 Традиционные проблемы тестирования. Использование в тестировании спецификации программы

Когда задана формальная спецификация программы, желаем основывать выбор тестов не только на анализе текста программы, но и на анализе текста спецификаций. Хотелось бы достичь формализации следующего критерия: *на наборе тестов T_n при проверке предусловий, тестовых прогонах и проверке постусловий должна быть использована вся информация о тексте программы и тексте спецификаций.*

- Пусть S —язык высокого уровня для написания спецификаций программ, φ —предусловие, и ψ —постусловие для $p \in L$, $\varphi, \psi \in S$ —одно и двухместный предикаты на D : $(\varphi [d]) \in \{\text{"True" "False"}\}$, $(\psi [d, res]) \in \{\text{"True" "False"}\}$, ‘True’, ‘False’—два различных значения из D , $\text{int} S \in R$ —интерпретатор языка S , написанный на языке R .
- Процесс тестирования p с заданной спецификацией $ps = (\varphi, \psi)$ можно рассматривать как процесс тестирования составной программы $\text{corr}(\varphi, p, \psi)$, с ожидаемой функцией: ‘True’ на всей области D (*пунктирная граница на слайде 3*).
- Будет удобно рассмотреть данную составную программу $\text{corr}(\varphi, p, \psi)$ явно, как программу, написанную на некотором языке JobSL, являющемся *надстройкой, языком заданий* над языками S, L .

Программа $\text{corr}(\varphi, p, \psi)$ на языке JobSL

```

corr:: (S,L,S) -> JobSL
corr(phi,p,psi)=
  (JOB [ Var"d" ]

    (ExecS phi [Var"d"] {-->-} (Var"phi d")
    (IF (Var "phi d")
-- THEN
      (ExecL p [Var"d"] {-->-} (Var"res")
      (ExecS psi [Var"d",Var"res"] {-->-} (Var"psi d res")
      (IF (Var "psi d res")
-- THEN
        (EXIT (Const "True")))
-- ELSE
        (EXIT (Const "False")) -- Error!
      )))
-- ELSE
  (EXIT (Const "True")))
) )) -- End of JOB

```

Синтаксис конструкций языка JobSL

$$\begin{aligned}
 job & ::= (\text{JOB } [jvar_1, \dots, jvar_n] jterm) && -- n \geq 0 \\
 jterm & ::= (\text{ExecS } progS [jexp_1, \dots, jexp_n] jvar && -- n \geq 0 \\
 & \quad jterm) && -- progS \in S \\
 & \quad | (\text{ExecL } progL [jexp_1, \dots, jexp_n] jvar && -- n \geq 0 \\
 & \quad jterm) && -- progL \in L \\
 & \quad | (\text{IF } jexp \ jterm_1 \ jterm_2) \\
 & \quad | (\text{EXIT } e\text{-}exp) \\
 jvar & ::= (\text{Var } name) \\
 jexp & ::= jvar \\
 & \quad | (\text{Const } d) && -- d \in D \\
 jbind & ::= (jvar := d) && -- d \in D \\
 jenv & ::= [jbind_1, \dots, jbind_n] && -- n \geq 0
 \end{aligned}$$

Вспомогательные функции интерпретатора языка JobSL

```
instance Eq JBind where
    ((Var n1):=_ ) == ((Var n2):=_ ) = (n1==n2)

mkJEnv :: [JVar] -> [D] -> JEnv
mkJEnv jvs ds = zipWith (\ jv d -> (jv:=d)) jvs ds

(+.) :: JEnv -> JEnv -> JEnv
je1 +. je2 = nub (je2++je1)

calcJExp :: JExp -> JEnv -> D
calcJExp (Const d) je=d
calcJExp (Var name) je=head[ d | ((Var n):=d)<-je, n==name]

calcJExps :: [JExp] -> JEnv -> [D]
calcJExps jexs je = map (\ jex -> calcJExp jex je) jexs
```

Интерпретатор языка JobSL (1/2)

```
intJobSL :: JobSL->[D]->D
intJobSL job@(JOB jvs jt0) d = evalJ jt0 je0
                                where je0 = mkJEnv jvs d
```

[illegible]

```

evalJ (ExecL p jexs jv jt') je = evalJ jt' je'
      where args = calcJExps jexs je
            res   = intL p args
            je'    = je +. [jv := res]

