LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Quantification in Algol-like Languages

Quantification in Algol-like Languages

by

R. D. Tennent

ECS-LFCS-86-6

LFCS Report Series

(also published as CSR-212-86)

June 1986

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Quantification in Algol-like Languages

R. D. Tennent[†]
Laboratory for the Foundations of Computer Science
Department of Computer Science
University of Edinburgh
Edinburgh, Scotland.

Keywords: quantifier, syntactic sugar, Algol-like language

Abstract A general notation is proposed which may be used to replace many specialized constructions in programming languages and logics, including variable-declaration blocks and quantified formulas.

> *In almost every language, a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic.*
>
> P. J. Landin

[†]On leave from the Department of Computing and Information Science, Queen's University, Kingston, Canada.

J. C. Reynolds[4] has suggested as a design principle for Algol-like languages that, except for "syntactic sugar" (i.e., language constructs that can be defined as abbreviations in terms of more basic constructs), the *only* identifier-binding mechanism should be the lambda expression. For example, the block form

let I **be** P **in** Q

may be treated, as suggested by Landin,[1] as syntactic sugar for the application

$(\lambda I : \theta . Q)P$

where P is a phrase of type θ. As another example, if τ denotes a data type, the variable-declaration block

new τ **var** I **in** C

is de-sugared by Reynolds[4] as

newvar$[\tau](\lambda I :$ **var**$[\tau] . C)$

for a suitable operator or constant **newvar**$[\tau]$.

These two examples of syntactic sugarings are, however, dissimilar in that Landin's **let** is a *general* form (for phrases P and Q of *arbitrary* types), whereas the variable-declaration block is specialized to *commands* C and *variable*-identifiers I. This would not be of concern if the variable-declaration block were the only such specialized construct; but, in fact, Reynolds[4] also discusses specialized syntactic sugarings for

- call by value and call by result,
- new array declarations,
- escapes, and
- for iterations,

and in Reynolds[3] there are additional sugarings for

- class declarations, and
- class instantiations.

Furthermore, quantification in both the assertion and the specification languages of Reynolds[5] and the block expression discussed in Tennent[6] would require de-sugaring as well.

This paper describes a simple and general notation comparable to Landin's **let** which can replace all of these specialized syntactic sugarings without loss of readability. The resulting language is thus syntactically simpler and more uniform, but still adheres to the principle that all binding can be reduced to lambda binding, because the proposed notation is definable in terms of lambda abstraction and application.

To simplify the presentation, we avoid coercions and adopt the type structure specified by the productions of Table 1, with the following abbreviations:

$$\mathbf{array}[\theta,0] = \theta$$

$$\mathbf{array}[\theta,n+1] = \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{array}[\theta,n]$$

$$\mathbf{array}[\theta] = \mathbf{array}[\theta,1]$$

To describe syntax, we use rules in natural-deduction format[2] as in Table 2.

The basic form of the rule for the new notation we propose is as follows:

Metavariables

τ	data types
θ	phrase types

Productions

$$\tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{real} \mid \ldots$$

$\theta ::= \mathbf{exp}[\tau]$	expressions
$\mid \mathbf{var}[\tau]$	variables
$\mid \mathbf{comm}$	commands
$\mid \mathbf{compl}$	completers
$\mid \mathbf{assert}$	assertions
$\mid \theta \rightarrow \theta$	procedures
$\mid \Pi(\ldots, I : \theta, \ldots)$	collections

Table 1. Types

Abstraction

$$\frac{\begin{array}{c}[I:\theta]\\ \vdots \\ P:\theta'\end{array}}{\lambda I:\theta.P:\theta\rightarrow\theta'}$$

Application

$$\frac{P:\theta\rightarrow\theta' \quad Q:\theta}{P\ Q:\theta'}$$

Collection Introduction

$$\frac{P_i:\theta_i \text{ for } i=1,\ldots,n}{<I_1:P_1,\ldots,I_n:P_n>:\Pi(I_1:\theta_1,\ldots,I_n:\theta_n)}$$

Collection Elimination

$$\frac{P:\Pi(\ldots,I:\theta,\ldots)}{P.I:\theta}$$

Landin's let

$$\frac{P:\theta \qquad \begin{array}{c}[I:\theta]\\ \vdots \\ Q:\theta'\end{array}}{\textbf{let } I \textbf{ be } P \textbf{ in } Q:\theta'}$$

Table 2. Syntax Rules.

