

ИНФОРМАТИКА

УДК 519 : 681

*А. П. Бельтюков***ПРОСТЫЕ ТИПИЗИРОВАННЫЕ ФУНКЦИОНАЛЬНЫЕ
ЯЗЫКИ ПРОГРАММИРОВАНИЯ**

Описан способ построения простых функциональных языков программирования, пригодных для определения классов рекурсивных функций ограниченной вычислительной сложности. Все обрабатываемые данные в языке обязательно ограничены типами, которые также могут быть построены средствами языка. Языки рассчитаны как на изучение сложных классов алгоритмов, так и на расширения для практического программирования.

Ключевые слова: языки программирования, функциональное программирование, вычислительная сложность алгоритмов.

Введение

В настоящей работе предлагается способ построения очень простых, но мощных языков программирования, которые можно использовать для определения алгоритмов ограниченной вычислительной сложности. Предлагаемые языки имеют жесткую типизацию данных. Более того, кроме обычных ограничений на типы данных возможно еще задание дополнительных ограничений, при которых задаются подмножества типов, в которые должны попадать одни данные при аналогичных ограничениях на другие данные.

Мы будем описывать программы, работающие с типами данных, которые принято называть конечными деревьями или термами. Конечно, при практическом использовании предлагаемой идеи можно применять и полностью абстрактные типы, поставляемые извне системы программирования.

Программа на любом из предлагаемых языков программирования представляет собой множество «предложений», каждое из которых описывает некоторое множество или отношение. Некоторые описываемые отношения являются алгоритмически задаваемыми функциями. Часть описаний такой функции и представляет собой ее алгоритм. Остальные описания предназначены для задания ограничений применимости функций и для гарантий того, что задаваемые алгоритмические процессы не слишком сложны.

Настоящая работа является продолжением работ автора [1–6].

§ 1. Бесконтекстный синтаксис языков

Определим бесконтекстный синтаксис строящегося языка программирования — множество «программ». Будем считать, что зафиксировано специальное множество N — множество «имен». В примерах в качестве имен будем рассматривать цепочки специальных символов — «букв», в качестве этих «букв» будем использовать строчные и прописные латинские буквы. Будем считать, что в алфавите языка имеются и другие символы, не встречающиеся в именах: знак равенства, круглые скобки, запятая и точка. В синтаксических определениях эти терминальные символы будем записывать так:

$$= () , .$$

Множество «программ» будем обозначать через F , через U — множество «предложений» этих программ, через T — множество «термов» (выражений), встречающихся в этих программах, через L — множество «списков» этих термов. Синтаксис этих множеств определяется следующим образом:

$$\begin{aligned} F &::= U, F \mid . \\ U &::= N=T \mid N(L)=T \\ T &::= N \mid N(L) \\ L &::= T \mid T, L \end{aligned}$$

Для наглядности приведем пример простейшей программы, в которой определяется понятие примитивной записи натурального числа в «палочковой» системе счисления и вводится операция сложения таких чисел:

```
Number = ZERO,
Number = NEXT(Number),
add(Number, Number) = Number,
add(x, ZERO) = x,
add(x, NEXT(y)) = NEXT(add(x, y)).
```

Эта программа состоит из пяти предложений, каждое из которых в примере занимает по одной строке. Предложения для удобства понимания программы сгруппированы в три части. В первой части описывается тип данных *Number* — записи чисел в «палочковой» системе счисления. Примеры таких записей чисел: *ZERO* — 0, *NEXT(ZERO)* — 1, *NEXT(NEXT(ZERO))* — 2 и так далее. Это описание является индуктивным и состоит, как обычно, из двух предложений, первое из которых описывает 0, а второе позволяет построить следующее число, если уже имеется предыдущее. Вторая часть программы описывает определяемую функцию как отображение множества пар записей натуральных чисел в