```


Интерпретатор языка JobSL (2/2)

```
evalJ (IF jex jt1 jt2)      je =  
                             if ((calcJExp jex je) == "True")  
                               then evalJ jt1 je  
                               else evalJ jt2 je
```

```
evalJ (EXIT jex)           je = calcJExp jex je
```

```
intL :: L->[D]->D
```

```
-- ... описание интерпретатора языка L
```

```
intS :: S->[D]->D
```

```
-- ... описание интерпретатора языка S
```

Окрестностное тестирование $\text{corr}(\varphi, p, \psi)$ (1/2)

$\text{nan intJobSL } (\text{corr}(\varphi, p, \psi) : d_i) \xRightarrow{*} (\text{res}_i, \mathcal{O}_i^{(\text{corr}(\varphi, p, \psi) : d_i)})$

$\text{decompose } \mathcal{O}_i^{(\text{corr}(\varphi, p, \psi) : d_i)} [\text{corr}(\varphi, p, \psi)] d_i \xRightarrow{*} (\mathcal{O}_i^{\text{corr}(\varphi, p, \psi)}, \mathcal{O}_i^{d_i})$

$\overline{\mathcal{O}}_i^{\text{corr}(\varphi, p, \psi)} = \mathcal{O}_1^{\text{corr}(\varphi, p, \psi)} * \mathcal{O}_2^{\text{corr}(\varphi, p, \psi)} * \dots * \mathcal{O}_i^{\text{corr}(\varphi, p, \psi)}$

$i = 1, 2, \dots$

- Единственным допустимым результатом является $\text{res}_i = \text{“True”}$.
- К тестированию $\text{corr}(\varphi, p, \psi)$ применимы все предыдущие результаты.
- Текст $\text{corr}(\varphi, p, \psi)$ включает в себя полный текст p и $\text{ps} = (\varphi, \psi)$, вычисление $\text{corr}(\varphi, p, \psi) d_i$ в общем случае включает в себя проверку предусловия φd_i , вычисление программы $p d_i \xRightarrow{*} \text{res}_i$ и проверку постусловия $\psi d_i \text{res}_i$.

Набор $T_n = [d_1, \dots, d_n]$ будет считаться достаточным для тестирования p с заданной спецификацией $\text{ps} = (\varphi, \psi)$, если на данных тестах в совокупности в процессах:

- проверок предусловий φd_i ;
- вычисления программы $p d_i$;
- проверок постусловий $\psi d_i \text{res}_i$;

была использована вся доступная информация о тексте программы p и о тексте спецификации $\text{ps} = (\varphi, \psi)$.

Окрестностное тестирование $\text{corr}(\varphi, p, \psi)$ (2/2)

Теорема 22 *Окрестностный критерий тестирования программы p с заданной спецификацией $ps=(\varphi, \psi)$ является надежным.* Пусть выполнено окрестностное тестирование программы $\text{corr}(\varphi, p, \psi)$ на конечном наборе тестов T_n , результаты всех тестовых прогонов $\text{res}_i = \text{‘True’}$, $i = 1, \dots, n$ и выполнен критерий окрестностного тестирования для набора тестов T_n . Тогда, программа p частично корректна по отношению к предусловию φ и постусловию ψ .

Следствие 23 *Неразрешимость окрестностного критерия тестирования.* Не существует алгоритма, который определяет, является ли окрестность \overline{O}_n^p неподвижной.

Глава 8. Нестандартные семантики

Рассмотрим схемы выполнения шага окрестностного тестирования и инверсного вычисления программ в произвольном языке программирования L :

$$\begin{aligned} \text{nbh intL } p \quad d &\xRightarrow{*} (\text{res}, \mathcal{O}^{(p:d)}) \\ \text{inv intL } p \quad (x, y) &\xRightarrow{*} [(s_1, r'_1), (s_2, r'_2), \dots] \end{aligned}$$

Можно связать с любой программой $p \in L$, кроме стандартной функции программы p , еще две функции p_{nbh} и p_{inv} :

$$\begin{array}{ll} p \quad d & \stackrel{\text{def}}{=} \text{intL } p \quad d & p & :: D \rightarrow D \\ p_{nbh} \quad d & \stackrel{\text{def}}{=} \text{nbh intL } p \quad d & p_{nbh} & :: D_{nbh} \rightarrow R_{nbh} \\ p_{inv} \quad (x, y) & \stackrel{\text{def}}{=} \text{inv intL } p \quad (x, y) & p_{inv} & :: D_{inv} \rightarrow R_{inv} \end{array}$$

где $D_{nbh}=D$, $R_{nbh}=(D, \text{Class})$, $D_{inv}=(\text{Class}, D)$, $R_{inv}=[(\text{Subst}, \text{Restr})]$.

