МОДЕЛЮВАННЯ СТРУКТУР ДАНИХ ТА ФУНКЦІЙ НАД НИМИ В КОМПОЗИЦІЙНО-НОМІНАТИВНІЙ МОВІ ACON

Тарас Володимирович Панченко

Київський національний університет імені Тараса Шевченка,
вул. Володимирська 64, 01033 Київ, Україна,
E-mail: pantaras@ukr.net

В роботі проводиться моделювання структур даних мов програмування та функцій над ними засобами композиційно-номінативних мов. Розглядаються всі імперативні конструктори типів RAISE. Надається модель типів, виражаються операції над даними та аналізується рівень абстрактності даних, на якому представляються ті чи інші структури.

Programming languages data structures and functions over them modelling is conducted in this paper via Composition-Nominative Languages. All RAISE data type imperative constructors are researched here. Type model is given here. Operations over data are expressed here. Abstractness level for all data structure presentations is analysed here too.

Вступ

Дана робота є продовженням теми представлення різноманітних структур даних мов програмування як уточнення композиційно-номінативних структур в рамках композиційного підходу. Відома теза, що типи даних – суть природні спеціалізації поняття іменного даного [1]. Робота є також розвитком і уточненням [2, 3], де розглянуто співвідношення RAISE [4, 5] і композиційного підходу [6-8] і встановлений факт виразимості всіх імперативних конструкторів типів з RSL [4] в термінах композиційного підходу взагалі та композиційно-номінативної мови (CNL – Composition Nominative Language) ACoN [2, 9] (надалі – [CNL] ACoN) зокрема.

Мова формальних специфікацій RSL [4] вважається підходящою основою для такої роботи, оскільки вона включає ряд функціональних можливостей існуючих методів формальних специфікацій (зокрема, VDM), які експерти визнають досить просунутими, і була апробована та зарекомендувала себе як зручна у практичному використанні [10, 11].

Актуальність моделювання структур даних та функцій над ними в CNL, що розглядається в роботі, випливає з багатьох робіт на цю тему [1, 7, 8, 12]. Складність питання полягає в тому, що необхідно для кожного типу структур даних визначити рівень, на якому можна виразити функції над ними. Часто деякі функції можна виразити на більш загальному рівні, але інші функції над цим типом вимагають суттєвого пониження рівня абстрактності. В роботі після кожного типу структур підводиться підсумок про необхідний рівень номінативних даних та композиційних систем, в термінах яких можна їх представити.

Композиційно-номінативна мова ACoN

Коротко нагадаємо, що CNL ACoN працює з номінативними даними *комплексно-номінативного* рівня [2], тобто дані визначаються як $D \equiv W \cup (V \to D)$, де D – номінативні дані, визначені над множиною імен V та множиною значень W, причому $V \cap D \neq \varnothing$ (тобто імена можуть бути як завгодно складно структурованими). Сигнатура ACoN є $\{\circ, \diamond, *, \nabla, \varnothing_d$, choice, $v{=>}, {=>}v, !v, \backslash v, \in W, \dagger, \backslash v, \searrow v, v{=>}_{Comp}, {=>}v_{Comp}, !v_{Comp}, \backslash v_{Comp}, \searrow v_{Comp}, =, \mapsto$, getnames$\}$[1] (всі позначення – стандартні для CNL, деталі див. в [2]).

Додаткові позначення і функції. Всі вираження будемо проводити в термінах композиційно-номінативної мови ACoN [2] *комплексно-номінативного* рівня [2].

Введемо позначення, які будемо використовувати для зручності запису та полегшення розуміння. Дужки будемо використовувати для підвищення читабельності, хоча їх можна опустити, коли вони не позначають перелік аргументів деякої функції і не впливають на порядок виконання обчислень.

Композицію $\diamond(a, b, c)$ будемо позначати також **if** *a* **then** *b* **else** *c*.

Далі, **if** *a* **then** *b* $\equiv$[2] **if** *a* **then** *b* **else** $\varnothing_d$

while *a* **do** *b* $\equiv *(a, b)$

$\mathrm{Id} \equiv \Rightarrow 1 \circ 1 \Rightarrow$

$a ; b \equiv a \nabla ((\mathrm{Id} \nabla a) \circ b)$

$a \,\&^3\, b \equiv$ **if** *a* **then** (**if** *b* **then** Id)

$a \vee b \equiv$ **if** *a* **then** Id **else** (**if** *b* **then** Id)

$\neg a \equiv$ **if** *a* **then** $\varnothing_d$ **else** Id

value(v) $\equiv$ **if** $!v$ **then** $v\Rightarrow$ **else** $\varnothing_d$

[1] $\varnothing_d$ позначає функцію, що повертає порожнє іменне дане, яке позначається []

[2] "є скороченням і позначенням для"

[3] логічні операції тут будемо розуміти в сенсі семантики зв'язок McCarthy

Необхідно введення рівності над даними. Розуміємо тут рівність в сенсі [2]. Вводиться рівність індуктивно – спочатку над W, потім – над всією сукупністю D [2].

Оскільки ми будемо виражати функції (композиції), що фактично є макропідстановками (в дужках перелічуємо **імена**, з якими працює функція, з іменного даного), над даними, то звернемо увагу на дві суттєво різні ситуації. В програмуванні вони відомі як передача параметрів [до функції] за посиланням (за іменем) або за значенням. Для розрізнення цих ситуацій ми будемо використовувати схожий на C синтаксис: коли імена даних передаються за посиланням, будемо в лівій частині визначення ставити символ & перед відповідним іменем, в протилежному випадку – ім'я передається за значенням (тобто, в програмуванні – значення такої змінної дублюється в локальну область функції, яка була викликана). У випадку передачі змінних за посиланням (за іменем), макропідстановка відбувається без істотних змін – як написано в правій частині, тільки імена аргументів в **правій частині** замінюються на **імена** аргументів, що були підставлені в якості фактичних параметрів на місцях формальних параметрів під час виклику функції. В іншому випадку – спочатку відбувається необхідна **реномінація** вхідних даних, а потім – підставляється текст **правої частини** відповідної функції **без змін**. Уточнимо останнє. Якщо записана функція $f(a_1, a_2, \ldots, a_n) \equiv expr$, то виклик її у вигляді $f(b_1, b_2, \ldots, b_n)$ означає наступне: $R^4[\, a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots, a_n \mapsto b_n \,] \circ expr \equiv (\, (\, b_1\Rightarrow \circ \Rightarrow a_1 \,) \nabla (\, b_2\Rightarrow \circ \Rightarrow a_2 \,) \nabla \ldots \nabla (\, b_n\Rightarrow \circ \Rightarrow a_n \,) \,) \circ expr$, причому імена $a_1, a_2, \ldots, a_n$, як правило, різні.

