ПРИКЛАДНАЯ ЛОГИКА
(Вычислительные системы)
1986 год Выпуск 116

УДК 519.681.2

СРЕДСТВА ОПИСАНИЯ ДАННЫХ В ЯЗЫКЕ СПЕЦИФИКАЦИЙ СЛЭНГ

С.П.Крицкий, В.Л.Хандрос

Введение

В последнее время программисты всего мира проявляют все больший интерес к выразительным средствам логики предикатов. Явным успехом работ в этом направлении было появление языка ПРОЛОГ и различных его диалектов, в которых логика первого порядка выступает непосредственно как язык программирования. Другим примером применения логики в программировании могут служить работы по использованию логических средств для определения абстрактных типов данных [I-4 и т.п.]. В этом случае логика выступает в качестве языка спецификаций. Попытка объединить эти подходы приводит к понятию "языка спецификаций как языка программирования", т.е. к такому использованию логики, когда программист получает средства не только для точного описания своей задачи (в терминах самой задачи), но и для непосредственного решения этой задачи в удобных и понятных ему терминах без кодирования алгоритма решения в определенном языке программирования.

Настоящая статья посвящена проекту такого языка спецификаций, называемого СЛЭНГ (Specification Language), а именно той его части, которая содержит средства описания данных. Язык СЛЭНГ является основой СЛЭНГ-системы, позволяющей разрабатывать и отлаживать программно-логические спецификации как в диалоговом, так и в пакетном режиме. Диалоговый режим предпочтительнее, поскольку он обеспечивает автоматический переход в режим редактирования спецификаций, что значительно облегчает процесс их создания и отладки. Предполагается, что, кроме самого языка, СЛЭНГ-система включает в себя также базу данных для хранения блоков спецификаций, допускающую различные способы доступа (в том числе и по образцу), систему

автоматического доказательства теорем, позволяющую проверять, обладают ли спецификации теми или иными свойствами, диалоговый редактор, имеющий доступ к базе данных, и систему технологической поддержки проектирования программ, основанную на методе декомпозиции модулей с автоматической генерацией утверждений в стиле Флойда-Хоара для компонент.

В основе механизмов описания данных в языке СЛЭНГ лежит логический подход к определению абстрактных типов данных, базирующийся на многосортных логиках с конструкторами. При этом абстрактные типы данных рассматриваются как многосортные алгебраические системы с (возможно) частичными операциями. Язык позволяет программисту иметь дело со спецификациями (определениями) таких структур, в которых носитель каждого сорта рассматривается как множество термов, построенных из конструкторов этого сорта, а другие операции и отношения вводятся с помощью индуктивных определений [5]. Таким образом, программист работает не с самими объектами, а с термами, эти объекты обозначающими. Это позволяет отлаживать спецификации программ на любом этапе их создания, проводя вычисления на уровне термов.

Язык СЛЭНГ поддерживает модульные иерархические спецификации типов данных (структур). Каждая система задается с помощью определения структуры, в которую вводятся конструируемые сорта объектов и операций и отношения над этими объектами. При определении конструируемых сортов могут использоваться первичные по отношению к ним сорта, определенные в других структурах. Например, для определения сорта "последовательность элементов" нам потребуется сорт "элемент", которым может быть, скажем, сорт целых чисел. Таким образом, на множестве всех определений структур естественным образом возникает некоторая иерархия. Первичные сорта передаются в определения структур посредством параметризации определений структур. На самом деле понятие параметризации шире, чем это описано выше. Передаваться в качестве параметров могут не только сорта, но и любые элементы определений структур, в том числе и сами определения.

Иногда программисту бывает не нужно полностью определять какую-то структуру, а достаточно зафиксировать только некоторые ее свойства. Для этого в языке есть средства, носящие название требований. Требования могут входить в определение структуры, могут определяться самостоятельно. Они могут также выступать в ка-

честве фактических параметров для параметризованных определений структур. Последнее позволяет использовать в языке такое мощное средство, как выбор в соответствии с образцом.

В язык встроены некоторые стандартные типы данных, такие, как множества, списки, записи, массивы, деревья, целые числа и символьные данные. Они могут выступать в качестве первичных сортов для любых определений структур. Доступ к ним осуществляется через стандартные спецификации. В отличие от всех других, встроенные типы данных реализуются непосредственно на ЭВМ и образуют базис, над которым могут быть построены "конкретные" исполнимые программы. Использование всех встроенных типов данных, за исключением множеств, которые не могут быть определены с помощью конструкторов, не обязательно. Программист может определить свои версии структур, соответствующие этим типам данных.

