М. С. Нікітченко

Семантика мов програмування

Розділи 1, 6, 7 підручника
«Теорія програмування»

1. ФОРМАЛІЗАЦІЯ ПРОСТОЇ МОВИ ПРОГРАМУВАННЯ

Центральним завданням теорії, зокрема теорії програмування, є визначення та дослідження її основних понять. Зробити це не так просто, оскільки в теоріях зазвичай використовується багато понять, які пов'язані одне з одним. Тому спочатку доцільно ввести основні поняття теорії програмування на невеликому прикладі.

Розглянемо програму знаходження найбільшого спільного дільника двох чисел. Програма записана деякою простою мовою. У цьому розділі мова буде формалізована, тобто буде описано її синтаксис і семантику та їх зв'язок. Також будуть досліджені деякі властивості програм цієї мови.

1.1. Неформальний опис простої мови програмування

Для посилання на певну мову треба дати їй ім'я. Для імені часто використовують абревіатуру короткої характеристики мови. Зважаючи на традиції використання латинських літер у науковій символіці, утворимо абревіатуру ***SIPL*** від характеристики ***SI****mple* ***P****rogramming* ***L****anguage* (Проста Мова Програмування).

Цю абревіатуру можна розшифрувати по-іншому, указуючи на імперативність або структурованість мови:

SIPL **– S**imple **I**mperative **P**rogramming **L**anguage,

SIPL **– S**tructured **I**mperative **P**rogramming **L**anguage.

Говорячи неформально, мова *SIPL* має числа та змінні цілого типу, над якими будуються арифметичні вирази та умови. Основними операторами є присвоювання, послідовне виконання, розгалуження, цикл.

Мова *SIPL* може розглядатися як надзвичайно спрощена традиційна мова програмування. У мові *SIPL* відсутні складні типи

даних, оператори введення-виведення, процедури та багато інших конструкцій традиційних мов програмування. Також немає явної типізації. Разом з тим ця мова досить потужна для програмування різних арифметичних функцій, більше того, у ній можуть бути запрограмовані всі обчислювані функції над цілими числами. Мета розгляду саме такої мови програмування полягає в тому, щоб її формалізація та дослідження були якомога простішими.

Приклад 1.1. Програма *GCD* знаходження найбільшого спільного дільника чисел *M* та *N* за алгоритмом Евкліда:

GCD ≡
begin
    while ¬ *M* = *N* **do**
        if *M* > *N* **then** *M* := *M* – *N* **else** *N* := *N* – *M*
end

Тут ¬ *M* = *N* означає *M* ≠ *N*. Оскільки програми призначені для обчислення результатів за вхідними даними, то розглянемо неформально процес виконання цієї програми на вхідних даних, у яких *M* має значення 8, а *N* – 16. Вважаємо, що дані записуються в пам'ять, а оператори виконуються деяким процесором. Тому розмітимо програму, позначаючи оператори мітками:

0: **begin**
    1: **while** ¬ *M* = *N* **do**
        2: **if** *M* > *N* **then** 3: *M* := *M* – *N* **else** 4: *N* := *N* – *M*
end

Процес виконання програми можна подати у вигляді таблиці (табл. 1.1), кожний рядок якої вказує на номер виконуваного оператора та нові значення змінних (деталі процесу виконання можна знайти у книжках із основ програмування).

***Таблиця* 1.1**

Мітка	Значення умови	Значення *M*	Значення *N*
0		8	16
1	¬ *M* = *N* – *true*		
2	*M* > *N* – *false*		

Мітка	Значення умови	Значення *M*	Значення *N*
4			8
1	$\neg\, M = N - false$		

Програма припиняє роботу зі значеннями *M* та *N*, рівними 8. Це число і є найбільшим спільним дільником.

Аналізуючи програму та процес її виконання, зазначаємо два аспекти:

- *синтаксичний* (текст програми);
- *семантичний* (смисл програми – те, що вона робить).

У нашому прикладі ні синтаксис, ні семантика не задані точно (формально). Ми маємо лише інтуїтивне розуміння програм мови *SIPL*. Тому важко відповідати на запитання, які вимагають точного опису мови. Наприклад незрозуміло, чи можна поставити крапку з комою між оператором і символом ***end***, що дозволяється робити в багатьох мовах програмування (синтаксичний аспект). Незрозуміло також, чи завжди наведена програма буде обчислювати найбільший спільний дільник для довільних значень *M* та *N* (семантичний аспект).

Щоб відповідь на такі запитання стала можливою, необхідно дати строге (формальне) визначення нашої мови програмування. Лише тоді можна буде застосовувати математичні методи дослідження програм і будувати відповідні системи програмування, зокрема транслятори та інтерпретатори.

Безумовно, уточнення (експлікацію) синтаксичного та семантичного аспектів можна робити по-різному. Спочатку розглянемо традиційні формалізми для опису синтаксису мов.

1.2. Формальний опис синтаксису мови SIPL

Для опису синтаксису мов зазвичай використовують БНФ (форми Бекуса – Наура). Програми (або їхні частини) виводять із метазмінних (нетерміналів), які записують у кутових дужках. Метазмінні задають синтаксичні класи. У процесі виведення метазмінні замінюються на праві частини правил, що задають ці

метазмінні. Праві частини для однієї метазмінної розділяються знаком альтернативи " | ". Процес породження припиняється, якщо всі метазмінні замінено на термінальні символи (тобто символи без кутових дужок).

Синтаксис мови *SIPL* можна задати за допомогою такої БНФ (табл. 1.2):

***Таблиця* 1.2**

Ліва частина правила – метазмінна	**Права частина правила**	**Ім'я правила**
<програма> ::=	**begin** <оператор> **end**	*NP*1
<оператор> ::=	<змінна> := <вираз> | <оператор> ; <оператор>| **if** <умова> **then** <оператор> **else** <оператор> | **while** <умова> **do** <оператор> | **begin** <оператор> **end** | **skip**	*NS*1 *NS*2 *NS*3 *NS*4 *NS*5 *NS*6
<вираз> ::=	<число>	<змінна>	<вираз> + <вираз>	 <вираз> – <вираз>	<вираз> * <вираз>	 (<вираз>)	*NA*1 ... *NA*6
<умова> ::=	<вираз> = <вираз>	<вираз> > <вираз>	 <умова> ∨ <умова>	¬ <умова>	 (<умова>)	*NB*1 ... *NB*5
<змінна> ::=	. . . *M*	*N*	. . .	*NV*…
<число> ::=	. . . –1	0	1	2	3	. . .	*NN*…

Наведена БНФ задає мову *SIPL* як набір речень (слів), що виводяться з метазмінної <програма>. Точне визначення виведення буде подано пізніше, зараз тільки зазначимо, що виведення можна подати у вигляді дерева.

Щоб переконатись у синтаксичній правильності програми, треба побудувати її виведення. Зокрема, для нашої програми *GCD* маємо дерево виведення, подане на рис. 1.1.

Побудоване дерево синтаксичного виведення дозволяє стверджувати синтаксичну правильність програми *GCD*. Дійсно, якщо записати зліва направо послідовність усіх термінальних сим-

волів, то отримаємо програму *GCD*, що свідчить про її вивідність у БНФ мови *SIPL*, тобто про її синтаксичну правильність.

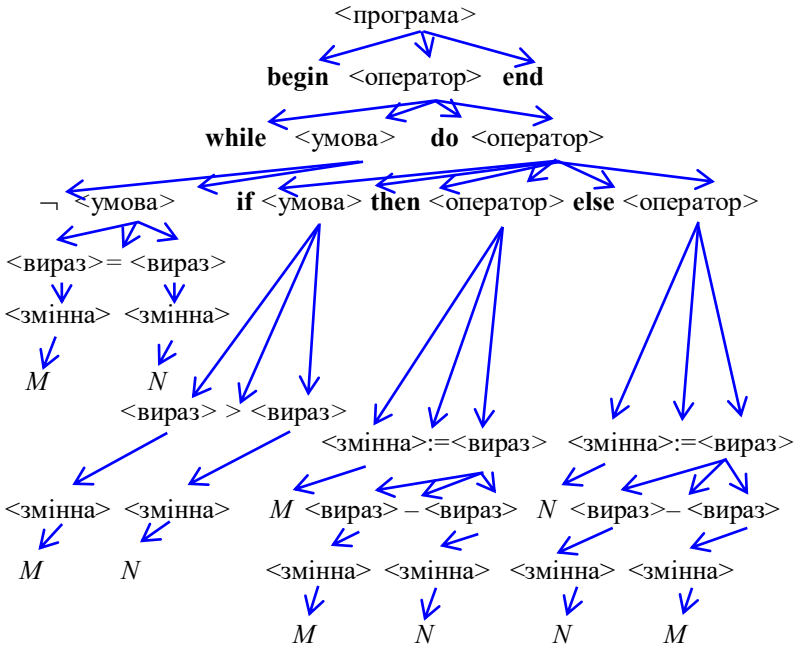

Рис. 1.1. Дерево синтаксичного виведення програми GCD

Аналізуючи далі наведену БНФ, можна зазначити, що вона досить просто описує основні конструкції мови *SIPL*. Однак зворотним аспектом цієї простоти є неоднозначність такої БНФ. Наприклад, вираз *X+Y*Z* має два різні дерева виведення, що ведуть до різних семантик (рис. 1.2). Тому неоднозначність БНФ значно ускладнює семантичний аналіз програм.

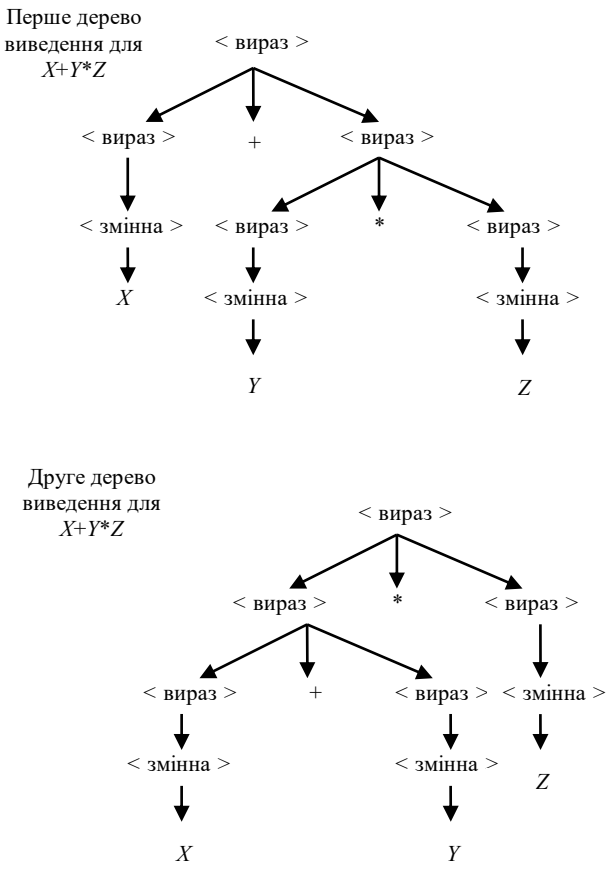

Рис. 1.2. Дерева синтаксичного виведення виразу *X+Y*Z*

Однозначності побудови дерева синтаксичного виведення можна досягти введенням пріоритетів операцій і правил асоціативності для операцій одного пріоритету. Слід зазначити, що в математиці при описі пріоритетів зазвичай не розрізняють синтаксичний і семантичний аспекти, тому часто кажуть, що пріоритети задають порядок виконання операцій (семантичний аспект) замість того, щоб говорити про порядок структурного аналізу виразу (синтаксичний аспект). Такий розгляд об'єкта, коли не виокремлюються його різні аспекти та складові, часто називають *синк-*

ретичним. У випадку мови *SIPL* однозначність аналізу може порушуватися при формуванні операндів операцій (синтаксичних конструкторів) різного типу. Будемо виокремлювати чотири типи операцій:

- арифметичні (бінарні операції +, –, *);
- порівняння (бінарні операції =, >);
- булеві (бінарна операція диз'юнкції $\vee$ та унарна операція заперечення $\neg$);
- оператори (присвоювання _:=_, оператор послідовного виконання _;_, умовний оператор **if_then_else**_ та оператор циклу **while_do**_).

Тут символ підкреслення позначає операнд (або параметр) оператора. Перші три групи операцій є традиційними, як і їх пріоритети. Четверта група потребує певного пояснення. Річ у тім, що, наприклад, синтаксична конструкція вигляду **if** *b* **then** *S*1 **else** *S*2 ; *S*3 може аналізуватися по-різному. Одне трактування може відносити до умовного оператора (його **else**-частини) послідовність операторів *S*2 ; *S*3, інше трактування – тільки один оператор *S*2. Різні трактування будуть задавати різну семантику. Те саме стосується і конструкцій вигляду **while** *b* **do** *S*1 ; *S*2 та *S*1 ; *S*2 ; *S*3. На семантику також впливає порядок застосування операцій одного типу. Наприклад, вираз $a1 - a2 - a3$ можна аналізувати як зліва направо (лівоасоціативно), що можна подати як $(a1 - a2) - a3$, так і справа наліво (правоасоціативно), що можна подати як $a1 - (a2 - a3)$ (= $(a1 - a2) + a3$). Щоб подолати неоднозначність синтаксичного аналізу, будемо вважати, що всі бінарні операції лівоасоціативні (обчислюються зліва направо) і що пріоритет операцій такий (у порядку зменшення):

1) множення *;
2) додавання + та віднімання –;
3) операції порівняння =, >;
4) заперечення $\neg$;
5) диз'юнкція $\vee$;
6) присвоювання _:=_;
7) оператор циклу **while_do**_;
8) умовний оператор **if_then_else**_;
9) послідовне виконання _;_.

Зауважимо ще раз, що однозначність синтаксичного аналізу не зовсім адекватно подається пріоритетами операцій, які мають семантичну природу. Тому можливі розбіжності між цими поняттями.

Є також друга можливість досягти однозначності БНФ мови *SIPL* – ввести нові метасимволи та нові правила їх визначення. Пропонуємо читачам зробити це самостійно.

Наведені вище позначення метазмінних не зовсім зручні, тому часто використовують форми, ближчі до математики. Уведемо позначення для всіх синтаксичних категорій і нові метазмінні, що вказують на представників цих категорій (табл. 1.3).

***Таблиця* 1.3**

Метазмінна	Синтаксична категорія	Нова метазмінна
<програма>	*Prog*	*P*
<оператор>	*Stm*	*S*
<вираз>	*Aexp*	*a*
<умова>	*Bexp*	*b*
<змінна>	*Var*	*x*
<число>	*Num*	*n*

У наведених позначеннях БНФ мови *SIPL* набуває такого вигляду (табл. 1.4):

***Таблиця* 1.4**

Ліва частина правила – метазмінна	Права частина правила	Ім'я правила
P::=	**begin** S **end**	$P1$
S::=	$x := a \mid S_1 ; S_2 \mid$ **if** b **then** S_1 **else** $S_2 \mid$ **while** b **do** $S \mid$ **begin** S **end** $\mid$ *skip*	$S1$–$S6$
a::=	$n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid (a)$	$A1$– $A6$
b::=	$a_1 = a_2 \mid a_1 > a_2 \mid b_1 \vee b_2 \mid \neg b \mid (b)$	$B1$ – $B5$
x::=	. . . $M \mid N \mid$ . . .	NV…
n::=	. . . –1 | 0 | 1 | 2 | 3 | . . .	NN…

Надалі будемо користуватися введеними позначеннями для запису структури програм та їхніх складових.

Зауваження 1.1. Табл. 1.3 та 1.4 дають можливість розглядати наведений формалізм подання синтаксису мову *SIPL* як певну

багатоосновну (синтаксичну) алгебру. Основами цієї алгебри є множини *Prog*, *Stm*, *Aexp*, *Bexp*, *Var* та *Num*, а операціями – перетворення, що задаються табл. 1.4.■

Дотепер ми користувались інтуїтивним розумінням БНФ. Разом із тим слід мати на увазі, що можуть існувати різні уточнення БНФ (про це йтиметься в розд. 4). Зараз приймемо, що БНФ дозволяє прописати індуктивне визначення синтаксичних категорій. Дійсно, наведена БНФ визначає шість синтаксичних категорій (класів): *Num*, *Var*, *Aexp*, *Bexp*, *Stm*, *Prog*. Вони можуть бути задані таким індуктивним визначенням:

1. База індукції:

- $Num = \{\ldots, -1, 0, 1, 2, 3, \ldots\}$;
- $Var = \{M, N, \ldots\}$;
- якщо $n \in Num$, то $n \in Aexp$;
- якщо $x \in Var$, то $x \in Aexp$;
- *skip* належить *Stm*.

2. Крок індукції:

- якщо a, a_1, $a_2 \in Aexp$, то записи (вирази)
  - $a_1 + a_2$,
  - $a_1 - a_2$,
  - $a_1 * a_2$,
  - (a)

належать *Aexp*,

- якщо a_1, $a_2 \in Aexp$, b , b_1, $b_2 \in Bexp$, то записи (умови)
  - $a_1 = a_2$,
  - $a_1 > a_2$,
  - $b_1 \vee b_2$,
  - $\neg b$,
  - (b)

належать *Bexp*,

- якщо $x \in Var$, $a \in Aexp$, $b \in Bexp$, S, S_1, $S_2 \in Stm$, то записи (оператори)
  - $x := a$,
  - S_1 ; S_2,
  - **if** b **then** S_1 **else** S_2,
  - **while** b **do** S,
  - **begin** S **end**

належать *Stm*,

- якщо $S \in Stm$, то запис **begin** *S* **end** належить *Prog*.

На підставі такого індуктивного визначення може бути задане визначення синтаксичних категорій за допомогою рекурентних співвідношень. Пропонуємо читачам зробити це самостійно.

Таким чином, можна стверджувати, що наведеними визначеннями синтаксис мови *SIPL* задано точно (формально).

Перейдемо тепер до визначення семантики мови *SIPL*.

1.3. Формальний опис семантики мови *SIPL*

Семантика задає значення (смисл) програми. Наш приклад показує, що смисл програми – перетворення вхідних *даних* у вихідні. У математиці такі перетворення називають *функціями*. Тому до семантичних понять відносять поняття даних, функцій і методів їх конструювання. Такі методи називаються *композиціями*, а відповідна семантика часто називається *композиційною*. Будемо вживати термін *композиційна семантика*, тому що саме композиції визначають її властивості. Композиційна семантика є певною конкретизацією *функціональної семантики* тому, що базується на тлумаченні програм як функцій.

Перейдемо до розгляду композиційної семантики мови *SIPL*.

1.3.1. Дані

Базові типи даних – множини цілих чисел, булевих значень та змінних (імен):

- $Int = \{ \ldots, -1, 0, 1, 2, \ldots \}$;
- $Bool = \{true, false\}$;
- $Var = \{\ldots, M, N, \ldots\}$.

Похідні типи – множина станів змінних (наборів іменованих значень, наборів змінних з їхніми значеннями):

- $State = Var \to Int$.

Приклади станів змінних:

$[M \mapsto 8, N \mapsto 16]$,

$[M \mapsto 3, X \mapsto 4, Y \mapsto 2, N \mapsto 16]$.

Визначимо тепер *операції* на базових типах даних. Щоб відрізнити операції від символів, якими вони позначаються, їх пишуть жирним шрифтом або вводять спеціальні позначення. Ми оберемо другий варіант, вводячи нові позначення для операцій на типах даних.

Операції на множині Int. Символам +, –, ∗ відповідають операції *add*, *sub*, *mult* (додавання, віднімання, множення). Це бінарні операції типу $Int^2 \to Int$.

Операції на множині Bool. Символам $\vee$, $\neg$ відповідають операції *or*, *neg*. Це бінарна операція типу $Bool^2 \to Bool$ (диз'юнкція) та унарна операція типу $Bool \to Bool$ (заперечення).

У мові також є *операції змішаного типу*. Символам операцій порівняння =, > відповідають операції *eq*, *gr*. Це бінарні операції типу $Int^2 \to Bool$.

Вивчення властивостей даних і операцій у математиці відбувається на основі поняття *алгебри* [3]. У першому наближенні алгебру можна тлумачити як множину із заданими на ній операціями. Для нашої мови маємо такі алгебри.

Алгебра цілих чисел: *A_Int* = <*Int*; *add*, *sub*, *mult*>.

Алгебра булевих значень: *A_Bool* = <*Bool*; *or*, *neg*>.

Якщо додати операції порівняння, отримаємо *двоосновну алгебру базових даних*:

A_Int_Bool = <*Int*, *Bool*; *add*, *sub*, *mult*, *or*, *neg*, *eq*, *gr*>.

Для опису мови *SIPL* треба додати ще одну основу – множину станів змінних. На цій основі задана бінарна операція *накладання* ∇ (іноді вживають термін *накладка*). Операція за двома станами будує новий стан змінних, до якого входять усі іменовані значення з другого стану і ті значення з першого стану, імена яких не входять до другого стану. Операція накладання подібна до операції копіювання каталогів зі спеціальним випадком однакових імен файлів у двох каталогах, коли файл із першого каталогу замінюється файлом з тим самим іменем із другого каталогу.

Наприклад:

$[M\mapsto 8, N\mapsto 16] \nabla [M\mapsto 3, X\mapsto 4, Y\mapsto 2] =$

$= [M\mapsto 3, N\mapsto 16, X\mapsto 4, Y\mapsto 2].$

Уведемо відразу також відношення розширення $\subseteq$ станів новими іменованими значеннями. Наприклад,

$[M\mapsto 8, N\mapsto 16] \subseteq [M\mapsto 8, X\mapsto 4, Y\mapsto 2, N\mapsto 16].$

Відношення не включаємо в алгебри, пов'язані з мовою *SIPL*, оскільки воно в цій мові безпосередньо не використовується, але буде корисним для доведення властивостей її програм.

Для створення й оперування зі станами змінних треба визначити дві функції: *іменування* $\Rightarrow x$: $Int \to State$ та *розіменування* $x\Rightarrow$: $State \to Int$, які мають параметр $x \in Var$:

- $\Rightarrow x(n) = [x\mapsto n]$;
- $x\Rightarrow(st) = st(x)$.

Тут і далі вважаємо, що $n \in Int$, $st \in State$. Перша функція іменує іменем x число n, створюючи стан $[x\mapsto n]$, друга бере значення імені x у стані st. Друга формула ґрунтується на тому факті, що стани змінних можуть тлумачитись як функції вигляду $Var \to Int$. Функція розіменування є частковою. Вона не визначена, якщо x не має значення у стані st.

Наприклад:

- $\Rightarrow M(5) = [M\mapsto 5]$;
- $Y\Rightarrow([M\mapsto 3, X\mapsto 4, Y\mapsto 2]) = 2$;
- $Y\Rightarrow([M\mapsto 3, X\mapsto 4])$ не визначене.

Крім того, введемо:

- *параметричну функцію-константу арифметичного типу* $\overline{n} : State \to Int$ таку, що $\overline{n}\,(st) = n$ $(n \in Int)$;
- *тотожну функцію* id: $State \to State$ таку, що $id(st) = st$.

Отримали багатоосновну *алгебру даних мови SIPL*

$$A\_Int\_Bool\_State = <Int, Bool, State;$$
$$add, sub, mult, or, neg, eq, gr, \Rightarrow x, x\Rightarrow, \overline{n}, id, \nabla>,$$

яка подана на рис. 1.3.

Зауваження 1.2. Є ще одна основа – *Var*. Оскільки ніяких операцій на ній у мові *SIPL* не задано, то до алгебри даних її не включаємо, але імена з *Var* використовуємо як параметри операцій іменування та розіменування.

1.3.2. Функції

Аналіз алгебри даних показує, що в мові *SIPL* можна вирізнити два види функцій: 1) *n*-арні функції на базових типах даних, 2) функції над станами змінних. Другий вид функцій будемо називати *номінативними функціями*. Назва пояснюється тим, що вони задані на наборах іменованих даних (латинське *nomen* – ім'я).

Визначимо тепер класи функцій, які будуть задіяні при визначенні семантики мови *SIPL*:

1. *n*-арні операції над базовими типами:
   - $FNA = Int^n \to Int$ – *n*-арні арифметичні функції (операції);
   - $FNB = Bool^n \to Bool$ – *n*-арні булеві функції (операції);
   - $FNAB = Int^n \to Bool$ – *n*-арні функції (операції) порівняння.
2. Функції над станами змінних:
   - $FA = State \to Int$ – *номінативні арифметичні функції*;
   - $FB = State \to Bool$ – *номінативні предикати*;
   - $FS = State \to State$ – *біномінативні функції-перетворювачі* (трансформатори) станів.

Для операцій мови *SIPL* зазвичай $n = 2$, а для булевої операції заперечення $n = 1$.