Отже, 1) після перейменування (виконання операції реномінації) робота іде з **зазначеними в реалізації** (тобто в правій частині виразу функції) **іменами** над номінативним даним, а 2) вхідні аргументи – імена – потрібні тільки для початкового перейменування, потім обчислення йдуть в точності, як записано, з фіксованими іменами (а не такими, що замінюються при підстановці – при виклику функції).

Введемо декілька додаткових функцій-позначень (тут *v* – ім'я з множини V): очищення імені *v*, збереження імені *v* (для іменного і номінативного – багатозначного – варіантів) з усіх можливих імен та виключення всіх іменних компонент з іменем *v* із номінативного даного:

empty(&*v*) ≡ $\varnothing_d \circ \Rightarrow v$

save1(&*v*) ≡ $v\Rightarrow \circ \Rightarrow v$

save(&*v*) ≡ $\Rightarrow 1 \circ$ **while** ($1\Rightarrow \circ\ !v$) **do** ((Id ○ \1) † (($1\Rightarrow \circ \searrow v$) ○ (Id ∇ ($2\Rightarrow \circ \Rightarrow v$)) ○ \2)) ○ \1

exclude(&*v*) ≡ **while** !*v* **do** *v*

Нам знадобиться функція розіменування спеціального вигляду getvalue, яка "дістає" значення з іменної пари з іменем *Name* із даного, іменованого *Data*, якщо така пара існує. Виразимо її наступним чином:

getvalue(*Name*, *Data*) ≡ (Id ∇ (*Data*⇒ ○ exclude(2) ○ ⇒*Data*2)) ○
if ((*Data*2⇒ ∇ (*Name*⇒ ○ ⇒1)) ○ !(1⇒)) **then** (
(*Data*⇒ ∇ (Name⇒ ○ ⇒2)) ○ (2⇒)⇒
) **else** (*Data*⇒ ○ 2⇒)

І ще деякі функції, які залежать від домену базових значень W, точніше – від представлення цілих чисел, і надають можливість моделювати (або працювати) з натуральними числами.

Якщо числа представляються кратним іменуванням (0 = [], i+1 = [1 ↦ i], де 1 – деяке ім'я з V), то визначимо:

null(&*v*) ≡ empty(*v*)

inc(&*v*) ≡ $v\Rightarrow \circ \Rightarrow 1 \circ \Rightarrow v$

dec(&*v*) ≡ $v\Rightarrow \circ$ (**if** !1 **then** $1\Rightarrow$) ○ $\Rightarrow v$

is_null(&*v*) ≡ **if** ($v\Rightarrow \circ$!1) **then** $\Rightarrow v$ **else** ∅

Якщо ж W є типізованим доменом, з визначеними операціями +1, ∸1 на піддомені цілих чисел (int) та можливістю представлення констант і порівняння, то

null(&*v*) ≡ ((int-const) 0) ○ $\Rightarrow v$

inc(&*v*) ≡ $v\Rightarrow \circ +1 \circ \Rightarrow v$

dec(&*v*) ≡ $v\Rightarrow \circ \dot{-}1 \circ \Rightarrow v$

is_null(&*v*) ≡ **if** (($v\Rightarrow$) = ((int-const) 0)) **then** $\Rightarrow v$ **else** ∅

Отже, тепер перейдемо до моделювання типів RAISE в ACoN .

Представлення структур RSL в ACoN

Добуток типів (Product). Добуток типів за RAISE – це впорядкована скінчена послідовність значень, можливо, різних типів. Позначається добуток типів $T_1, T_2, \ldots, T_n$ як $T_1 \times T_2 \times \ldots \times T_n$. Представник такого типу має вигляд $(v_1, v_2, \ldots, v_n)$, де кожне v_i є значенням типу T_i. Представники типу **Product**

[4] звичайна операція реномінації в сенсі, напр. [13]

моделюються в термінах CNL як поіменовані *n* компонент добутку, де кожна компонента має ім'я, що відповідає її номеру та відповідне значення [2]. (Наприклад, якщо *v* типу **Bool × Int** має значення (***true***, 5), то синтаксично в термінах CNL цей факт буде мати вигляд: [*v* ↦ [1 ↦ ***true***, 2 ↦ 5]]. [2])

Хоча в RAISE не визначено жодної операції над типом **Product** – виразимо деякі ключові функції над даними цього типу.

Взяття *i*-ї (за номером) компоненти добутку в даному *d*:
get_component(*i*, *d*) ≡ getvalue(*i*, *d*)

Встановлення *i*-ї компоненти добутку в даному *d* значенням *val* і повернення добутку-результату:
set_component(*d*, *i*, *val*) ≡ *d*⇒ ∇ (*i* ↦ *val*)

Кількість компонент добутку *d* знаходиться наступним чином:
size(*d*) ≡ (Id ∇ (null(*counter*) ○ inc(*counter*))) ○
while (!*d* & ((*d*⇒ ∇ save1(*counter*)) ○ !(*counter*⇒))) **do** (Id ∇ inc(*counter*))
○ dec(*counter*) ○ *counter*⇒

Конструктор добутку – за добутками *d*1 та *d*2 будує добуток *d*1 × *d*2:
product(*d*1, *d*2) ≡ (Id ∇ (size(*d*1) ○ ⇒*counter*1) ∇ (null(*counter*2) ○ inc(*counter*2))) ○
while (!*d*2 & ((*d*2⇒ ∇ save1(*counter*2)) ○ !(*counter*2⇒))) **do** (
(Id ∇ inc(*counter*1) ∇ (get_component(*counter*2, *d*2) ○ ⇒*val*)) ○
(Id ∇ (set_component(*d*1, *counter*1, *val*) ○ ⇒*d*1) ∇ inc(*counter*2))
) ○ *d*1⇒

Зауважимо, що без введення відношення (операції) рівності над даними, над типом добутку можна виразити всі функції, але для представлення set_component(*d*, *i*, *val*) та product(*d*1, *d*2) необхідно використовувати функцію ↦, що виводить представлення даних типу **Product** на *комплексно-номінативний* рівень.