Quantification

$$\frac{\begin{array}{c}[I:\theta'] \\ \vdots \\ P:\theta''\end{array} \qquad Q:(\theta' \to \theta'') \to \theta}{\# Q\ I.P:\theta}\ ,$$

and the construction is de-sugared by the following equivalence:

$$\# Q\ I.P = Q(\lambda I:\theta'.P)\ .$$

Phrase Q is termed the *quantifier* part of the construction. Note that the de-sugared form, like that for Landin's **let**, is an application, but that it is the operand part, rather than the operator part, which is a lambda expression.

As an example of its use, the variable-declaration block discussed earlier may be replaced by the construct

$$\#\ \mathbf{newvar}[\tau]\ I.C\ ,$$

where C is a command and **newvar**[τ] is a constant of type (**var**[τ] → **comm**) → **comm**. It is convenient to allow quantification to be *iterated* when $\theta''=\theta$, as follows:

Iterated Quantification

$$\frac{\begin{array}{c}[I_1,\ldots,I_n:\theta'] \\ \vdots \\ P:\theta\end{array} \qquad Q:(\theta' \to \theta) \to \theta}{\# Q\ I_1,\ldots,I_n.P:\theta}\ ,$$

with the equivalence

$$\# Q\ I_1,\ldots,I_n.P = \ \# Q\ I_1.\ldots\ \# Q\ I_n.P\ .$$

Then several variables of the same type may be declared at once by the construction

$$\#\ \mathbf{newvar}[\tau]\ I_1,\ldots,I_n.C\ .$$

It may even be desirable to allow the data-type name τ to be used as a variable-allocating phrase of type (**var**[τ] → **comm**) → **comm** so as to permit the Algol 60-like block

$$\#\ \tau\ I_1,\ldots,I_n.C\ .$$

The "quantifier" terminology derives from treating constants **forall**[τ] and **exists**[τ] of type (**exp**[τ] → **assert**) → **assert** as quantifiers in the obvious way, and similarly for iterator **for**(E_1,E_2) of type $(\mathbf{exp[int]} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$, where integer expressions E_1 and E_2 determine the iteration limits.

Many other examples may be given. To declare new (one-dimensional) array variables, we introduce

$$\mathbf{array}[\tau](E_1,E_2)$$

of type $(\mathbf{array}[\mathbf{var}[\tau]] \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$, where the integer expressions E_1 and E_2 determine the subscript bounds, and similarly for higher-dimensioned arrays. Note that the quantification notation makes it clear that the bound expressions are not in the scope of the quantifier. The operator **escape** of type $(\mathbf{compl} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$ described by Reynolds[4] may be used as a quantifier as follows,

$$\#\,\mathbf{escape}\ I.C\ ,$$

in order to establish a way of exiting from command C. The block expression

$$\mathbf{result}\ I:\tau\ \mathbf{of}\ C$$

discussed in Tennent[6] may be replaced by

$$\#\,\mathbf{result}[\tau]\ I.C$$

using a constant **result**$[\tau]$ of type $(\mathbf{var}[\tau] \rightarrow \mathbf{comm}) \rightarrow \mathbf{exp}[\tau]$. Note that this quantifier may *not* be iterated because $\mathbf{comm} \neq \mathbf{exp}[\tau]$.