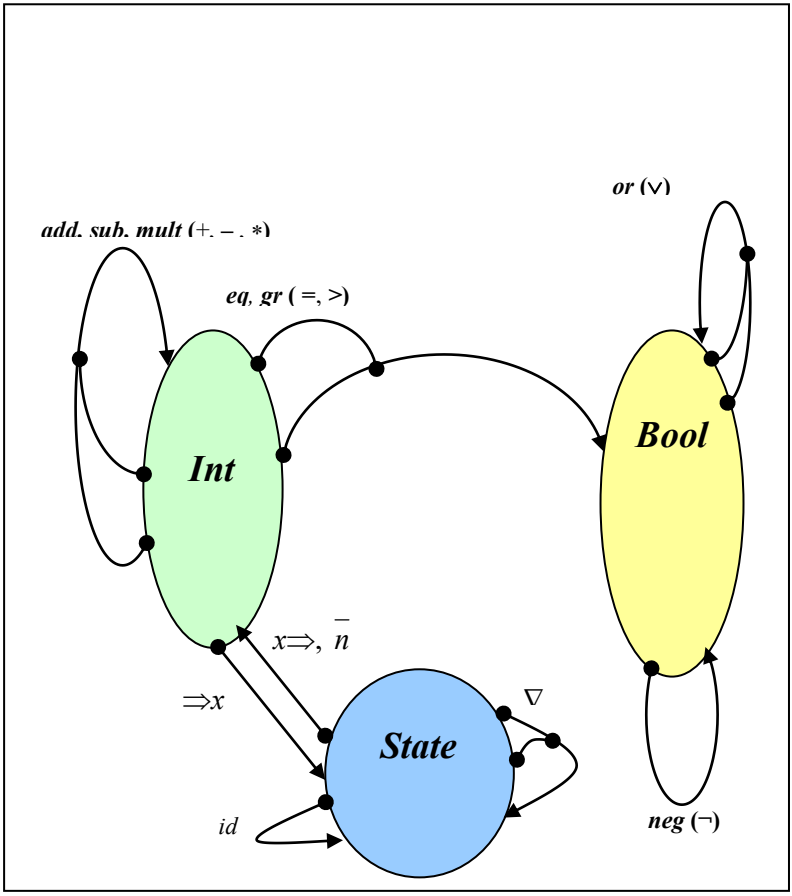

Рис. 1.3. Алгебра даних мови *SIPL*

1.3.3. Композиції

Нагадаємо, що композиції формалізують методи побудови програм. Аналіз мови *SIPL* дає підстави говорити про те, що будуть уживатися композиції різних типів, а саме:

- композиції, які пов'язані з номінативними функціями та предикатами;
- композиції, які пов'язані з біномінативними функціями.

Перший клас композицій використовується для побудови семантики арифметичних виразів і умов, другий – операторів.

Перший клас композицій складається із композицій *суперпозиції* в n-арні функції, які задані на різних основах (класах функцій):

- суперпозиція номінативних арифметичних функцій у n-арну арифметичну функцію має тип S^n: $FNA \times F^n \to FA$;
- суперпозиція номінативних арифметичних функцій у n-арну функцію порівняння має тип S^n: $FNAB \times FA^n \to FB$;
- суперпозиція номінативних предикатів у n-арну булеву функцію має тип S^n: $FNB \times FB^n \to FB$.

Зауваження 1.3. Суперпозиції різного типу позначаємо одним знаком.

Суперпозиція задається формулою

$$(S^n(f, g_1,\ldots, g_n))(st) = f(g_1(st),\ldots, g_n(st)),$$

де f – n-арна функція, $g_1,\ldots, g_n$ – номінативні функції відповідного типу.

Другий клас композицій складається із таких композицій.

- *Присвоювання* AS^x: $FA \to FS$ (x – параметр).

Присвоювання задається формулою

$$AS^x(fa)(st) = st\,\nabla[x \mapsto fa(st)].$$

- *Послідовне виконання* $\bullet$: $FS^2 \to FS$.

Послідовне виконання задається формулою

$$(fs_1 \bullet fs_2)(st) = fs_2(fs_1(st)).$$

- *Умовний оператор* (розгалуження): IF: $FB \times FS^2 \to FS$, задається формулою

$$IF(fb, fs_1, fs_2)(st) = \begin{cases} fs_1(st), & \text{якщо } fb(st) = true, \\ fs_2(st), & \text{якщо } fb(st) = false. \end{cases}$$

- *Цикл* (*ітерація з передумовою*): WH: $FB \times FS \to FS$, задається рекурентно (індуктивно) таким чином:

$$WH(fb, fs)(st) = st_n,$$

де $st_0 = st$, $st_1 = fs(st_0)$, $st_2 = fs(st_1),\ldots, st_n = fs(st_{n-1})$, причому

$fb(st_0) = true, fb(st_1) = true,\ldots, fb(st_{n-1}) = true, fb(st_n) = false.$

Важливо зазначити, що для циклу наведена послідовність визначається однозначно. Позначимо число n (кількість ітерацій циклу) як $NumItWH((fb, fs), st)$. Однозначність визначення n дозволяє розглядати $NumItWH((fb, fs), st)$ як тернарне відображення, яке залежить від fb, fs, st. Якщо цикл не завершується, то $NumItWH((fb, fs), st)$ вважається невизначеним. Це відображення буде використовуватись у індуктивних доведеннях властивостей циклу.

1.3.4. Програмні алгебри

Побудовані композиції дозволяють стверджувати, що отримано *алгебру функцій* (*програмну алгебру*)

$A\_Prog = <FNA, FNB, FNAB, FA, FB, FS; S^n, AS^x, \bullet, IF, WH, x\Rightarrow, id>$.

Функції іменування $\Rightarrow x$ і накладання ∇ не включені до переліку операцій цієї алгебри, тому що вони представлені композицією присвоювання. Також не включаємо функції-константи $\overline{n}$, які мають специфічну природу. Разом із тим до переліку композицій включено функції (нуль-арні композиції) розіменування й тотожна функція. Це пояснюється тим, що вказані нуль-арні композиції мають не предметну (специфічну), а логічну (загальнозначущу) природу, що дозволяє розглядати їх саме як загальні композиції (нехай і нуль-арні), а не як специфічні функції. Утім, цей розподіл досить умовний.

Наведена алгебра (рис. 1.4) у своїй основі містить багато "зайвих" функцій, які не можуть бути породжені в мові *SIPL*. Аналіз мови дозволяє стверджувати (цей факт буде доведено пізніше в теоремі 1.1), що функції, які задаються мовою *SIPL*, породжуються в алгебрі *A_Prog* з таких *базових функцій*:

- $add, sub, mult \in FNA$;
- $or, neg \in FNB$;
- $eq, gr \in FNAB$;
- $\overline{n} \in FA$ ($n \in Int$).

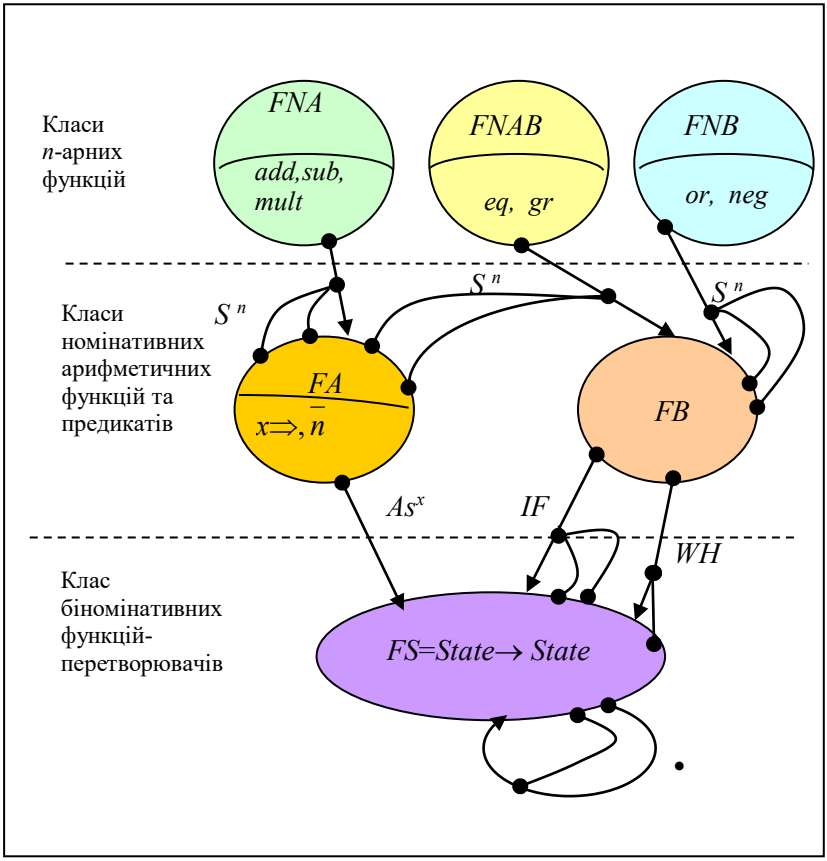

Рис. 1.4. Алгебра функцій (програмна алгебра)

Підалгебру алгебри *A_Prog*, породжену наведеними базовими функціями, назвемо *функціональною алгеброю мови SIPL* і позначимо *A_SIPL*.

Зауважимо, що всі функції алгебри *A_SIPL* є *однозначними* (детермінованими) функціями. Це випливає з того, що всі базові функції є однозначними, а композиції зберігають цю властивість. Прохання до читача довести зазначену властивість самостійно.

Формули для обчислення композицій і функцій алгебри *A_Prog* подамо в табл. 1.5 (тут f – n-арна функція, fa, $g_1,\ldots, g_n$ – номінативні арифметичні функції, fb – номінативний предикат, fs, fs_1, fs_2 – біномінативні функції, st – стан, n – число).

***Таблиця* 1.5**

Композиція	**Формула обчислення**	**Ім'я формули**
Суперпозиція	$(S^n(f, g_1,\ldots, g_n))(st) = f(g_1(st),\ldots, g_n(st))$	*AF_S*
Присвоювання	$AS^x(fa)(st) = st\,\nabla[x \mapsto fa(st)]$	*AF_AS*
Послідовне виконання	$fs_1 \bullet fs_2(st) = fs_2(fs_1(st))$	*AF_SEQ*
Умовний оператор	$IF(fb, fs_1, fs_2)(st) = \begin{cases} fs_1(st), & \text{якщо } fb(st) = true, \\ fs_2(st), & \text{якщо } fb(st) = false. \end{cases}$	*AF_IF*
Цикл	$WH(fb, fs)(st) = st_n$, де $st_0 = st$, $st_1 = fs(st_0)$, $st_2 = fs(st_1)$, …, $st_n = fs(st_{n-1})$, причому $fb(st_0) = true, fb(st_1) = true,\ldots,$ $fb(st_{n-1}) = true, fb(st_n) = false$.	*AF_WH*
Функція розіменування	$x\Rightarrow(st) = st(x)$	*AF_DNM*
Тотожна функція	$id(st) = st$	*AF_ID*

Зауваження 1.4. Наведені формули слід тлумачити з урахуванням частковості функцій, а саме: якщо значення однієї з функцій, що фігурує у формулі, не є визначеним, то і результат не буде визначеним. Наприклад, для формули $fs_1 \bullet fs_2(st) = fs_2(fs_1(st))$ вважаємо, що якщо $fs_1(st)$ або $fs_2(fs_1(st))$ не визначені, то і результат не є визначеним.

Для роботи з частковими функціями використовують такі позначення:

- $fs(st)\uparrow$ – значення fs на st не визначене,
- $fs(st)\downarrow$ – значення fs на st визначене,
- $fs(st)\downarrow = r$ –значення fs на st визначене і дорівнює r.

З урахуванням частковості формулу для послідовного виконання можна записати таким чином:

$$fs_1 \bullet fs_2(st) = \begin{cases} r, \text{ якщо } fs_1(st) \downarrow = r' \text{ та } fs_2(r') \downarrow = r, \\ \text{не визначено в інших випадках.} \end{cases}$$

Формула для умовного оператора матиме вигляд

$$IF(fb, fs_1, fs_2)(st) = \begin{cases} r_1, \text{ якщо } fb(st) \downarrow = true \text{ та } fs_1(st) \downarrow = r_1, \\ r_2, \text{ якщо } fb(st) \downarrow = false \text{ та } fs_2(st) \downarrow = r_2, \\ \text{не визначено в інших випадках.} \end{cases}$$

Інші формули можуть бути переписані аналогічно. Наведений вигляд формули зберігають і у випадку багатозначних (недетермінованих) функцій. Однак тут такі функції розглядати не будемо.

Зазначимо, що наведені формули можна було б записувати з уживанням сильної рівності, яка задає невизначеність лівої частини при невизначеності правої. ■

Побудована алгебра *A_SIPL* дозволяє тепер формалізувати семантику програм мови *SIPL*, задаючи їх функціональними виразами (семантичними термами) алгебри *A_SIPL*.

1.3.5. Визначення семантичних термів

Досі ми не дуже чітко розрізняли функціональний вираз алгебри *A_Prog* і функцію, що задається цим виразом. Наприклад, запис $(f \bullet g) \bullet h$ можна тлумачити як функцію, і тоді її можна застосувати до стану *st*, або як вираз, і тоді, наприклад, вивчати тотожність $(f \bullet g) \bullet h = f \bullet (g \bullet h)$. У математичній логіці таке розрізнення роблять явним, вважаючи, що $(f \bullet g) \bullet h$ є виразом (термом, формулою), а не функцією. Саму ж функцію, яка задається цим виразом, позначають, наприклад, $(f \bullet g) \bullet h_I$, де I – інтерпретація символів *f*, *g*, *h* в алгебрі *A_Prog*.

Таке розрізнення можна було б зробити і для мови *SIPL*. У такому випадку для опису (дескрипції) функцій, які задаються програмами мови *SIPL*, можуть використовуватися функціональні вирази алгебри *A_SIPL*, які називаються *термами* цієї алгебри.

Такі класи термів будуються індуктивно, аналогічно класам синтаксичних категорій. Терми програмної алгебри будемо також називати *семантичними термами*. Ми зазвичай не будемо використовувати різні позначення для термів і функцій, сподіваючись, що читач із контексту зрозуміє, про яке поняття йдеться. Однак слід пам'ятати, що таке розрізнення є важливим у теорії програмування, яка чітко виокремлює синтаксис і семантику програм.

Зауважимо, що можна використовувати різні визначення термів алгебри, у тому числі:

- індуктивне;
- рекурсивне;
- аналітичне;
- БНФ;
- графічне тощо.

Прохання до читача побудувати індуктивне визначення множини термів самостійно.

Позначимо множини термів, які задають арифметичні вирази, умови, оператори (і програми) мови *SIPL* відповідно як *TFA*, *TFB*, *TFS*.

1.3.6. Побудова семантичного терму програми

Програма мови *SIPL* може бути перетворена на семантичний терм (терм програмної алгебри), який задає її семантику (семантичну функцію), перетвореннями такого типу:

- *sem_P*: *Prog* → *TFS*;
- *sem_S*: *Stm* → *TFS*;
- *sem_A*: *Aexp* → *TFA*;
- *sem_B*: *Bexp* → *TFB*.

Ці перетворення (табл. 1.6) задаються рекурсивно (за структурою програми). Тому побудова семантичного терму залежить від вибору структури синтаксичного запису програми. Тут треба зважати на неоднозначність обраної нами граматики, що може зумовити різну семантику програм. Для досяг-

нення однозначності треба користуватися пріоритетами операцій і типом їх асоціативності.

***Таблиця* 1.6**

Правило заміни	**Номер правила**
sem_P: *Prog* → *TFS* задається правилами:	
sem_P(**begin** *S* **end**) = *sem_S*(*S*)	*NS_Prog*
sem_S: *Stm* → *TFS* задається правилами:	
$sem\_S(x := a) = AS^{x}(sem\_A(a))$ $sem\_S(S_1 ; S_2) = sem\_S(S_1) \bullet sem\_S(S_2)$ *sem_S*(**if** *b* **then** S_1 **else** S_2) = $= IF(sem\_B(b), sem\_S(S_1), sem\_S(S_2))$ *sem_S*(**while** *b* **do** *S*) = *WH*(*sem_B*(*b*), *sem_S*(*S*)) *sem_S*(**begin** *S* **end**) = (*sem_S*(*S*)) *sem_S*(*skip*) = *id*	*NS_Stm_As* *NS_Stm_Seq* *NS_Stm_If* *NS_Stm_Wh* *NS_Stm_BE* *NS_Stm_skip*
sem_A: *Aexp* → *TFA* задається правилами:	
$sem\_A(n)) = \overline{n}$ $sem\_A(x)) = x\Rightarrow$ $sem\_A(a_1 + a_2) = S^{2}(add, sem\_A(a_1), sem\_A(a_2))$ $sem\_A(a_1 - a_2) = S^{2}(sub, sem\_A(a_1), sem\_A(a_2))$ $sem\_A(a_1 * a_2) = S^{2}(mult, sem\_A(a_1), sem\_A(a_2))$ $sem\_A((a)) = sem\_A(a)$	*NS_A_Num* *NS_A_Var* *NS_A_Add* *NS_A_Sub* *NS_A_Mult* *NS_A_Par*
sem_B: *Bexp* → *TFB* задається правилами:	
$sem\_B(a_1 = a_2) = S^{2}(eq, sem\_A(a_1), sem\_A(a_2))$ $sem\_B(a_1 > a_2) = S^{2}(gr, sem\_A(a_1), sem\_A(a_2))$ $sem\_B(b_1 \vee b_2) = S^{2}(or, sem\_B(b_1), sem\_B(b_2))$ $sem\_B(\neg b) = S^{1}(neg, sem\_B(b))$ $sem\_B((b)) = sem\_B(b)$	*NS_B_eq* *NS_B_gr* *NS_B_or* *NS_B_neg* *NS_B_Par*

Наведені правила слід розглядати як загальні правила, які в логіці називають *схемами правил*. Щоб із *загального* правила (метаправила) отримати *конкретне* (об'єктне) правило, слід замість синтаксичних метасимволів, таких як a, b, S, P, n, x, підставити конкретні синтаксичні елементи (записи), наприклад замість a підставити $N - M$, замість $b - M > N$ і т. д. Далі ліва частина конкретного правила замінюється на його праву частину і т. д.

Приклад 1.2. Побудуємо семантичний терм виразу $X + Y * Z$. Побудова терму полягає в обчисленні значення $sem\_A(X + Y * Z)$. Щоб зробити перший крок такого обчислення, треба знайти правило, ліва частина якого буде збігатися із записом $sem\_A(X + Y * Z)$ при відповідній конкретизації цього правила. Такий процес називається *уніфікацією* двох записів (термів). У нашому випадку можлива уніфікація з лівою частиною правил *NS_A_Add* та *NS_A_Mult*. У першому випадку уніфікація $sem\_A(X + Y * Z)$ та $sem\_A(a_1 + a_2)$ можлива при заміні a_1 на X та a_2 на $Y * Z$, а в другому – уніфікація $sem\_A(X + Y * Z)$ та $sem\_A(a_1 * a_2)$ можлива при заміні a_1 на $X + Y$ та a_2 на Z. Наведені заміни (підстановки) називаються *уніфікаторами* і зазвичай позначаються $[a_1/X, a_2/Y*Z]$ та $[a_1/X+Y, a_2/Z]$ або $[a_1 \mapsto X, a_2 \mapsto Y*Z]$ та $[a_1 \mapsto X+Y, a_2 \mapsto Z]$. Зазначимо, що друга уніфікація порушує пріоритет операцій, тому розглядатимемо лише першу. Застосування першого уніфікатора до правила *NS_A_Add* породжує таке конкретне (об'єктне) правило:

NS_A_Add': $sem\_A(X + Y * Z) = S^2(add, sem\_A(X), sem\_A(Y * Z))$.

Застосування цього правила дозволяє перетворити запис $sem\_A(X + Y * Z)$ на запис $S^2(add, sem\_A(X), sem\_A(Y * Z))$. Останній запис містить два підзаписи: ($sem\_A(X)$ та $sem\_A(Y * Z)$), до яких можна застосувати перетворення. Підзапис $sem\_A(X)$ уніфікується з лівою частиною правила *NS_A_Var* за допомогою уніфікатора $[x/X]$, а підзапис $sem\_A(Y * Z)$ – із *NS_A_Mult* за допомогою уніфікатора $[a_1/Y, a_2/Z]$. Застосування цих уніфікаторів породжує два нові конкретні правила:

NS_A_Var': $sem\_A(X) = X \Rightarrow$ та

NS_A_Mult': $sem\_A(Y * Z) = S^2(mult, sem\_A(Y), sem\_A(Z))$.

Застосовуючи їх до виразу $S^2(add, sem\_A(X), sem\_A(Y * Z))$ отримаємо $S^2(add, X\Rightarrow, S^2(mult, sem\_A(Y), sem\_A(Z)))$. Залишилося конкретизувати правило *NS_A_Var*, щоб отримати остаточний результат:

$$sem\_A(X + Y * Z) = S^2(add, X\Rightarrow, S^2(mult, Y\Rightarrow, Z\Rightarrow)).\blacksquare$$

Отже, процес побудови семантичного терму програми полягає в послідовному перетворенні запису, для якого будується семантичний терм. Ці перетворення вимагають таких дій:

- вибір загального правила, яке можна застосувати до підзапису поточного виразу з урахуванням пріоритету операцій;
- знаходження уніфікатора лівої частини правила з обраним підзаписом;
- отримання конкретного правила застосуванням уніфікатора до обох частин загального правила;
- заміна в поточному записі лівої частини конкретного правила на його праву частину.

Формальніше процес перетворень такого типу сформульований у теорії переписуючих правил.

Приклад 1.3. Побудувати семантичний терм програми *GCD*. Побудову будемо робити згідно з вищенаведеними правилами. Деталі не вказуємо.

$sem\_P(GCD) =$
$= sem\_P($**begin**
while $\neg\, M = N$ **do**
if $M > N$ **then** $M := M - N$ **else** $N := N - M$
end$) =$
$= sem\_S($**while** $\neg\, M = N$ **do if** $M > N$ **then** $M := M - N$ **else** $N := N - M) =$
$= WH(sem\_B(\neg\, M{=}N), sem\_S($**if** $M{>}N$ **then** $M := M - N$ **else** $N := N - M)) =$
$= WH(S^1(neg, sem\_A(M = N),$
$IF(sem\_B(M > N), sem\_S(M := M - N), sem\_S(N := N - M)))=$
$= WH(S^1(neg, S^2(eq, sem\_A(M), sem\_A(N))),$
$IF(sem\_B(M > N), sem\_S(M := M - N), sem\_S(N := N - M)))=$
$= WH(S^1(neg, S^2(eq, M\Rightarrow, N\Rightarrow)), IF(S^2(gr, M\Rightarrow, N\Rightarrow),$
$AS^M(sem\_A(M{-}N)), AS^N(sem\_A(N{-}M)))) =$
$= WH(S^1(neg, S^2(eq, M\Rightarrow, N\Rightarrow)), IF(S^2(gr, M\Rightarrow, N\Rightarrow),$
$AS^M(S^2(sub, sem\_A(M), sem\_A(N))),$
$AS^N(S^2(sub, sem\_A(N), sem\_A(M))))) =$
$= WH(S^1(neg, S^2(eq, M\Rightarrow, N\Rightarrow)),$
$IF(S^2(gr, M\Rightarrow, N\Rightarrow),$

$$AS^{M}(S^{2}(sub, M\Rightarrow, N\Rightarrow)),$$
$$AS^{N}(S^{2}(sub, N\Rightarrow, M\Rightarrow)))).\blacksquare$$

Повернемося тепер до доведення того факту, що семантика довільної програми мови *SIPL* задається термом алгебри *A_SIPL*. Дійсно, аналізуючи табл. 1.6, бачимо, що там фігурують лише композиції та базові функції алгебри *A_SIPL*. Більш строге доведення можна отримати індукцією за структурою програми *SIPL*. Отже, справедливе таке твердження.

Теорема 1.1. Для довільної програми мови *SIPL* її семантична функція задається термом алгебри *A_SIPL*.

Зауваження 1.5. Відображення побудови семантичного терму можна розглядати також у іншому аспекті – алгебраїчному. У такому випадку програма розглядається як терм синтаксичної алгебри, який відображається в семантичну алгебру. Відображення буде гомоморфізмом синтаксичної алгебри в семантичну. Доведення випливає з аналізу правил, заданих у табл. 1.6. Прохання до читача довести цей факт самостійно.

1.3.7. Обчислення значень семантичних термів

Семантичні терми програми є точно (формально) заданими об'єктами, які формалізують семантику програм у термінах відповідних семантичних алгебр. Такі алгебри й терми є головними об'єктами дослідження в нашому підручнику. Далі ми будемо вивчати різні властивості таких алгебр і відповідних термів. Зараз розглянемо найпростішу властивість термів, зважаючи на те, що вони задають деякі функції (тобто повертаємось знову до синкретичного тлумачення терму як функції). Ця властивість називається *аплікацією* і полягає в застосуванні функції, що задається термом, до певних вхідних даних. Аплікація є аналогом (абстракцією) тестування програм.

Приклад 1.4. Обчислимо значення семантичного терму програми *GCD* на вхідному даному $[M \mapsto 8, N \mapsto 16]$.

Процес обчислення значення задається формулами табл. 1.5. Ці формули, як і формули табл. 1.6, є загальними правилами обчислень. Щоб їх застосовувати, треба виконувати конкретизацію

загальних правил подібно до того, як описано в прикл. 1.3. Отже, завдання полягає в отриманні значення такої аплікації:

$$WH(S^{1}(neg, S^{2}(eq, M\Rightarrow, N\Rightarrow)), IF(S^{2}(gr, M\Rightarrow, N\Rightarrow),$$
$$AS^{M}(S^{2}(sub, M\Rightarrow, N\Rightarrow)),$$
$$AS^{N}(S^{2}(sub, N\Rightarrow, M\Rightarrow)))) ([M \mapsto 8, N \mapsto 16]).$$

Правила для обчислення циклу *AF_WH* свідчать про необхідність поступового обчислення станів st_0, st_1, … та перевірки відповідних умов. Отримуємо уніфікатор

$$[fb/S^{1}(neg, S^{2}(eq, M\Rightarrow, N\Rightarrow)),$$
$$fs/ IF(S^{2}(gr, M\Rightarrow, N\Rightarrow)),$$
$$AS^{M}(S^{2}(sub, M\Rightarrow, N\Rightarrow), AS^{N}(S^{2}(sub, N\Rightarrow, M\Rightarrow))),$$
$$st/[M \mapsto 8, N \mapsto 16], st_0/[M \mapsto 8, N \mapsto 16]].$$

Переходимо до обчислення $fb(st_0)$, яке конкретизується аплікацією

$$S^{1}(neg, S^{2}(eq, M\Rightarrow, N\Rightarrow))([M\mapsto 8, N\mapsto 16]).$$

Ця аплікація обчислюється згідно із загальним правилом *AF_S* для обчислення суперпозиції. Після відповідної уніфікації та конкретизації спочатку отримуємо, що

$$S^{1}(neg, S^{2}(eq, M\Rightarrow, N\Rightarrow))([M\mapsto 8, N\mapsto 16]) =$$
$$= neg(S^{2}(eq, M\Rightarrow, N\Rightarrow)([M\mapsto 8, N\mapsto 16])).$$

Застосовуючи правило обчислення суперпозиції ще раз і правила обчислення функції розіменування, отримуємо, що

$$S^{2}(eq, M\Rightarrow, N\Rightarrow)([M\mapsto 8, N\mapsto 16]) =$$
$$= eq(M\Rightarrow([M\mapsto 8, N\mapsto 16]), N\Rightarrow([M\mapsto 8, N\mapsto 16])) = eq(8,16) = false.$$

Остаточно маємо:

$$S^{1}(neg, S^{2}(eq, M\Rightarrow, N\Rightarrow))([M\mapsto 8, N\mapsto 16]) = neg(false) = true.$$

Правила обчислення для циклу свідчать про необхідність обчислення за правилом $st_1 = fs(st_0)$, що конкретизується таким чином:

$$IF(S^{2}(gr, M\Rightarrow, N\Rightarrow), AS^{M}(S^{2}(sub, M\Rightarrow, N\Rightarrow)),$$

$$AS^N(\ S^{\,2}(sub, N\Rightarrow, M\Rightarrow)))([M\mapsto 8, N\mapsto 16]).$$

Обчислення цієї аплікації полягає у застосуванні правила *AF_IF* для обчислення умовного оператора. Спочатку обчислюємо умову

$$S^{\,2}(gr, M\Rightarrow, N\Rightarrow))([M\mapsto 8, N\mapsto 16]) = gr(M\Rightarrow([M\mapsto 8, N\mapsto 16]),$$

$$N\Rightarrow([M\mapsto 8, N\mapsto 16])) = gr(8, 16) = false.$$

За хибної умови за правилом *AF_IF* обчислюємо

$AS^N(\ S^{\,2}(sub, N\Rightarrow, M\Rightarrow))([M\mapsto 8, N\mapsto 16])$.