Множини (Set). Множина за RAISE – це невпорядкований набір різних значень однакового типу. В [2] вказано, що множини можна представляти двома способами – як ідентифіковані дані (це розглянуто в [13, 14]) і як мультиномінативні дані [8]. Оскільки множина – невпорядкований набір, то її можна подати як іменований набір компонент (ім'я – значення), що мають однакове ім'я (для визначеності серед можливих претендентів візьмемо *elem*, аби підкреслити, що маємо справу з елементами множини) [8], але різні значення. Тобто множина S={1, 2, 5} представлятиметься як [S ↦ [*elem* ↦ 1, *elem* ↦ 2, *elem* ↦ 5]]. Таке представлення відповідає інтенсіоналу поняття множини [8].

Таким чином, дане типу **Set** знаходиться на *номінативному* рівні, тобто такого рівня даних достатньо для адекватного представлення типу множини. Виразимо операції, визначені в RAISE над типом множин.

Виразимо функції над множинами.

Належність елемента до множини – повертає непорожнє дане[5], якщо *v*∈ *V* (дане з іменем *v* міститься серед елементів множини, іменованої *V*):
v∈ *V* ≡ **while** (!*V* & (*V*⇒ ○ !*elem*)) **do**
(((*V*⇒ ○ ↘*elem*) ∇ save1(*v*)) ○
(**if** (2⇒) = (*v*⇒) **then** exclude(*v*) **else** ((1⇒ ○ ⇒*V*) ∇ save1(*v*))))
○ **if** !*v* **then** ∅$_d$ **else** ⇒1

Входження множин – повертає непорожнє дане, якщо *V*⊆*W* (множина з іменем *V* є підмножиною множини, іменованої *W*):
V⊆*W* ≡ **while** (!*V* & (*V*⇒ ○ !*elem*)) **do** (((*V*⇒ ○ ↘*elem*) ∇ save1(*W*)) ○
if 2∈ *W* **then** ((1⇒ ○ ⇒*V*) ∇ save1(*W*)) **else** ∅)
○ **if** (!*V* & !*W*) **then** ⇒1 **else** ∅$_d$

Функції "не належності", рівності та "є власною підмножиною" визначаються просто:
v∉ *V* ≡ ¬ (*v*∈ *V*)

[5] мається на увазі, що результатом функції буде деяке, довільне, дане – головне, що воно не є порожнім

$V{=}W \equiv (V{\subseteq}W)\ \&\ (W{\subseteq}V)$
$V{\subset}W \equiv (V{\subseteq}W)\ \&\ \neg\ (W{\subseteq}V)$
$V{\supseteq}W \equiv W{\subseteq}V$
$V{\supset}W \equiv W{\subset}V$

Об'єднання множин – повертає множину, що складається з елементів (без повторень) двох заданих (своїми іменами *V* та *W*) множин:

$V{\cup}W \equiv V{\Rightarrow}\ \dagger\ W{\Rightarrow}$

Перетин множин – повертає множину, що складається з елементів, присутніх в обох заданих (своїми іменами *V* та *W*) множинах:

$V{\cap}W \equiv$ (Id ∇ empty(*Result*)) $\circ$
while (!*V* & (*V*$\Rightarrow$ $\circ$!*elem*)) **do** (((*V*$\Rightarrow$ $\circ$ $\searrow$*elem*) ∇ save1(*W*) ∇ save1(*Result*)) $\circ$
(((*Result*$\Rightarrow$ $\dagger$ (**if** 2$\in$ *W* **then** (2$\Rightarrow$ $\circ$ $\Rightarrow$*elem*) **else** $\varnothing$)) $\circ$ $\Rightarrow$*Result*) ∇
(1$\Rightarrow$ $\circ$ $\Rightarrow$*V*) ∇ save1(*W*)))
$\circ$ *Result*$\Rightarrow$

Різниця множин – повертає множину, що складається з елементів, присутніх в першій множині і відсутніх в другій (множини задані своїми іменами *V* та *W*):

$V{\setminus}W \equiv$ (Id ∇ empty(*Result*)) $\circ$
while (!*V* & (*V*$\Rightarrow$ $\circ$!*elem*)) **do** (((*V*$\Rightarrow$ $\circ$ $\searrow$*elem*) ∇ save1(*W*) ∇ save1(*Result*)) $\circ$
(((*Result*$\Rightarrow$ $\dagger$ (**if** 2$\notin$ *W* **then** (2$\Rightarrow$ $\circ$ $\Rightarrow$*elem*) **else** $\varnothing_{\mathrm{d}}$)) $\circ$ $\Rightarrow$*Result*) ∇
(1$\Rightarrow$ $\circ$ $\Rightarrow$*V*) ∇ save1(*W*)))
$\circ$ *Result*$\Rightarrow$

Потужність множини – card(*V*) – повертає кількість елементів вказаної [своїм іменем] множини у відповідному вигляді (див. зауваження щодо функцій null(*v*) та inc(*v*) вище):

card(*V*) $\equiv$ (Id ∇ null(*Result*)) $\circ$
while (!*V* & (*V*$\Rightarrow$ $\circ$!*elem*)) **do** ((*V*$\Rightarrow$ $\circ$ *elem* $\circ$ $\Rightarrow$*V*) ∇ inc(*Result*))
$\circ$ *Result*$\Rightarrow$

Взяття елементу з множини – повертає деякий елемент, якщо множина непорожня:

get_elem_partial(*V*) $\equiv$ *V*$\Rightarrow$ $\circ$ *elem*$\Rightarrow$

get_elem_total(*V*) $\equiv$ **if** (!*V* & (*V*$\Rightarrow$ $\circ$!*elem*)) **then** (get_elem_partial(*V*)) **else** $\varnothing$
відповідно часткова та тотальна функції.