Обобщая данный вывод, предложена следующая формальная модель модификатора семантик и нестандартных семантик.

Определение 10 Модификатор семантик *языков программирования*—произвольный алгоритм m , реализующий функцию со следующим типом:

$$m :: \text{ProgR} \rightarrow D \rightarrow D_m \rightarrow R_m.$$

Следующий язык L_m назовем модификацией языка L , порожденной модификатором m (m -модифицированным языком L):

- синтаксис программ: совпадает с синтаксисом языка L ;
- предметная область: входные данные программ— D_m , результаты вычисления программ— R_m ;
- семантика: с каждой программой $p \in D$ свяжем функцию p_m :

$$p_m \text{ d } \stackrel{\text{def}}{=} m \text{ intL } p \text{ d} \qquad p_m :: D_m \rightarrow R_m$$

Нестандартная семантика языка L , порожденная модификатором m (m -нестандартная семантика языка L)■

Приводятся несколько примеров “осмысленных” модификаторов семантик и дается описание методики построения подобных модификаторов семантик.

8.1 Использование специализации для эффективной реализации нестандартных семантик

- Неэффективность вычисления “напрямую” схемы $m \text{ intL } p \text{ d.} \Rightarrow$
Предложены методы *автоматического* построения эффективных реализаций нестандартных семантик, основанные на применении специализации программ.
- Специализация: основное свойство специализатора s (для случая двуместной специализируемой программы):

$$\begin{aligned} (s[p^{SD}, x]) \quad [y] &= p[x, y] & (s[p^{DS}, y]) \quad [x] &= p[x, y] \\ (s[p^{SS}, x, y]) \quad [] &= p[x, y] & (s[p^{DD} \quad]) \quad [x, y] &= p[x, y] \end{aligned}$$

- Специализация схемы вычисления m -нестандарной функции программы:

$$\begin{aligned} p_m \text{ d} &= m \text{ [intL, p, d]} = s[m^{SDD}, \text{intL}] \text{ [p, d]} \\ &= s[s^{SD}, m^{SDD}] \text{ [intL]} \text{ [p, d]} \\ &= s[s^{SD}, s^{SD}] \text{ [m}^{SDD}] \text{ [intL]} \text{ [p, d]} \end{aligned}$$

$$\begin{aligned} p_m \text{ d} &= m \text{ [intL, p, d]} = s[m^{SSD}, \text{intL, p}] \text{ [d]} \\ &= s[s^{SSD}, m^{SSD}, \text{intL}] \text{ [p]} \text{ [d]} \\ &= s[s^{SSD}, s^{SSD}, m^{SSD}] \text{ [intL]} \text{ [p]} \text{ [d]} \\ &= s[s^{SSD}, s^{SSD}, s^{SSD}] \text{ [m}^{SSD}] \text{ [intL]} \text{ [p]} \text{ [d]} \end{aligned}$$

Полученные проекции описывают схемы автоматического построения $(\forall m, \forall L, \forall \text{int}L)$:

- Трех программных средств одноуровневой m -нестандартной интерпретации:
 1. m -нестандартного интерпретатора языка L ;
 2. генератора m -нестандартных интерпретаторов;
 3. генератора генераторов нестандартных интерпретаторов.
- Четырех средств поддержки m -нестандартной компиляции с языка L в язык R :
 1. результата m -нестандартной компиляции программы p с языка L в язык R ;
 2. m -нестандартного компилятора с языка L на язык R ;
 3. генератора m -нестандартных компиляторов;
 4. генератора генераторов нестандартных компиляторов.

Приведенные выше семь проекций пока еще не были осуществлены на практике. Практическая реализация данных проекций потребует дальнейшего развития методов метавычислений. К счастью, совершенствовать методы метавычислений можно беспредельно. Точно так же не видно ограничений в создании все новых и новых методов применения метавычислений в практическом программировании.

*Таким образом, метавычисления всегда останутся
необъемлемым и благодатным полем для новых исследований.*