Отримуємо за правилом *AF_AS*

$$AS^N(\ S^{\,2}(sub, N\Rightarrow, M\Rightarrow))([M\mapsto 8, N\mapsto 16]) =$$

$$= [M\mapsto 8, N\mapsto 16]\nabla[N\mapsto S^{\,2}(sub, N\Rightarrow, M\Rightarrow)([M\mapsto 8, N\mapsto 16])] =$$

$$= [M\mapsto 8, N\mapsto 16]\nabla[N\mapsto sub(N\Rightarrow([M\mapsto 8, N\mapsto 16]),$$

$$M\Rightarrow([M\mapsto 8, N\mapsto 16]))] = [M\mapsto 8, N\mapsto 16]\nabla[N\mapsto sub(16,8)] =$$

$$= [M\mapsto 8, N\mapsto 16]\nabla[N\mapsto 8] = [M\mapsto 8, N\mapsto 8].$$

Отже, st_1 конкретизується як $[M\mapsto 8, N\mapsto 8]$. Тепер за правилом *AF_WH* обчислюємо умову циклу на отриманому стані:

$S^{\,1}(neg, S^{\,2}(eq, M\Rightarrow, N\Rightarrow))([M\mapsto 8, N\mapsto 8]) = false$.

Таким чином, остаточним станом є $[M\mapsto 8, N\mapsto 8]$.■

1.3.8. Загальна схема формалізації мови *SIPL*

Підсумуємо схему формалізації, яка була застосована для мови *SIPL*. Зазначимо, що в першому наближенні мова *L* задається як трійка вигляду (*Synt*, *Sem*, *interpretation*), де *Synt* – опис синтаксичного аспекту (текстів програм), *Sem* – семантичного аспекту (смислу програм); *interpretation*: *Synt*→ *Sem* – інтерпретація програм, яка кожній програмі зіставляє її смисл (значення). Інтерпретацію також називають денотацією.

Спочатку ми мали початковий опис мови ($Synt_0$, Sem_0, $interpretation_0$), який складався з формального подання синтаксису

$Synt_0$ у вигляді БНФ і неформального опису семантики Sem_0. Інтерпретація $interpretation_0$: $Synt_0 \rightarrow Sem_0$ також була неформальною.

Для формалізації композиційної семантики було вибрано формалізм функціональних (програмних) алгебр. Цей формалізм також можна трактувати як певну мову, синтаксис якої задається термами алгебри $Synt_1$, семантика Sem_1 – функціями з носія алгебри, а інтерпретація $interpretation_1$: $Synt_1 \rightarrow Sem_1$ є просто інтерпретацією термів у функціональній алгебрі.

Визначення формальної семантики дозволяє дати формальне визначення мови *SIPL* таким чином:

- синтаксис $Synt_0$ задається БНФ;
- семантика Sem_1 задається алгеброю функцій;
- інтерпретація $interpretation_{SIPL}$: $Synt_0 \rightarrow Sem_1$ є добутком відображень sem_P та $interpretation_1$, тобто

$$interpretation_{SIPL} = sem\_P \bullet interpretation_1.$$

(Тут множення • тлумачиться як композиція довільних відображень.)

Таким чином, мова *SIPL* уточнюється трійкою

$$(Synt_0, Sem_1, sem\_P \bullet interpretation_1).$$

Кожен компонент трійки визначений формально. Указана схема формалізації наведена на рис. 1.5.

У схемі похідну стрілку $Sem_0 \rightarrow Sem_1$ можна тлумачити як уточнення (експлікацію) семантики. Така схема буде використана далі для введення формальної семантики програм.

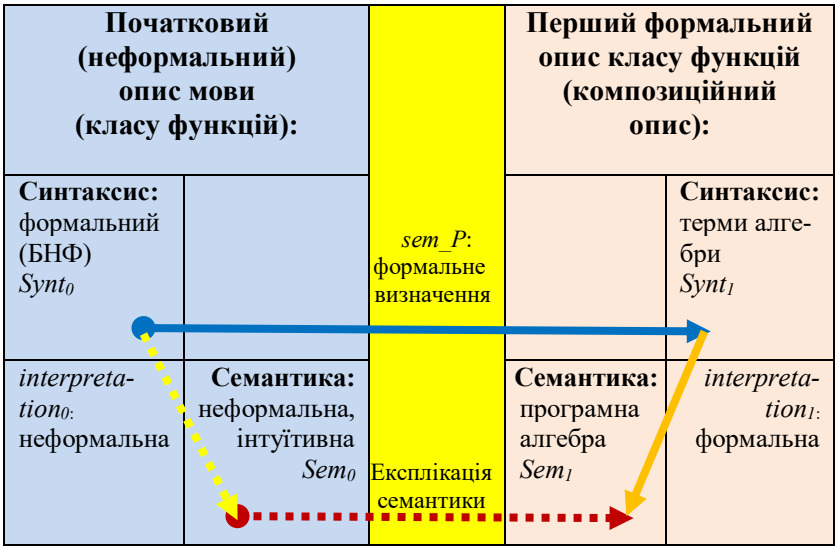

Рис. 1.5. Схема визначення формальної семантики мови *SIPL*

1.4. Властивості програмної алгебри

Побудована програмна алгебра дозволяє сформулювати властивості програм, досліджуючи властивості функцій цієї алгебри, заданих її термами (функціональними виразами). Доведемо кілька таких властивостей.

Зауваження 1.6. У програмній алгебрі рівність розглядається як рівність функцій.

Лема 1.1. Доведемо властивість асоціативності послідовного виконання, тобто справедливість такої тотожності ($f, g, h \in FS$):

$$(f \bullet g) \bullet h = f \bullet (g \bullet h).$$

Доведення. Оскільки в лемі формулюється рівність функцій, то слід довести, що на одному й тому самому даному обидві функції мають бути 1) одночасно невизначеними або 2) одночасно визначеними, і в цьому випадку давати однакові резуль-

тати (сильна рівність функціональних виразів). Використовуючи позначення $fs(st)\downarrow$ (функція fs визначена на st) та $fs(st)\uparrow$ (функція fs не визначена на st), наведене формулювання рівності можна задати таким чином:

$fs_1 = fs_2$ тоді й тільки тоді, коли для довільного стану $st \in State$

1) $((fs_1(st)\uparrow \,\&\, fs_2(st)\uparrow)$ або

2) $(fs_1(st)\downarrow \,\&\, fs_2(st)\downarrow \,\&\, fs_1(st) = fs_2(st))$.

Стосовно леми це означає:

$(f \bullet g) \bullet h = f \bullet (g \bullet h))$ тоді й тільки тоді, коли для довільного стану $st \in State$

1) $((f \bullet g) \bullet h)(st)\uparrow \,\&\, (f \bullet (g \bullet h))(st)\uparrow$ або

2) $((f \bullet g) \bullet h)(st)\downarrow \,\&\,$

$$\&\, (((f \bullet g) \bullet h)(st) = (f \bullet (g \bullet h))(st)).$$

Перейдемо до доведення. Беремо довільний стан $st \in State$. Спочатку доведемо, що якщо $((f \bullet g) \bullet h)(st)\uparrow$, то $(f \bullet (g \bullet h))(st)\uparrow$. Дійсно, для $((f \bullet g) \bullet h)(st)$ маємо формулу $h(g(f(st)))$. Тому значення $((f \bullet g) \bullet h)(st)$ буде невизначеним, якщо або $f(st)$, або $g(f(st))$, або $h(g(f(st)))$ не визначене. Однак у кожному з цих трьох випадків буде невизначеним і значення $(f \bullet (g \bullet h))(st)$. Має місце також зворотне. Отже, якщо одне зі значень, $((f \bullet g) \bullet h)(st)$ або $(f \bullet (g \bullet h))(st)$, не визначене, то і друге значення також не визначене, тому формула $((fs_1(st)\uparrow \,\&\, fs_2(st)\uparrow)$ буде істинною. Звідси випливає, що якщо значення одного з функціональних виразів є визначеним, то є визначеним і значення іншого функціонального виразу.

Отже, якщо одне зі значень буде визначеним, то обидва значення будуть визначеними і рівними, оскільки

$$((f \bullet g) \bullet h)(st) = h(g(f(st))) \text{ та } (f \bullet (g \bullet h))(st) = h(g(f(st))).$$

Лему доведено.■

Надалі не будемо детально аналізувати випадки невизначеності функцій, сподіваючись, що читач сам зможе це зробити.

Зауваження 1.7. Рівність функцій можна доводити як рівність їхніх графіків. Однак це вимагає переозначення композицій із функціональних у теоретико-множинні терміни. Тут цього робити не будемо.

Далі доведемо твердження, яке часто буде використовуватися для доведення коректності програм із циклами.

Лема 1.2 (про властивості циклу). Для довільних функцій $fs \in FS$ та $fb \in FB$ і довільного стану $st \in State$ мають місце такі властивості:

- $WH(fb,fs) = IF(fb, fs \bullet WH(fb,fs), id)$.
- $NumItWH((fb, fs), st) = 0$ тоді й тільки тоді, коли $fb(st)\downarrow = false$ (це також означає, що $WH(fb, fs)(st) = st$).
- Якщо $NumItWH((fb, fs), st) > 0$, то
  - $fb(st)\downarrow = true$,
  - $WH(fb, fs)(st) = WH(fb, fs)(fs(st))$ та
  - $NumItWH((fb, fs), fs(st)) = NumItWH((fb, fs), st) - 1$.

Доведення. Спочатку доведемо першу властивість, що задає певну тотожність. Візьмемо довільний стан $st \in State$. Припустимо, що $WH(fb, fs)(st)$ визначено. Можливі два варіанти:

1) $fb(st) = false$;

2) $fb(st) = true$.

У першому випадку тіло циклу не виконується ($NumItWH((fb, fs), st) = 0$), тому $WH(fb, fs)(st) = st$. При обчисленні $IF(fb, fs \bullet WH(fb, fs), id)(st)$ потрібно обчислити $id(st)$, що дає також значення st. Тому лема для цього випадку справедлива.

У другому випадку визначеність $WH(fb, fs)(st)$ означає, що є послідовність станів і значень предиката (тому $NumItWH((fb, fs), st))>0$), яка задовольняє такі умови:

$st_0 = st$, $st_1 = fs(st_0)$, $st_2 = fs(st_1)$,…, $st_n = fs(st_{n-1})$, причому

$fb(st_0) = true$, $fb(st_1) = true$,…, $fb(st_{n-1}) = true$, $fb(st_n) = false$.

Розглянемо, яке значення має $WH(fb, fs)(st_1)$. Зрозуміло, що послідовність обчислень повторює вищенаведену послідовність, але початковим станом є st_1, тому $WH(fb, fs)(st_1) = st_n$. Звідси також випливає, що $NumItWH((fb, fs), st_1) = n-1$.

Розглянемо тепер обчислення функціонального виразу (праву частину тотожності)

$$IF(fb, fs \bullet WH(fb, fs), id)(st).$$

Оскільки $fb(st) = true$, то

$$IF(fb, fs \bullet WH(fb, fs), id)(st) = (fs \bullet WH(fb, fs))\ (st) =$$
$$= WH(fb, fs)(fs(st)) = WH(fb, fs)\ (st_1) = st_n.$$

Отже, і в цьому випадку лема справедлива.

Аналогічно доводиться, що якщо визначена ліва частина тотожності, то буде визначена і права частина з тим самим результатом. Тотожність доведено. З доведення випливають також інші дві властивості, сформульовані в лемі.■

Зауваження 1.8. Тотожність

$$WH(fb, fs) = F(fb, fs \bullet WH(fb, fs), id)$$

стверджує, що $WH(fb, fs)$ є розв'язком відносно X (можливо, одним із розв'язків) функціонального рівняння

$$X = IF(fb, fs \bullet X, id).$$

Дослідження рівнянь такого типу буде виконане в розд. 5, присвяченому рекурсії.■

Леми 1.1 та 1.2 характеризують властивості побудованої алгебри функцій *A_Prog*. Таких властивостей досить багато. Їхня ідентифікація (формулювання) – важлива проблема. Зазначені тотожності дозволяють здійснювати еквівалентні перетворення програм з метою доведення їх властивостей або проведення оптимізації, трансляції, інтерпретації тощо.

Перейдемо тепер до визначення підалгебр алгебри *A_Prog*. Серед можливих підалгебр виділимо підалгебру, яка індукована аналізом відношення розширення (збагачення) станів новими змінними з їх значеннями. Наприклад, знаючи, що $sem\_P(GCD)([M \mapsto 8, N \mapsto 16]) = [M \mapsto 8, N \mapsto 8]$, запитаємо про значення $sem\_P(GCD)([M \mapsto 8, N \mapsto 16, L \mapsto 9])$, тобто про значення функції $sem\_P(GCD)$ на новому стані, який має нову змінну L. Більшість програмістів погодяться із тим, що результуючий стан буде просто розширенням попереднього результуючого стану цією новою змінною, тобто

$$sem\_P(GCD)([M \mapsto 8, N \mapsto 16, L \mapsto 9]) = [M \mapsto 8, N \mapsto 8, L \mapsto 9]).$$

Інакше кажучи, функція $sem\_P(GCD)$ є монотонною щодо відношення розширення станів $\subseteq$. Монотонність функцій визначається таким чином.

Функція $fs \in FS$ називається *монотонною*, якщо із $fs(st)\downarrow = str$ та $st \subseteq st'$ випливає, що $fs(st)\downarrow = str'$ та $str \subseteq str'$. Підклас монотонних функцій позначимо *MFS*.

Така властивість функцій використовується програмістами при збільшенні пам'яті комп'ютера, оскільки програми на збільшеній пам'яті працюватимуть так само (у будь-якому разі ми на це сподіваємось), як і на меншій пам'яті.

А чи є подібна властивість для функцій, породжених арифметичними виразами й умовами мови *SIPL*? Виявляється, при розширенні станів значення арифметичних функцій і предикатів не змінюється. Така властивість називається еквітонністю ("екві" означає "рівний"). Наведемо її точне означення.

Функція $f \in FA$ називається *еквітонною*, якщо із $f(st)\downarrow = r$ та $st \subseteq st'$ випливає, що $f(st')\downarrow = r$. Підклас еквітонних функцій позначимо *EFA*, підклас еквітонних предикатів – *EFB*.

Як бачимо, поняття монотонної та еквітонної функцій дуже важливі, тому розглянемо поведінку таких функцій стосовно композицій мови *SIPL*.

Виявляється (і зараз це буде доведено), що введені класи утворюють еквітонно-монотонну підалгебру

$$A\_EM\_Prog = <FNA, FNB, FNAB, EFA, EFB, MFS;$$
$$S^n, AS^x, \bullet, IF, WH, x\Rightarrow, id>$$

алгебри функцій *A_Prog*.

Потрібно довести, що класи *EFA*, *EFB*, *MFS* замкнені відносно заданих на них композицій.

Спочатку доведемо, що клас *EFA* замкнений відносно композицій алгебри *A_Prog*, заданих на *EFA*, тобто що він замкнений відносно композиції суперпозиції S^n та функції розіменування. Нехай f – n-арна функція з *FNA*, $g_1,\ldots, g_n \in EFA$. Треба довести, що $S^n(f, g_1,\ldots, g_n) \in EFA$. Беремо довільні st та st' такі, що $st \subseteq st'$. Вважаємо також, що $(S^n(f, g_1,\ldots, g_n))(st)\downarrow = r$. Доведемо, що

$$(S^n(f, g_1,\ldots, g_n))(st')\downarrow = r.$$

Дійсно, ураховуючи, що $g_1(st') = g_1(st), \ldots, g_n(st') = g_n(st)$, маємо

$$(S^n(f, g_1,\ldots, g_n))(st') = f(g_1(st'),\ldots, g_n(st')) = f(g_1(st),\ldots, g_n(st)) = r.$$

Неважко також переконатися, що функція розіменування є еквітонною функцією з класу *EFA*.

Аналогічно доводиться, що і клас еквітонних предикатів *EFB* замкнений відносно відповідних композицій суперпозиції.

Залишилося довести, що клас монотонних функцій *MFS* замкнений відносно композицій присвоювання, послідовного виконання, умовного оператора, циклу і тотожної функції. Розглянемо всі ці випадки (для довільних st та st' таких, що $st \subseteq st'$).

1. *Присвоювання*. Нехай $fa \in EFA$ та $AS^x(fa)(st)\downarrow = str$. За визначенням $AS^x(fa)(st) = st\ \nabla[x \mapsto fa(st)]$. Обчислимо $AS^x(fa)(st')$, ураховуючи, що для $fa \in EFA\ fa(st') = fa(st)$. Маємо

$$AS^x(fa)(st') = st'\ \nabla[x \mapsto fa(st')] = st'\ \nabla[x \mapsto fa(st)].$$

Порівнюючи $st\ \nabla[x \mapsto fa(st)]$ та $st'\ \nabla[x \mapsto fa(st)]$, можна стверджувати, що $st\ \nabla[x \mapsto fa(st)] \subseteq st'\ \nabla[x \mapsto fa(st)]$, оскільки значення змінної x у цих станах однакові, а $st \subseteq st'$.

2. *Послідовне виконання*. Нехай fs_1, $fs_2 \in MFS$ та $fs_1 \bullet fs_2(st)\downarrow = str$. За визначенням $fs_1 \bullet fs_2(st) = fs_2(fs_1(st))$. Обчислимо $fs_1 \bullet fs_2(st')$, ураховуючи, що fs_1, $fs_2 \in MFS$. Маємо $fs_1 \bullet fs_2(st') = fs_2(fs_1(st'))$. Оскільки $fs_1(st) \subseteq fs_1(st')$, то і $fs_2(fs_1(st)) \subseteq fs_2(fs_1(st'))$, що і треба було довести.

3. *Умовний оператор*. Нехай $fb \in EFB$, fs_1, $fs_2 \in MFS$ та $IF(fb, fs_1, fs_2)(st)\downarrow = str$. За визначенням $IF(fb, fs_1, fs_2)(st)$ дорівнює $fs_1(st)$, якщо $fb(st) = true$, або $fs_2(st)$, якщо $fb(st) = false$. Обчислимо $IF(fb, fs_1, fs_2)(st')$, ураховуючи, що $fb \in EFB$ (тобто $fb(st') = fb(st)$), а fs_1, $fs_2 \in MFS$. Тому, якщо $fb(st') = true$, то $IF(fb, fs_1, fs_2)(st') = fs_1(st')$, і оскільки $fs_1(st) \subseteq fs_1(st')$, то

$$IF(fb, fs_1, fs_2)(st) \subseteq IF(fb, fs_1, fs_2)(st').$$

Аналогічно для випадку $fb(st') = false$ отримуємо

$$IF(fb, fs_1, fs_2)(st) \subseteq IF(fb, fs_1, fs_2)(st').$$

4. *Цикл*. Доведення випливає з того, що послідовності станів при обчисленні $WH(fb, fs)(st')$ відповідають послідовності станів при обчисленні $WH(fb, fs)(st)$, причому весь час має місце співвідношення $st_i \subseteq st_i'$ $(i = 0, \ldots, n)$.

5. *Тотожна функція* є монотонною з класу *MFS*.

Отже, доведено таку лему.

Лема 1.3. Еквітонно-монотонна алгебра *A_EM_Prog* є підалгеброю алгебри функцій *A_Prog*.

Доведена лема дозволяє дати точніший опис функцій, породжуваних мовою *SIPL*. Для цього слід показати, що алгебра *A_SIPL* є підалгеброю алгебри *A_EM_Prog*. Слід переконатися, що функції-константи $\bar{n}$ є еквітонними функціями (що очевидно). Отже, справедливо таке:

- якщо $a \in Aexp$, то $sem\_A(a) \in EFA$;
- якщо $b \in Bexp$, то $sem\_B(b) \in EFB$;
- якщо $S \in Stm$, то $sem\_S(S) \in MFS$;
- якщо $P \in Prog$, то $sem\_P(P) \in MFS$.

Отримані твердження можна сформулювати у вигляді теореми.

Теорема 1.2. Вирази й умови мови *SIPL* породжують еквітонні функції, а програми й оператори – монотонні функції.

З теореми випливає, що якщо програма мови *SIPL* завершується з якимись результатами на певному стані, то вона завершиться й на розширеному стані, причому однакові змінні в обох результуючих станах будуть мати однакові результати.

1.5. Часткова та повна коректність програм

Доведемо тепер коректність програми *GCD*. Інтуїтивно програма є коректною, якщо для вхідних даних, які задовольняють початкові вимоги, результати будуть задовольняти заключні вимоги. Виокремлюють часткову і тотальну коректність. Для часткової коректності завершуваності програми не вимагають, для тотальної коректності програма має завершитись.

Зазначимо, що оскільки в нас поки що є лише одна – композиційна – семантика, то коректність програми доводиться відносно неї. Тому програмістські терміни слід тлумачити саме в термінах композиційної семантики. Отже, завершуваність програми означає, що функція, яка задає її семантику, визначена на відповідному стані.

Позначимо функцію взяття найбільшого спільного дільника двох чисел як *gcd*.

Теорема 1.3 (про часткову коректність програми *GCD*). Нехай стан *st* такий, що $M\Rightarrow(st) = m$, $N\Rightarrow(st) = n$ ($m, n > 0$). Тоді якщо $sem\_P(GCD)(st)\downarrow = str$, то $M\Rightarrow(str) = N\Rightarrow(str) = gcd(m, n)$.

Зауваження 1.9. Згідно з теоремою 1.2 можна взяти стан, який має лише дві змінні – *M* та *N*. У цьому випадку теорему можна сформулювати простіше:

якщо $sem\_P(GCD)([M\mapsto m, N\mapsto n])\downarrow = [M\mapsto r1, N\mapsto r2]$, то

$$r1 = r2 = gcd(m, n).$$

Доведення. Побудуємо семантичний терм функції *sem_P*(*GCD*). Маємо

$sem\_P(GCD) =$

$= sem\_P($**begin while** $\neg\, M = N$ **do**

if $M > N$ **then** $M := M - N$ **else** $N := N - M$ **end**$) =$

$= WH(S^1(neg, S^2(eq, M\Rightarrow, N\Rightarrow)),$

$IF(S^2(gr, M\Rightarrow, N\Rightarrow),$

$AS^M(\, S^2(sub, M\Rightarrow, N\Rightarrow)),$

$AS^N(\, S^2(sub, N\Rightarrow, M\Rightarrow))$

$)$

$)$

Таким чином, семантика програми задається семантичним термом з композицією циклу як головною операцією. Для спрощення позначимо

$p\_g = S^1(neg, S^2(eq, M\Rightarrow, N\Rightarrow)),$

$f\_g = IF(S^2(gr, M\Rightarrow, N\Rightarrow),$

$AS^M(\, S^2(sub, M\Rightarrow, N\Rightarrow)),$

$AS^N(\, S^2(sub, N\Rightarrow, M\Rightarrow))).$

У цьому випадку $sem\_P(GCD) = WH(p\_g, f\_g)$.

Оскільки значення $WH(p\_g, f\_g)(st)$ визначене, то $NumItWH((p\_g, f\_g), st) \geq 0$. Нехай $NumItWH((p\_g, f\_g), st) = k$. Теорему будемо доводити індукцією за *k* (тобто за кількістю ітерацій циклу). *Індуктивна гіпотеза* фактично повторює формулювання теореми:

якщо $m > 0$, $n > 0$, $M\Rightarrow(st) = m$, $N\Rightarrow(st) = n$,

$WH(p\_g, f\_g)(st)\downarrow = str$, то $M\Rightarrow(str) = N\Rightarrow(str) = gcd(m, n)$.

База індукції. Нехай $k = 0$. Згідно з лемою 1.2

$p\_g(st)\downarrow = false$ та $WH(p\_g, f\_g)(st) = st$.

Що означає умова $p\_g(st)\downarrow = false$? Щоб це з'ясувати, виконаємо такі обчислення:

$$p\_g(st) = S^1(neg, S^2(eq, M\Rightarrow, N\Rightarrow))(st) =$$
$$= neg(eq(M\Rightarrow(st), N\Rightarrow(st))) = neg(eq(m, n)) = false.$$

Це означає, що $m = n$, тому $M\Rightarrow(st) = N\Rightarrow(st) = gcd(m, n)$.