Зауважимо, що без введення відношення (операції) рівності над даними, над типом множин можна реалізувати (виразити) лише функції об'єднання *V*$\cup$*W*, знаходження потужності множини card(*V*) та взяття елементу get_elem_partial(*V*) і get_elem_total(*V*). Майже всі класичні операції над множинами з теорії множин вимагають введення функції-предикату рівності над елементами множини. Можна також виразити всі функції за допомогою предикату належності елемента до множини, якщо взяти його базовим, і позбутися явного використання предикату рівності над даними.

Також відзначимо, якщо взяти представлення множин у вигляді ідентифікованих даних [8, 13] за основу, то для вираження всіх функцій над множинами необхідно спочатку множину з такого представлення перетворити в множину розглянутого вище представлення за допомогою функції getnames, а результат, коли потрібно, перетворити в зворотному напрямку за допомогою функції set2id(*V*):

set2id(*V*) $\equiv$ (save1(*V*) ∇ null(*Result*)) $\circ$
while (!*V* & (*V*$\Rightarrow$ $\circ$!*elem*)) **do** (((*V*$\Rightarrow$ $\circ$ $\searrow$*elem*) ∇ save1(*Result*)) $\circ$
(((*Result*$\Rightarrow$ ∇ (2 $\mapsto$ 2)) $\circ$ $\Rightarrow$*Result*) ∇ (1$\Rightarrow$ $\circ$ $\Rightarrow$*V*)))
) $\circ$ *Result*$\Rightarrow$

Але слід зауважити, що представлення множин у вигляді ідентифікованих даних виводить ці дані на *комплексно-номінативний* рівень [2], в той час як розглянуте вище представлення вимагає лише *метаномінативного* рівня даних. Лише для функцій get_elem_partial(*V*) та get_elem_total(*V*) достатньо *номінативного* рівня даних.

Списки (List). Тип списку (**List**) за RAISE визначається як послідовність значень одного типу, що, можливо, містить дублікати. В [2] зафіксовано представлення даних типу **List** в CNL у вигляді іменованої пари елемента-голови списку з іменем *Head* і хвоста (що є, в свою чергу, також списком) з іменем *Tail*. Оскільки традиційно першими операціями для списків визначають **hd** (голова) та **tl** (хвіст), то такий інтенсіонал списку (тобто: кожний список складається з голови – першого елемента – і хвоста – решти елементів, що мають таку ж структуру, як і сам [весь] список) адекватно відображається обраним представленням. Значення порожнього списку представляється пустим даним. Так, наприклад, список v=<1, 5, 17> буде представлений наступним чином: [v ↦ [*Head* ↦ 1, *Tail* ↦ [*Head* ↦ 5, *Tail* ↦ [*Head* ↦ 17, *Tail* ↦ []]]]].

Покажемо, як виражаються функції, визначені в RAISE над типом **List**, в TCNL.

Виразимо функції над списками:

Голова списку:

hd(L) ≡ L⇒ ○ *Head*⇒

Хвіст списку:

tl(L) ≡ L⇒ ○ *Tail*⇒

Таким чином, наприклад, LISP-конструкція CADDR(L) = L⇒ ○ *Tail*⇒ ○ *Tail*⇒ ○ *Head*⇒.

Елемент за індексом (повертає потрібний елемент списку):

index(L, *idx*) ≡ L[*idx*] ≡

while (¬ is_null(*idx*)) **do** ((L⇒ ○ *Tail*⇒ ○ ⇒L) ∇ dec(*idx*)) ○ L⇒ ○ *Head*⇒

Довжина списку (повертає кількість елементів списку):

len(L) ≡ (Id ∇ null(*Result*)) ○

while (L⇒) **do** ((L⇒ ○ *Tail*⇒ ○ ⇒L) ∇ inc(*Result*))

○ *Result*⇒

Індекси елементів списку (множина):

inds(L) ≡ (Id ∇ empty(*Result*) ∇ (null(*Counter*) ○ inc(*Counter*))) ○

while (L⇒) **do** ((L⇒ ○ *Tail*⇒ ○ ⇒L) ∇ inc(*Counter*) ∇

((*Result*⇒ † (*Counter*⇒ ○ ⇒*elem*)) ○ ⇒*Result*))

○ *Result*⇒

Елементи списку (множина):

elems(L) ≡ (Id ∇ empty(*Result*)) ○

while (L⇒) **do** ((L⇒ ○ *Tail*⇒ ○ ⇒L) ∇

((*Result*⇒ † ((L⇒ ○ *Head*⇒) ○ ⇒*elem*)) ○ ⇒*Result*))

○ *Result*⇒

Для вираження функції конкатенації списків (дописування другого списку в кінець першого, позначається в RAISE як ^) представимо додатково допоміжні функції додавання елементу в голову списку (addfirst), в кінець хвоста (addlast) та “розворот” або реверсію (reverse) списку:

addfirst(*elem*, L) ≡ (*elem*⇒ ○ ⇒*Head*) ∇ (L⇒ ○ ⇒*Tail*)

reverse(L) ≡ (Id ∇ empty(*Result*)) ○

while (L⇒) **do** ((L⇒ ○ *Tail*⇒ ○ ⇒L) ∇

(((*Result*⇒ ○ ⇒*Tail*) ∇ (L⇒ ○ save1(*Head*))) ○ ⇒*Result*))

○ *Result*⇒

addlast(*elem*, L) ≡

((reverse(L) ○ ⇒L) ∇ save1(*elem*)) ○ (addfirst(*elem*, L) ○ ⇒L) ○ (reverse(L))

тоді

cat(V, W) ≡ ((reverse(V) ○ ⇒L) ∇ save1(W)) ○

while (*L*⇒) **do** ((*L*⇒ ○ *Tail*⇒ ○ ⇒*L*) ∇
(((*W*⇒ ○ ⇒*Tail*) ∇ (*L*⇒ ○ save1(*Head*))) ○ ⇒*W*))
○ *W*⇒

Зауважимо, що без введення відношення (операції) рівності над даними, над типом списків можна реалізувати (виразити) всі функції [з наведених вище], окрім взяття (знаходження) значення елемента за його індексом index(*L*, *idx*) у випадку int ⊆ W. Дані типу **List** знаходяться на *метаномінативному* рівні даних, хоча для вираження всіх функцій, окрім тих, що працюють з множинами (inds(*L*) та elems(*L*) – індекси та елементи списку) достатньо *номінативного* рівня даних.