Благодаря наличию встроенных типов данных, у программиста появляется возможность получать обычные программы путем отображения своих структур во встроенные типы. Более того, язык предоставляет возможность отображать друг в друга произвольные структуры или, иначе говоря, строить представления одних структур в других. Программист может отображать свои структуры не непосредственно во встроенные типы, а строить иерархию представлений, нижним уровнем которой могут служить встроенные типы данных. Таким образом, язык предоставляет программисту возможность "строить" свои программы из спецификаций, причем система будет контролировать правильность такого построения.

Соглашения о терминологии

Прежде чем приступать к описанию языка, зафиксируем некоторые общие термины. В целом мы с небольшими изменениями следуем терминологии работы [5].

Под многосортным языком первого порядка с конструкторами будем понимать язык $\langle S_0, S, Cons_S, Select_{Cons}, F, P, \sigma \rangle$, где S_0 – множество первичных сортов; S – множество конструируемых сортов; $Cons_S$ – индексированное семейство множеств конструкторов, для каждого $s \in S$ $Cons_s$ – это множество конструкторов сорта s; $Select_{Cons}$ – индексированное семейство множеств селекторов, для каждого $s \in S$ и каждого $c \in Cons_s$ $Select_c$ – это множество селекторов для конструктора c; F и P – множества функций и отношений соответственно; σ – функция, сопоставляющая каждому элементу из $Cons_S \cup Select_{Cons} \cup F \cup P$ его тип: запись вида $\langle s_1, \ldots, s_n \rangle$.

В дальнейшем мы будем опускать индексы при Cons и Select , если это не будет приводить к двусмысленности.

Пусть Σ - многосортный язык первого порядка с конструкторами. Базисным языком языка Σ будем называть подъязык $\Sigma_0 = \langle S_0, S, Cons \rangle$ языка Σ.

Структурой на множестве A будем называть совокупность $\langle A, F, P \rangle$, где F и P - множества функций и отношений на A.

Σ-структурой (на A) будем называть структуру $\langle A, F', P' \rangle$, для которой существует такая интерпретация I языка Σ, что

1) $A = \bigcup_{s \in S} A_s$, где $A_s = I(s)$ (носитель сорта s); $A_s \neq \emptyset$;

2) $F' = \{I(f)/f \in F\}$, причем если $\sigma(f) = \langle s_1, \ldots, s_n, s \rangle$, то $I(f): A_{s_1} x \ldots x A_{s_n} \to A_s$;

3) $P' = \{I(p)/p \in P\}$, причем если $\sigma(p) = \langle s_1, \ldots, s_n \rangle$, то $I(p) \subseteq A_{s_1} x \ldots x A_{s_n}$.

Множество A будем называть носителем этой Σ-структуры.

Абстрактной Σ-структурой языка Σ называется Σ-структура, носителем которой является множество всех термов базисного языка.

Остальные термины будут введены по ходу изложения.

Замкнутые определения структур

Абстрактные типы данных рассматриваются в языке СЛЭНГ как некоторые абстрактные структуры. Более точно, абстрактный тип данных со множеством сортов объектов S, множеством операций F и множеством отношений P - это некоторая абстрактная $\langle S_0, S, Cons, Select, F, P, \sigma \rangle$-структура, где S_0 - множество (возможно, пустое) первичных сортов, участвующих в конструировании сортов из S, Cons и Select - семейства множеств конструкторов и селекторов соответственно, и σ - функция, определяющая тип операций и отношений. Отсюда легко видеть, что для задания спецификации абстрактного типа данных необходимо специфицировать его синтаксическую часть - некоторый многосортный язык первого порядка с конструкторами и задать определения его операций и отношений. Так как тип данных - это абстрактная структура, носитель определяется синтаксически как множество термов базисного языка. В дальнейшем мы будем использовать термин "структура", опуская слово "абстрактная", поскольку речь будет идти только о таких структурах; термин "тип данных" - вместо термина "абстрактный тип данных" и термин "определение структуры" - вместо термина "спецификация абстрактного типа данных".

Как уже отмечалось, в определении структуры могут участвовать (за счет параметризации) элементы определений других структур. Такие определения будем называть открытыми, а остальные - замкнутыми. В этом разделе речь пойдет только о замкнутых определениях структур.

Итак, замкнутое определение структуры состоит из определения языка структуры и задания ее операций и отношений. Определение языка задается в явном виде простым перечислением имен сортов, конструкторов (возможно, с указанием имен селекторов), операций и отношений с указанием их типов. Определение операций и отношений осуществляется с помощью индуктивных определений [5], т.е. предложений вида $f(x_1,\dots,x_n) = y \leftrightarrow \varphi(x_1,\dots,x_n,y)$ для операций и предложений вида $p(x_1,\dots,x_n) \leftrightarrow \varphi(x_1,\dots,x_n)$ для отношений, где f и p - n-местные операция и отношение, а φ - правильно построенная формула некоторого специального вида.