Теорема виконується для розглянутого випадку.

Крок індукції. Нехай теорема справедлива для всіх станів st таких, що $NumItWH((p\_g, f\_g), st) = k$ (обчислення $WH(p\_g, f\_g)(st)$ вимагає k застосувань функції f_g на відповідних станах). Доведемо, що тоді теорема буде справедлива для станів st, на яких обчислення $WH(p\_g, f\_g)(st)$ вимагає k+1 кроків. У такому випадку (згідно з лемою 1.2) це означає, що $p\_g(st)\downarrow = true$ (тобто $m \neq n$),

$$WH(p\_g, f\_g)(st) = WH(p\_g, f\_g)(f\_g\ (st)) \text{ та}$$
$$NumItWH((p\_g, f\_g), f\_g\ (st)) = k.$$

Виконаємо обчислення $f\_g\ (st) = IF(S^2(gr, M\Rightarrow, N\Rightarrow), AS^M(S^2(sub, M\Rightarrow, N\Rightarrow)), AS^N(S^2(sub, N\Rightarrow, M\Rightarrow)))(st)$.

Маємо два випадки:

1) $S^2(gr, M\Rightarrow, N\Rightarrow)(st)\downarrow = true$ (це означає, що $m > n$). Тоді

$$f\_g\ (st) = AS^M(S^2(sub, M\Rightarrow, N\Rightarrow))(st) =$$
$$= st\nabla[M\mapsto S^2(sub, M\Rightarrow, N\Rightarrow)(st)] =$$
$$= st\nabla[M\mapsto sub(M\Rightarrow(st), N\Rightarrow(st)] =$$
$$= st\nabla[M\mapsto sub(m, n)] = st\nabla[M\mapsto m - n]$$

(для стану st із двома змінними можна записати, що результатом є $[M\mapsto m, N\mapsto n]\ \nabla[M\mapsto m - n] = [M\mapsto m - n, N\mapsto n]$);

2) $S^2(gr, M\Rightarrow, N\Rightarrow)(st)\downarrow = false$ (це означає, що $m \leq n$, але враховуючи, що $m \neq n$, маємо $m < n$). Тоді

$$f\_g\ (st) = AS^N(S^2(sub, N\Rightarrow, M\Rightarrow))(st) =$$
$$= st\nabla[N\mapsto S^2(sub, N\Rightarrow, M\Rightarrow)(st)] =$$
$$= st\nabla[N\mapsto sub(N\Rightarrow(st), M\Rightarrow(st)] = st\nabla[N\mapsto sub(n, m)] =$$
$$= st\nabla[N\mapsto n - m]$$

(для стану st із двома змінними можна записати, що результатом є $[M \mapsto m, N \mapsto n] \nabla [N \mapsto n - m] = [M \mapsto m, N \mapsto n - m]$).

З теорії чисел випливає, що за вказаних умов $gcd(m, n) = gcd(m, m - n)$, якщо $m > n$, та $gcd(m, n) = gcd(n - m, n)$, якщо $m < n$.

Дійсно, не обмежуючи загальності, розглянемо випадок $m > n$. Тут M має значення $m - n$, а $N - n$. Покажемо, що $gcd(m, n) = gcd(m, m - n)$. Для цього слід показати, що множина спільних дільників m та n і множина спільних дільників $m - n$ та n збігаються. Нехай q – спільний дільник m та n, тобто $m = q * k$, $n = q * r$ для деяких натуральних чисел k, r. Тоді $m - n = q * k - q * r = q * (k - r)$, тобто q – спільний дільник $m - n$ та n. Аналогічно доводиться і зворотне твердження. Оскільки множини спільних дільників двох пар чисел збігаються, то і найбільший спільний дільник у них теж збігається.

Отже, у стані $st_1 = f\_g(st)$ значення змінних M та N додатні (більше нуля) і дорівнюють відповідно m та $m - n$ або $n - m$ та n. Таким чином, для обох випадків маємо $M \Rightarrow (st_1) > 0$, $N \Rightarrow (st_1) > 0$ та $gcd(M \Rightarrow (st_1), N \Rightarrow (st_1)) = gcd(M \Rightarrow (st), N \Rightarrow (st)) = gcd(m, n)$.

Тепер можна скористуватися індуктивною гіпотезою для стану st_1, оскільки всі її засновки виконуються. Тому отримуємо, що $M \Rightarrow (str) = N \Rightarrow (str) = gcd(M \Rightarrow (st_1), N \Rightarrow (st_1))$.

З урахуванням $gcd(M \Rightarrow (st_1), N \Rightarrow (st_1)) = gcd(m, n)$ маємо, що $M \Rightarrow (str) = N \Rightarrow (str) = gcd(m, n)$.

Теорему доведено.■

Зазначимо дві обставини, пов'язані з теоремою коректності.

По-перше, теорему доведено за умови додатності значень змінних M та N. А як буде працювати програма, коли ця умова не виконуватиметься? Якщо значення змінних M та N не будуть додатними та рівними між собою (наприклад, значення змінних M та N дорівнюють –5), то програма зупиниться, не змінюючи стан. Однак наявне від'ємне число (або 0), що є значенням змінних, не може вважатися їх найбільшим спільним дільником. Коли ж значення змінних різні й хоча б одне з них від'ємне або дорівнює нулю, програма зациклюється. Отже, на правильний ре-

зультат можна сподіватися лише для додатних значень змінних M та N.

По-друге, теорему коректності доведено за умови завершуваності програми. Така коректність називається *частковою коректністю*. *Повна* (*тотальна*) *коректність* програми P для певного класу ST вхідних даних означає, що програма є частково коректною та завершується на всіх вхідних даних із цього класу. Якщо позначити формулу завершуваності програми P на стані $st \in ST$ як $termination(P)(st)$, а коректність – як $correctness(P)(st)$, то умова часткової коректності задається формулою

$$\forall st \in ST\ (termination(P)(st) \Rightarrow correctness(P)(st)),$$

а тотальної коректності –

$$\forall st \in ST\ (termination(P)(st) \wedge correctness(P)(st)).$$

Теорема 1.4 (про повну коректність програми *GCD*). Нехай стан st такий, що $M\Rightarrow(st) = m$, $N\Rightarrow(st) = n$ та $m, n > 0$. Тоді для деякого стану str маємо:

$$sem\_P(GCD)(st)\downarrow = str \text{ та } (M\Rightarrow(str)) = (N\Rightarrow(str)) = gcd(m, n).$$

Щоб отримати повну коректність програми для додатних значень змінних M та N, треба довести її завершуваність (точніше визначеність значення функції $WH(p\_g, f\_g)(st)$). Це означає, що треба довести визначеність значення $NumItWH((p\_g, f\_g), st)$, якщо $m, n > 0$.

Розглянемо послідовність станів змінних:

$$st_0 = st,\ st_1 = fs(st_0),\ st_2 = fs(st_1),\ \ldots .$$

Оскільки при обчисленні нових станів задіяна лише функція віднімання, яка є всюди визначеною, то кожен елемент послідовності є визначеним, а сама послідовність – нескінченною.

Доведемо, що в цій послідовності є стан, у якому значення змінних M та N збігаються. Доведення виконаємо від супротивного. Припустимо, що в послідовності для кожного стану значення M та N різні. Розглянемо тепер послідовність сум значень цих змінних, тобто послідовність

$$t_0 = M\Rightarrow(st_0) + N\Rightarrow(st_0),\ t_1 = M\Rightarrow(st_1) + N\Rightarrow(st_1),$$
$$t_2 = M\Rightarrow(st_2) + N\Rightarrow(st_2),\ \ldots$$

У початковому стані значення M та N додатні, тому $t_0 > 0$. За відсутності рівних значень M та N додатними мають бути й усі інші числа t_0, t_1, t_2, … . Разом із тим послідовність є строго спадною, тобто $t_0 > t_1 > t_2 > \ldots$, тому що нові значення M та N отримуються відніманням меншого з чисел від більшого. За наведених умов послідовність t_0, t_1, t_2,… не може бути нескінченною, оскільки обмежена знизу числом 2, що є найменшою сумою двох додатних чисел.

Отримане протиріччя свідчить про те, що в послідовності станів змінних $st_0 = st$, $st_1 = fs(st_0)$, $st_2 = fs(st_1)$, … є стан з однаковими значеннями M та N, на якому цикл зупиняється, тобто значення $WH(p\_g, f\_g)(st)$ буде визначеним.

Це означає, що для додатних значень змінних M та N доведено повну коректність нашої програми.■

Зауваження 1.10. При доведенні теореми ми інколи застосовували програмістську термінологію, говорячи, наприклад, про ітерації циклу, завершуваність програми, а не про обчислення послідовності певних значень функцій, як того вимагає композиційна семантика. Зроблено це для того, щоб доведення краще відповідало програмістській інтуїції. Утім, сподіваємося, що читач зможе розпізнати і трансформувати програмістські терміни в точні математичні формулювання.

Варто підкреслити ще одну обставину, а саме те, що без наявності формальної семантики й синтаксису ми не могли б навіть говорити про доведення різних властивостей програм, зокрема їх коректності. На інтуїтивному рівні розуміння програм може бути лише інтуїтивне обґрунтування їхніх властивостей.

Розглянемо ще один приклад: побудуємо програму мовою *SIPL*, яка за допомогою операції додавання обчислює результат піднесення числа x до натурального степеня n ($n \geq 0$), і доведемо коректність побудованої програми.

Вважатимемо, що $0^0 = 1$.

У наведеній нижче програмі *EXP* обчислення x^n використовуються змінні X та N для позначення відповідно значень x та n, а також змінна R для повернення результату r:

$EXP \equiv$

begin
  $R :=\ 1$;
  while $N > 0$ **do**
    begin
      $R :=\ R * X$;
      $N :=\ N - 1$
    end
end

Тепер побудуємо семантичний терм програми. Отримаємо:
$sem\_P(EXP) =$
$= sem\_P($**begin** $R :=\ 1;$
  while $N > 0$ **do begin** $R := R * X;\ N := N - 1$ **end end**$) =$
$= AS^R(\overline{1}) \bullet WH(S^2(gr, N\Rightarrow, \overline{0}),$
$AS^R(S^2(mult, R\Rightarrow, X\Rightarrow) \bullet AS^N(S^2(sub, N\Rightarrow, \overline{1})))$
Для спрощення позначимо
$p\_e = S^2(gr, N\Rightarrow, \overline{0})$,
$f\_e = AS^R(S^2(mult, R\Rightarrow, X\Rightarrow) \bullet AS^N(S^2(sub, N\Rightarrow, \overline{1}))$.
У цьому випадку $sem\_P(EXP) = AS^R(\overline{1}) \bullet WH(p\_e, f\_e)$.

Перейдемо до доведення часткової коректності програми *EXP*.

Теорема 1.5 (про часткову коректність програми *EXP*). Нехай стан *st* такий, що $X\Rightarrow(st) = x$, $N\Rightarrow(st) = n$ ($n \geq 0$). Тоді якщо $sem\_P(EXP)(st)\downarrow = str$, то $R\Rightarrow(str) = x^n$.

Доведення виконаємо двома методами: *прямим* і *зворотним*. Прямий метод доведення означає, що індуктивне твердження для циклу формулюється для *i*-го стану від початку циклу, а зворотний – за *j* ітерацій до завершення циклу.

Для прямого методу *індуктивна гіпотеза* має вигляд:

якщо $X\Rightarrow(st_0) = x$, $N\Rightarrow(st_0) = n$ ($n \geq 0$), $R\Rightarrow(st_0) = 1$, цикл виконався i разів ($i \geq 0$), сформувавши стан st_i, то

$$X\Rightarrow(st_i) = x,\ N\Rightarrow(st_i) = n - i,\ R\Rightarrow(st_i) = x^i.$$

База індукції. Нехай $i = 0$. Тоді

$p\_e(st_0) = S^2(gr, N\Rightarrow, \overline{0})(st_0) = gr(N\Rightarrow(st_0), \overline{0}\,(st_0)) = gr(n,0) = false$,

а це означає, що $n = 0$, тому

$$X\Rightarrow(st_0) = x,\ N\Rightarrow(st_0) = n,\ R\Rightarrow(st_0) = 1 = x^0.$$

Індуктивна гіпотеза для бази індукції доведена.

Крок індукції. Нехай індуктивна гіпотеза є правильною для деякого $i > 0$. Доведемо, що вона є правильною для $i + 1$.

За індуктивним припущенням

$$X\Rightarrow(st_i) = x,\ N\Rightarrow(st_i) = n - i,\ R\Rightarrow(st_i) = x^i.$$

Оскільки цикл виконується $i + 1$ разів, то обчислимо

$$f\_e(st_i) = AS^R(S^2(mult, R\Rightarrow, X\Rightarrow) \bullet AS^N(S^2(sub, N\Rightarrow, \overline{1}\,))(st_i) =$$

$$= AS^N(S^2(sub, N\Rightarrow, \overline{1}\,))(\, AS^R(S^2(mult, R\Rightarrow, X\Rightarrow)\,(st_i)) =$$

$$= AS^N(S^2(sub, N\Rightarrow, \overline{1}\,))(\, st_i\nabla[\, R\mapsto S^2(mult, R\Rightarrow, X\Rightarrow)\,(st_i)]) =$$

$$= AS^N(S^2(sub, N\Rightarrow, \overline{1}\,))(\, st_i\nabla[\, R\mapsto mult(R\Rightarrow(st_i), X\Rightarrow(st_i))]) =$$

$$= AS^N(S^2(sub, N\Rightarrow, \overline{1}\,))(\, st_i\nabla[\, R\mapsto mult(x^i, x))]) =$$

$$= AS^N(S^2(sub, N\Rightarrow, \overline{1}\,))(\, st_i\nabla[\, R\mapsto x^{i+1}]) =$$

$$= (\, st_i\nabla[\, R\mapsto x^{i+1}])\ \nabla[\, N\mapsto S^2(sub, N\Rightarrow, \overline{1}\,)(\, st_i\nabla[\, R\mapsto x^{i+1}])]) =$$

$$= (st_i\nabla[R\mapsto^{i+1}])\nabla[N\mapsto sub(N\Rightarrow(st_i\nabla[R\mapsto x^{i+1}]), \overline{1}\,(st_i\nabla[\, R\mapsto x^{i+1}])]) =$$

$$= (st_i\nabla[R\mapsto x^{i+1}])\nabla[N\mapsto sub(n - i),1]) =$$

$$= (st_i\nabla[R\mapsto x^{i+1}])\nabla[N\mapsto(n - i) - 1]) =$$

$$= st_i\nabla[R\mapsto x^{i+1}, N\mapsto n - (\, i + 1)]).$$

Позначимо отриманий стан st_{i+1}. Для цього стану

$$X\Rightarrow(st_{i+1}) = x,\ N\Rightarrow(st_{i+1}) = n - (i + 1),\ R\Rightarrow(st_{i+1}) = x^{i+1}.$$

А це означає, що індуктивна гіпотеза доведена.

Залишилось довести часткову правильність програми *EXP*.

Дійсно, нехай програма *EXP* має *st* як початковий стан і завершується за k кроків виконання циклу в стані st_k. Це означає, що виконання оператора $AS^R(\overline{1}\,)$ зумовлює стан st_0. Тому за індуктивною гіпотезою маємо, що $X\Rightarrow(st_k) = x$, $N\Rightarrow(st_k) = n - k$, $R\Rightarrow(st_k) = x^k$. Однак оскільки програма завершується за k кроків, то $n - k = 0$, тобто $n = k$. Тому $R\Rightarrow(st_k) = x^n$, тобто програма частково коректна.

Повна коректність вимагає доведення завершуваності програми, яка випливає з факту, що кожна ітерація циклу зменшує n на 1, тобто обов'язково буде досягнуто значення 0. А це означає, що програма завершиться. Тому справедлива така теорема.

Теорема 1.6 (про повну коректність програми *EXP*). Нехай стан st такий, що $X{\Rightarrow}(st) = x$, $N{\Rightarrow}(st) = n$ ($n \geq 0$). Тоді програма *EXP* завершується в певному стані *str* (тобто $sem\_P(EXP)(st){\downarrow} = str$) такому, що $R{\Rightarrow}(str) = x^n$.

Доведення часткової коректності програми зворотним методом ґрунтується на такій *індуктивній гіпотезі*:

нехай стан st_j такий, що $X{\Rightarrow}(st_j) = x$, $N{\Rightarrow}(st_j) = k$, $R{\Rightarrow}(st_j) = m$, $NumItWH((p\_e,\ f\_e),\ st_j) = j$, $WH((p\_e,\ f\_e)(st_j) = str$, тоді $k = j$, $X{\Rightarrow}(str) = x$, $N{\Rightarrow}(str) = 0$, $R{\Rightarrow}(str) = m * x^j$.

Доведення подібне доведенню індуктивної гіпотези для програми *GCD*, тому його не наводимо.

Втім, формальне доведення можна побачити в прикладі 6.9.

Висновки

У цьому розділі був розглянутий простий приклад програми в мові *SIPL*. Зазначена мова є одним із варіантів простих мов подібного типу, наприклад мови *WHILE* [22, 25]. Фактично такі мови є мовами структурного програмування. Однак тут була дана композиційна семантика мови *SIPL*, яка базується на запропонованому В. Н. Редьком композиційному програмуванні [14]. Отже, основні результати розділу полягають у такому:

1. Дано неформальний і формальний описи мови *SIPL*.
2. Для формального визначення синтаксису мови запропоновано її БНФ і розглянуто її властивості (дерева виведення, однозначність і неоднозначність БНФ).
3. Для формального визначення семантики мови побудовано алгебру даних мови *SIPL* і низку функціональних алгебр. Операціями цих алгебр є композиції, що формалізують засоби конструювання програм.
4. Визначено відображення побудови семантичного терму програми.
5. Досліджено низку тотожностей у алгебрі програм.

6. Продемонстрована можливість доведення часткової та повної коректності на підставі введених формалізацій.

Разом із тим виникають деякі запитання. Можливо, усі введені поняття й методи спрацьовують лише у випадку простої мови, а для складніших мов це не підійде? Чи є введені поняття необхідними й суттєвими, чи, може, вони випадкові? Які поняття ще слід увести, щоб працювати з потужнішими мовами програмування?

Запитання такого типу є загальнометодологічними. Їх розглянемо в наступному розділі.

Контрольні запитання

1. Які недоліки неформального опису мов програмування?
2. Яким чином задається синтаксис мови *SIPL*?
3. Що є деревом синтаксичного виведення програми?
4. Коли програма є синтаксично правильною?
5. Що таке неоднозначна БНФ?
6. Для чого вводиться пріоритет операцій?
7. Які пріоритети введено для операцій мови *SIPL*?
8. Які синтаксичні категорії та метазмінні введені в мові *SIPL*?
9. Які типи даних використовуються в мові *SIPL*?
10. Які алгебри даних пов'язані з мовою *SIPL*?
11. Як визначаються операції іменування та розіменування?
12. Які класи функцій використовуються для формалізації семантики мови *SIPL*?
13. Які типи номінативних функцій використовуються для формалізації семантики мови *SIPL*?
14. Як визначається суперпозиція в n-арну функцію?
15. Як визначається композиція присвоювання?
16. Як визначається композиція послідовного виконання?
17. Як визначається композиція розгалуження?
18. Як визначається композиція циклу?
19. Які програмні алгебри пов'язані з мовою *SIPL*?
20. За якими критеріями в перелік композицій включають функції?

21. Як визначається підалгебра алгебри *A_Prog*, породжена мовою *SIPL*?
22. Як частковість функцій впливає на визначення композицій?
23. Як визначається семантичний терм програми?
24. Як будується семантичний терм програми?
25. Які властивості виконуються для програмних алгебр?
26. Як визначаються монотонні функції?
27. Як визначаються еквітонні функції?
28. Як визначають програми мови *SIPL*?
29. Що таке часткова коректність програм?
30. Що таке повна коректність програм?

Вправи

1. Довести однозначність синтаксичного аналізу при використанні правил пріоритету й асоціативності операцій.
2. Побудувати рекурентні співвідношення для визначення синтаксичних категорій.
3. Навести приклади неоднозначного аналізу програм мови *SIPL*.
4. Навести індуктивне визначення множин термів програмної алгебри.
5. Побудувати програми мовою *SIPL* для таких задач (x, y, n – додатні цілі числа):

- обчислення $x * y$ з використанням функцій +, – (не використовуючи функції *);
- обчислення $n!$;
- обчислення $x - y$ з використанням функції віднімання одиниці (–1);
- перевірки парності числа n;
- обчислення $x \; div \; y$ з використанням функцій +, –;
- обчислення x^y з використанням функцій *, +, –;
- обчислення $[lg\ n]$ з використанням функцій *div*, *mod*, +, –;
- обчислення $x \; mod \; y$ з використанням функцій +, –;
- обчислення 3^x з використанням функцій *, +, –.

6. Перевірити синтаксичну правильність програм, створених у вправі 5, побудувавши їхні дерева синтаксичного виведення. Побудувати семантичні терми цих програм і застосувати їх до певних вхідних даних.

7. Довести часткову й повну коректність програм, створених у вправі 5.

8. Довести, що програми мови *SIPL* задають однозначні (детерміновані) функції.

9. Довести лему про збіжність значень для програм мови *SIPL*, а саме: визначити за програмою мови *SIPL* змінні, від яких залежить програма (індукцією за структурою програми), і довести, що значення програми будуть однаковими на станах з однаковими значеннями таких змінних.

10. Побудувати консервативні розширення мови *SIPL*. Тут неформально вважаємо, що розширення *SIPL_C* є консервативним розширенням *SIPL*, якщо для кожної програми з мови *SIPL_C* можна побудувати еквівалентну програму з мови *SIPL*. Включити до консервативного розширення *n*-арні функції *div*, *mod*, *abs*; композиції **repeat_until_**, **if_then_**, цикли **for** тощо. Надати формалізацію розширеної мови.

6. ФОРМАЛЬНА СЕМАНТИКА МОВ ПРОГРАМУВАННЯ

Формальною семантикою програми (або деякого мовного виразу) називається смислове значення програми (або мовного виразу), подане в термінах певної формальної (математичної) моделі. Наявність формальної семантики дозволяє досліджувати властивості програм з використанням математичних методів. Наприклад, формальна верифікація програм (доведення їхньої коректності) можлива лише за наявності формальної семантики.

У цьому розділі додатково розглянемо три традиційні методи подання семантики мов програмування: денотаційну, операційну й аксіоматичну семантики.

6.1. Денотаційна семантика

Композиційна семантика мови *SIPL*, викладена в першому розділі, може розглядатися як спеціальний випадок денотаційної (денотативної) семантики. Така семантика для мов програмування була запропонована в роботах Кристофера Стречі (Christopher Strachey) і Дана Скотта (Dana Scott) у 60-ті рр. минулого століття.

У денотаційній семантиці денотат (смислове значення) програми тлумачиться як функція, яка переводить вхідні дані у вихідні. Денотат складної програми подається як композиція денотатів її складових (принцип композиційності). Пізніше Дана Скотт запропонував теорію доменів для подання денотатів програм. Тут під доменом будемо розуміти спеціальну множину, на якій задано відношення часткового порядку, для якого додатково виконуються певні аксіоми. ω-області, розглянуті в попередньому розділі, є прикладами доменів.

Домени можна будувати з базових доменів за допомогою операцій розміченого об'єднання, декартового добутку, побудови різних класів функцій (зокрема класів неперервних функцій). Тому за допомогою доменів можна подати семантику різних мов програмування, які дозволяють рекурсивність, недетермінізм, складні структури даних тощо.

Семантика рекурсивних програм подається за допомогою оператора найменшої нерухомої точки. Тому семантика циклу подається такою формулою:

$$WH(p,f) = lfp_X IF(p, f \bullet X, id).$$

Побудуємо денотаційну семантику для мови *SIPL*.

Використовуємо граматику з табл. 1.4. Денотат програмної конструкції *e* позначаємо подвійними дужками $[\![e]\!]$ (ці подвійні дужки використовуємо без типізації за складовими програми, тому відображення *sem_P*, *sem_S*, *sem_A*, *sem_B* подаємо такими дужками).

Таблицю для денотаційної семантики мови отримуємо з табл. 1.6, змінивши лише визначення оператора циклу (табл. 6.1).

***Таблиця* 6.1**

Правило заміни	**Назва правила**
$[\![P]\!]$: $Prog \to TFS$ задається правилами:	
$[\![$**begin** S **end**$]\!] = [\![S]\!]$	*DS_Prog*
$[\![S]\!]$: $Stm \to TFS$ задається правилами:	
$[\![x{:=}a]\!] = AS^x([\![a]\!])$ $[\![S_1; S_2]\!] = [\![S_1]\!] \bullet [\![S_2]\!]$ $[\![$**if** b **then** S_1 **else** $S_2]\!] = IF([\![b]\!], [\![S_1]\!], [\![S_2]\!])$ $[\![$**while** b **do** $S]\!] = lfp_X IF([\![b]\!], [\![S]\!] \bullet X, id)$ $[\![$**begin** S **end**$]\!] = [\![S]\!]$ $[\![skip]\!] = id$	*S_Stm_As* *DS_Stm_Seq* *DS_Stm_If* *DS_Stm_Wh* *DS_Stm_BE* *DS_Stm_skip*

Закінчення табл. 6.1

Правило заміни	Назва правила
$[\![A]\!]$: $Aexp \to TFA$ задається правилами:	
$[\![n]\!] = \overline{n}$ $[\![x]\!] = x\Rightarrow$ $[\![a_1 + a_2]\!] = S^2(add, [\![a_1]\!], [\![a_2]\!])$ $[\![a_1 - a_2]\!] = S^2(sub, [\![a_1]\!], [\![a_2]\!])$ $[\![a_1 * a_2]\!] = S^2(mult, [\![a_1]\!], [\![a_2]\!])$ $[\![(a)]\!] = [\![a]\!]$	*DS_A_Num* *DS_A_Var* *DS_A_Add* *DS_A_Sub* *DS_A_Mult* *DS_A_Par*
$[\![B]\!]$: $Bexp \to TFB$ задається правилами:	
$[\![a_1 = a_2]\!] = S^2(eq, [\![a_1]\!], [\![a_2]\!])$ $[\![a_1 > a_2]\!] = S^2(gr, [\![a_1]\!], [\![a_2]\!])$ $[\![b_1 \vee b_2]\!] = S^2(or, [\![b_1]\!], [\![b_2]\!])$ $[\![\neg b]\!] = S^1(neg, [\![b]\!])$ $[\![(b)]\!] = [\![b]\!]$	*DS_B_eq* *DS_B_gr* *DS_B_or* *DS_B_neg* *DS_B_Par*

Теорема 6.1 (про еквівалентність композиційної та денотаційної семантик програм мови *SIPL*). Для довільної програми *P* мови *SIPL* її композиційна семантика збігається з її денотаційною семантикою, тобто $sem\_P(P) = [\![P]\!]$.