Також для типу **List** в [2] вказане інше представлення – у вигляді "нумерованого списку", де кожен елемент іменується його номером у списку. Але це представлення, хоча і здається простим, зразу виводить дані на *комплексно-номінативний* рівень, оскільки вимагає суттєвого застосування специфічних для останнього рівня даних функцій ↦ та getnames. Таке представлення, до того ж, майже не відрізняється від моделі добутку типів **Product**. Для вираження функцій над таким представленням, достатньо скористатися функціями перетворення list2product та product2list:

list2product(*L*) ≡ (Id ∇ empty(*Result*) ∇ (null(*Counter*) ○ inc(*Counter*))) ○
while (*L*⇒) **do** ((*L*⇒ ○ *Tail*⇒ ○ ⇒*L*) ∇ inc(*Counter*) ∇
((*Result*⇒ ∇ ((Id ∇ (*L*⇒ ○ *Head*⇒ ○ ⇒*elem*)) ○ (*Counter* ↦ *elem*))) ○ ⇒*Result*))
○ *Result*⇒

product2list(*d*) ≡ (Id ∇ (null(*counter*) ○ inc(*counter*)) ∇ empty(*L*)) ○
while (!*d* & ((*d*⇒ ∇ save1(*counter*)) ○ !(*counter*⇒))) **do** (
(Id ∇ (getvalue(*counter*, *d*) ○ ⇒*val*)) ○ (Id ∇ (addfirst(*val*, *L*) ○ ⇒*L*) ∇ inc(*counter*))
) ○ reverse(*L*)

Слід лише зауважити, що деякі функції, тим не менше, можна виразити простіше, аніж сперш застосування функції product2list, потім виконання операції над попереднім представленням списку, а потім – якщо потрібно – використання list2product для зворотного перетворення типів (якщо результат функції є списком). Так, hd(*L*) ≡ 1⇒.

Відображення (Map). Тип відображення (**Map**) за RAISE – це структура, що ставить у відповідність значенням одного типу значення іншого типу. Наприклад, представником типу **Map**: **Int→Bool** буде [1 ↦ ***true***, 1 ↦ ***false***, 6 ↦ ***true***] в RAISE-нотації [4]. Для типу **Map** можна побудувати два представлення, причому обидва мають право на існування (див. [2, 7, 8, 13]). Оскільки ці представлення [2] лежать на різних рівнях абстракції, то розглянемо детально обидва.

З одного боку, відображення – це бінарне відношення, тобто підмножина декартового добутку двох множин, а, отже – множина пар. Звідси представник типу **Map** може мати наступну структуру: множина двокомпонентних пар, іменованих однаковим іменем (*Map*), де перша компонента з іменем *arg* містить елемент з домену аргументів відображення, а друга компонента – з іменем *res* – відповідний йому елемент з домену значень цього відображення. Так, наведений вище приклад відображення (нехай з іменем *v*) матиме наступну нотацію:[*v* ↦ [*Map* ↦ [*arg* ↦ 1, *res* ↦ ***true***], *Map* ↦ [*arg* ↦ 1 , *res* ↦ ***false***], *Map* ↦ [*arg* ↦ 6 , *res* ↦ ***true***]]].

В цьому представленні функції над типом **Map** будуть виражатися наступним чином.

Застосування (application) відображення *m* до елементу *el* з області визначення відображення (в RAISE, як і в математиці, позначається *m*(*el*)):
apply(*m*, *el*) ≡
while (!*m* & (*m*⇒ ○ !*Map*)) **do** (((*m*⇒ ○ ↘*Map*) ∇ save1(*el*)) ○
(**if** (2⇒ ○ *arg*⇒) = (*el*⇒) **then** (2⇒) **else** ((1⇒ ○ ⇒*m*)∇ save1(*el*))))
○ *res*⇒

Область визначення відображення (множина):
dom(*m*) ≡ (Id ∇ empty(*Result*)) ○
while (!*m* & (*m*⇒ ○ !*Map*)) **do** (((*m*⇒ ○ ↘*Map*) ∇ save1(*Result*)) ○ (
((*Result*⇒ † (2⇒ ○ *arg*⇒ ○ ⇒*elem*)) ○ ⇒*Result*) ∇ (1⇒ ○ ⇒*m*)))
○ *Result*⇒

Область значень відображення (множина):

rng(*m*) ≡ (Id ∇ empty(*Result*)) ○

while (!*m* & (*m*⇒ ○ !*Map*)) **do** (((*m*⇒ ○ ↘*Map*) ∇ save1(*Result*)) ○ (

((*Result*⇒ † (2⇒ ○ *res*⇒ ○ ⇒*elem*)) ○ ⇒*Result*) ∇ (1⇒ ○ ⇒*m*)))

○ *Result*⇒

Накладання відображення *mo* на *m* (мається на увазі, що елементи (пари) другого відображення перекривають елементи першого відображення при однакових перших компонентах пар (тобто однакових значеннях *arg*), в RAISE позначається *m* † *mo*):

override(*m*, *mo*) ≡ (Id ∇ (dom(*mo*) ○ ⇒*mo_dom*)) ○

while (!*m* & (*m*⇒ ○ !*Map*)) **do** (((*m*⇒ ○ ↘*Map*) ∇ save1(*mo*) ∇ save1(*mo_dom*)) ○