ПРИМЕР. Определение структуры NAT натуральных чисел:

```
define struct  NAT
  sorts nat
   constr  0:→nat
           s:nat→nat
  operations
    -+-,-⁻-,-*-,-/-:⟨ nat nat ⟩→ nat
  relations
    -≤- :⟨ nat,nat ⟩
  specifications
     var x,y,z  over  nat
  +: x+y=z↔ x=0 & z=y ∨∃u(x=s(u)& z=s(u+y))
  -: x-y=z↔ x=z+y
  *: x*y=z↔ x=0 & z=0 ∨∃u(x=s(u) & z=y+(u*y))
  /: x/y=z↔ x=z*y  & y≠0
  ≤: x≤y↔ x=0∨∃u,v(x=s(u)&  y=s(v) & u ≤v)
 end  NAT
```

В данном примере тип NAT натуральных чисел определяется как некоторая абстрактная $\langle \emptyset,\{nat\},\{0,s\}, Select,\{+,-,*,/\},\{\le\},\sigma\rangle$-структура. Семейство селекторов в данном случае явно не задано. Однако если обозначить селектор для конструктора s через s^- (конструктор 0 селекторов, очевидно, не имеет поскольку, является нуль-арным конструктором), то семейство Select будет состоять из единственного множества $\{s^-\}$. Функция σ определяется следующим образом:

$\sigma(0) = \langle nat \rangle$, $\sigma(s) = \langle nat, nat \rangle$,

$\sigma(+)=\sigma(-)=\sigma(*)=\sigma(/)= \langle nat, nat, nat \rangle$, $\sigma(\leq)= \langle nat, nat \rangle$.

Синтаксис для операций и отношения использует инфиксную форму записи. Вообще для определения синтаксиса может использоваться любая форма записи, в том числе и "миксфиксная".

Заметим, что данная спецификация исполнима в том смысле, что для любой операции (отношения) по этой спецификации может быть построен алгоритм, который по любым термам базисного языка (элементам носителя структуры) строит такой терм базисного языка, что при подстановке этих термов в соответствующее определение операции (отношения) мы получим истинное утверждение. В случае операций такой терм будем называть значением операции на данных аргументах. В общем случае, если такого терма не существует (операция не определена на данных аргументах), алгоритм может искать его бесконечно долго. В данном же случае можно показать, что алгоритм обязательно остановится после конечного числа шагов (при этом если значение операции существует, то оно будет найдено).

Можно сделать еще несколько замечаний относительно определений операций и отношения в данном примере.

Во-первых, эти определения могут быть записаны в более привычной для программиста нотации с помощью условных и выбирающих выражений. Например, определение операции "+" могло бы выглядеть так:

а) x+y=**if** x=0 **then** y
elif x=s(u) **then** s(u+y) **fi** .

Заметим, что мы используем "неполное" условное выражение - в подвыражении **if** x=s(u) **then** s(u+y) отсутствует **else**-ветвь. Это позволяет нам определять частичные операции. (В данном случае операция "+" тотальна, поскольку любой объект сорта nat либо является нулем, либо получен применением конструктора s к некоторому другому объекту этого сорта. Таким образом, в данном случае мы могли бы написать просто

x+y=**if** x=0 **then** y **else** $s(s^{-}(x)+y)$ **fi**,

где s^{-} - селектор для конструктора s.)

б) x+y=**case of** x=0::y
x=s(u)::s(u+y)
end case .

В силу приведенных выше рассуждений мы могли бы написать просто

```
x+y = case of x=0: y
          otherwise s(s⁻(x)+y)
      end case.
```

Все приведенные формы записи эквивалентны с точки зрения их исполнимости и потому равноправны. Однако та форма, которая была использована в тексте примера, обладает тем преимуществом, что она является логической формулой и может быть использована для доказательства. Конечно, любая из этих форм записи может быть приведена к виду логической формулы.

Во-вторых, эти определения могут быть легко оттранслированы в следующие предложения языка ПРОЛОГ:

```
+(0,x,x):--
+(sx,y,sz):-- +(x,y,z).
```

В-третьих, из этих определений могут быть доказаны такие свойства операций и отношений, как коммутативность и ассоциативность операций "+" и "*", транзитивность отношения "≤".

В замкнутые определения структур, кроме аксиом, определяющих операции и отношения, может входить еще одно множество аксиом произвольного вида, выражающих требования, которым эти операции и отношения должны удовлетворять. Эти аксиомы могут использоваться системой в качестве некоторого критерия правильности определений, а также как вспомогательное средство при доказательстве утверждений относительно объектов определяемой структуры. Подробнее мы остановимся на их описании при обсуждении механизма требований.