Доведення. Спочатку доводимо, що для довільного арифметичного виразу *a* та довільної умови *b* маємо, що

$$sem\_A(a) = [\![a]\!] \text{ та } sem\_B(b) = [\![b]\!] .$$

Використовуємо індукцію за структурою *a* та *b*. Твердження випливає з табл. 1.6 та 6.1, які задають однакові значення для складових *a* та *b*. Далі доводимо *sem_S* (*S*) = ⟦*S*⟧ індукцією за структурою оператора *S*. Табл. 1.6 та 6.1 задають однакові значення (крім циклу) для складових *S*. Що стосується циклу, то з теореми 5.5 випливає *sem_ S* (**while** *b* **do** *S*) = ⟦**while** *b* **do** *S*⟧ . Звідси отримуємо *sem_P*(*P*) = ⟦*P*⟧ .■

Денотаційна семантика широко використовується для опису різноманітних мов програмування, тому вона є одним з головних методів формалізації програм.

Ще одним поширеним методом є операційна семантика.

6.2. Операційна семантика

Денотаційна семантика тлумачить програму як функцію, що переводить вхідні дані у вихідні. Така функція подається в алгебрі програм.

Операційна семантика також тлумачить програму як функцію, але ця функція подається іншим способом, а саме як транзиційне відношення на множині даних. Операційну семантику ще називають *натуральною* семантикою.

Таке відношення подається множиною *індивідних* формул вигляду $\langle S, st\rangle \mapsto st'$, де *S* – оператор (чи програма), *st* та *st′* – стани. Такі формули називаємо *індивідними*, оскільки в них прописані конкретні (індивідні) стани. Також ці формули можна називати *тестами*. Формула (тест) $\langle S, st\rangle \mapsto st'$ вважається істинною, якщо обчислення *S* на стані *st* завершується у стані *st′*.

Зауважимо, що ми використовуємо формулу вигляду $\langle S, st\rangle \mapsto st'$ у лінійному вигляді замість формули $st \xrightarrow{S} st'$, яка чіткіше демонструє транзиційне відношення між станами.

Подібні формули використовуються також для подання значень виразів ($\langle S, st\rangle \mapsto n$) та умов ($\langle S, st\rangle \mapsto r$). Тут *n* – ціле число, *r* – булеве (логічне) значення.

Транзиційне відношення подається у вигляді правил операційної семантики, які мають вигляд

$$\frac{\Phi_1,\Phi_2,...,\Phi_n}{\Psi} \qquad \text{або} \qquad \frac{\Phi_1 \mid \Phi_2 \mid ... \mid \Phi_n}{\Psi},$$

де $\Phi_1,\Phi_2,...,\Phi_n$ – засновки правил, Ψ – висновки. Якщо засновків багато, то пишемо їх у кілька рядків, виокремлюючи вертикальною рискою:

$$\begin{array}{l} \Phi_1 \\ \mid \Phi_2 \\ \mid \ ... \\ \mid \Phi_n \\ \hline \Psi \end{array}$$

Аксіоми будемо записувати без засновків. Правила фактично задають обчислення заключного стану програми на підставі обчислень її складових. Отже, дерева виведення фактично подають порядок виконання операцій. Тому в цьому розділі будемо вживати вирази "побудувати дерево виведення" та "обчислити програму" як синоніми.

Для мови *SIPL* правила операційної семантики подаються в табл. 6.2.

***Таблиця* 6.2**

Назва правила	**Правило операційної семантики**
Правила для програми та операторів:	
PR	$\dfrac{< S, st >\mapsto}{< \textbf{begin}\ S\ \textbf{end}, st >\mapsto}$
AS	$\dfrac{< a, st >\mapsto}{< x := a, st >\mapsto \qquad \mapsto}$
SEQ	$\dfrac{< S_1, st >\mapsto \qquad\qquad\qquad \mapsto}{< S_1; S_2, st >\mapsto}$
IF*true*	$\dfrac{< b, st >\mapsto \qquad\qquad\qquad \mapsto}{< \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, st >\mapsto}$

IF*false*	$\dfrac{< b, st >\mapsto}{< \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2, st >\mapsto}$
WH*false*	$\dfrac{< b, st >\mapsto}{< \textbf{while } b \textbf{ do } S, st >\mapsto}$

Закінчення табл. 6.2

Назва правила	**Правило операційної семантики**
WH*true*	$\dfrac{< b, st >\mapsto \quad \mid < S, st >\mapsto \quad \mid < \textbf{while } b \textbf{ do } S, st'' >\mapsto}{< \textbf{while } b \textbf{ do } S, st >\mapsto}$
BEG	$\dfrac{< S, st >\mapsto}{< \textbf{begin } S \textbf{ end}, st >\mapsto}$
skip	$< skip, st >\mapsto$
Правила для виразів:	
Num	$< n, st >\mapsto$
Var	$< x, st >\mapsto$
A+	$\dfrac{< a_1, st >\mapsto}{< a_1 + a_2, st >\mapsto \quad , n_2)} \mapsto$
A–	$\dfrac{< a_1, st >\mapsto}{< a_1 - a_2, st >\mapsto \quad n_2)} \mapsto$
A*	$\dfrac{< a_1, st >\mapsto}{< a_1 * a_2, st >\mapsto \quad , n_2)} \mapsto$
A()	$\dfrac{< a, st >\mapsto}{< (a), st >\mapsto}$
Правила для умов:	
B=	$\dfrac{< a_1, st >\mapsto}{< a_1 = a_2, st >\mapsto \quad)} \mapsto$
B>	$\dfrac{< a_1, st >\mapsto}{< a_1 > a_2, st >\mapsto \quad)} \mapsto$
B$\vee$	$\dfrac{< b_1, st >\mapsto}{< b_1 \vee b_2, st >\mapsto \quad)} \mapsto$

B¬	$\dfrac{< b\ , st >\mapsto}{< \neg b\ , st >\mapsto}$
B()	$\dfrac{< b, st >\mapsto}{< (b), st >\mapsto}$

Для програми P її операційну семантику позначаємо $Sem\_P_{OP}(P)$.

Приклад 6.1. Обчислити в операційній (натуральній) семантиці умову $\neg M = N$ на стані $st = [M \mapsto 8, N \mapsto 16]$.

Розв'язання. Будемо будувати дерево виведення прямим методом – від аксіом до висновків. Питання полягає в тому, які аксіоми вибирати. Орієнтуємось на процес обчислення умови $\neg M = N$. Для обчислення цієї умови треба обчислити значення змінних M та N на стані $[M \mapsto 8, N \mapsto 16]$. Це задається правилом (схемою правил) Var. Назви правил будемо писати з лівого боку. Ураховуючи, що

$st(M) = [M \mapsto 8, N \mapsto 16](M) = 8$ та $st(N) = [M \mapsto 8, N \mapsto 16](N) = 16$,

отримуємо дві аксіоми:

$$\text{Var: } < M, [M \mapsto \quad \mapsto \quad \mapsto \quad \text{та}$$

$$\text{Var: } < N, [M \mapsto \quad \mapsto \quad .$$

Далі слід застосувати правило B= для обчислення умови $M = N$. Ураховуючи, що умова $8 = 16$ дає значення *false*, отримуємо таке дерево виведення:

$$\text{B=}: \dfrac{\begin{array}{l}\text{Var:} < M, [M \mapsto \quad \mapsto \quad \mapsto \\ \text{Var:} < N, [M \mapsto \quad \mapsto \quad \end{array}}{< M = N, [M \mapsto \quad \mapsto}$$

Тепер надійшла черга застосувати правило B¬. Оскільки $neg(false) = true$, то маємо:

$$\text{B=}: \dfrac{\begin{array}{l}\text{Var:} < M, [M \mapsto \quad \mapsto \quad \mapsto \\ \text{Var:} < N, [M \mapsto \quad \mapsto \quad \mapsto \end{array}}{< M = N, [M \mapsto \quad \mapsto \quad \mapsto}$$

$$\text{B}\neg : \dfrac{< M = N, [M \mapsto \quad \mapsto \quad , \quad \neg false)}{< \neg M = N, [M \mapsto \quad \mapsto \quad , \quad \neg false)}$$

Приклад 6.2. Обчислити в операційній (натуральній) семантиці вираз $N - M$ на стані $st = [M \mapsto 8, N \mapsto 16]$.

Розв'язання. Як і в попередньому випадку, застосовуємо аксіоми для значень змінних, а потім – правило B–. При застосуванні цього правила обчислюємо значення 16 – 8, яке дорівнює 8. Такі вирази будемо писати як коментар у дужках у висновку правила. Отримуємо таке дерево:

$$\text{B}-:\frac{\begin{array}{l}\text{Var}:<M,[M\mapsto \quad \mapsto \\ \text{Var}:<N,[M\mapsto \quad \mapsto \end{array}}{<N-M,[M\mapsto \quad \mapsto \qquad\qquad -8)}$$

Приклад 6.3. Обчислити в натуральній семантиці оператор $N := N - M$ на стані $st = [M \mapsto 8, N \mapsto 16]$.

Розв'язання. Після обчислення виразу $N - M$ застосовуємо правило AS, яке задає новий стан таким чином:

$$[M\mapsto \quad \mapsto \qquad\qquad \mapsto \quad \mapsto \ .$$

Отримуємо дерево виведення:

$$\text{AS}:\frac{\text{B}-:\dfrac{\begin{array}{l}\text{Var}:<M,[M\mapsto \quad \mapsto \\ \text{Var}:<N,[M\mapsto \quad \mapsto \end{array}}{<N-M,[M\mapsto \quad \mapsto}}{<N:=N-M,[M\mapsto \quad \mapsto \qquad \mapsto \quad \mapsto}$$

■

Розглянуті приклади демонструють певні складнощі прямого методу (від засновків до висновків) при побудові дерев виведення. А саме, необхідно якимось чином обирати аксіоми, а потім правила для подальшої побудови дерева. Тому замість прямого розглянемо зворотний метод (від висновків до засновків). У цьому методі для зручності нотації при побудові дерева правила записують у "перевернутому" вигляді (міняють місцями засновки та висновки). Зворотний метод означає, що від складного висновку ми переходимо до простіших засновків, у той час як для прямого методу – від простіших засновків до складного висновку.

Однак і у випадку зворотного методу нас чекають свої труднощі: на початку обчислення у формулі $<S, st> \mapsto st'$ нам відомі S

та *st*, але не відоме *st'*. Такі стани будемо позначати змінними, які у процесі побудови дерева набувають конкретних значень.

Крім того, не зрозуміло, як обирати альтернативні правила WH*true* та WH*false* (IF*true* та IF*false*). Тут допоможе та обставина, що ці правила вимагають спочатку обчислення умови. Тому при побудові дерева зворотним методом будемо рисувати риску для відокремлення висновків від засновків, потім – обчислювати умову і, залежно від отриманого значення, – обирати необхідне правило. Отже, якщо значенням умови є *true*, то обираємо правило WH*true* (IF*true*); якщо значенням умови є *false*, то обираємо правило WH*false* (IF*false*).

Приклад 6.4. Обчислити в натуральній семантиці умову $\neg M = N$ на стані $st = [M \mapsto 8, N \mapsto 16]$ зворотним методом.

Розв'язання. Потрібно побудувати виведення для формули $< \neg M = N, st > \mapsto$, де стан $st = [M \mapsto 8, N \mapsto 16]$ відомий, а логічне значення r – ні. Структура виразу $\neg M = N$ свідчить, що основною операцією є операція заперечення, тому при зворотному методі слід застосувати правило B¬. Застосувавши це правило для нашої формули $< \neg M = N, st > \mapsto$, отримуємо дерево

$$\text{B}\neg: \frac{< \neg M = N, st > \mapsto}{< M = N, st > \mapsto}$$

До умови $M = N$ тепер можна застосувати правило B=. Отримуємо:

$$\text{B}\neg: \frac{\dfrac{< \neg M = N, st > \mapsto}{\text{B}=: \dfrac{< M = N, st > \mapsto}{< M, st > \mapsto}}}{\qquad\qquad \mapsto}$$

З'явились нові змінні n_1 та n_2 такі, що $neg(r) = eq(n_1, n_2)$. Отже, маємо три невідомі: r, n_1 та n_2.

Застосувавши два рази правило Var, отримуємо, що $n_1 = 8$ та $n_2 = 16$. Рухаючись вверх по побудованому дереву, знаходимо, що $neg(r) = eq(8, 16) = false$, $r = neg(false) = true$.

Значення всіх невідомих знайдено, що дозволяє переписати отримане дерево у зворотному вигляді (у кінці дерева два засновки подані на двох рядках). Позначимо це дерево як (RB):

(RB):

$$\text{B}\neg:\frac{<\neg M=N,[M\mapsto \quad \mapsto \quad \mapsto}{\text{B}=:\dfrac{<M=N,[M\mapsto \quad \mapsto \quad \mapsto}{\begin{array}{l}\text{Var:}<M,[M\mapsto \quad \mapsto\\ \mid\ \text{Var:}<N,[M\mapsto \quad \mapsto \quad \mapsto\end{array}}}$$

Отримане дерево (RB) є деревом виведення формули $<\neg M=N,[M\mapsto \quad \mapsto$.

Це дерево можна переписати як виведення і для прямого методу:

$$\text{B}=:\frac{\begin{array}{l}\text{Var:}<M,[M\mapsto \quad \mapsto \quad \mapsto\\ \mid\ \text{Var:}<N,[M\mapsto \quad \mapsto\end{array}}{\text{B}\neg:\dfrac{<M=N,[M\mapsto \quad \mapsto \quad \mapsto}{<\neg M=N,[M\mapsto \quad \mapsto}}$$

Приклад 6.5. Обчислити в натуральній семантиці оператор

$$\textbf{if } M>N \textbf{ then } M:=M-N \textbf{ else } N:=N-M$$

на стані $st = [M\mapsto 8, N\mapsto 16]$ зворотним методом.

Розв'язання. Потрібно побудувати виведення для формули

$$<\textbf{if } M>N \textbf{ then } M:=M-N \textbf{ else } N:=N-M, st>\mapsto \quad ,$$

де стан $st = [M\mapsto 8, N\mapsto 16]$ відомий, а стан st' – ні.

Структура умовного оператора свідчить, що при зворотному методі слід застосувати правило IF*true* або IF*false*. Яке саме правило слід застосувати, залежить від значення умови $M>N$ на стані *st*. Цю умову можна обчислити окремо й потім продовжити побудову дерева за потрібним правилом IF*true* або IF*false*. Також можна обчислювати умову безпосередньо в дереві виведення. Для цього введемо ще одну змінну *r*, яка буде вказувати на правило IF*true* або IF*false*. Таке тимчасове правило позначимо IF*r*. Отримуємо дерево

$$\text{IF}r:\frac{<\textbf{if } M>N \textbf{ then } M:=M-N \textbf{ else } N:=N-M, st>\mapsto}{<M>N, st>\mapsto}$$

Продовжуємо розкривати дерево виведення умови. Отримуємо

$$\mathrm{IF}r:\frac{< \textbf{if } M > N \textbf{ then } M := M - N \textbf{ else } N := N - M, st > \mapsto}{\mathrm{B}>:\dfrac{< M > N, st > \mapsto}{\mathrm{Var}:< M, st > \mapsto}} \qquad \mapsto$$

Тепер знаходимо, що $n_1 = 8$, $n_2 = 16$, $r = false$. Таким чином, треба застосувати правило IF*false*. Замінюючи тимчасове правило IFr на правило IF*false*, отримуємо:

$$\mathrm{IF}false:\frac{< \textbf{if } M > N \textbf{ then } M := M - N \textbf{ else } N := N - M, st > \mapsto}{\mathrm{B}>:\dfrac{< M > N, st > \mapsto}{\mathrm{Var}:< M, st > \mapsto}} \qquad (\mathrm{RA}) < N := N - M, st > \mapsto$$

$$\Big|\ \mathrm{Var}:< N, st > \mapsto$$

Тут (RA) позначає дерево виведення формули $< N := N - M, st > \mapsto$, яке має вигляд

(RA)

$$\mathrm{AS}:\frac{< N := N - M, st > \mapsto \qquad \mapsto}{\mathrm{A}-:\dfrac{< N - M, st > \mapsto \qquad :b(n_2, n_1))}{\mathrm{Var}:< N, st > \mapsto} \qquad \mapsto}$$

Тут $n_2 = 16$, $n_1 = 8$, $n_3 = 8$, $st' = [M \mapsto \qquad \mapsto$.

Підставивши це дерево в попереднє дерево, отримуємо повне дерево виведення для нашого прикладу, яке позначимо (RI). Саме дерево тут не наводимо, оскільки ширина сторінки не дозволяє його подати. Однак це дерево (при заміні операторів на простіші позначення) подамо в прикл. 6.6.

Отже, ми обчислили $st' = [M \mapsto \qquad \mapsto$. Це означає, що ми довели формулу

$< \textbf{if } M > N \textbf{ then } M := M - N \textbf{ else } N := N - M, [M \mapsto \qquad \mapsto \qquad \mapsto$

$[M \mapsto \qquad \mapsto$

Приклад 6.6. Побудувати в операційній семантиці дерево обчислення програми *GCD* (розд. 1) на стані $st = [M \mapsto 8, N \mapsto 16]$ зворотним методом.

Розв'язання. Програма *GCD* обчислення найбільшого спільного дільника за алгоритмом Евкліда має такий вигляд:

GCD ≡

begin

while ¬ $M = N$ **do**

if $M > N$ **then** $M := M - N$ **else** $N := N - M$

end

Нам потрібно побудувати виведення для формули

$$< GCD, st > \mapsto$$

де стан $st = [M \mapsto 8, N \mapsto 16]$ відомий, а st' – ні.

Далі для скорочення запису будемо вживати такі позначення фрагментів програми *GCD*:

- W = **while** ¬$M = N$ **do if** $M > N$ **then** $M := M - N$ **else** $N := N - M$
- I = **if** $M > N$ **then** $M := M - N$ **else** $N := N - M$

Очевидно, що першим необхідно застосувати правило BEG:

$$\text{BEG}: \frac{< \textbf{begin } W \textbf{ end}, st > \mapsto}{< W, st > \mapsto}$$

Далі потрібно розкривати цикл W одним із правил WH*true* або WH*false*. Оскільки значення умови циклу невідоме, то вводимо тимчасове правило WH*r*. Отримуємо:

$$\text{BEG}: \frac{< \textbf{begin } W \textbf{ end}, st > \mapsto}{\text{WH}r: \dfrac{< W, st > \mapsto}{< \neg M = N, st > \mapsto}}$$

Дерево для формули $< \neg M = N, st > \mapsto$ побудоване у прикл. 6.4. Отримане значення умови дорівнює *true*, тому слід замінити правило WH*r* на правило WH*true*. У цьому правилі є три засновки:

- $< \neg M = N, st > \mapsto$ (обчислення умови циклу);
- $< \textbf{if } M > N \textbf{ then } M := M - N \textbf{ else } N := N - M, st > \mapsto$

(обчислення тіла циклу; цю формулу позначаємо також як $< I, st > \mapsto$);

- $< W, st'' > \mapsto$ (обчислення циклу на стані st'').

Отримуємо таке дерево:

$$\mathrm{BEG}:\frac{<\textbf{begin}\ W\ \textbf{end}, st>\mapsto}{\mathrm{WH}true:\dfrac{<W, st>\mapsto}{\begin{array}{l}(\mathrm{RB})<\neg M=N, st>\mapsto\\ \quad|\ (\mathrm{RI})<I, st>\mapsto\\ \quad|\ (\mathrm{RW})<W, st''>\mapsto\end{array}}}$$

Відповідні дерева (RB) та (RI) для умови й умовного оператора було побудовано раніше (прикл. 6.4 та 6.5, у (RI) треба замінити st' на st'', отримуємо $st''=[M\mapsto\quad\mapsto\quad)$, а дерево для третього засновку (RW) побудуємо зараз.

Спочатку запишемо правило WHr:

$$\mathrm{WH}r:\frac{<W, st''>\mapsto}{<\neg M=N, st''>\mapsto}$$

Виведення формули $<\neg M=N, st''>\mapsto$ дає $st''=[M\mapsto\quad\mapsto\quad$, $r=false$ і подається деревом

$$\mathrm{B}\neg:\frac{<\neg M=N, st''>\mapsto}{\mathrm{B}=:\dfrac{<M=N, st''>\mapsto}{\begin{array}{l}\mathrm{Var}:<M, st''>\mapsto\\ \quad|\ \mathrm{Var}:<N, st''>\mapsto\end{array}}}$$

Тому треба застосувати правило WH*false*. Звідси

$$\mathrm{WH}false:\frac{<W, st''>\mapsto}{<\neg M=N, st''>\mapsto}$$

Підставивши дерево виведення для $<\neg M=N, st''>\mapsto$, отримуємо (RW).

Наводимо остаточний результат прикладу, підставляючи $[M\mapsto\quad\mapsto\quad$ замість st, $[M\mapsto\quad\mapsto\quad$ замість st'' та st':

$$\text{BEG}:\frac{< GCD,[M\mapsto \quad \mapsto \quad \mapsto \quad \mapsto \quad \mapsto}{\text{WH}true\dfrac{< W, st >\mapsto \quad \mapsto \quad \mapsto}{(\text{RB}) < \neg\, M = N,[M\mapsto \quad \mapsto \quad \mapsto}}$$

$$\big|\ (\text{RI}) < I,[M\mapsto \quad \mapsto \quad \mapsto \quad \mapsto \quad \mapsto$$

$$\big|\ (\text{RW}) < W,[M\mapsto \quad \mapsto \quad \mapsto \quad \mapsto \quad \mapsto$$

Залишилось вказати дерева (RB), (RI), (RW).

(RB):

$$\text{B}\neg:\frac{< \neg\, M = N,[M\mapsto \quad \mapsto \quad \mapsto}{\text{B}=:\dfrac{< M = N,[M\mapsto \quad \mapsto \quad \mapsto}{\text{Var:} < M,[M\mapsto \quad \mapsto}}$$

$$\big|\ \text{Var:} < N,[M\mapsto \quad \mapsto \quad \mapsto$$

(RI):

$$\text{IF}false:\frac{< I,[M\mapsto \quad \mapsto \quad \mapsto \quad \mapsto \quad \mapsto}{\text{B}>:\dfrac{< M > N,[M\mapsto \quad \mapsto \quad \mapsto}{\text{Var} :< M,[M\mapsto \quad \mapsto \quad \mapsto}}$$

$$\big|\ \text{Var} :< N,[M\mapsto \quad \mapsto \quad \mapsto$$

$$\Big|\quad \text{AS}:\frac{< N := N - M,[M\mapsto \quad \mapsto \quad \mapsto \quad \mapsto \quad \mapsto}{\text{A}-:\dfrac{< N - M,[M\mapsto \quad \mapsto \quad \mapsto}{\text{Var} :< N,[M\mapsto \quad \mapsto \quad \mapsto}}$$

$$\Big|\quad \big|\text{Var} :< M,[M\mapsto \quad \mapsto \quad \mapsto$$

Нагадуємо, що тут дерева, які починаються правилами B> та AS, є деревами двох засновків правила IF*false*.

(RW):

$$\text{WH}false:\frac{< W,[M\mapsto \quad \mapsto \quad \mapsto \quad \mapsto}{\text{B}\neg:\dfrac{< \neg M = N,[M\mapsto \quad \mapsto}{\text{B}=:\dfrac{< M = N,[M\mapsto \quad \mapsto}{\text{Var:} < M,[M\mapsto \quad \mapsto}}}$$

$$\big|\ \text{Var:} < N,[M\mapsto \quad \mapsto \quad \mapsto$$

Приклад 6.7. Побудувати в натуральній семантиці дерево обчислення програми *EXP* (розд. 1) на стані $st = [N \mapsto 2, X \mapsto 8]$ зворотним методом.

Розв'язання. У програмі *EXP* обчислення x^n використовуються змінні X та N для позначення відповідно значень x та n, а також змінна R для повернення результату r:

EXP ≡
begin
$R := 1$;
while $N > 0$ **do**
  begin
    $R := R * X$;
    $N := N - 1$
  end
end

Побудуємо виведення для формули $< EXP, st > \mapsto$, де стан $st = [N \mapsto 2, X \mapsto 8]$ відомий, а стан st' – ні.

Уведемо такі позначення для фрагментів програми *EXP*:

- W = **while** $N > 0$ **do begin** $R := R * X$; $N := N - 1$ end
- BW = $R := R * X$; $N := N - 1$

Починаємо будувати дерево виведення:

$$\text{BEG}: \frac{< \textbf{begin}\ R{:=}1; W\ \textbf{end}, st > \mapsto}{\text{SEQ}: \dfrac{< R{:=}1; W, st > \mapsto}{\text{AS}: \dfrac{< R{:=}1, st > \mapsto}{\text{Num}: < 1, st > \mapsto}}} \qquad \text{‘‘V1)} < W, st" > \mapsto$$

З побудованого дерева маємо, що

$st" = [N \mapsto \quad \mapsto \quad \mapsto \quad \mapsto \quad \mapsto \quad \mapsto \quad .$

Далі деякі отримані значення писатимемо в дужках прямо в дереві.