(((*mo*⇒ † ((Id ∇ (2⇒ ○ *arg*⇒ ○ ⇒*newarg*)) ○

(**if** *newarg*∈ *mo_dom* **then** $\varnothing_d$ **else** (2⇒ ○ ⇒*Map*)))) ○ ⇒*mo*) ∇

(1⇒ ○ ⇒*m*) ∇ save1(*mo_dom*)))

○ *mo*⇒

Об'єднання відображень (як графіків функцій) *m* та *mo* (в RAISE позначається *m* ∪ *mo*) буде виражатись наступним чином.

union(*m*, *mo*) ≡ (*m*⇒ † *mo*⇒)

Зріз відображення *m* множиною *s* (в RAISE позначається *m* \ *s*):

cutby(*m*, *s*) ≡ (Id ∇ empty(*Result*)) ○

while (!*m* & (*m*⇒ ○ !*Map*)) **do** (((*m*⇒ ○ ↘*Map*) ∇ save1(*Result*) ∇ save1(*s*)) ○ (

((*Result*⇒ † ((Id ∇ (2⇒ ○ *arg*⇒ ○ ⇒*newelem*)) ○

(**if** *newelem*∉ *s* **then** (2⇒ ○ ⇒*Map*) **else** $\varnothing_d$))

) ○ ⇒*Result*) ∇ (1⇒ ○ ⇒*m*) ∇ save1(*s*)))

○ *Result*⇒

Обмеження відображення *m* множиною *s* (в RAISE позначається *m* / *s*):

restrict(*m*, *s*) ≡ (Id ∇ empty(*Result*)) ○

while (!*m* & (*m*⇒ ○ !*Map*)) **do** (((*m*⇒ ○ ↘*Map*) ∇ save1(*Result*) ∇ save1(*s*)) ○ (

((*Result*⇒ † ((Id ∇ (2⇒ ○ *arg*⇒ ○ ⇒*newelem*)) ○

(**if** *newelem*∈ *s* **then** (2⇒ ○ ⇒*Map*) **else** $\varnothing_d$))

) ○ ⇒*Result*) ∇ (1⇒ ○ ⇒*m*) ∇ save1(*s*)))

○ *Result*⇒

Для вираження композиції відображень *m*1 та *m*2 (в RAISE позначається *m*1 ○ *m*2) введемо функцію образу елементу за відображенням, яка повертає множину (**Set**) значень даного типу **Map** (значення *res*) для відповідного вхідного значення *arg*:

image(*elem*, *map*) ≡ (Id ∇ empty(*Result*)) ○

while (!*map* & (*map*⇒ ○ !*Map*)) **do** (((*map*⇒ ○ ↘*Map*) ∇ save1(*elem*)

∇ save1(*Result*)) ○ (((*Result*⇒ † ((Id ∇ (2⇒ ○ *arg*⇒ ○ ⇒*newarg*) ∇

(2⇒ ○ *res*⇒ ○ ⇒*newres*)) ○

(**if** ((*newarg*⇒) = (*elem*⇒)) **then** (*newres*⇒ ○ ⇒*elem*) **else** $\varnothing_d$))

) ○ ⇒*Result*) ∇ (1⇒ ○ ⇒*map*) ∇ save1(*elem*)))

○ *Result*⇒

тоді

compos(*m*1, *m*2) ≡ (Id ∇ empty(*Result*)) ○

while (!*m*1 & (*m*1⇒ ○ !*Map*)) **do** (((*m*1⇒ ○ ↘*Map*) ∇ save1(*m*2) ∇

save1(*Result*)) ○ (((*Result*⇒ † ((Id ∇ (2⇒ ○ *arg*⇒ ○ ⇒*curarg*) ∇

(2⇒ ○ *res*⇒ ○ ⇒*curres*)) ○ (Id ▽ (image(*curres*, *m*2) ○ ⇒*Image*)) ○
if (*Image*⇒) **then** create_map(*curarg*, *Image*) **else** $\varnothing_d$)) ○
⇒*Result*) ▽ (1⇒ ○ ⇒*m*1) ▽ save1(*m*2)))
○ *Result*⇒

де

create_map(*Arg*, *R_set*) ≡ (Id ▽ empty(*Result*)) ○
while (!*R_set* & (*R_set*⇒ ○ !*elem*)) **do** (
((*R_set*⇒ ○ ↘*elem*) ▽ save1(*Arg*) ▽ save1(*Result*)) ○
(((*Result*⇒ † (((*Arg*⇒ ○ ⇒*arg*)▽ (2⇒ ○ ⇒*res*)) ○ ⇒*Map*))
○ ⇒*Result*) ▽ (1⇒ ○ ⇒*R_set*) ▽ save1(*Arg*))
) ○ *Result*⇒

Слід зауважити, що застосування композиції відображень (наприклад, *m*1 та *m*2) до елементу (нехай *elem*) можна визначити простіше, ніж “напряму” – спочатку побудова нового відображення-композиції, а потім застосування. А саме – можна виразити безпосередньо послідовне застосування і замість

(Id ▽ (compos(*m*1, *m*2) ○ ⇒*m*)) ○ apply(*m*, *elem*)

отримати:

apply_to_composition(*m*1, *m*2, *elem*) ≡
(Id ▽ (image(*elem*, *m*1) ○ ⇒*s*)) ○ restrict(*m*2, *s*) ○ *Map*⇒ ○ *res*⇒

До речі, з урахуванням введених вище функцій,

apply(*m*, *elem*) ≡ image(*elem*, *m*) ○ *elem*⇒

Зауважимо, що без введення відношення (операції) рівності над даними, над наведеним представленням типу відображень можна реалізувати (виразити) лише функції області визначення і значень dom(*m*) та rng(*m*) та об’єднання union(*m*, *mo*), а з допоміжних – create_map(*Arg*, *R_set*). Всі інші функції вимагають порівнянь. Дані типу **Map** в такому представленні знаходяться на *метаномінативному* рівні даних, хоча для вираження функції об’єднання відображень union(*m*, *mo*) достатньо *номінативного* рівня даних.