Конструкторы и селекторы

Мы уже сказали о том, что носителями рассматриваемых структур являются множества термов базисных языков этих структур, т.е. термов, построенных из конструкторов (и, возможно, также термов первичных сортов). Под конструкторами мы понимаем некоторые всюду определенные функции, удовлетворяющие условию однозначной конструируемости - конструкторы инъективны, и множества их значений не пересекаются, и условию конечной конструируемости - любой терм получается с помощью конечного числа применений конструкторов. Эти довольно жесткие ограничения делают носитель каждого сорта рекурсивным множеством, что позволяет считать, что для каждого сорта объектов структуры неявно определен рекурсивный характеристический предикат.

Более того, мы могли бы считать, что для каждого сорта объектов структуры задана некоторая контекстно-свободная грамматика термов, единственным нетерминальным символом которой является имя этого сорта, и рассматривать конструкторы не как функции, а как терминальные символы этой грамматики. В некоторых случаях такая трактовка конструкторов удобна и, по существу, мы используем ее, задавая базисный язык структуры. Однако в общем случае она неудовлетворительна, поскольку при определении отображения одной структуры в другую мы должны будем рассматривать образы конструкторов при этом отображении именно как функции.

Нуль-арные конструкторы и конструкторы, в область определения которых входят только первичные сорта, будем называть константными. Заметим, что в силу условия однозначной конструируемости множество всех термов одного сорта является фундированным по отношению "быть значением конструктора", или, в терминах термов, "быть подтермом". При этом константные конструкторы определяют минимальные элементы.

Наряду с конструкторами программист может использовать обратные к ним селекторные функции (селекторы). Предполагается, что для любого конструктора селекторные функции определены неявно, и предусматриваются два способа именования таких функций, позволяющих задать для каждого селектора общее и индивидуальное имя. Пусть cons - имя n-арного конструктора некоторого сорта. Тогда общим именем для его i-го селектора будет имя cons⁻i (для унарного конструктура - просто cons⁻). Индивидуальные имена присваиваются селекторам при задании базисного языка в определении структуры. Опять пусть cons - имя n-арного конструктора сорта s. Тогда при задании имени самого конструктора cons индивидуальное имя его i-му селектору может быть присвоено следующим образом: cons: $\langle s_1, \ldots, s_{i-1}$, select: $s_i, s_{i+1}, \ldots, s_n \rangle \to s$, где select - индивидуальное имя селектора.

ПРИМЕР. Определение структуры стеков:

```
define struct STACK (elem: sort)
    sorts stack
      constr nil:→ stack
             push: ⟨ top:elem,pop:stack ⟩ → stack
end STACK
```

В данном примере определяется структура, соответствующая обычному понятию стека. Три традиционные операции над стеками описываются как конструктор push и два его селектора top и pop.

Заметим, что при этом, благодаря свойствам конструкторов и селекторов, операции top и pop не применяются к пустому стеку nil.

Первичные сорта

Мы говорим, что некоторый сорт является первичным по отношению к данной структуре, если он является конструируемым сортом некоторой другой структуры или сортом встроенного типа данных и в данной структуре используется для построения объектов конструируемых сортов. Первичные сорта могут использоваться только в параметризованных определениях структур. В приведенном выше примере элементами описываемых стеков являются объекты сорта elem. Этот сорт является первичным по отношению к структуре STACK и передается в нее в качестве параметра. На использование первичных сортов накладывается единственное ограничение - при определении сортов не должно быть циклов, т.е. при определении некоторого конструируемого сорта нельзя использовать в качестве первичного такой сорт, для определения которого применялись сорта, использующие этот конструируемый сорт в качестве первичного.

Операции над носителями структур

Конструируемые сорта могут строиться не только с помощью конструкторов, но также с помощью теоретико-множественных операций над носителями первичных сортов. Допускаются следующие операции: объединение, пересечение, разность, взятие подмножества, декартово произведение и факторизация на основе некоторого отношения эквивалентности. В случае операций факторизации и выделения подмножества на основе характеристического свойства рассматриваемое отношение может выражаться в терминах отношений и операций над первичными сортами с помощью индуктивных определений. Проверка рекурсивности этого отношения возлагается на программиста.

Индуктивные определения

Формула φ называется позитивной относительно функционального символа f (предикатного символа p), если ее дизъюнктивная нормальная форма не содержит подформул вида

$$\neg f(t_1,\ldots,t_n)=t \quad (\neg p(t_1,\ldots,t_n)).$$

Формула φ называется экзистенциальной, если в префикс ее пренексной формы входят только кванторы существования.

Индуктивным определением операции f (отношения p) называется формула вида $f(x_1,\dots,x_n)=y \leftrightarrow \varphi(x_1,\dots,x_n,y)$ $(p(x_1,\dots,x_n) \leftrightarrow \leftrightarrow \varphi(x_1,\dots,x_n))$, где φ – позитивная относительно определяемых символов экзистенциальная формула. (Определяемыми символами в некотором определении структуры являются все символы операций и отношений, вводимые в этом определении.) Эту формулу будем называть посылкой определения.