Переходимо до розкриття циклу (RW1) шляхом введення тимчасового правила:

$$\text{WH}r: \frac{< W, st" > \mapsto}{\text{B}>: \dfrac{< N > 0, st" > \mapsto}{\text{Var}: < N, st" > \mapsto}} \qquad \mapsto$$

Отримуємо, що $r = true$. Тому для розкриття (RW1) необхідно застосувати правило WH*true*:

$$\text{WH}true:\frac{\begin{array}{c}<W, st''> \mapsto \\ \hline <N > 0, st''> \mapsto \end{array}}{\text{B}>:\frac{}{\text{Var}:<N, st''> \mapsto \quad \text{ım}:<0, st''> \mapsto}}$$

$$\big|\ (\text{RS1}) <R := R * X; N := N - 1, st''> \mapsto$$

$$\big|\ (\text{RW2}) <W, st_1> \mapsto$$

Розкриваємо (RS1):

$$\text{SEQ:}\frac{< R := R * X; N := N - 1, st" >\mapsto}{\text{AS:}\dfrac{< R := R * X, st" >\mapsto}{\text{A*:}\dfrac{< R * X, st" >\mapsto}{\text{Var:} < R, st" >\mapsto}}} \quad \mapsto \quad \mapsto$$

$$\Big|\ \text{AS:}\frac{< N := N - 1, st_2 >\mapsto}{\text{A}-:\dfrac{< N - 1, st_2 >\mapsto}{\text{Var:} < N, st_2 >\mapsto \quad \text{m:} <1, st_2 >\mapsto}} \quad \mapsto \quad \mapsto$$

Звідси

$st_2 = [N \mapsto \quad \mapsto \quad \mapsto \quad ,$

$st_1 = [N \mapsto \quad \mapsto \quad \mapsto$

Продовжуємо розкриття (RW2):

$$\text{WH}true:\frac{< W, st_1 >\mapsto}{\text{B}>:\dfrac{< N > 0, st_1 >\mapsto}{\text{Var} :< N, st_1 >\mapsto}} \quad \overline{gr(1,0))} \quad \mapsto$$

$$\Big|\ (\text{RS2}) < R := R * X; N := N - 1, st_1 >\mapsto$$

$$\Big|\ (\text{RW3}) < W, st_3 >\mapsto$$

Розкриваємо (RS2):

$$\text{SEQ:}\frac{< R := R * X; N := N - 1, st_1 >\mapsto}{\text{AS:}\dfrac{< R := R * X, st_1 >\mapsto}{\text{A*:}\dfrac{< R * X, st_1 >\mapsto \quad \cdot)}{\text{Var:} < R, st_1 >\mapsto}}} \quad \mapsto \quad \mapsto$$

$$\Big|\ \text{AS:}\frac{< N := N - 1, st_4 >\mapsto}{\text{A}-:\dfrac{< N - 1, st_4 >\mapsto}{\text{Var:} < N, st_4 >\mapsto}} \quad \mapsto \quad \mapsto$$

Звідси

$st_4 = [N \mapsto \quad \mapsto \quad \mapsto \quad ,$

$st_3 = [N \mapsto \quad \mapsto \quad \mapsto \quad .$

Розкриваємо (RS3):

$$\text{WH}\mathit{false}:\frac{< W, st_3 > \mapsto \qquad t_3)}{\text{B}> : \dfrac{< N > 0, st_3 > \mapsto}{\text{Var} :< N, st_3 > \mapsto} \qquad = gr(0,0)) \mapsto}$$

Виведення завершено у стані

$$st_3 = st' = [N \mapsto \quad \mapsto \quad \mapsto \quad .$$

Цей стан задає правильне значення *R*.

Теорема 6.2 (про еквівалентність композиційної та операційної семантик програм мови *SIPL*). Для довільної програми *P* мови *SIPL* її композиційна семантика збігається з її операційною семантикою, тобто $sem\_P(P) = Sem\_P_{OP}(P)$.

Доведення. Спочатку доводимо, що для довільного арифметичного виразу *a* та довільної умови *b* маємо, що $sem\_A(a) = Sem\_P_{OP}(a)$ та $sem\_B(b) = Sem\_P_{OP}(b)$. Використовуємо індукцію за структурою *a* та *b*. Твердження випливає з табл. 1.6 та 6.2, які задають однакові значення для складових *a* та *b*. Далі доводимо $sem\_S(S) = Sem\_P_{OP}(S)$. Також використовуємо індукцію за структурою *S*. Однак випадок циклу вимагає ще однієї індукції за структурою дерева виведення.■

Зазначимо, що операційна семантика визначалась безпосередньо для мови *SIPL*, а не для семантичної алгебри мови.

6.3. Аксіоматична семантика

Операційна семантика, яка була розглянута в попередньому підрозділі, задавала семантику програм індивідними формулами (тестами) вигляду $<S, st> \mapsto st'$, де *S* – оператор (чи програма), *st* – початковий стан, *st′* – заключний стан. Така формула вважається істинною, якщо обчислення *S* на початковому стані *st* завершується у стані *st′*.

Індивідні формули задають значення програми на конкретних вхідних даних (конкретному початковому стані). Вони не зручні для формулювання властивостей програм, оскільки властивості

визначають поведінку програм на *класах* станів. Тому, якщо говорити про властивості програм у цілому, то слід від конкретних станів перейти до їхніх класів. Залежно від того, яким чином уточнюються класи, матимемо різні типи формул. Зокрема, можна класи тлумачити як множини певних станів. У цьому випадку можна розглядати формули вигляду $<S, St> \mapsto St'$ (або $St \xrightarrow{S} St'$), де St, St' – деякі класи (множини) станів. Однак таке тлумачення класів не зовсім адекватне природі мов програмування, для яких характернішим є функціональний спосіб подання класів за допомогою предикатів. Тому розглядатимемо формули вигляду $<S, P> \mapsto P'$ (або $P \xrightarrow{S} P'$), де P, P' – деякі предикати на множині станів. Такі формули будемо подавати у вигляді $\{P\}\ S\ \{P'\}$ і називати *програмними асерціями*, або *трійками Хоара*. Предикат P називається *передумовою*, а P' – *післяумовою*. Надалі будемо вживати термін *асерція* замість терміна *програмна асерція*.

Розглянутий метод подання семантики програм за допомогою асерцій був використаний Робертом Флойдом для верифікації програм. Пізніше його метод був удосконалений Тоні Хоаром, акий запропонував числення асерцій. Отримана логіка називається *логікою Хоара*, або *логікою Флойда – Хоара*. Отже, далі розглядатимемо аксіоматичну семантику програм як логіку Флойда – Хоара.

Семантичне тлумачення *істинності* асерції $\{P\}\ S\ \{P'\}$ таке: для довільного стану st, для якого передумова P є істинною, а програма S на цьому стані завершується у стані st', післяумова P' має бути істинною на st'. Наведене визначення фактично задає певну властивість програми.

Перейдемо до опису правил аксіоматичної семантики для мови *SIPL*.

Вираз $P[x \mapsto a]$ означає синтаксичну підстановку в предикат P замість змінної x виразу a. Наприклад, якщо P має вигляд $gcd(m, n) = gcd(M, N)$, то в результаті виконання $P[M \mapsto M - N]$ отримуємо предикат $gcd(m, n) = gcd(M - N, N)$.

Уведене позначення для підстановки дозволяє записати таке правило (аксіому) для оператора присвоювання:

$$\{P[x\mapsto a]\}\ x:=a\ \{P\}.$$

Дійсно, семантика оператора присвоювання $x:=a$ полягає у формуванні нового стану, в якому значення змінної x дорівнює значенню виразу a на початковому стані. Тому, якщо на новому стані післяумова P є істинною, то на початковому стані має бути істинною така передумова, яка при заміні x на a буде істинною на новому стані, тобто буде післяумовою P.

Наприклад, якщо ви поклали на верхню полицю шафи светр і тепер на ній лежать чотири светри, то до цієї дії на полиці було три светри. Дійсно, якщо позначити верхню полицю як x, дію додавання светра на полицю – як $x:=x+1$, післяумову $x=4$ – як P, то можна записати асерцію $\{P[x\mapsto x+1]\}x:=x+1\{P\}$, яка є $\{(x=4)[x\mapsto x+1]\}\ x:=x+1\ \{x=4\}$. Зробивши підстановку, отримуємо $\{(x+1=4)\}\ x:=x+1\{x=4\}$. Це дорівнює асерції $\{(x=3)\}\ x:=x+1\{x=4\}$, у якій передумовою є $x=3$.

Зауважимо, що тут побудова цієї аксіоми (асерції) здійснюється зворотним методом – від післяумови до передумови.

Для оператора *skip* очевидним є таке правило (аксіома):

$$\{P\}\ skip\ \{P\}.$$

Достатньо очевидними є правила для оператора послідовного виконання й умовного оператора:

$$\frac{\{P\}\ S1\ \{Q\},\{Q\}\ S2\ \{R\}}{\{P\}\ S1;S2\ \{R\}},$$

$$\frac{\{b\wedge P\}\ S1\ \{Q\},\{\neg b\wedge P\}\ S2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\{Q\}}.$$

Складнішим є правило для оператора циклу:

$$\frac{\{b\wedge P\}S\{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{\neg b\wedge P\}}.$$

Це правило ґрунтується на понятті інваріанта циклу. *Інваріантом циклу* називається предикат (формула), який є істинним як до виконання тіла циклу, так і після його завершення. Тому ін-

варіантом для циклу **while** b **do** S буде така формула P, для якої асерція $\{b \wedge P\}\ S\ \{P\}$ є істинною.

Слід брати до уваги, що може бути багато інваріантів для одного циклу. Побудова інваріантів є складною задачею, яку тут досліджувати не будемо.

Дотепер ми розглядали лише правила, які описували властивості операторів (композицій) програм. Однак потрібне ще одне правило, яке дозволяє змінювати перед- і післяумови. Це правило називається *правилом наслідку* (consequence) і має такий вигляд:

$$\frac{\{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}},\ \text{якщо}\ P \Rightarrow P',\ Q' \Rightarrow Q.$$

Як бачимо, це правило посилює передумову висновку P відносно передумови засновку P' (оскільки $P \Rightarrow P'$) і послаблює післяумову висновку Q відносно післяумови засновку Q' (оскільки $Q' \Rightarrow Q$). Тому правило наслідку інколи розбивають на два правила: *посилення передумови* та *послаблення післяумови*.

Необхідно також додати правило для операторних дужок **begin-end**:

$$\frac{\{P\}\ S\ \{Q\}}{\{P\}\ \textbf{begin}\ S\ \textbf{end}\ \{Q\}}.$$

Для мови *SIPL* правила аксіоматичної семантики подані в табл. 6.3.

***Таблиця* 6.3**

Правило виведення	**Позначення правила**
$\{P[x \mapsto a]\}\ x{:=}a\ \{P\}$	*AS*
$\{P\}\ skip\ \{P\}$	*skip*
$\dfrac{\{P\}\ S1\ \{Q\}, \{Q\}\ S2\ \{R\}}{\{P\}\ S1; S2\ \{R\}}$	*S*
$\dfrac{\{b \wedge P\}\ S1\ \{Q\}, \{\neg b \wedge P\}\ S2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$	*IF*

$\dfrac{\{b \wedge P\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{\neg b \wedge P\}}$	*WH*
$\dfrac{\{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}}$, якщо $P \Rightarrow P'$, $Q' \Rightarrow Q$	*C*
$\dfrac{\{P\}\ S\ \{Q\}}{\{P\}\ \textbf{begin}\ S\ \textbf{end}\ \{Q\}}$	*BE*

Зауважимо, що ці правила коректні лише для всюди визначених (тотальних) предикатів, які задають перед- та післяумови. У випадку часткових предикатів слід уживати складніші правила.

Продемонструємо застосування логіки Флойда – Хоара (аксіоматичної семантики) для верифікації програм. Для правила наслідку умови $P \Rightarrow P'$, $Q' \Rightarrow Q$ у дереві виведення вказувати не будемо – вони зрозумілі із запису правила, а їх доведення буде вказано поза деревом виведення.

Приклад 6.8. Довести в логіці Флойда–Хоара часткову коректність програми *GCD* обчислення найбільшого спільного дільника за алгоритмом Евкліда.

Розв'язання. Текст програми *GCD* наведено в прикл. 1.1:

GCD ≡

begin

while ¬ $M = N$ **do**

if $M > N$ **then** $M := M - N$ **else** $N := N - M$

end

Спочатку сформулюємо асерцію, яка задає коректність програми *GCD*. Для цього введемо бінарну функцію *gcd*: $Int^2 \to Int$, яка задає найбільший спільний дільник двох її аргументів. Тому запис *gcd* (*m*, *n*) задає найбільший спільний дільник чисел *m* та *n*. Вважатимемо, що початковий стан програми має такий вигляд: [$M \mapsto m$, $N \mapsto n$]. Це означає, що ми розглядаємо стан зі змінними *M* та *N* і з параметрами *m* та *n*. Наше припущення можна записати у вигляді предиката $M > 0 \wedge N > 0 \wedge M = m \wedge N = n$, який і буде передумовою потрібної асерції. Тоді післяумову можна записати у вигляді $M = N \wedge M = gcd(m,n)$, і вона стверджує, що після завершення програми значення змінних *M* та *N* рівні та дорівнюють значенню *gcd* (*m*, *n*).

Отже, потрібно довести таку асерцію, яку запишемо у трьох рядках:

$$\{M > 0 \wedge N > 0 \wedge M = m \wedge N = n\}$$
$$GCD$$
$$\{M = N \wedge M = gcd(m, n)\}.$$

Ідея побудови доведення полягає в пошуку інваріанта циклу. Попередній розгляд програми *GCD* у розд. 1 свідчить про те, що віднімання меншого числа від більшого не змінює найбільшого спільного дільника. Тому за інваріант можна взяти предикат $gcd(m, n) = gcd(M, N)$, який позначимо P; він буде інваріантом циклу.

Уведене позначення P дозволяє записати таку асерцію для оператора присвоювання $M := M - N$:

$$\{P[M \mapsto \quad]\}\ M := M - N\ \{P\}.$$

Оскільки $M > N \wedge P[M \mapsto \quad \mapsto \quad$, то за правилом наслідку отримуємо таку асерцію:

$$\{M > N \wedge P[M \mapsto \quad]\}\ M := M - N\ \{P\}.$$

Тепер доведемо, що $M > N \wedge P$ імплікує $M > N \wedge P[M \mapsto \quad]$. Розкриваючи P, розуміємо, що треба довести таку імплікацію:

$$M > N \wedge gcd(m,n) = gcd(M,N)[M \mapsto \quad] \Rightarrow$$
$$\Rightarrow gcd(m,n) = gcd(M,N).$$

Таке доведення було виконано в розд. 1.

Застосовуючи правило наслідку, отримуємо асерцію $\{M > N \wedge P\}\ M := M - N\ \{P\}$.

Проведені виведення можна подати таким деревом (позначення аксіоми *AS* пишемо в рядку висновку):

$$C: \frac{AS: \{P[M \mapsto \quad]\}\ M := M - N\ \{P\}}{C: \dfrac{\{M > N \wedge P[M \mapsto \quad]\}\ M := M - N\ \{P\}}{\{M > N \wedge P\}\ M := M - N\ \{P\}}}$$

Аналогічно отримуємо дерево виведення і для другого оператора присвоювання:

$$C:\frac{AS:\{P[N\mapsto \quad]\}\ N:=N-M\ \{P\}}{C:\dfrac{\{\neg(M>N)\wedge P[N\mapsto \quad]\}\ N:=N-M\ \{P\}}{\{\neg(M>N)\wedge P\}\ N:=N-M\ \{P\}}}$$

Висновки двох проведених виведень, а саме асерції $\{M>N\wedge P\}\ M:=M-N\ \{P\}$ та $\{\neg(M>N)\wedge P\}\ N:=N-M\ \{P\}$, є засновками правила *IF*. Застосовуючи це правило, отримуємо асерцію

$$\{P\}\ \textbf{if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M\ \{P\}.$$

Щоб застосувати правило для циклу, посилимо передумову до $\neg(M=N)\wedge P$ і, застосувавши правило наслідку, виведемо таку асерцію:

$$\{\neg(M=N)\wedge P\}$$
$$\textbf{if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M$$
$$\{P\}$$

Виведена асерція доводить, що P є інваріантом циклу. Крім цього, така асерція є засновком правила для оператора циклу. Застосувавши це правило, маємо:

$$\{P\}$$
$$\textbf{while}\ \neg(M=N)\ \textbf{do if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M$$
$$\{\neg\neg M=N\wedge P\}$$

Очевидно, що $\neg\neg M=N\wedge P$ дорівнює $M=N\wedge P$.

Тепер посилимо передумову P до

$$M>0\wedge N>0\wedge M=m\wedge N=n\wedge P.$$

Залишилось ще раз посилити отриману передумову до

$$\{M>0\wedge N>0\wedge M=m\wedge N=n\}.$$

Це очевидно, оскільки при $M=m$ та $N=n$ маємо, що

$$gcd(m,n)=gcd(M,N),$$

тобто

$$M>0\wedge N>0\wedge M=m\wedge N=n\Rightarrow$$
$$\Rightarrow M>0\wedge N>0\wedge M=m\wedge N=n\wedge P.$$

Зауважимо, що ці умови еквівалентні.

Скориставшись правилом наслідку, посилимо передумову до $\{M > 0 \wedge N > 0 \wedge M = m \wedge N = n\}$ і послабимо післяумову до $M = N \wedge M = gcd(m,n)$. Отримаємо

$\{M > 0 \wedge N > 0 \wedge M = m \wedge N = n\}$

while $\neg(M = N)$ **do if** $M > N$ **then** $M := M - N$ **else** $N := N - M$

$\{M = N \wedge M = gcd(m,n)\}$.

Застосувавши правило *BE*, отримуємо асерцію, яку необхідно було вивести:

$$\{M > 0 \wedge N > 0 \wedge M = m \wedge N = n\}$$

$$GCD$$

$$\{M = N \wedge gcd(M,N) = gcd(m,n)\}$$

Подамо проведені виведення у вигляді дерева:

$$C:\dfrac{AS:\{P[M\mapsto \quad]\}M:=M-N\{P\}}{C:\dfrac{\{M>N\wedge P[M\mapsto \quad]\}M:=M-N\{P\}}{\{M>N\wedge P\}M:=M-N\{P\}}}$$

$$IF:\dfrac{\begin{array}{|l}C:\dfrac{AS:\{P[N\mapsto \quad]\}N:=N-M\{P\}}{C:\dfrac{\{\neg(M>N)\wedge P[N\mapsto \quad]\}N:=N-M\{P\}}{\{\neg(M>N)\wedge P\}N:=N-M\{P\}}}\end{array}}{C:\dfrac{\{P\}\ \textbf{if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M\ \{P\}}{\{\neg(M=N)\wedge P\}}}$$

$$\textbf{if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M$$

$$WH:\dfrac{\{P\}}{\{P\}}$$

$$\textbf{while}\ \neg(M=N)\ \textbf{do}$$

$$\textbf{if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M$$

$$C:\dfrac{\{\neg\neg M=N\wedge P\}}{\{P\}}$$

$$\textbf{while}\ \neg(M=N)\ \textbf{do}$$

$$\textbf{if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M$$

$$C:\dfrac{\{M=N\wedge P\}}{\{M>0\wedge N>0\wedge M=m\wedge N=n\wedge P\}}$$

$$\textbf{while}\ \neg(M=N)\ \textbf{do}$$

$$\textbf{if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M$$

$$C:\dfrac{\{M=N\wedge P\}}{\{M>0\wedge N>0\wedge M=m\wedge N=n\}}$$

$$\textbf{while}\ \neg(M=N)\ \textbf{do}$$

$$\textbf{if}\ M>N\ \textbf{then}\ M:=M-N\ \textbf{else}\ N:=N-M$$

$$BE:\dfrac{\{M=N\wedge M=gcd(m,n)\}}{\{M>0\wedge N>0\wedge M=m\wedge N=n\}}$$

$$GCD$$

$$\{M=N\wedge M=gcd(m,n)\}$$

Для доведення повної коректності треба ускладнити правила логіки, які б ураховували завершуваність програм та операторів. Тут деталі наводити не будемо.

Приклад 6.9. Довести в логіці Флойда – Хоара часткову коректність програми *EXP* обчислення x^n.

Розв'язання. У програмі *EXP* обчислення x^n (розд. 1) використовуються змінні X та N для позначення відповідно значень x та n, а також змінна R для повернення результату r:

```
EXP ≡
begin
R := 1;
while N > 0 do
   begin
      R :=  R *X;
      N :=  N – 1
   end
end
```

Спочатку сформулюємо асерцію, яка задає коректність програми *EXP*.

Передумовою є предикат $N \geq 0 \wedge X = x \wedge N = n$, післяумовою – $R = x^n$. Отже, треба побудувати виведення асерції:

$$\{N \geq 0 \wedge X = x \wedge N = n\}\ EXP\ \{R = x^n\}.$$

Доведемо, що предикат $R * X^N = x^n$ є інваріантом циклу.

Використаємо зворотний метод для побудови передумов. Спочатку будуємо асерцію для оператора присвоювання $N := N - 1$:

$$\{(R * X^N = x^n)[N \mapsto \qquad N := N - 1\ \{R * X^N = x^n\}.$$

Обчислюємо $(R * X^N = x^n)[N \mapsto$. Отримуємо передумову $R * X^{N-1} = x^n$.

Тепер розглянемо цю передумову як післяумову оператора присвоювання $R := R * X$. Дістаємо асерцію

$$\{(R * X^{N-1} = x^n)[R \mapsto \qquad \}\ R := R * X\ \{R * X^{N-1} = x^n\}.$$

Обчислюємо нову передумову $(R * X^{N-1} = x^n)[R \mapsto$. Одержуємо предикат $(R * X) * X^{N-1} = x^n$, який дорівнює $R * X^N = x^n$.

Застосувавши правило для оператора послідовного виконання, отримуємо асерцію

$$\{R * X^N = x^n\} R := R * X; N := N - 1 \{R * X^N = x^n\}.$$

Щоб застосувати правило для циклу, посилимо передумову до $N > 0 \wedge R * X^N = x^n$, і за правилом наслідку виводимо таку асерцію:

$$\{N > 0 \wedge R * X^N = x^n\} R := R * X; N := N - 1 \{R * X^N = x^n\}.$$

За правилом *BE* отримуємо

$$\{N > 0 \wedge R * X^N = x^n\} \textbf{begin } R := R * X; N := N - 1 \textbf{ end } \{R * X^N = x^n\}.$$

Звідси випливає, що предикат $R * X^N = x^n$ дійсно є інваріантом циклу.

Застосувавши правило циклу, маємо

$$\{R * X^N = x^n\}$$
$$\textbf{while } N > 0 \textbf{ do begin } R := R * X; N := N - 1 \textbf{ end}$$
$$\{\neg(N > 0) \wedge R * X^N = x^n\}.$$

Тепер залишилось вивести формулу

$$\neg(N > 0) \wedge R * X^N = x^n \Rightarrow R = x^n.$$

Ця формула еквівалентна такій формулі

$$N \le 0 \wedge R * X^N = x^n \Rightarrow R = x^n.$$

Але ця формула не є істинною. Наприклад, при значеннях N рівному –1, n рівному 1, X рівному x, R рівному X^2, формула є хибною. (Зауважимо, що значення X^{-1} може бути невизначеним, але ми обмежились випадками, коли всі предикати є всюди визначеними.)

Така ситуація виникає досить часто при верифікації програм. Причина полягає в тому, що обраний інваріант $R * X^N = x^n$ є занадто слабким. Треба його посилити умовою $N \ge 0$, яка буде гарантувати, що цикл завершиться зі значенням N, яке буде дорівнювати 0.

Тому розглянемо новий предикат

$$N \geq 0 \wedge R * X^N = x^n$$

та перевіримо, чи буде він інваріантом циклу.

Будуємо асерцію для оператора присвоювання $N := N - 1$:

$\{(N \geq 0 \wedge R * X^N = x^n)[N \mapsto \quad N := N - 1\{N \geq 0 \wedge R * X^N = x^n\}.$

Обчислюємо $(N \geq 0 \wedge R * X^N = x^n)[N \mapsto$. Отримуємо

$$N - 1 \geq 0 \wedge R * X^{N-1} = x^n.$$

Тепер розглядаємо цю передумову як післяумову оператора присвоювання $R := R * X$. Отримуємо асерцію

$$\{(N - 1 \geq 0 \wedge R * X^{N-1} = x^n)[R \mapsto \quad \}$$
$$R := R * X \quad .$$
$$\{N - 1 \geq 0 \wedge R * X^{N-1} = x^n\}.$$

Обчислюємо нову передумову

$$(N - 1 \geq 0 \wedge R * X^{N-1} = x^n)[R \mapsto \quad .$$

Отримуємо предикат $N - 1 \geq 0 \wedge R * X * X^{N-1} = x^n$, який дорівнює $N \geq 1 \wedge R * X^N = x^n$.

Застосувавши правило для оператора послідовного виконання, отримуємо асерцію

$$\{N \geq 1 \wedge R * X^N = x^n\} R := R * X; N := N - 1 \{N \geq 0 \wedge R * X^N = x^n\}.$$

Тепер посилимо передумову предикатом $N \geq 0$:

$$\{N \geq 1 \wedge N \geq 0 \wedge R * X^N = x^n\}$$
$$R := R * X; N := N - 1$$
$$\{N \geq 0 \wedge R * X^N = x^n\}.$$

Застосовуємо правило для операторних дужок:

$$\{N \geq 1 \wedge N \geq 0 \wedge R * X^N = x^n\}$$
$$\textbf{begin } R := R * X; N := N - 1 \textbf{ end}$$
$$\{N \geq 0 \wedge R * X^N = x^n\}.$$

Отримана асерція підтверджує, що $N \geq 0 \wedge R * X^N = x^n$ є інваріантом циклу. Застосовуємо відповідне правило, враховуючи, що $N \geq 1$ еквівалентно формулі $N > 0$:

$\{N \geq 0 \wedge R * X^N = x^n\}$

while N>0 **do begin** $R := R * X; N := N - 1$ **end**

$\{\neg N>0 \wedge N \geq 0 \wedge R * X^N = x^n\}$.