The class-instantiation construct

$$\mathbf{newelement}\ I:Q\ \mathbf{in}\ C$$

of Reynolds[3] may be replaced by

$$\#Q\ I.C\ ,$$

and, furthermore, may be iterated, because Q has a type of the form $(\Pi(\ldots) \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$. Class *declaration* is more problematical; Reynolds[3] suggests the notation

$$\mathbf{class}\ I(D;C_0;I_1:P_1;\ \ldots\ ;I_n:P_n)\ \mathbf{in}\ C_1\ \ ,$$

where C_0 and C_1 are commands, D is a sequence of "declarations-for-commands", and the construct de-sugars as

$$
\begin{array}{l}
\mathbf{let}\ I\ \mathbf{be} \\
\quad \lambda I' : \Pi(I_1:\theta_1,\ldots,I_n:\theta_n) \rightarrow \mathbf{comm}. \\
\qquad D\ C_0; \\
\qquad I' \langle I_1:P_1;\ \ldots\ ;I_n:P_n \rangle \\
\mathbf{in}\ C_1
\end{array}
$$

where I' is a new identifier. To handle this with our quantification notation, we first introduce a notation for

Exportation

$$\frac{C\colon \textbf{comm} \qquad P\colon \theta}{C\ \textbf{export}\ P\colon (\theta \to \textbf{comm}) \to \textbf{comm}}$$

defined by the equivalence

$C\ \textbf{export}\ P \;=\; \lambda I\colon \theta \to \textbf{comm}.(C; I(P))$,

where I is not free in C or P; then, we generalize the applicability of the quantification notation to allow:

Higher-order Quantification

$$\frac{\begin{array}{c}[I\colon \theta'] \\ \vdots \\ P\colon \theta_1 \to \ldots \to \theta_n \to \theta''\end{array} \qquad Q\colon (\theta' \to \theta'') \to \theta}{\#\, Q\ I.P\colon \theta_1 \to \ldots \to \theta_n \to \theta}$$

with the equivalence

$\#\, Q\ I.P \;=\; \lambda I_1\colon \theta_1.\ldots.\lambda I_n\colon \theta_n.Q(\lambda I\colon \theta'.P(I_1)\ldots(I_n))$,

where the I_i are not free in P or Q. If $n=0$, this reduces to the basic rule. Using exportation and higher-order quantification (possibly iterated), one may then write

let I **be**
  D
  C_0
export
  $\langle I_1 : P_1 ; \ldots ; I_n : P_n \rangle$
in C_1

where D is a sequence of quantifications with result type **comm**.

The "call-by-value" operator

τ **value** I **in** C

proposed by Reynolds[4] is also problematical because the (first) occurrence of I is both free and bound. Perhaps a reasonable alternative would be the generalized operator **value**[τ] of type **exp**[τ] $\to$ (**var**[τ] $\to$ **comm**) $\to$ **comm** defined by

value[τ](E)(P) $=$ $\#\,\tau I.(I{:=}E; P(I))$

for I not free in E or P. This could be used to simulate the effect of

Reynolds's operator as follows,

$\#\,\mathbf{value}[\tau](I)\, I \bullet C$,

and similarly for call by result.

In conclusion, it has been shown that simpler and more uniform syntax may be designed for an Algol-like language by replacing many specialized constructs with a general notation which is derived from the concept of quantification in predicate logic and is definable in terms of the more basic constructions of lambda abstraction and application.

Acknowledgements

Financial support was provided by an operating grant from the Natural Sciences and Engineering Research Council of Canada, and a fellowship from the Alvey Directorate and the British Science and Engineering Research Council. I am grateful to Robin Milner for arranging the fellowship, and for his hospitality.

References

(1) P. J. Landin, "The next 700 programming languages", *Comm. ACM* 9 (3), pp. 157-166 (1966).

(2) D. Prawitz, *Natural Deduction: A Proof-Theoretical Study,* Almquist and Wiksell, Stockholm (1965).

(3) J. C. Reynolds, "Syntactic control of interference", *Conf. Record 5th ACM Symp. on Principles of Programming Languages,* pp. 39-46, ACM, New York (1978).

(4) J. C. Reynolds, "The essence of Algol", in *Algorithmic Languages* (J. W. de Bakker and J. C. van Vliet, Eds.), pp. 345-372, North-Holland (1981).

(5) J. C. Reynolds, *The Craft of Programming,* Prentice-Hall International, London (1981).

(6) R. D. Tennent, "Functor-category semantics of programming languages and logics", to appear in the Proc. of the Workshop on Category Theory and Computer Programming, University of Surrey, Guildford, England, Sept. 16-20, 1985 (D. Pitt, Ed.), Lecture Notes in Computer Science, Springer.