В работе [5] показано, что операционная семантика логического вывода для спецификаций, основанных на индуктивных определениях, обеспечивает перечислимость (вычислимость) определяемых отношений и операций. Это обосновывает наше утверждение об исполнимости спецификаций в языке СЛЭНГ.

С другой стороны, одной перечислимости часто бывает недостаточно, нужна рекурсивность. Например, при определении отношения эквивалентности, на основе которого будет проводиться факторизация. С этой целью мы выделяем (синтаксически) класс индуктивных определений, обеспечивающих рекурсивность определяемых отношений и операций.

Заметим, что термы естественным образом представляются деревьями. Так, терм вида $f(t_1,\dots,t_n)$ представляется корневым упорядоченным деревом, корень которого помечен символом f. Сыновьями корня являются деревья, представляющие термы $t_1,\dots,t_n$ и упорядоченные слева направо по возрастанию индексов при термах. Символ, помечающий корень дерева, представляющего некоторый терм, будем называть головой этого терма. Глубиной терма t относительно символа f будем называть высоту максимального поддерева, корень которого помечен символом f, в дереве, представляющем терм t. Общей глубиной терма будем называть его глубину относительно его головы.

Наряду с обычными термами, мы будем использовать термы с переменными. Множества свободных переменных терма t и формулы φ будем обозначать $FV(t)$ и $FV(\varphi)$ соответственно.

Пусть f и g – символы операций или отношений. Будем говорить, что f непосредственно зависит от g, если символ g входит в посылку определения f. Отношение зависимости между функциональными и предикатными символами получается рефлексивно-транзитивным замыканием отношения непосредственной зависимости.

Пусть $f \in F$, $\sigma(f) = \langle s_1,\dots,s_n,s \rangle$. Будем говорить, что операция f определена индукцией по структуре аргументов, если посылка определения f имеет вид $\varphi \equiv \varphi_1 \vee \dots \vee \varphi_m$, где φ_i – это формула вида

$$(*) \qquad \exists \bar{u} \quad \&_{1\le j\le k,\, 1\le i_j\le n} \; x_{i_j} = t_j \;\&\; y = t \;\&\; \phi \;,$$

где $\bar{u}$ – совокупность различных переменных $\bar{u} \cap \{x_1,\dots,x_n\} = \emptyset$; t_j – термы сортов s_{i_j} ; t – терм сорта s; ϕ – позитивная относительно определяемых символов формула;

$$FV(t_j) \subseteq \bar{u} \cup \{x_1,\dots,x_n\} \setminus \{x_{i_j}\};$$

$$FV(t) \subseteq \cup_j FV(t_j) \cup \{x_i \mid \forall_j\, i \neq i_j\};$$

$$FV(\phi) \subseteq \cup_j FV(t_j) \cup \{x_i \mid \forall_j\, i \neq i_j\} \cup \{y\};$$

и t_j не входят в t в качестве подтермов.

Будем говорить, что операция f определена индукцией по структуре объектов, если для всех j термы t_j, входящие в $(*)$, являются термами базисного языка с переменными (объектными термами), и что f определена индукцией по структуре функций – в противном случае.

Пусть операция f определена индукцией по структуре объектов. Будем говорить, что определение для f допускает конечное вычисление, если для всех φ_i в $(*)$ выполняется следующее условие: либо t и все термы, входящие в ϕ, являются объектными термами, либо если они содержат подтерм вида $g(t_1',\dots,t_n')$ и g зависит от f, то глубина этого подтерма относительно g меньше или равна глубине терма $f(x_1,\dots,x_n)[t_j/x_{i_j}]$ относительно f и определение для g допускает конечное вычисление.

Пусть операция f определена индукцией по структуре функций. Будем говорить, что определение для f допускает конечное вычисление, если для всех φ_i в $(*)$ выполняется следующее условие: если термы t_j, терм t и термы, входящие в ϕ, содержат подтермы вида $g_i(t_1',\dots,t_n')$, то определения для g_i допускают конечное вычисление, и если при этом g_i зависят от f, то глубина этих подтермов относительно g_i меньше или равна глубине терма $f(x_1,\dots,x_n) \times [t_j/x_{i_j}]$ относительно f.

Для предикатных символов определения, допускающие конечные вычисления, определяются аналогичным образом.

Можно показать, что такие определения задаются общерекурсивными операциями и рекурсивными отношениями. Более того, легко заметить, что такие определения могут быть оттранслированы в предложения языка ПРОЛОГ. Причем вид наших определений гарантирует, что даже при эффективной (в глубину) стратегии поиска доказательства этот процесс не является расходящимся.