Розглянемо інше представлення елементів типу **Map**. По суті, **Map** – це відображення, тобто деяка функція, що зіставляє елементові з області визначення деякий елемент (або декілька) з області значень. Більш того, для елементів типу **Map** характерні ті ж операції, що і для функцій в математиці (область визначення, область значення, композиція, об’єднання графіків функції (union) і т.д.). Тобто інтенсіонал типу **Map** є дуже близьким до поняття функції в математиці. Ця теза породжує, ймовірно, більш адекватне представлення елементів типу **Map** в термінах композиційного числення, а саме: **Map** – це множина пар, де ім’я є елементом з області визначення відображення, а значення – відповідним йому елементом з області значення цього відображення. Тоді наведений вище приклад відображення (нехай з іменем *v*) буде мати наступну нотацію: [*v* ↦ [1 ↦ ***true***, 1 ↦ ***false***, 6 ↦ ***true***]]. Аналіз і порівняння двох представлень можна знайти в [2].

Виразимо операції над типом **Map** в останньому варіанті представлення даних цього типу.

Введемо допоміжні функції[6]: функцію знаходження образу елементу з dom за відображенням – image та функцію створення відображення за елементом (аргументом) та його образом (множиною результатів, значень) – create_map:

image(*elem*, *map*) ≡ (save1(*elem*) ▽ empty(*Result*) ▽
(*map*⇒ ○ exclude(*elem*) ○ ⇒*map*) ▽ (*map*⇒ ○ ⇒*map*2)) ○
if (!*map* & ((*map*⇒ ▽ (*elem*⇒ ○ ⇒1)) ○ !(1⇒))) **then** (
while (!*map* & ((*map*⇒ ▽ save1(*elem*)) ○ !(*elem*⇒))) **do** (
(((*map*⇒ ▽ save1(*elem*)) ○ ↘(*elem*⇒)) ▽ save1(*elem*) ▽ save1(*Result*)) ○
(((*Result*⇒ † (2⇒ ○ ⇒*elem*)) ○ ⇒*Result*) ▽

[6] Зауваження. Функція image виражається досить громіздко, бо при вказаному представленні **Map** можлива ситуація, коли стандартні імена 1 і 2 будуть задіяні в якості елементів з області визначення відображення (якщо воно, наприклад, має вигляд **Int→Bool**, див. приклад вище), а отже необхідно робити додаткові перевірки і розгалуження для розгляду варіантів і коректного їх опрацювання. Аналогічне зауваження має місце і для інших виразів (наприклад, getvalue та apply нижче).

(1⇒ ○ ⇒*map*) ∇ save1(*elem*))
)) **else** (
while (!*map*2 & (*map*2⇒ ○ !*elem*)) **do** (
((*map*2⇒ ○ ↘*elem*) ∇ save1(*Result*)) ○
(((*Result*⇒ † (2⇒ ○ ⇒*elem*)) ○ ⇒*Result*) ∇ (1⇒ ○ ⇒*map*2))
)) ○ *Result*⇒

create_map(*Arg*, *R_set*) ≡ (Id ∇ empty(*Result*)) ○
while (!*R_set* & (*R_set*⇒ ○ !*elem*)) **do** (
((*R_set*⇒ ○ ↘*elem*) ∇ save1(*Arg*) ∇ save1(*Result*)) ○
(((*Result*⇒ † (*Arg* ↦ 2)) ○ ⇒*Result*) ∇ (1⇒ ○ ⇒*R_set*) ∇ save1(*Arg*))
) ○ *Result*⇒

Отже, застосування (application) відображення *m* до елементу з області визначення *el*:
apply(*m*, *el*) ≡ **if** (((*m*⇒ ○ exclude(*el*)) ∇ (*el*⇒ ○ ⇒1)) ○ !(1⇒)) **then** (
(Id ∇ (*m*⇒)) ○ ((*el*⇒) ⇒)) **else** (*m*⇒ ○ *el*⇒)

Область визначення відображення:
dom(*m*) ≡ *m*⇒ ○ getnames

Область значень відображення:
rng(*m*) ≡ (Id ∇ empty(*Result*) ∇ (*m*⇒ ○ getnames ○ ⇒*Names*)) ○
while (!*Names* & (*Names*⇒ ○ !*elem*)) **do** ((Id ∇ (*Names*⇒ ○ ↘*elem*)) ○
(Id ∇ ((*Result*⇒ † (image(2, *m*))) ○ ⇒*Result*)) ○
(Id ∇ (1⇒ ○ ⇒*Names*))) ○ *Result*⇒

Накладання відображення *mo* на *m*:
override(*m*, *mo*) ≡ *m*⇒ ∇ *mo*⇒

Об’єднання відображень *m* та *mo*:
union(*m*, *mo*) ≡ *m*⇒ † *mo*⇒

Зріз відображення *m* множиною *s*:
cutby(*m*, *s*) ≡ (Id ∇ (*m*⇒ ○ getnames ○ ⇒*Names*) ∇ empty(*Result*)) ○
while (!*Names* & (*Names*⇒ ○ !*elem*)) **do** ((Id ∇ (*Names*⇒ ○ ↘*elem*)) ○
(Id ∇ **if** (2∉ *s*) **then** ((*Result*⇒ † (
((image(2, *m*) ○ ⇒*R_set*) ∇ save1(2)) ○ create_map(2, *R_set*)
)) ○ ⇒*Result*) **else** $\varnothing_d$) ○
(Id ∇ (1⇒ ○ ⇒*Names*))) ○ *Result*⇒

Обмеження відображення *m* множиною *s*:
restrict(*m*, *s*) ≡ (Id ∇ (*m*⇒ ○ getnames ○ ⇒*Names*) ∇ empty(*Result*)) ○
while (!*Names* & (*Names*⇒ ○ !*elem*)) **do** ((Id ∇ (*Names*⇒ ○ ↘*elem*)) ○
(Id ∇ **if** (2∈ *s*) **then** ((*Result*⇒ † (
((image(2, *m*) ○ ⇒*R_set*) ∇ save1(2)) ○ create_map(2, *R_set*)
)) ○ ⇒*Result*) **else** $\varnothing_d$) ○
(Id ∇ (1⇒ ○ ⇒*Names*))) ○ *Result*⇒