натуральные числа. Последняя часть программы задает функцию сложения *add* в соответствии с традиционной схемой примитивной рекурсии:

$$\begin{aligned}x + 0 &= x, \\x + (y + 1) &= (x + y) + 1.\end{aligned}$$

Такое определение функции называется рекурсивным. Например, вычисление $1 + 2$ в соответствии с этим определением можно записать в виде:

```
add(NEXT(ZERO), NEXT(NEXT(ZERO))) =
NEXT(add(NEXT(ZERO), NEXT(ZERO))) =
NEXT(NEXT(add(NEXT(ZERO), ZERO))) =
NEXT(NEXT(NEXT(ZERO)))
```

В программах выделяются два рода предложений. Предложения первого рода имеют вид $N = T$. Это «описание типа». Имя перед знаком равенства называется именем «типа». В приведенном примере это имя *Number*. Для наглядности эти имена в примерах будем писать с прописной буквы. Предложения второго рода имеют вид $N(L) = T$. Это описание «соответствия». Имя перед скобкой называется именем «соответствия». В приведенном примере это имя *add*.

Для удобства чтения программы предложения, начинающиеся с одного имени, будем писать подряд. Имена, встречающиеся только в правых частях описаний типа, называются «символами». В приведенном примере это имена *ZERO* и *NEXT*. Для наглядности имена символов в примерах будем писать прописными буквами.

Все остальные имена в программе являются именами переменных. В приведенном примере это имена x и y . Для наглядности имя каждой переменной в примерах будем записывать в виде одной строчной буквы.

Соответствия делятся на «функции» и «ограничения». «Ограничениями» называются соответствия, имеющие описания с одинаковыми левыми частями (до знака равенства) или имеющие описания, содержащие имена типов в правой части (после знака равенства) и при этом не содержащие таких имен в левой части. Остальные соответствия называются функциями. К описаниям функций далее предъявляются некоторые дополнительные требования.

Имя *add* в приведенном ранее примере задает функцию. Описаний ограничений в приведенном примере нет. Примеры описаний ограничений могут быть получены добавлением к приведенному примеру следующих предложений:

```
BNumber(Number) = Number,
BNumber(ZERO) = ZERO,
BNumber(NEXT(x)) = ZERO,
BNumber(NEXT(x)) = NEXT(BNumber(x)),
```

```

min(Number,Number) = Number,
min(x,y) = BNumber(x),

min(ZERO,y) = ZERO,
min(NEXT(x),ZERO) = ZERO,
min(NEXT(x),NEXT(y)) = NEXT(min(x,y)),

subt(Number,Number) = Number,
subt(x,BNumber(x)) = BNumber(x),

subt(ZERO,ZERO) = ZERO,
subt(NEXT(x),ZERO) = NEXT(subt(x,ZERO)),
subt(NEXT(x),NEXT(y)) = NEXT(subt(x,y)),

Nstack = BOTTOM,
Nstack = TOP(Number,Nstack),

BNstack(Number) = Nstack,

BNstack(ZERO) = BOTTOM,
BNstack(NEXT(x)) = TOP(Number,BNstack(x)).

```

Здесь $BNumber$ и $BNstack$ — имена ограничений, $BNumber(x)$ — натуральные числа, не превосходящие x , $Nstack$ — имя типа «числовой стек», $Nstack(x)$ — числовой стек заданного размера x . Функция min дает минимум аргументов, для которого данная программа гарантирует ограничение сверху первым аргументом. Функция $subt$ выполняет вычитание из одного своего аргумента второго аргумента, ограниченного первым. Результат при этом также ограничен первым аргументом.

Описания функций, не содержащие имен типов и ограничений, будем называть «определяющими». Они образуют «алгоритм» функции. В приведенном примере определяющие описания это

```

add(x,ZERO) = x
add(x,NEXT(y)) = NEXT(add(x,y))

```

Остальные описания будем называть «ограничивающими». Например, это описание

```

add(Number,Number) = Number

```

Для наглядности имена функций в примерах будем писать строчными буквами. Имена ограничений для наглядности будут начинаться с пары прописных букв. Число термов в скобках после имени соответствия называется числом его аргументов. В приведенном примере функция add имеет два аргумента, а ограничения $BNstack$ и $BNumber$ — по одному.