Спрощуючи післяумову отримуємо $N = 0 \wedge R = x^n$, що дає наступну асерцію:

$\{N \geq 0 \wedge R * X^N = x^n\}$

while N>0 **do begin** $R := R * X; N := N - 1$ **end**

$\{N = 0 \wedge R = x^n\}$.

Розглядаємо $\{N \geq 0 \wedge R * X^N = x^n\}$ як післяумову оператора присвоювання $R := 1$:

$\{(N \geq 0 \wedge R * X^N = x^n)[R \mapsto \quad : 1\} \{N \geq 0 \wedge R * X^N = x^n\}$.

Зробивши підстановку отримуємо

$\{N \geq 0 \wedge X^N = x^n\}\ R := 1\ \{N \geq 0 \wedge R * X^N = x^n\}$.

Пригадаємо передумову програми: $N \geq 0 \wedge X = x \wedge N = n$. Цей предикат сильніший за передумову $N \geq 0 \wedge X^N = x^n$, тобто формула $N \geq 0 \wedge X = x \wedge N = n \Rightarrow N \geq 0 \wedge X^N = x^n$ є істинною. Це означає, що

$\{N \geq 0 \wedge X = x \wedge N = n\}\ R := 1\ \{N \geq 0 \wedge R * X^N = x^n\}$.

Тепер застосовуємо правило для послідовного оператора:

$\{N \geq 0 \wedge X = x \wedge N = n\}$

$R := 1$; **while** N>0 **do begin** $R := R * X; N := N - 1$ **end**

$\{N = 0 \wedge R = x^n\}$.

Далі застосовуємо правило *BE*:

$\{N \geq 0 \wedge X = x \wedge N = n\}\ EXP\ \{N = 0 \wedge R = x^n\}$.

На останньому кроці виведення залишилось застосувати правило послаблення післяумови, щоб отримати асерцію, яку слід було вивести для доведення часткової коректності програми *EXP*:

$$\{N \geq 0 \wedge X = x \wedge N = n\}\ EXP\ \{R = x^n\}.$$

Подамо проведені виведення у вигляді дерева. Оскільки таке дерево буде занадто великим, щоб поміститись на одній сторінці, подамо його двома піддеревами. Спочатку побудуємо дерево (*TW*) виведення асерції для циклу:

(*TW*):

$$S: \frac{\begin{array}{l} AS: \{N-1 \geq 0 \wedge R * X^{N-1} = x^n\}\, N := N-1 \{N \geq 0 \wedge R * X^N = x^n\} \\ \left| AS: \{N \geq 1 \wedge R * X^N = x^n\}\, R := R * X \{N-1 \geq 0 \wedge R * X^{N-1} = x^n\} \right. \end{array}}{\begin{array}{l} \{N \geq 1 \wedge R * X^N = x^n\} \\ R := R * X; N := N-1 \end{array}}$$

$$C: \frac{\begin{array}{l} \{N \geq 0 \wedge R * X^N = x^n\} \end{array}}{\begin{array}{l} \{N \geq 1 \wedge N \geq 0 \wedge R * X^N = x^n\} \\ R := R * X; N := N-1 \end{array}}$$

$$BE: \frac{\begin{array}{l} \{N \geq 0 \wedge R * X^N = x^n\} \end{array}}{\begin{array}{l} \{N \geq 1 \wedge N \geq 0 \wedge R * X^N = x^n\} \\ \textbf{begin}\ R := R * X; N := N-1\ \textbf{end} \end{array}}$$

$$WH: \frac{\begin{array}{l} \{N \geq 0 \wedge R * X^N = x^n\} \end{array}}{\begin{array}{l} \{N \geq 0 \wedge R * X^N = x^n\} \\ \textbf{while}\ N{>}0\ \textbf{do begin}\ R := R * X; N := N-1\ \textbf{end} \end{array}}$$

$$C: \frac{\begin{array}{l} \{\neg N{>}0 \wedge N \geq 0 \wedge R * X^N = x^n\} \end{array}}{\begin{array}{l} \{N \geq 0 \wedge R * X^N = x^n\} \\ \textbf{while}\ N{>}0\ \textbf{do begin}\ R := R * X; N := N-1\ \textbf{end} \end{array}}$$

$$\{N = 0 \wedge R = x^n\}.$$

Тепер використовуємо побудоване дерево (*TW*) як другий засновок правила для послідовного виконання операторів присво-

єння та циклу. Далі продовжуємо побудову дерева виведення до отримання бажаної асерції:

$$C:\frac{\{(N\geq 0\wedge R*X^N=x^n)[R\mapsto \qquad 1\{N\geq 0\wedge R*X^N=x^n\}}{C:\dfrac{\{N\geq 0\wedge X^N=x^n\}R:=1\{N\geq 0\wedge R*X^N=x^n\}}{\{N\geq 0\wedge X=x\wedge N=n\}R:=1\{N\geq 0\wedge R*X^N=x^n\}}}$$

$$S:\frac{\big|\,(TW)}{\{N\geq 0\wedge X=x\wedge N=n\}}$$

$$R:=1;\textbf{while } N>0 \textbf{ do begin } R:=R*X;N:=N-1 \textbf{ end}$$

$$BE:\frac{\{N=0\wedge R=x^n\}}{\{N\geq 0\wedge X=x\wedge N=n\}\ EXP\ \{N=0\wedge R=x^n\}}$$

Зауважимо, що правила аксіоматичної семантики застосовуються безпосередньо до програм та складових мови *SIPL*, а не до їх семантичних термів. Це дозволяє використовувати класичну логіку першого порядку для запису перед- та післяумов. У випадку більш складних програм доцільно застосовувати правила до семантичних термів та використовувати логіки, орієнтовані на таки терми – композиційно-номінативні логіки [11, 12].

Додаткові питання з семантики розглянуто в [19, 21, 25].

Висновки

У цьому розділі було розглянуто традиційні методи подання семантики програм. До таких методів зазвичай відносять денотаційну, операційну й аксіоматичну семантики. У денотаційній семантиці денотат (смислове значення) програми тлумачиться як функція, що переводить вхідні дані у вихідні. Денотат складної програми подається як композиція денотатів її складових (принцип композиційності). В операційній семантиці програма також тлумачиться як функція, але ця функція подається в інший спосіб, а саме як транзиційне відношення на множині даних. Таке відношення задається множиною індивідних формул (тестів) вигляду $\langle S, st\rangle \mapsto st'$, де S – оператор (програма), st та st' – стани. В аксіоматичній семантиці семантика програми пода-

ється формулами загальнішого вигляду, ніж в операційній семантиці, а саме формулами вигляду $\{P\}\ S\ \{P'\}$ (програмними асерціями, трійками Хоара), де S – оператор (програма), P, P' – предикати на множині станів. Трійки Хоара визначають властивості програм. Для однієї програми може бути багато істинних трійок Хоара. Денотаційна й операційна семантики мови *SIPL* еквівалентні, бо вони описують один і той же клас функцій, індукований програмами мови *SIPL*. Що ж стосується аксіоматичної семантики, то її зв'язок із денотаційною й операційними семантиками складніший, оскільки аксіоматична семантика напряму не задає клас функцій, індукованих програми, а визначає їх властивості.

Контрольні запитання

1. Що таке формальна семантика програми?
2. Які методи подання семантики програми вважаються традиційними?
3. Які основні принципи денотаційної семантики?
4. Який вигляд мають правила денотаційної семантики?
5. Як пов'язані композиційна та денотаційна семантики програм мови *SIPL*?
6. Які основні принципи операційної семантики?
7. Який вигляд мають правила операційної семантики?
8. Як пов'язані композиційна та операційна семантики програм мови *SIPL*?
9. Які основні принципи аксіоматичної семантики?
10. Який вигляд має програмна асерція (трійка Хоара)?
11. Яке семантичне тлумачення терміна "асерція" ("трійка Хоара")?
12. Який вигляд мають правила аксіоматичної семантики (логіки Флойда – Хоара)?
13. Як пов'язані композиційна й аксіоматична семантики програм мови *SIPL*?

Вправи

1. Написати *SIPL*-програму знаходження *n*-го числа Фібоначчі, побудувати її терм у денотаційній семантиці й обчислити його значення на стані $st = [N \mapsto 5]$.
2. Написати *SIPL*-програму знаходження *n*-го числа Фібоначчі та побудувати дерево її обчислення в операційній семантиці на стані $st = [N \mapsto 5]$ прямим і зворотним методом.
3. Написати *SIPL*-програму знаходження *n*-го числа Фібоначчі й довести її часткову і повну коректність у логіці Флойда – Хоара (аксіоматичній семантиці).

7. Формалізація мов програмування та мов специфікацій програм

У першому розділі було розглянуто просту мову програмування *SIPL*. Ця мова є дійсно простою, оскільки в ній явно визначено тільки один простий тип даних – цілі числа, простий набір базових функцій і композицій. Сучасні мови програмування та специфікацій використовують розвинену систему типів даних, чималу кількість базових функцій і композицій, рекурсію, об'єктно-орієнтовані конструкції, паралелізм, квантифікацію тощо. У цьому розділі ми покажемо, як формалізувати складніші мови. Зокрема, ми продемонструємо, як формалізувати типи даних і ввести нові функції та композиції.

7.1. Типи даних

Найпростіше тлумачення типу даних полягає в тому, що це – певна множина значень. Однак таке тлумачення занадто просте, оскільки необхідно вказати ще операції над значеннями. Це означає, що тип даних слід тлумачити як певну *алгебру*.

Наприклад, для мови *SIPL* у першому розділі ми визначили таку алгебру цілих чисел:

$$A\_Int = <Int; add, sub, mult>.$$

Утім, цієї алгебри недостатньо, оскільки ще треба ввести алгебру булевих значень

$$A\_Bool = <Bool; or, neg>.$$

Оскільки формалізація мови вимагає введення операцій порівняння, то отримаємо *двоосновну алгебру базових даних*:

$$A\_Int\_Bool = <Int, Bool; add, sub, mult, or, neg, eq, gr>.$$

Робимо висновок, що типи даних можна уточнювати через поняття багатоосновної (багатосортної) алгебри.

Точне визначення є таким.

Нехай $A_1, A_2, \ldots, A_m$ – деякі множини (*носії алгебри*), а $f_1, f_2, \ldots, f_k$ – *скінченноарні операції*. Це означає, що кожна операція f_i задана на декартовому добутку певних множин $A_{i1} \times A_{i2} \times \ldots \times A_{in}$, а її результат належить деякій множині A_{i0}. У такому випадку кажуть, що операція f_i має тип $A_{i1} \times A_{i2} \times \ldots \times A_{in} \to A_{i0}$, що позначають f_i: $A_{i1} \times A_{i2} \times \ldots \times A_{in} \to A_{i0}$.

Наприклад, для алгебри *A_Int_Bool* її носіями є множини *Int* та *Bool*, а операції мають такі типи:

- *add*, *sub*, *mult*: $Int \times Int \to Int$,
- *or*: $Bool \times Bool \to Bool$,
- *neg*: $Bool \to Bool$,
- *eq*, *gr*: $Int \times Int \to Bool$.

Зауважимо, що наведене визначення алгебри *A_Int_Bool* насправді не є повним, оскільки ми не дали формального визначення носіїв та операцій. Наше знання про цю алгебру ґрунтується на попередньому досвіді, і коли виникне потреба довести якісь її властивості, ми відчуємо брак формалізації. Пояснимо це детальніше.

Яким чином можна формалізувати носії? Для цілих чисел можна використовувати різні системи числення: позиційні, непозиційні, змішані, факторіальні, Фібоначчі тощо. Наприклад, цілі числа можна подавати в десятковій, двійковій системах або просто послідовністю одиниць. Наприклад, число 5 (десяткова

система) у двійковій системі буде 101, а як послідовність одиниць це буде 11111. Це означає, що можна говорити про множину Int_{10} цілих чисел, заданих у десятковій системі, множину Int_2 цілих чисел, заданих у двійковій системі, множину Int_1 цілих чисел, заданих послідовністю одиниць тощо.

Відповідно для кожного способу подання цілих чисел будуть різні визначення (зазвичай алгоритмічні) операцій над цілими числами. Наприклад, для визначення додавання в позиційних системах описують алгоритм add_{col} додавання у стовпчик, а для подання додавання послідовністю одиниць – приписування (конкатенацію) чисел add_{con}. Це означає, що насправді ми матимемо багато різних алгебр, які визначають цілі числа й операції з ними та можуть розглядатись як реалізації цього типу.

Для аналізу, перетворення та верифікації програм використовуватимемо *властивості операцій*, заданих на різних типах.

Підсумовуючи, можемо сказати, що тип даних визначається такими складовими:

- ім'ям типу;
- набором операцій, які можна застосовувати до елементів даного типу;
- співвідношеннями між операціями (властивостями операцій типу);
- реалізацією – конкретним поданням елементів та операцій цього типу (алгеброю типу).

Оскільки реалізацій (алгебр) може бути багато, то це зумовлює наявність їхніх різних властивостей, наприклад кількість символів у поданні числа 5 буде різною в різних алгебрах. Звідси випливає проблема побудови інваріантної (незалежної від подання) теорії цілих чисел. Це насправді дуже непроста проблема (згадаймо теорему Гьоделя про неповноту формальної арифметики). Не вдаючись у проблематику, зазначимо, що, увівши поняття алгебр, ми перейшли до визначення їхніх властивостей, тобто ми йшли *від алгебр до їхніх властивостей*. Це наштовхує на думку, що можливий і зворотний шлях – *від власти-*

востей до алгебр. У такому випадку властивості стають абстрактним засобом визначення алгебр.

Цей шлях має певні переваги:

- ми отримуємо інваріантні властивості, істинні в різних алгебрах, які коректно реалізують тип;
- маючи різні алгебри, можна обирати ті, що матимуть кращу ефективність для розв'язання певної задачі.

З наведених міркувань випливає поняття *абстрактних типів даних (АТД)*.

Ідея абстрактних типів даних полягає в тому, щоб відокремити семантичні (поведінкові) властивості типу даних від їх реалізації. У цьому випадку програміст працює з даними лише через визначений інтерфейс і не має доступу до внутрішнього подання даних. В АТД властивості типу задаються як певні аксіоми, а інші властивості є похідними від аксіом.

Поняття АТД ґрунтується на понятті Σ-алгебри. Нехай S – деяка множина, елементи якої називаються *сортами* (*іменами типів*, *символами типів*), Op – множина *імен скінченноарних операцій*, $type_{Op}$: $Op \to (S^* \times S)$ – відображення *типізації*, яке кожному символу операції зіставляє його тип (точніше складне ім'я його типу). Тоді кортеж $\Sigma = (S, Op, type_{Op})$ називається *сигнатурою*. Сигнатура є описом символів (імен), тобто сортів та імен операцій, які вживаються в алгебрі.

Маючи сигнатуру Σ, можна перейти до визначення Σ-алгебри. Нехай T – певний клас множин (клас базових типів, клас основ), I_S: $S \to T$ – відображення інтерпретації сортів, I_{Op}: $Op \to \cup_{n \in Nat} (T^n \to T)$ – відображення інтерпретації операцій, яке узгоджене з відображенням $type_{Op}$ таким чином: якщо $type_{Op}(op) = (s_1 s_2 \ldots s_n, s)$, то $I_{Op}(op) \in I_S(s_1) \times I_S(s_2) \times \ldots \times I_S(s_n) \to I_S(s)$, де $op \in Op$.

Тоді Σ-*алгебра* – це пара $<I_S; I_{Op}>$, яку можна подати в більш традиційному вигляді $<\{ I_S(s) \mid s \in S\}; \{ I_{Op}(op) \mid op \in Op\}>$.

Складні об'єкти в Σ-алгебрі подаються за допомогою *термів* – виразів, побудованих із типізованих змінних і операцій алгебри.

Нехай V – множина предметних змінних (предметних імен), $type_V$: $V \to S$ – відображення типізації змінних.

Терми (традиційний вигляд) визначаються в такий спосіб:

– якщо $v \in V$, $type_V(v) = s$, то v є термом сорту s;

– якщо $op \in Op$, $type_{Op}(op) = (s_1 s_2 \ldots s_n, s)$, t_1 є термом сорту s_1, t_2 є термом сорту s_2,…, t_n є термом сорту s_n, то вираз $op(t_1, t_2, \ldots, t_n)$ є термом сорту s.

Для семантичного запису термів використовуємо суперпозицію в n-арну функцію, тоді терм $op(t_1, t_2, \ldots, t_n)$ записуємо у вигляді $S^n(op, t_1, t_2, \ldots, t_n)$.

Далі можна будувати *формули*. Для цього вводимо сорт *Bool* булевих значень. Символи операцій з типом вигляду $type_{Op}(op)=(s_1 s_2 \ldots s_n, Bool)$ називаються *символами предикатів*.

Формули будуються таким чином:

– якщо $op \in Op$, $type_{Op}(op) = (s_1 s_2 \ldots s_n, Bool)$, t_1 є термом сорту s_1, t_2 є термом сорту s_2,…, t_n є термом сорту s_n, то вираз $op(t_1, t_2, \ldots, t_n)$ є формулою;

– якщо Φ, Ψ є формулами, $v \in V$, t_1 та t_2 є термами одного сорту, то $\Phi \vee \Psi$, $\neg\Phi$, $\exists v\, \Phi$, $t_1 = t_2$ є формулами.

За заданих відображень типізації змінних та інтерпретацій сортів і символів операцій можна побудувати інтерпретації термів і формул, які кожному терму зіставляють функцію, а кожній формулі – предикат у Σ-алгебрі.

Уведені поняття Σ-алгебр, термів і формул та їхніх інтерпретацій є доволі громіздкими, тому не будемо виписувати всі деталі, а розглянемо кілька прикладів.

Приклад 7.1 (множина цілих чисел і булевих значень як АТД). Спочатку визначимо сигнатуру алгебри – сорти: *INT* та *BOOL*; символи операцій: +, –, ×, ∨, ¬, =, >; відображення типізації задається таким чином:

- для операцій +, –, × типізацією є (*INT INT*, *INT*);
- для операції ∨ типізацією є (*BOOL BOOL*, *BOOL*);
- для операції ¬ типізацією є (*BOOL*, *BOOL*);
- для операцій =, > типізацією є (*INT INT*, *BOOL*).

Властивості операцій задаються аксіомами, які зазвичай є тотожностями. Для цієї алгебри (алгебраїчної системи) випишемо

лише деякі тотожності (x, y, z мають типізацію *INT*; a, b, c мають типізацію *BOOL*):

– $x + (y + z) = (x + y) + z$;
– $x \times (y \times z) = (x \times y) \times z$;
– $x + y = y + x$;
– $x \times y = y \times x$;
– $(x + y) \times z = x \times z + y \times z$;
– $(x - y) = z \leftrightarrow x = z + y$;
– $a \vee (b \vee c) = (a \vee b) \vee c$;
– $a \vee b = b \vee c$;
– …

Вважається, що аксіоми універсально квантифіковані, тобто мають виконуватись для всіх x, y, z та a, b, c.

Для даного прикладу в кожній алгебрі вказаної сигнатури мають виконуватись усі аксіоми.

Прикладом конкретної Σ-алгебри буде алгебра

$$A\_Int\_Bool = <Int, Bool; add, sub, mult, or, neg, eq, gr>.$$

У цій алгебрі сорт *INT* інтерпретується як носій *Int,* скажімо, як множина цілих чисел у десятковій системі числення; сорт *BOOL* інтерпретується як множина $Bool = \{T, F\}$: символи операції +, –, ×, ∨, ¬, =, > інтерпретуються як операції *add*, *sub*, *mult*, *or*, *neg*, *eq*, *gr*, задані на відповідних носіях.

У цій алгебрі виконуються всі аксіоми, описані у відповідному АТД.

Іншим прикладом буде алгебра

$A\_Int_2\_Bool_2 = <In_2t, Bool_2; add_2, sub_2, mult_2, or_2, neg_2, eq_2, gr_2>$,

де Int_2 – множина цілих чисел у двійковій системі числення, $Bool_2 = \{1, 0\}$; add_2, sub_2, $mult_2$, or_2, neg_2, eq, gr_2 – операції, задані на носіях Int_2 та $Bool_2$.

Для цієї алгебри також виконуються всі аксіоми, описані в АТД.

Приклад 7.2 (список як АТД). Спочатку визначимо сигнатуру алгебри списків – сорти: *ELEM*, *LIST* та *BOOL*; символи операцій: *Empty*, *Is_empty*, *Cons*, *Head*, *Tail*, *Conc*, *Equal*; відображення типізації задається таким чином:

- для операції *Empty* (створення порожнього списку, операція нульової арності) типізацією є (*ELEM*);

- для операції *Is_empty* (перевірка, чи є список порожнім) типізацією є (*LIST*, *BOOL*);
- для операції *Cons* (додавання елемента в голову списку) типізацією є (*ELEM LIST*, *LIST*);
- для операції *Head* (отримання першого елемента списку) типізацією є (*LIST*, *ELEM*);
- для операції *Tail* (отримання списку без першого елемента) типізацією є (*LIST*, *LIST*);
- для операції *Conc* (конкатенація двох списків) типізацією є (*LIST LIST*, *LIST*);
- для операції *Equal* (поелементне порівняння списків) типізацією є (*LIST LIST*, *BOOL*).

Аксіоми (тотожності) для списку такі (змінні *E* та *E1* мають тип *ELEM*, змінні *L* та *L1* мають тип *LIST*):

- *Is_empty*(*Empty*) = *T*;
- *Is_empty* (*Cons*(*E*, *L*)) = *F*;
- *Head*(*Cons*(*E*, *L*)) = *E*;
- *Tail*(*Cons*(*E*, *L*)) = *L*;
- *Conc*(*Empty*, *L*) = *L*;
- *Conc*(*Cons*(*E*, *L*), *L1*) = *Cons*(*E*, *Conc* (*L*, *L1*));
- *Equal*(*Empty*, *Empty*) = *T*;
- *Equal*(*Empty*, *Cons*(*E*, *L*)) = *F*;
- *Equal*(*Cons*(*E*, *L*), *Empty*) = *F*;
- *Equal*(*Cons*(*E*, *L*), *Cons*(*E1*, *L1*)) = (*E*=*E1*) $\wedge$ *Equal*(*L*, *L1*).

Приклад 7.3 (одновимірний масив як АТД). Сигнатура складається із сортів *ELEM*, *INDEX* та *ARRAY*; символу операції *Select* із типізацією (*ARRAY INDEX* , *ELEM*) та символу операції *Replace* з типізацією (*ARRAY INDEX ELEM*, *ARRAY*). Операція *Select* вибирає значення елемента масиву за його індексом, операція *Replace* задає нове значення елементу масиву за його індексом.

Аксіоми (тотожності) для одновимірого масиву такі (змінна *A* має тип *ARRAY*, змінна *E* має тип *ELEM*, змінні *i* та *j* мають тип *INDEX*):

- *Select*(*Replace*(*A*, *i*, *E*), *i*) = *E*;

- *Select*(*Replace*(*A*, *i*, *E*), *j*) = *Select*(*A*, *j*), якщо $i \neq j$.

Перша аксіома означає, що при виборі значення за його індексом ми отримаємо останнє оновлене значення; за другою аксіомою оновлення значення за його індексом не змінює значень з іншими індексами.

Приклад 7.4 (двовимірний масив – матриця – як АТД). Сигнатура складається із сортів *ELEM*, *INDEX1*, *INDEX2* та *MATR*; символів операцій *Select* із типізацією (*MATR INDEX1 INDEX2*, *ELEM*) та *Replace* з типізацією (*MATR INDEX1 INDEX2 ELEM*, *MATR*). Операція *Select* вибирає значення елемента матриці за його індексами, операція *Replace* задає нове значення елементу матриці за його індексами.

Аксіоми (тотожності) для матриці такі (змінна *M* має тип *MATR*, змінна *E* має тип *ELEM*, змінні *i* та *j* мають тип *INDEX1*, змінні *k* та *l* мають тип *INDEX2*):

- *Select*(*Replace*(*A*, *i*, *k*, *E*), *i*, *k*) = *E*;
- *Select*(*Replace*(*A*, *i*, *k*, *E*), *j*, *l*) = *Select*(*A*, *j*, *l*),

якщо $i \neq j$ або $k \neq l$.

Перша аксіома означає, що при виборі значення за його індексами ми отримаємо останнє оновлене значення; за другою аксіомою оновлення значення за його індексами не змінює значення компонентів з іншими індексами.

АТД можна ввести в мови програмування (подібно тому, як це зроблено для цілих чисел у мові *SIPL*). Це дозволяє виразити певні конструкції мов програмування. Наприклад, для формалізації операторів введення-виведення ми припускаємо наявність двох списків (файлів) з іменами *input* та *output*. Тоді семантика оператора введення read(*x*) полягає в такому: прочитати перше значення списку *input*, присвоїти його змінній *x* та видалити перше значення зі списку *input*. Семантика оператора write(*a*) – обчислити значення виразу *a* та додати його до файлу *output*.

Наведені приклади демонструють подання деяких типів як АТД. У мовах програмування використовують багато різних структур даних: записи, вказівники, дерева тощо. Виникає природне запитання: чи можна подати різні структури даних як похідні певного універсального типу? Це значно спростило б формалізацію структур даних і доведення їхніх властивостей.

Аналіз різних структур даних свідчить, що основними операціями їхньої обробки є операції взяття й оновлення значень. Ці операції мають своїми параметрами імена тих компонентів, значення яких вибираються або оновлюються. Іншими словами, указані операції ґрунтуються на відношенні *ім'я*$\mapsto$ *значення*, яке називаємо відношенням *номінації* (*іменування*). Ця ідея веде до поняття *номінативних даних*.