Композиція відображень *m*1 та *m*2:
compos(*m*1, *m*2) ≡ (Id ∇ (*m*1⇒ ○ getnames ○ ⇒*Names*) ∇ empty(*Result*)) ○
while (!*Names* & (*Names*⇒ ○ !*elem*)) **do** (

(Id ∇ (*Names*⇒ ○ ↘*elem*)) ○ (Id ∇ (image(2, *m*1) ○ ⇒*Image*1) ∇
(1⇒ ○ ⇒*Names*) ∇ (2⇒ ○ ⇒*curarg*)) ○
while (!*Image*1 & (*Image*1⇒ ○ !*elem*)) **do** (
(Id ∇ (*Image*1⇒ ○ ↘*elem*)) ○ (Id ∇ (1⇒ ○ ⇒*Image*1) ∇
((*Result*⇒ † ((Id ∇ (image(2, *m*2) ○ ⇒*Image*2)) ○
if (*Image*2⇒) **then** create_map(*curarg*, *Image*2) **else** $\varnothing_d$)) ○
⇒*Result*)))
) ○ *Result*⇒

Слід зауважити, що застосування композиції відображень (наприклад, *m*1 та *m*2) до елементу (нехай *elem*) можна визначити простіше, ніж "напряму" – спочатку побудова нового відображення-композиції, а потім застосування. А саме – можна виразити безпосередньо послідовне застосування і замість

(Id ∇ (compos(*m*1, *m*2) ○ ⇒*m*)) ○ apply(*m*, *elem*)

отримати:

apply_to_composition(*m*1, *m*2, *elem*) ≡
(Id ∇ (image(*elem*, *m*1) ○ ⇒*s*)) ○ restrict(*m*2, *s*) ○ ⇒*map* ○
(Id ∇ (*map*⇒ ○ getnames ○ *elem*⇒ ○ ⇒*Name*)) ○ getvalue(*Name*, *map*)

До речі, з урахуванням введених вище функцій,

apply(*m*, *elem*) ≡ image(*elem*, *m*) ○ *elem*⇒

Отже, без введення відношення (операції) рівності над даними, над останнім представленням типу відображень можна реалізувати (виразити) всі функції окрім тих, що безпосередньо працюють з множинами – cutby(*m*, *s*) та restrict(*m*, *s*), зріз та обмеження відображення множиною. Дані типу **Map** в останньому представленні знаходяться на *комплексно-номінативному* рівні, оскільки суттєво вимагають таких операцій як ↦ та getnames для вираження функцій над ними, хоча для функцій apply(*m*, *el*) та union(*m*, *mo*) достатньо *метаномінативного* рівня, а для функції override(*m*, *mo*) – навіть *номінативного* рівня.

Заключна частина

Розглянуте питання про представлення структур даних (імперативних конструкторів типів даних) RSL в термінах CNL. Всі основні структури даних RSL (складні типи даних) є уточненнями номінативних даних і лежать на різних рівнях даних – від *номінативного* до *комплексно-номінативного*. Специфічні типи структур RSL (абстрактні типи даних, об'єкти та деякі інші) будуть досліджені в подальших роботах.

Література

1. *Редько В.Н.* Семантические структуры программ // Программирование. – 1981. – №1. – С. 3-19.
2. *Нікітченко М. С., Панченко Т. В.* Структури даних в композиційних мовах програмування // Вісник Київського університету. Серія: фіз.-мат. науки. – 2004. – вип. 2. (подано до друку).
3. *Панченко Т. В.* Типізація в композиційно-номінативних мовах // Матеріали Міжнародної науково-практичної конференції студентів, аспірантів та молодих вчених "Шевченківська весна. Сучасний стан науки, досягнення, проблеми та перспективи розвитку". Збірник тез. – 2003. – С. 66-68.
4. *The RAISE Language Group.* The RAISE Specification Language. BCS Practitioner Series. Prentice Hall, 1992. – 397 p.
5. *The RAISE Method Group.* The RAISE Development Method. BCS Practitioner Series. Prentice Hall, 1995. – 493 p.
6. *Редько В. Н.* Основания композиционного программирования // Программирование. – 1979. – № 3. – С. 3-13.
7. *Басараб И. А., Никитченко Н. С., Редько В. Н.* Композиционные базы данных. – К.: Либідь, 1992. – 191 с.
8. *N. Nikitchenko.* A Composition Nominative Approach to Program Semantics. – Technical Report IT-TR: 1998-020. – Technical University of Denmark. – 1998. – 103 p.
9. *Panchenko T. V.* Composition Approach to Software Systems Modeling and its Support Tools // International Conference on Dynamical System Modeling and Stability Investigation. Thesis of Conference Reports, May 27-30, 2003. – P. 421.
10. *Панченко Т. В.* Використання формальних специфікацій для розробки програмних систем // Вісник Київського університету. Серія: фіз.-мат. науки. – 2002. – вип. 2. – С. 245-256.
11. *Панченко Т. В.* Використання формальних специфікацій для розробки Електронної біржі Ощадного банку // Проблемы программирования. - 2002. - №1-2. - С. 161-167.
12. *Никитченко Н. С.* Интенсиональные аспекты понятия программы // Проблемы программирования. – 2001.– № 3–4.– С. 5-13.
13. *Никитченко Н. С., Омельчук Л. Л., Шкильняк С. С., Янченко О. И.* Аксиоматические системы спецификаций программ над номинативными данными // Проблемы программирования. Специальный выпуск: Материалы второй международной научно-практической конференции по программированию.– 2000.– N.1-2. – С. 259-272.
14. *Нікітченко М. С., Шкільняк С. С., Омельчук Л.Л.* Програми над ідентифікованими даними та їх Σ-визначеність. Київ, 1999.– Деп. в ДНТБ України 01.10.99, N 242–Ук99. – 82 с.