§ 2. Общие контекстные условия

Конкретный вид языка программирования определяется дополнительными («контекстными») условиями, накладываемыми на программы. Некоторые из таких условий являются универсальными для всех рассматриваемых здесь языков программирования. Это следующие условия.

2.1. Условия однозначности смысла имен

Первое условие этого рода состоит в том, что имя типа нельзя использовать как имя соответствия. Как уже было отмечено, мы будем использовать разные классы имен для типов и соответствий.

Второе условие: соответствие должно иметь всегда одно и то же число аргументов. Например, ошибкой является описание вида

$$\text{minus}(x) = \text{minus}(\text{ZERO}, x).$$

Это вводит запрет на так называемую «перегрузку» имен.

2.2. Условия определенности значений переменных

Это условие состоит в том, что каждая переменная в описании соответствия должна ровно один раз встретиться в его левой части (до знака равенства), заметим, что это исключает одношаговое сравнение обрабатываемых объектов, которое используется в таких языках, как Рефал. Ошибочными, например, являются описания

$$\begin{aligned} \text{BPair}(x) &= \text{PAIR}(x, y), \\ \text{compare}(x, x) &= \text{TRUE}, \end{aligned}$$

где x и y — переменные. В первом предложении переменная y не встретила в левой части.

2.3. Условия определенности типов

Первое условие определенности типов заключается в том, что каждое соответствие должно иметь тип: каждое соответствие с именем F должно иметь ровно одно описание вида $F(X) = T$, где X — последовательность имен типов, разделенных запятыми, T — имя типа. Такое описание называется «первичной спецификацией» соответствия. В приведенном примере и его дополнении это описания

$$\begin{aligned} \text{add}(\text{Number}, \text{Number}) &= \text{Number}, \\ \text{BNumber}(\text{Number}) &= \text{Number}, \\ \text{min}(\text{Number}, \text{Number}) &= \text{Number}, \\ \text{subt}(\text{Number}, \text{Number}) &= \text{Number}, \\ \text{BStack}(\text{Number}) &= \text{Nstack}, \end{aligned}$$

которые не могут быть опущены. Второе условие определенности типов заключается в том, что типы переменных должны также определяться однозначно в описаниях соответствий. Например, ошибочным является следующий фрагмент:

```
List = ONE(Pair),
List = ONE(List),
List = TWO(List),
cutlist(List) = List,
cutlist(ONE(x)) = NULL,
cutlist(TWO(x)) = x.
```

2.4. Условие предикативности определений

Условиями предикативности обычно называются условия, гарантирующие отсутствие «порочных кругов» в определениях. Условие этого рода в нашем случае состоит в том, что в описании типа не могут содержаться имена соответствий. Тип определяется только из символов и типов. Тем не менее определение типа может быть индуктивным, то есть сложные данные некоторого типа могут содержать в своем составе более простые данные того же типа, как в этом примере:

```
Number = ZERO,
Number = NEXT(Number),
Nstack = BOTTOM,
Nstack = TOP(Number, Nstack).
```

Возможно и совместное индуктивное определение нескольких типов, как в следующем примере:

```
Term = VAR(Variable),
Term = APPLY(Function, List),
List = NULL,
List = PAIR(Term, List).
```

2.5. Условие локальности алгоритмов

Локальность алгоритма это «элементарность» каждого его шага. «Алгоритмом» функции в нашем языке является совокупность ее описаний, не содержащих имен типов и ограничений, то есть «определяющих» описаний. Условие локальности состоит в том, что левая часть любого предложения алгоритма не содержит имен соответствий (кроме однократного вхождения имени самой описываемой функции в самом начале). Нарушением условия локальности является, например, следующее предложение алгоритма:

```
predecessor(successor(x)) = x,
```

где *successor* — имя функции.

