SUPERHIGH-LEVEL LANGUAGE PARIS

V. D. Korneev, N. N. Mirenkov, and A. Sh. Nepomnyashchaya

The paper describes the main constructs of the language PARIS designed for artificial intelligence applications. The basic ideas underlying the construction of the language are outlined.

INTRODUCTION

The advent of parallel computers raises the issue of efficient adaptation of various algorithms to the architecture of these machines. Superhigh-level languages play an important role in this context.

In this paper, we describe the basic ideas and main constructs of the superhigh-level language PARIS designed for parallel implementation (primarily on associative and semiassociative processors). The language constructs are selected from the following considerations. The increasing intelligence level of the man–machine interaction requires sufficiently expressive and readable languages with a relatively small number of universal concepts and expressive tools for these concepts, which provide a basis for the construction of special-purpose languages. The language constructs should allow for the actual physical structures and operating features of modern and future computers and also support the ideas of functional and logic programming.

The language PARIS was developed by a stepwise implementation strategy. The core was created in the first stage, with most of the core constructs admitting parallel interpretation. Various problem-oriented shells will be developed in subsequent stages.

The entire development process fundamentally utilizes the properties of existing high-level languages. Analysis of the languages MODULA-2 [1], ADA [2], PROLOG [3], LISP [4], SEQUEL [5], and the SETL language family [6, 7, 8] shows that these languages already contain many of the constructs that we need for our language (although not all of them). However, none of the languages provides an option for "selection" of constructs (nor do they contain the new extensions incorporated in our language). The purpose of the analysis was to identify a collection of constructs which allow efficient interpretation of the existing languages. Yet, the name PARIS, which is an acronym for PARallel Interpretation of SETL, was chosen to emphasize the basic role of data types and of a number of SETL constructs in our new language. The language SETL was extended with constructs of fundamental importance — set and tuple transformations, which are useful for manipulation of large data files. Moreover, the set of tuple instructions was extended; in particular, special operations were included for manipulation of names when they appear as tuple elements. Operators that specify explicit (external) parallelism and support object-oriented programming were introduced. An operator was introduced for specifying the names of external objects, including new data types. The semantics of operations on basic types was sharpened to allow for their parallel implementation.

We should also stress that the constructs in PARIS are designed to support conflicting language properties: proceduralness and nonproceduralness, typing and typelessness, use of both dynamic and static objects at the same time. In our view, such dialectic properties are essential in a language designed for artificial intelligence applications.

DESCRIPTION OF THE CORE OF THE LANGUAGE PARIS

As the basic types and operations on basic types, PARIS adopts part of the SETL tools. This choice was made possible a) by the existence of expressively powerful data types in SETL — sets and tuples, which are useful for representation of a variety

of complex data structures; b) by the existence of constructs that say "what to do" without specifying "how" (e.g., set or tuple generators); c) by the possibility of fairly efficient parallel implementation of operations on basic types using associative and semiassociative processors (the existence of internal parallelism).

Basic Types

Basic types in PARIS are not declared by a special operator and the interpreting system recognizes them by their syntax. There is thus no "rigid" linkage between object names, on the one hand, and object values or types of these values, on the other. The specific features of the basic types are a compromise between the language level and the need for efficient implementation of the language on modern parallel computers. Let us consider the features of operations on sets and tuples.

A set and a tuple consist of elements, each of which belongs either to a basic type or to a new defined type. An element of a set or a tuple may be a name of an object, if it is assigned a specific value. When set operations are performed, every element which is an object name is replaced with the corresponding value. Operations on sets and tuples manipulate values, and not object names. For tuples, however, there are special operations capable of manipulating object names.

Note that set-theoretical operations are fairly efficiently implemented on both MIMD and SIMD computers.

Operations on Basic Types

Basic types consist of standard types (integer, real, logical, atomic) and composite types (symbolic, bit, set, tuple). Operations on standard types, with the exception of type atom, are conventional. The atomic type is defined as in [7]. We should only note that atoms may take the undefined value om (from "omega"), which may be an element of a tuple or a set. Operations on composite types are divided into two classes: generators and transformers. Generators construct new objects, without altering the initial object, while transformers alter the initial objects.

String Operations. Let C and C' be symbolic strings, i and j integers. We list the main generators for symbolic strings:

- C + C' — concatenation of the strings C and C';
- i * C — concatenation of i copies of the string C;
- C(i) — extraction of the i-th symbol of string C;
- C(i..j) — cutting the substring from symbol i to j inclusive from the string C (1 ≤ i ≤ j);
- #C — the length of the string C.

Let α and β be bit strings of equal length. Alongside the standard string operations, we also have the following bit strings generators, which are defined conventionally: |α, α V β, α ^ β.

Symbolic (or bit) strings are transformed by applying an assignment operator of the form C(i) := C' or of the form C(i..j) := C', which inserts the string C' in place of symbol i or in place of the cutting from symbol i to symbol j in string C.

Tuple Operations. Let K and K' be tuples, i and j integers. We list the main tuple generators:

- K + K' — concatenation of the tuples K and K';
- i * K — concatenation of i copies of the tuple K;
- #K — the index of the last element in K which is not om; K(i) — extraction of the i-th element in K (1 ≤ i ≤ #K); K(i..j) — cutting the section between elements i and j in K (1 ≤ i ≤ j ≤ #K).

The main tuple transformers are the following:

- K with x — the value x is adjoined at the end of the tuple K;
- x tohd K — the value x is adjoined to the head of the tuple K;
- z frtl K — the last element which is not om is deleted from K and z := K(#K);
- z frhd K — the first element is deleted from K and z := K(1).

Note that the operations K with x, z frtl K, and z frhd K can be used to construct various structures, such as stack, queue, and deque. If x is an object name to be adjoined to the tail or the head of the tuple K, then we use the transformers K with name(x), name(x) tohd K.

A tuple is also transformed by applying the assignment operator, which is defined similarly to that for strings: K(i) := K' and K(i..j) := K'.

A tuple is defined as an enumeration of its elements or alternatively by a constructor or a transformer. A transformer of tuple elements has the form

\[ \ast \langle H(x) : x \text{ from } K \mid P(x) \rangle \ast , \]