7.2. Номінативні дані

Інтуїтивно номінативні дані можна трактувати як ієрархічно побудовані об'єкти, в основі яких лежить бінарне відношення *ім'я*$\mapsto$ *значення*. Це відношення можна тлумачити як певну функцію, тому для номінативних даних будемо використовувати переважно функціональні позначення. Самі дані можна подавати в текстовому (лінійному) або графічному вигляді. Візьмемо, наприклад, такий текстовий запис:

$[A \mapsto\ [1 \mapsto [1 \mapsto 9,\ 2 \mapsto 8,\ 3 \mapsto 7],\ 2 \mapsto [1 \mapsto 6,\ 2 \mapsto 5,\ 3 \mapsto 4]]]$.

Йому відповідає певне орієнтоване дерево, подане на рис. 7.1, з дугами, розміченими іменами, і кінцевими вершинами (листками), розміченими значеннями.

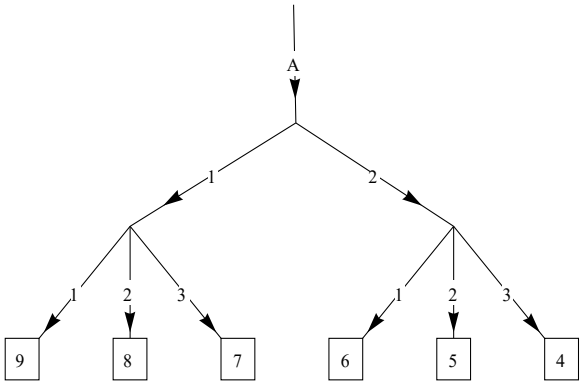

Рис. 7.1. Графічне подання номінативного даного

Математично *номінативні дані* визначаються індуктивно на основі множини імен V та множини базових значень (атомів) W таким чином:

$ND_0(V,W) = W$ – номінативні дані рангу нуль,

$ND_{k+1}(V,W) = W \cup (V \xrightarrow{n} ND_k(V,W))$,

де $k \geq 0$ – номінативні дані рангу не більше $k+1$. Тоді $ND(V,W) = \bigcup$ ') – усі номінативні дані.

Стрілка $\xrightarrow{n}$ означає побудову множини функцій зі скінченним графіком. Такі функції можуть бути багатозначними. До даних рангу 0 належить і порожнє номінативне дане $\varnothing$.

Зауважимо, що введені класи номінативних даних різного рангу утворюють ланцюг, тобто

$$ND_0(V, W) \subseteq ND_1(V, W) \subseteq ND_2(V, W) \subseteq \ldots .$$

Номінативні дані можна також визначити рекурсивним рівнянням

$$ND(V,W) = W \cup (V \xrightarrow{n} ND(V,W)).$$

Конкретизації номінативних даних дозволяють подати традиційні структури даних, такі як записи, файли, масиви, множини, реляції, бази даних тощо. Наприклад, можна вважати, що на рис. 7.1 подано певний масив з описом

var *A*: **array** [1..2] **of array** [1..3] **of** integer.

Список вигляду (a_1, a_2,…,a_m) можна подати в такий спосіб:

[1↦ ↦ ↦ ↦ ↦ ,

де 1 та 2 – певні стандартні імена.

Для подання множини вигляду {a_1, a_2,…,a_m} використовуємо багатозначні функції іменування. Тоді таку множину можна подати номінативним даним вигляду

[1↦ ↦ ↦ ,

де 1 є стандартним іменем.

Оскільки номінативними даними можна подати різні структури даних, які використовуються у програмуванні, то клас номінативних даних ми розглядаємо як *універсальний*, *прагматично повний клас*.

Для обробки номінативних даних зазвичай використовують параметричну унарну операцію *розіменування* $v\Rightarrow$ (*вибору компонента даного*) і параметричну бінарну операцію *оновлення компонента* ∇^v (накладання), які продемонструємо на прикладах.

Нехай

$$d = [x\mapsto [1\mapsto 9,\ 2\mapsto 8,\ 3\mapsto 7],\ y\mapsto [1\mapsto 6,\ 3\mapsto 4]].$$

Тоді

$x\Rightarrow(d) = [1\mapsto 9,\ 2\mapsto 8,\ 3\mapsto 7]$,

$y\Rightarrow(d) = [1\mapsto 6,\ 3\mapsto 4]$,

$v\Rightarrow(d)$ не визначено,

$d\ \nabla^x [1\mapsto 6,\ 2\mapsto 3] = [x\mapsto [1\mapsto 6,\ 2\mapsto 3],\ y\mapsto [1\mapsto 6,\ 3\mapsto 4]]$,

$d\ \nabla^v [1\mapsto 6,\ \ 2\mapsto 3] = [x\mapsto [1\mapsto 9,\ \ 2\mapsto 8,\ \ 3\mapsto 7],\ \ y\mapsto [1\mapsto 6,\ 3\mapsto 4],\ v\mapsto [1\mapsto 6,\ 2\mapsto 3]]$.

Як бачимо, операція розіменування $v\Rightarrow$ вибирає значення імені v з номінативного даного, а якщо такого значення немає, то результат буде невизначений. Операція оновлення з параметром v у номінативному даному (першому аргументі операції) замінює значення імені v на значення, яке задається другим аргументом, а якщо значення v відсутнє в номінативному даному, то другий аргумент з іменем v додається до номінативного даного.

Наявність операції оновлення дозволяє визначити оператор присвоювання (як у мові *SIPL*) такою формулою: $AS^x(fa)(d)=d\ \nabla^x(fa(d))$, де fa – функція, яка задається правою частиною оператора присвоювання, а d – номінативне дане.

Тоді семантика оператора введення read(x) полягає в такому: прочитати перше значення списку (файлу) *input*, присвоїти його змінній x та видалити з файлу *input* перший елемент:

$$AS^x(S^1(head,\ input\Rightarrow)\bullet AS^{input}(S^1(tail,\ input\Rightarrow)).$$

Семантика оператора write(fa) полягає в такому: обчислити значення фунції fa та додати його до файлу *output*:

$$AS^{output}(S^2(cons,\ fa,\ output\Rightarrow)).$$

Уведені операції розіменування й оновлення компонента мають своїми параметрами прості імена. Водночас у мовах програмування використовуються і складні імена. Наприклад, для доступу чи оновлення компонента ієрархічного запису (ієрархічної структури) використовуються імена вигляду *x.y*. Вважаємо, що складні імена вигляду $x_1.x_2. \ldots .x_m$ утворюються за допомогою асоціативної операції конкатенації, яку позначаємо крапкою ".". Це дозволяє узагальнити на складні імена операцію розіменування $x_1.x_2. \ldots .x_m\Rightarrow$ таким чином:

$$x_1.x_2. \ldots .x_m\Rightarrow(d) = x_{\mathrm{m}}\Rightarrow(\ldots(x_2\Rightarrow(x_1\Rightarrow(d)))\ldots).$$

Також можна узагальнити на складні імена операцію оновлення $\nabla^{x_1.x_2.\ldots.x_m}$. Формальне визначення доволі громіздке, тому обмежимось інтуїтивним описом: для обчислення значення $d\ \nabla^{x_1.x_2.\ldots.x_m} d'$ треба віднайти в номінативному даному d "місце", яке задається іменем $x_1.x_2. \ldots .x_m$, і надати йому значення номінативного даного d'. Тепер можна узагальнити оператор присвоювання формулою

$$As^{x_1.x_2.\ldots.x_m}(fa)(d) = d\ \nabla^{x_1.x_2.\ldots.x_m}(fa(d)).$$

Зауважимо, що поки ми ввели складні імена як параметри операцій над номінативними даними. Можна піти далі й увести номінативні дані зі складними іменами.

Наприклад, номінативне дане, яке розглядалось раніше (рис. 7.1), можна перетворити на такі дані зі складними іменами:

(1) [A.1 ↦ [1 ↦ 9, 2 ↦ 8, 3 ↦ 7], A.2 ↦ [1 ↦ 6, 2 ↦ 5, 3 ↦ 4]];

(2) [A.1.1 ↦ 9, A.1.2 ↦ 8, A.1.3 ↦ 7, A.2.1 ↦ 6, A.2.2 ↦ 5, A.2.3 ↦ 4];

(3) [A.1.1 ↦ 9, A.1.2 ↦ 8, A.1.3 ↦ 7, A.2 ↦ [1 ↦ 6, 2 ↦ 5, 3 ↦ 4]].

Графічно такі дані подано на рис. 7.2.

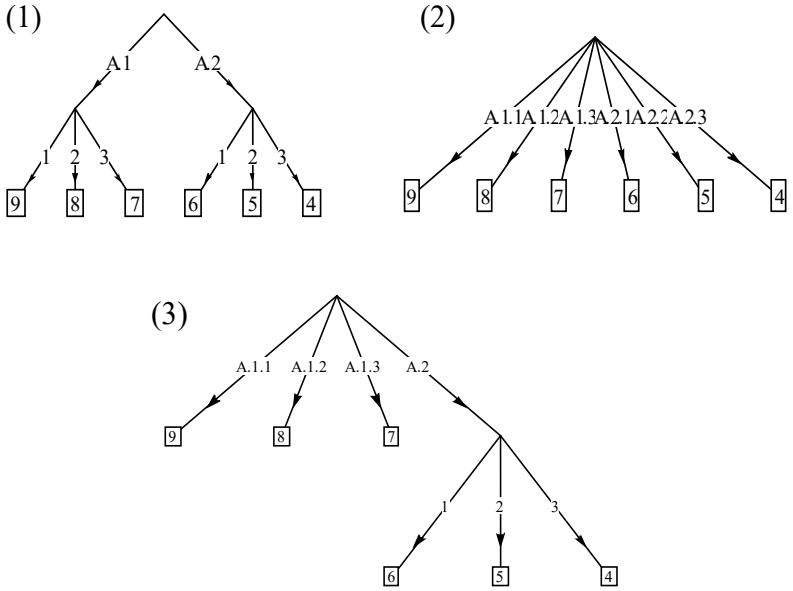

Рис. 7.2. Номінативні дані зі складними іменами

Усі раніше введені операції над номінативними даними трактували імена як відмінні від значень, тобто $V \cap W = \varnothing$. Це означає, що ми розглядали лише пряму адресацію, проте в мовах програмування використовується і непряма адресація, наприклад вказівники вказують на ім'я, яке вказує на певне значення. Подібна ситуація виникає при роботі з масивами. Наприклад, якщо ми звертаємось до компонента $A[i + 1]$, то $i + 1$ не є прямим іменем компонента, а є виразом, значення якого буде іменем компонента.

З наведених міркувань доходимо висновку, що необхідно ввести нові операції та композиції, які дозволять спочатку обчислювати імена, а потім вибирати або оновлювати їхні значення. Будемо вважати, що $V \subseteq W$, тобто що імена є значеннями. Тепер уведемо унарну композицію *обчислювального розіменування*

^, яка задається такою формулою (*fv* – функція, значення якої є іменем, *d* є номінативним даним):

$$^(fv)(d) = v\Rightarrow(d), \text{ де } v = fv(d).$$

Ця формула означає, що спочатку обчислюється значення функції *fv* на даному *d*, отримане значення *v* розглядається як ім'я, тому результатом буде значення цього імені в *d*.

Бінарна композиція *обчислювального присвоювання* ^*AS* задається таким чином (*fv* – функція, значення якої є іменем, функція *fa* задає значення, яким оновлюється компонент, *d* є номінативним даним):

$$^AS\ (fv, fa)(d) = d\ \nabla^{v}(fa(d)), \text{ де } v = fv(d).$$

Ця формула означає, що спочатку обчислюється значення функції *fv* на даному *d*, отримане значення *v* розглядається як ім'я, яке буде оновлено значенням функції *fa* на *d*.

Зауважимо, що ^*AS* є символом композиції обчислювального присвоювання, тому символ ^ у ^*AS* не є символом композиції обчислювального розіменування.

Приклад 7.5 (формалізація операцій над масивами). Треба записати семантику оператора присвоювання A[i]:=A[i+1].

Будемо використовувати функції-константи $\overline{i}$ та $\overline{A}$, які на довільному даному мають результат *i* та *A*, відповідно. Бінарну функцію конкатенації позначаємо *conc*. Ліва частина оператора присвоювання A[i+1] означає, що треба оновити компонент, який задається виразом i+1, тому застосовуємо композицію обчислювального розіменування. Отримуємо

$$^(S^2(conc, \overline{A}, S^2(add, i \Rightarrow, \overline{1}))) .$$

Формалізація правої частини дає формулу $S^2(conc, \overline{A}, i \Rightarrow))$, а весь оператор присвоювання подається формулою

$$^AS(S^2(conc, \overline{A}, i \Rightarrow), ^(S^2(conc, \overline{A}, S^2(add, i \Rightarrow, \overline{1})))).$$

Обчислимо отриману функцію на номінативному даному

$$d=[A \mapsto [1 \mapsto 9,\ 2 \mapsto 8,\ 3 \mapsto 7,\ 4 \mapsto 2],$$
$$i \mapsto 2,\ B \mapsto [1 \mapsto 6,\ 2 \mapsto 5,\ 3 \mapsto 4],\ j \mapsto 4].$$

Спочатку обчислюємо праву частину:

$$^(S^2(conc, \overline{A}, S^2(add, i \Rightarrow, \overline{1})))(d) = A.3 \Rightarrow (d) = 7 ,$$

оскільки

$$S^2(conc, \overline{A}, S^2(add, i \Rightarrow, \overline{1}))(d) = conc(\overline{A}(d), S^2(add, i \Rightarrow, \overline{1})(d)) =$$

$$= conc(A, add(i \Rightarrow (d), \overline{1}(d))) = conc(A, add(2,1)) = conc(A, 3) = A.3.$$

Тепер обчислюємо ліву частину:

$$S^2(conc, \overline{A}, i \Rightarrow))(d) = conc(\overline{A}(d), i \Rightarrow (d)) = conc(A, 2) = A.2 .$$

Далі обчислюємо сам оператор присвоювання:

$$\wedge AS(S^2(conc, \overline{A}, i \Rightarrow), \wedge(S^2(conc, \overline{A}, S^2(add, i \Rightarrow, \overline{1}))))(d) =$$

$$= d\ \nabla^{A.2}(A.3 \Rightarrow (d)) =$$

$$= [A \mapsto [1 \mapsto 9,\ 2 \mapsto 8,\ 3 \mapsto 7,\ 4 \mapsto 2],\ i \mapsto 2,$$

$$B \mapsto [1 \mapsto 6,\ 2 \mapsto 5,\ 3 \mapsto 4],\ j \mapsto 4]\ \nabla^{A.2} 7 =$$

$$= [A \mapsto [1 \mapsto 9,\ 2 \mapsto 7,\ 3 \mapsto 7,\ 4 \mapsto 2],\ i \mapsto 2,$$

$$B \mapsto [1 \mapsto 6,\ 2 \mapsto 5,\ 3 \mapsto 4],\ j \mapsto 4].$$

Зауваження 7.1. При порівнянні традиційного запису оператора присвоювання A[i]:=A[i+1] та формалізованого запису

$$\wedge AS(S^2(conc, \overline{A}, i \Rightarrow), \wedge(S^2(conc, \overline{A}, S^2(add, i \Rightarrow, \overline{1}))))$$

виникає природне запитання, чому формалізований запис видається таким складним. Тут треба розуміти, що формалізований запис явно визначає чітку семантику оператора присвоювання та зазначає роль кожного символу в цьому записі, але це явно не відображається у традиційному записі. Наприклад, традиційний запис A[i+2] не дає можливості визначити його семантику. Це можна зробити лише після аналізу контексту: якщо цей запис є лівою частиною оператора присвоювання, то він означає ім'я, яке має бути оновленим; якщо ж запис є частиною певного виразу, то його треба обчислити, щоб отримати відповідне значення. Формалізований запис є більш складним, тому що контекст уже врахований у самому записі. Це означає, що формалізований запис є композиційним, тому семантика його компонент не залежить від контексту.

Аналогічно можна вводити інші конструкції мов програмування, наприклад вказівники, які реалізують принцип непрямого іменування.

7.3. Композиційно-номінативні мови та логіки програм

Основним засобом подання програм є мови програмування. Головними їх недоліками з погляду теорії є те, що вони зазвичай неформалізовані та великі за описом. Це фактично робить неможливим будь-які математичні доведення властивостей їхніх програм. Тому для наших цілей (розбудови формальної теорії програмування) необхідні мови, які задовольняють такі критерії:

- *формальність* (математична строгість);
- *дескриптивна простота* (невеликий опис мови);
- *функціональна повнота* (можливість запрограмувати будь-яку обчислювану функцію);
- *структурна адекватність* (можливість відобразити програмою структуру алгоритму розв'язання задачі).

Ясно, що ці вимоги не є експлікованими, тому можливі різні їх тлумачення. Утім, перші три є інтуїтивно зрозумілими, а що стосується структурної адекватності, то будемо тлумачити її як композиційну адекватність мови, тобто її набір композицій має виразити основні засоби конструювання програм.

Для побудови мов, що відповідають наведеним критеріям, ми використовуємо композиційно-номінативний підхід [22]. Цей підхід базується на принципах композиційності та номінативності. Принцип *композиційності* трактує засоби побудови програм (функцій, предикатів) як алгебраїчні операції. *Принцип номінативності* вимагає використання відношень іменування для побудови семантичних моделей і опису програм (їхніх даних, функцій та композицій).

Наведені принципи дозволяють подавати семантику програм за допомогою двох алгебр: алгебри даних та алгебри функцій. Головним структурним типом даних є номінативні дані. На їх основі можна вводити різні конструкції мов програмування, формалізуючи їхню семантику в термінах операцій і композицій

над номінативними даними. Приклади були розглянуті в попередньому підрозділі.

Більше того, можна формалізувати типи даних, функцій і композицій, які можуть вводитися користувачем. Типи даних можна розглядати як спеціальні підкласи номінативних даних, нові функції можуть задаватись рекурсивними визначеннями (див. розд. 5), нові композиції також можуть вводитись рекурсивними визначеннями. Наприклад, композиція $RU(f, p)$ – **repeat** f **until** p – задається рекурсивним визначенням

$$RU(f, p) = f \bullet IF(p, id, RU(f, p)).$$

Ця композиція також може бути визначена за допомогою композиції циклу WH:

$$RU(f, p) = f \bullet WH(\neg p, f).$$

Мову *SIPL* можна розглядати як просту композиційно-номінативну мову. Складніші композиційно-номінативні мови розглянуто в [2, 22, 23].

Отже, композиційно-номінативні мови є потужним, але не занадто складним формалізмом подання мов програмування.

Однак лише мов програмування недостатньо для дослідження (аналізу, верифікації) програм. Для цього необхідні мови *специфікацій* та *логіки програм*. Для мов специфікацій обчислюваність композицій не вимагається. Тому серед композицій у мовах специфікацій є композиції універсальної та екзистенціальної квантифікації предикатів. Це дозволяє використовувати потужні мови специфікацій програм для їх аналізу та верифікації.

Ідея програмних логік полягає у створенні формалізмів для опису властивостей програм. У цьому сенсі логіки подібні до абстрактних типів даних, але, на відміну від останніх, які концентруються на описі властивостей операцій, програмні логіки передусім описують властивості композицій. Це добре видно для правил логіки Флойда–Хоара (табл. 6.3), що описують властивості композицій мови *SIPL*.

Подібні логіки можна визначити для більш потужних, ніж *SIPL*, композиційно-номінативних мов [20, 22, 23]. Такі логіки використовуються для доведення властивостей програм [15].

Висновки

Формалізація мов програмування означає побудову формальної (математичної) моделі, яка:

- дозволяє точне й однозначне тлумачення програм;
- надає можливість використання математичних методів для дослідження властивостей програм, їх аналізу та верифікації;
- створює підстави для автоматизації дослідження програм.

При формалізації мов програмування важливими є принципи композиційності та номінативності. Це дозволяє подати семантику програм як функцій над номінативними даними. Такі функції будуються за допомогою композицій. У цьому випадку семантика програм задається двома алгебрами: даних і функцій. Формалізм алгебр даних і функцій дозволяє формалізувати основні конструкції мов програмування та мов специфікацій. Зрозуміло, що для простих мов програмування та специфікацій ми отримуємо достатньо прості формальні моделі програм (алгебри даних і функцій); для більш складних мов алгебри будуть складнішими.

Вивчення загальних властивостей програм полягає у вивченні властивостей операцій над даними, а також властивостей композицій. Коли дані визначаються властивостями їхніх операцій, ми отримуємо поняття абстрактного типу даних; коли ми вивчаємо загальні властивості композицій програм, отримуємо поняття програмної логіки. Ці формалізми використовуються для доведення властивостей програм.

Додаткові питання з формалізації мов програмування та її використання в розробці програм розглянуто в [7, 8, 15, 17, 24].

Контрольні запитання

1. Що таке формальна модель програми?
2. Яка мета формалізації мов програмування?
3. Які алгебри використовуються для формалізації мов програмування?
4. Що таке абстрактний тип даних?
5. Яка ідея покладена у визначення поняття номінативних даних?
6. Як визначається клас номінативних даних?
7. Які операції над номінативними даними є основними?
8. Що означає принцип номінативності?
9. Що означає принцип композиційності?
10. Яка мета введення програмних логік?

Вправи

1. Побудувати абстрактний тип даних "стек".
2. Побудувати абстрактний тип даних "черга".
3. Побудувати абстрактний тип даних "бінарне дерево".
4. Побудувати абстрактний тип даних "дерево".
5. Побудувати абстрактний тип даних "номінативне дане".
6. Побудувати розширення мови *SIPL* булевим типом.
7. Побудувати розширення мови *SIPL* символьним типом.
8. Побудувати розширення мови *SIPL* структурами (записами).
9. Побудувати розширення мови *SIPL* масивами.
10. Побудувати розширення мови *SIPL* процедурами.

СПИСОК ЛІТЕРАТУРИ

1. Ахо А. Теория синтаксического анализа, перевода и компиляции. Т. 1. Синтаксический анализ / А. Ахо, Дж. Ульман. – М. : Мир, 1978.
2. Басараб И. А. Композиционные базы данных / И. А. Басараб, Н. С. Никитченко, В. Н. Редько. – К. : Либідь, 1992.
3. Глушков В. М. Алгебра. Языки. Программирование / В. М. Глушков, Г. Е. Цейтлин, Е. Л. Ющенко.– К. : Наукова думка, 1974.
4. Грис Д. Наука программирования / Д. Грис. – М. : Мир, 1982.
5. Гросс М. Теория формальных грамматик / М. Гросс, А. Лантен. – М. : Мир, 1971.
6. Дейкстра Э. Дисциплина программирования / Э. Дейкстра. – М. : Мир, 1976.
7. Камкин А. С. Введение в формальные методы верификации программ : учеб. пособие / А. С. Камкин. – М. : МАКС Пресс, 2018.
8. Кривий С. Л. Вступ до методів створення програмних продуктів / С. Л. Кривий. – К. : НУКМА, 2018.
9. Лавров С. Программирование. Математические основы, средства, теория / С. Лавров. – СПб. : БХВ-Петербург, 2001.
10. Нікітченко М. С. Теорія програмування. Ч. 1. / М. С. Нікітченко. – Ніжин, 2010.

11. Нікітченко М. С. Математична логіка та теорія алгоритмів / М. С. Нікітченко, С. С. Шкільняк. – К. : ВПЦ "Київський університет", 2008.
12. Нікітченко М. С. Прикладна логіка / М. С. Нікітченко, С. С. Шкільняк. – К. : ВПЦ "Київський університет", 2013.
13. Пентус А. Е. Теория формальных языков : учеб. пособие / А. Е. Пентус, М. Р. Пентус. – М. : Изд-во ЦПИ при мех.-мат. ф-те МГУ, 2004.
14. Редько В. Н. Композиции программ и композиционное программирование / В. Н. Редько // Программирование. – 1978. – № 5. – С. 3–24.
15. Формальні методи специфікації програм : навч. посіб. / А. Ю. Дорошенко, К. А. Жереб, Є. В. Іванов, М. С. Нікітченко, О. А. Яценко. – К. : ВПЦ "Київський університет", 2018.
16. Хопкрофт Дж. Введение в теорию автоматов, языков и вычислений / Дж. Хопкрофт, Р. Мотвани, Д. Ульман. – М. : Вильямс, 2002.
17. Abrial J.-R. Assigning Programs to Meanings / J.-R. Abrial. – Cambridge University press, 1996.
18. Hehner E. C. R. A Practical Theory of Programming. Texts and Monographs in Computer Science / E. C. R. Hehner. – N.Y. : Springer, 1993.
19. Hoare C. A. R. Unifying Theories of Programming / C. A. R. Hoare, H. Jifeng. – Prentice Hall Europe, 1998.

20. Formalization of the algebra of nominative data in mizar. Federated Conference on Computer Science and Information Systems (FedCSIS) / A. Korniłowicz, A. Kryvolap, M. Nikitchenko, I. Ivanov. – Prague, 2017. – P. 237–245.
21. Nielson H. R. Semantics with Applications. A Formal Introduction / H. R. Nielson, F. Nielson. – John Wiley & Sons, Chichester, England, 1992.
22. Nikitchenko N. A. Composition Nominative Approach to Program Semantics. Technical Report IT-TR: 1998-020 / N. A. Nikitchenko. – Technical University of Denmark, 1998.
23. Extended Floyd-Hoare Logic over Relational Nominative Data / M. Nikitchenko, I. Ivanov, A. Korniłowicz, A. Kryvolap // Communications in Computer and Information Science. – Springer, Cham, 2018. – Vol. 826. – P. 41–64.
24. Spivey J. M. Understanding Z: A specification language and its formal semantics / J. M. Spivey. – Cambridge University press, 1988.
25. Winskel G. The Formal Semantics of Programming Languages: An Introduction / G. Winskel. – The MIT Press, London : England, 1993.