2.6. Условие однозначности алгоритмического процесса

Это условие заключается в том, что левые части предложений алгоритмов функций должны быть устроены так, что при любом значении аргумента функции выполнимым является не более чем одно предложение. Ошибкой является, например, алгоритм

$$\begin{aligned} \text{or}(x, \text{TRUE}) &= \text{TRUE}, \\ \text{or}(\text{TRUE}, y) &= \text{TRUE}, \\ \text{or}(\text{FALSE}, \text{FALSE}) &= \text{FALSE}. \end{aligned}$$

Правильно можно записать, например, так:

$$\begin{aligned} \text{or}(\text{TRUE}, x) &= \text{TRUE}, \\ \text{or}(\text{FALSE}, x) &= x. \end{aligned}$$

Это условие нетрудно сформулировать чисто синтаксически.

2.7. Условия соответствия алгоритмов спецификациям и ограничениям

Первое из этих условий состоит в том, что использование функций должно соответствовать их ограничивающим описаниям. Это означает, что функции всегда получают аргументы типов, заданных в первичной спецификации, в предположении, что используемые в аргументах функции выдают значения соответствующих типов. Например, описанную выше функцию *add* нельзя применять не к числам:

$$\text{add}(\text{BOTTOM}).$$

Второе условие соответствия означает, что определяющие описания функций задают результат типа, заданного в ее первичной спецификации, в предположении, что аргументы описываемой функции также имеют специфицированные типы.

Например, ошибкой будет добавить к вышеприведенному алгоритму для функции *subt* описание вида

$$\text{subt}(\text{ZERO}, \text{NEXT}(y)) = \text{BOTTOM}.$$

Первое и второе условия соответствия нетрудно задать чисто синтаксически.

Третье условие соответствия означает, что определяющие описания функций должны соответствовать их ограничивающим описаниям. То есть мы можем каждому терму в выражении, определяющем значение функции, поставить в соответствие множество выражений, задающих ограничения значения этого терма. В результате выполнения такой разметки терм

правой части предложения алгоритма в качестве ограничений должен получить все ограничения, наложенные на вычисляемую функцию. Разумеется, это выполняется в предположении, что и на переменные накладываются ограничения, вытекающие из соответствующего ограничивающего описания. Синтаксическая реализация данного условия может быть уточнена различными способами, что дает возможность порождать различные семейства языков данного типа. Далее будут приведены некоторые примеры того, как можно выполнять наложенные ограничения.

2.8. Условие полноты определений функций

Требуется, чтобы алгоритм работал во всех случаях, заданных совокупностью ограничений, наложенных на функцию. Синтаксическая реализация данного условия также может быть осуществлена несколькими способами.

Это условие иллюстрируется примером

$$\begin{aligned} \text{BNumber}(\text{Number}) &= \text{Number}, \\ \text{BNumber}(\text{ZERO}) &= \text{ZERO}, \\ \text{BNumber}(\text{NEXT}(x)) &= \text{ZERO}, \\ \text{BNumber}(\text{NEXT}(x)) &= \text{NEXT}(\text{BNumber}(x)), \\ \text{subt}(\text{Number}, \text{Number}) &= \text{Number}, \\ \text{subt}(x, \text{BNumber}(x)) &= \text{BNumber}(x), \\ \text{subt}(\text{ZERO}, \text{ZERO}) &= \text{ZERO}, \\ \text{subt}(\text{NEXT}(x), \text{ZERO}) &= \text{NEXT}(\text{subt}(x, \text{ZERO})), \\ \text{subt}(\text{NEXT}(x), \text{NEXT}(y)) &= \text{NEXT}(\text{subt}(x, y)), \\ \text{Nstack} &= \text{BOTTOM}, \\ \text{Nstack} &= \text{TOP}(\text{Number}, \text{Nstack}). \end{aligned}$$

Здесь не требуется определять

$$\text{subt}(\text{ZERO}, \text{NEXT}(x)) = \dots,$$

так как такое сочетание аргументов не допускается наложенным ограничением.