Параметризованные определения структур

Как уже отмечалось ранее, программист имеет возможность строить новые определения структур, используя уже имеющиеся. Одним из средств, обеспечивающих такую возможность, является параметризация определений. Примером параметризованного определения структуры может служить приведенное выше определение структуры STACK. В заголовке этого определения явно указано, что имя сорта elem является формальным параметром, т.е. выступает в роли переменной на множестве имен сортов объектов, а не в качестве имени конкретного сорта. Теперь определение структуры стеков натуральных чисел может быть получено как конкретизация определения структуры STACK путем подстановки в качестве фактического параметра сорта nat структуры NAT:

```
define struck STACK_OF_NAT
   is STACK (nat of NAT)
end STACK_OF_NAT.
```

При этом все вхождения имени elem в определение структуры STACK будут заменены на вхождения имени nat of NAT, т.е. будет произведена обычная текстуальная подстановка и новая структура получит имя STACK_OF_NAT.

Такое использование параметризованных определений ориентировано на построение определений структур по принципу "сверху вниз", когда программист сначала осознает потребность в определении класса однородных структур, затем строит единственное параметризованное определение структуры и после этого получает определения интересующих его структур как различные конкретизации этого параметризованного определения.

Предположим теперь, что у нас нет определения структуры STACK и что мы хотим сразу определить стеки натуральных чисел. Для этого мы могли бы воспользоваться следующим определением:

```
define struct STACK_OF_NAT
 sorts stack,nat of NAT
  constr nil:→ stack,push: ⟨ top:nat,pop:stack ⟩→ stack
end STACK_OF_NAT
```

В этом случае имя nat является именем конкретного сорта, а определение задает конкретный вид стеков - стеки натуральных чисел. Однако если мы захотим теперь определить стеки целых чисел, то ничто не мешает нам заменить сорт nat на сорт int, определенный в некоторой структуре INT:

```
define struct STACK_OF_INT
   is STACK_OF_NAT with nat => int of INT
end STACK_OF_INT
```

В результате мы получим определение стеков целых чисел, отличающееся от определения стеков натуральных чисел только тем, что все вхождения имени nat в определение структуры STACK_OF_NAT будут заменены на вхождения имени int в определении структуры STACK_OF_INT. Таким образом, мы видим, что хотя имя nat и является именем конкретного сорта, оно тем не менее может выступать в роли формального параметра определения стека. В этом случае мы будем говорить, что значение формального параметра задано по умолчанию (в данном случае им является сорт nat структуры NAT).

Такие параметризованные определения могут использоваться при построении определений структур по принципу "снизу вверх", когда программист сначала строит конкретное определение структуры, а затем, если ему понадобится определить аналогичную структуру, рассматривает это определение как параметризованное с формальными параметрами, заданными по умолчанию.

Мы рассмотрели примеры, когда в качестве формального параметра выступал сорт объектов. На самом деле формальным параметром определений может быть любой элемент структуры, в том числе и полная структура.

Вынесение формального параметра в заголовок определения структуры означает, что значение этого параметра по умолчанию не задается. В этом случае программист должен явно указывать вид этого параметра (в случае стека явно указывается, что в качестве фактического параметра будет подставлено имя сорта объектов некоторой структуры).

Особый случай представляет собой подстановка в качестве фактического параметра элементов неконкретизированного параметризованного определения структуры. В этом случае конкретизация состоит в построении нового параметризованного определения, вид формальных параметров которого определяется этим фактическим параметром. То же самое происходит, когда фактические параметры подставляются не для всех формальных параметров, значения которых не заданы по умолчанию.

Программист может использовать механизм именования для того, чтобы синтаксически разделять различные конкретизации одного и того же параметризованного определения структуры. Например, строки

символов могут быть определены с помощью такой последовательности определений:

```
define struct SEQ
   is STACK with
   replace (stack → seq,push→_._,top → head,pop → tail)
end SEQ,
```

SEQ - структура последовательностей элементов произвольного вида;

```
define struck STRINGS
   is SEQ (char of SYMBOLS) with
   replace (seq → strings,_._ → __,
            head → first,tail → rest)
end STRINGS
```

STRINGS - конкретизация определения структуры SEQ . В качестве фактического параметра подставляется сорт char некоторой структуры SYMBOLS . Такое определение структуры STRINGS эквивалентно следующему параметризованному определению с формальным параметром, заданным по умолчанию:

```
define struct STRINGS
   sorts strings,char of SYMBOLS
    constr nil: → strings
           __:⟨ first: char,rest: strings ⟩ → strings
end STRINGS.
```

Требования

Часто бывают ситуации, когда программист, уже зная общие свойства определяемых им объектов, еще не может точно определить их. С другой стороны, возможны ситуации, когда ему на самом деле и не нужно знать всех особенностей объектов некоторой структуры, а достаточно знать,что они обладают определенной группой свойств. Так, например, при работе с массивами нам зачастую достаточно знать только, что множество их индексов линейно упорядочено, при этом нет необходимости вдаваться в конкретную его природу.