§ 3. Выполнение программы

Выполнение программы задается алгоритмами описанных функций. В силу условий локальности алгоритмов и однозначности алгоритмического процесса каждый терм, не содержащий имен типов и ограничений, инициирует процесс преобразования этого терма. Этот процесс можно сделать

однозначно определенным, если вычислять подтермы слева направо (изнутри наружу). В силу наложенных контекстных условий этот процесс не может «зайти в тупик», то есть остановиться при наличии в терме имен функций. Процесс преобразования считается завершенным, если в терме не осталось вхождений имен функций. Из теории алгоритмов известно, что не существует универсального синтаксического условия, гарантии завершаемости алгоритмического процесса. Все такие условия должны ограничивать сложность вычислений. Такие условия и формулируются в следующем параграфе.

§ 4. Условие завершаемости алгоритмов

Простейшие ограничения сложности алгоритмов — ограничения на рекурсию. Будем считать, что функции могут рекурсивно определяться только по одной, то есть запретим одновременные рекурсивные определения нескольких функций. Второе ограничение на рекурсию — ее «примитивность». Она заключается в том, что вектор аргументов функции обязательно должен при вычислении на каждом шаге лексикографически «убывать», то есть в правой части определяющего описания функция применяется к более простым аргументам, чем в левой части. Лексикографический полный порядок на конечных деревьях, которыми являются обрабатываемые данные, нетрудно определить очевидным и естественным способом. Таким образом, получается, что наши алгоритмы могут определять функции из некоторого субрекурсивного класса.

§ 5. Пример: оценка формул «слабой прототетики»

Под языком «слабой прототетики» будем иметь в виду язык логики высказываний с «оператором присваивания», конструкцией, которая вводит логическую переменную, присваивая ей значение, заданное другой формулой. Синтаксис и семантику формул слабой прототетики опишем следующим образом (натуральные числа используются как индексы переменных):

```
Index = ZERO,  
Index = NEXT(Index).
```

Формулы (тождественная ложь, переменная, импликация и «оператор присваивания») определяются так:

```
Formula = NO,  
Formula = VAR(Index),  
Formula = IMP(Formula,Formula),  
Formula = FOR(Formula,Formula).
```

Последний вариант формулы вычисляется так: значение первой подформулы добавляется в вершину стека, на котором вычисляется значение всей формулы. После этого вычисляется вторая подформула. Значение переменной на стеке определяется как значение, отстоящее на заданное число от вершины стека.

Логические значения определяются обычным путем:

```
Bool = FALSE,
Bool = TRUE.
```

Единственная используемая логическая функция определяется как

```
impl(Bool, Bool) = Bool,
impl(FALSE, x) = TRUE,
impl(TRUE, x) = x.
```

Определим понятие стека логических значений:

```
Stack = BOTTOM,
Stack = TOP(Bool, Stack),
```

Нам потребуется извлечение значения из заданной глубины стека

```
extract(Index, Stack) = Bool,
extract(ZERO, TOP(v, s)) = v,
extract(NEXT(x), TOP(v, s)) = extract(x, s),
extract(NEXT(x), BOTTOM) = FALSE.
```

Значение формулы на стеке можно определить следующим способом:

```
fsv(Formula, Stack) = Bool,
fsv(NO, s) = FALSE,
fsv(VAR(x), s) = extract(x, s) ,
fsv(IMP(a, b), s) = impl(fsv(a, s), fsv(b, s)),
fsv(FOR(a, b), s) = fsv(b, TOP(fsv(a, s), s)).
```

Далее приведен пример ограничения функции fsv — функция $bfsv$. Эта функция определена так, что из этого определения нетрудно доказать ограниченность времени ее вычисления (числа шагов) многочленом от размера исходных данных.

Сначала произведем некоторые вспомогательные построения. Определим стеки, ограниченные индексом:

```
BStack(Index) = Stack,
BStack(ZERO) = BOTTOM,
BStack(NEXT(x)) = TOP(Booleam, BStack(x)).
```

Определим ограничение высоты стека индексом:

```

cut(Index,Stack) = Stack,
cut(x,s) = BStack(x),

cut(ZERO,s) = BOTTOM,
cut(NEXT(x),BOTTOM) = BOTTOM,
cut(NEXT(x),TOP(v,s)) = TOP(v,cut(x,s)).