В таких случаях язык СЛЭНГ позволяет программисту охарактеризовать определяемые им структуры относительно интересующей его группы свойств, не задавая определения структуры в целом. Такое задание структуры называется определением требований.

Существуют два вида требований - встроенные и внешние. В обоих случаях их природа одинакова - группа аксиом произвольного

вида, задающая свойства определяемых объектов. Однако в первом случае требования являются частью определения структуры и могут служить в качестве некоторого критерия правильности определения. В качестве такого рода требований мы могли бы включить в определение структуры NAT обычные аксиомы, выражающие коммутативность, ассоциативность и дистрибутивность операций "+" и "*", а также транзитивность отношения "≤". При обработке такого определения система будет пытаться доказать эти утверждения, используя в качестве аксиом спецификации операций и отношений, а также аксиомы для конструкторов. В случае неудачи она выдаст диагностическое сообщение о том, что структура определена неправильно. В противном случае аксиомы требований будут в дальнейшем использоваться системой для доказательства утверждений о натуральных числах наряду с аксиомами, определяющими операции или отношения.

Внешние требования играют в языке несколько иную роль. Они могут предъявляться к любым элементам структур, в том числе и к полным структурам.

При определении внешних требований программист должен указать имя этих требований, некоторый образец того объекта, к которому эти требования предъявляются, и группу аксиом, выражающих сами требования. Задание образца внешне ничем не отличается от задания той части языка структуры, которая соответствует объекту требований, за исключением того, что образец задает некоторую схему определения соответствующего элемента, в которой часть специфицированных имен выступает в роли переменных над соответствующими объектами, а остальные имена (отмеченные специальным образом) играют роль заданных по умолчанию параметров, аналогичных параметрам параметризованных определений.

ПРИМЕР.

```
define requirements ORD
   pattern of structure
     sorts obj
     relations _R_: ⟨ obj,obj ⟩
end pattern
 axioms
    var x,y,z over obj
    x R x,
    x R y & y R x → x = y
    x R y & y R z → x R z
end ORD
```

Данный пример является иллюстрацией определения требований, предъявляемых, как указывает образец, к некоторой структуре. Эта структура должна содержать по крайней мере один конструируемый сорт, на котором должно быть определено некоторое бинарное отношение. Все эти сведения содержатся в описании образца. Имена (obj и R) являются переменными, принимающими значения на множествах сортов и бинарных отношений соответственно. В том случае, когда необходимо определить требования к структуре, в которой имеется конструируемый сорт как раз с именем obj, мы будем использовать запись sorts obj: fixed, указывая на то, что имя obj является фиксированным. Отмеченные таким образом имена можно рассматривать как формальные параметры определения требований, заданные по умолчанию, аналогично тому, как это делается для параметризованных определений.

При попытке показать, что некоторая структура удовлетворяет требованиям, описание языка структуры будет сопоставляться с образцом, и если в нем найдутся соответствующие образцу элементы (в данном случае сорт и определенное на нем бинарное отношение), то система будет пытаться доказать, что аксиомы требований (в данном случае обычные аксиомы частичного порядка) являются следствием аксиом, определяющих операции и отношения, аксиом для конструкторов и аксиом встроенных требований. В случае удачи эти аксиомы будут перенесены в структуру в качестве дополнительных встроенных требований.

При разработке спецификаций внешние требования могут выполнять различные функции. Во-первых, они могут служить "техническими заданиями" на разработку соответствующих определений структур. Во-вторых, используя требования, программист может выбрать среди имеющихся (например, в базе данных) нужные ему определения структур, опираясь на "содержательно" сформулированные критерии (выбор из базы данных по образцу). В-третьих, внешние требования могут указываться в качестве фактических параметров при конкретизации параметризованных определений структур. В этом случае конкретизация будет состоять в построении нового параметризованного определения, параметры которого должны будут удовлетворять этим требованиям. При последующих конкретизациях такого определения фактические параметры будут проверяться на соответствие заданным требованиям, и в случае несоответствия система будет выдавать диагностическое сообщение о невозможности конкретизации этого определения структуры данными параметрами.

Расширения определений

В языке СЛЭНГ программист может определять расширения определений структур, добавляя к уже имеющимся определениям новые элементы или доопределяя уже существующие.