```

Определим значение формулы на стеке, ограниченном индексом:

```

bfsva(Index,Formula,Stack) = Bool,
bfsva(x,a,Bs(x,a)) = Bool,

bfsva(x,NO,s) = FALSE,
bfsva(x,VAR(x),s) = extract(x,s),
bfsva(x,IMP(a,b),s) = impl(bfsva(x,a,s),bfsva(x,b,s)),
bfsva(x,FOR(a,b),s) = bfsva(x,b,cut(x,TOP(bfsva(x,a,s),s))),

```

Определим «размер» формулы:

```

fn(Formula) = Index,

fn(NO) = ZERO,
fn(VAR(x)) = x,
fn(IMP(NO,b)) = fn(b),
fn(IMP(VAR(x),b)) = fn(b),
fn(IMP(IMP(a,b),c)) = fn(IMP(a,IMP(b,c))),
fn(IMP(FOR(a,b),c)) = NEXT(fn(IMP(a,IMP(b,c)))),
fn(FOR(a,b)) = NEXT(fn(IMP(a,b))).

```

Глубина стека вычисляется так:

```

sdep(Stack) = Index,

sdep(BOTTOM) = ZERO,
sdep(TOP(v,s)) = NEXT(sdep(s)).

```

Теперь мы можем определить значение формулы на стеке, используя только ограниченную рекурсию:

```

bfsv(Formula,Stack) = Bool

bfsv(a,s) = bfsva(fn(IMP(a,VAR(sdep(s))))),a,
            cut(fn(IMP(a,VAR(sdep(s))))),s)

```

Заключение

Таким образом, построен язык программирования, позволяющий формулировать определения функций с помощью различных видов ограниченной рекурсии. На следующих этапах работы в этой области можно получить семейства подязыков для порождения алгоритмов для различных сложностных классов на деревьях. На практике такие языки можно использовать, если требуется получать программы с гарантированно ограниченными сложностными характеристиками. Конечно, абстрактного полиномиального ограничения сложности вычисления на практике не достаточно. Поэтому одним из направлений работы может быть разработка алгоритмов «измерения» конкретных степеней и коэффициентов ограничивающих многочленов.

СПИСОК ЛИТЕРАТУРЫ

1. Бельтюков А. П. Дедуктивная оптимизация программ // Оптимизация и преобразование программ: Материалы всесоюз. семинара. Новосибирск, 1983. Ч. 1. С. 105–113.
2. Бельтюков А. П. Язык дедуктивного программирования // Теория языков программирования: Сб. науч. тр. Ижевск, 1983. С. 3–18.
3. Бельтюков А. П. Эффективное расширение языка РЕФАЛ // Вычислительные методы и информационное обеспечение пакетов прикладных программ. Устинов, 1985. С. 50–54.
4. Beltiukov A. P. Intuitionistic formal theories with realizability in subrecursive classes // Annals of Pure and Applied Logic. 1997. Vol. 89. P. 3–15.
5. Бельтюков А. П. Слабая конструктивная арифметика второго порядка с привлечением алгоритмов полиномиальной временной сложности // Записки семинаров ПОМИ. СПб. 2003. Т. 304. С. 7–12.
6. Beltiukov A. P. A strong induction scheme that leads to polynomially computable realizations // Theoretical Computer Science. 2004. Vol. 322. P. 17–39.

Поступила в редакцию 01.11.06

A. P. Beltiukov

Simple typed functional programming languages

The paper presents a way of constructing simple functional programming languages that fit to define classes of recursive functions with bounded computational complexity. All processed data in the language are restricted with types, that also can be constructed in the language. The languages can be used both to concern complexity algorithms classes and to be expanded for practical programming use.

Бельтюков Анатолий Петрович
Удмуртский государственный университет
426004, Россия, г. Ижевск,
ул. Университетская, 1
E-mail: belt@uni.udm.ru