Определение расширения содержит список расширяемых определений структур, имя получаемого расширения и дополнительные определения элементов структур. К последним относятся определения сортов, операций и отношений и встроенных требований. Построение расширения происходит следующим образом. Расширяемые определения переносятся элемент за элементом в новое определение. При этом если для некоторого элемента имеется дополнительное определение, то определение этого элемента соответствующим образом расширяется. Например, к определениям сортов могут добавляться новые конструкторы, к определениям операций и отношений - новые индуктивные определения. После того как все элементы расширяемых определений уже перенесены, переносятся дополнительные определения, вводящие новые сорта, операции, отношения и встроенные требования, и новому расширению присваивается новое имя.

ПРИМЕР.

```
define struct LEXORD
  is extension of SEQ (obj of ORD)
  with replace R of ORD ↦_ ≤ _
  operations
     sort: seq → seq
    _without_: ⟨ seq, obj ⟩ → seq
  relations
     sorted: seq
    _perm_: ⟨ seq, seq ⟩
    _include_: ⟨ seq,seq ⟩
    _member_: ⟨ obj,seq ⟩
    _≤≤ _: ⟨ seq,seq ⟩
  specifications var x,y over seq, e over obj
     sort: sort (x) = y⟷ sorted (y) & x perm y,
     sorted: sorted (x) is the case of
                        x = nil,
                        let e = head (x),u = tail (x) in
                        (u = nil,
                         e ≤ head (u) & sorted (u))
                         end case,
```

```
perm: x perm y⟷ x include y & y include x,
include: x include y is the case of
                    y = nil,
                    x = y,
                    let e = head (y) in
                    e member x & (x without e) include (tail (y))
                    end case,
member: e member x⟷ e = head (x) ∨ e member (tail (x)),
without: x without e = if x = nil then nil
                       else let e1 = head (x), u = tail (x) in
                       if e = e1   then u
                       else e1.  (u without e)
                       fi
                     fi,
        << :  x<< y is the case of
                    x = nil,
                    let e1 = head (x), e2 = head (y) in
                    e1 ≤ e2 & (e1 = e2 → tail (x) << tail(y))
                    end case

end LEXORD
```

З а к л ю ч е н и е

Язык СЛЭНГ не ориентирован на поддержку какой-либо определенной методологии проектирования. Выше в тексте было показано, что он может поддерживать как восходящую, так и нисходящую технологии. Кроме того, использование механизма расширений в какой-то мере согласуется с техникой сосредоточенного описания рассредоточенных действий, известной как РД-технология. Мы надеемся,что использование этого языка позволит точно описать все стадии проек - тирования программы, начиная с формулировки требований,предъявляемых к проекту, и кончая построением программы на некотором алго - ритмическом языке программирования. Последнее требует наличия в языке развитых средств представлений одних структур другими, и разработка таких средств является ближайшей задачей на будущее.

Заметим, что определение операции sort из структуры LEXORD не задает конкретного алгоритма ее исполнения. В этом смысле мы можем говорить, что структура LEXORD специфицирует задачу сортировки последовательности элементов, на которых определен частич -

ный порядок (требования ORD). С другой стороны, мы могли бы определить операцию сортировки следующим образом:

```
sort(x) = if sorted(x) then x
          else let e = head(x)  u = tail(x) in
               if sorted(u)
               then head(u).sort(e.tail(u))
               else sort(e. sort(u)) fi
          fi
```

Такое определение явно указывает, что для исполнения этой операции используется конкретный алгоритм сортировки, а именно сортировка вставкой. Если бы мы использовали в определении структуры LEXORD такое определение этой операции, то можно было бы говорить о том, что LEXORD специфицирует алгоритм (программу) сортировки последовательности.

Возможность представления одних структур другими, гарантирующая некоторым образом правильность такого представления, позволит строить "правильные" спецификации программ (и собственно программы, представляя структуры во встроенных типах данных), исходя из "правильных" спецификаций задач. А некоторая уверенность в правильности последних может быть получена (в силу исполняемости спецификаций) с помощью обычного тестирования, аналогично тому, как это делается для обычных программ.

Литература

I. CARTWRIGHT R. Towards a Logical Theory of Program Data. - LNCS,1982,v.131,p. 37-51.

2. NOURANI F. A model-theoretic approach to specification,extension and implementation.-LNCS,1980,v.83,p.282-297.

3. BERTONI A.,MAURI G.,MIGLIOLI P. Towards a theory of abstract data types: a discussion on problems and tools. - LNCS,1980, v.83,p.44-58.

4. NAKAJIMA R.,HONDA M., NAKAHARA H. Hierarchical Program Specification and Verification - A Many-sorted Logical Approach.- Acta Informatica,1980,v.14, p.135-155.

5.КРИЦКИЙ С.П. Индуктивные определения в логических спецификациях.- В кн.: Логические вопросы теории типов данных (Вычислительные системы,вып.II4). Новосибирск,1986,с.40-58.

Поступила в ред.-изд.отд.
22 октября 1985 года