A LINGUAGEM CONSTRUCTOR PARA AMBIENTES COMPUTACIONAIS

P. KEGLEVICH DE BUZIN *

SUMARIO

Neste trabalho são apresentadas algumas das características mais marcantes da linguagem CONSTRUCTOR, com alguns poucos exemplos. Tal linguagem foi projetetada tendo vista quatro tipos de ambientes computacionais: seqüencial, concorrente, parelelo e distribuído.

ABSTRACT

This paper presents some of the most important features of the CONSTRUCTOR programming language, with some few examples. Such language was designed aiming four kinds of computational environments: sequential, concurrent. parellel and distributed.

* MsC. Ciência da Computação (UFRGS, 1984); Sistemas Distribuídos, controle de processos e sistemas em temp-real, engenharia de software, projeto de linguagens de programação. UFRGS-PGCC, Av. Osvaldo Aranha, 99 - Porto Alegre, RS - CEP 90000.

1. INTRODUÇÃO

O fornecimento de recursos computacionais através de uma linguagem de alto nível confere uma enorme versatilidade e confiabilidade a um sistema, provendo um meio eficiente de descrever as computações feitas pela máquina. Além disso, constitui-se em um meio efetivo para descrever e divulgar computações entre pessoas.

A seguir será feita uma descrição suscinta da linguagem de alto nível CONSTRUCTOR projetada recentemente para suporte e desenvolvimento de sistemas em ambientes computacionais. Maiores detalhes, e as considerações que nortearam o projeto desta linguagem, os quais são omitidos neste trabalho, podem ser encontrados em |KEG 84a, KEG 84b|.

2. A LINGUAGEM CONSTRUCTOR

Todos os comandos da linguagem seguem a filosofia dos comandos guardados, com a única diferença que a conferência das guardas e feita seqüencialmente. A idéia a que esta mesma estrutura seja mantida pelo código virtual gerado pelo compilador.

O projeto da linguagem não pode deixar de preocupar-se com a implementação. Assim, o conjunto completo da linguagem foi subdividido em oito subconjunto de complexidade gradativa de modo que é possível haver oito versões intermediárias da linguagem, até chegar a versão final. Este esquema é similar ao adotado na implementação da linguagem SP/K. A linguagem CONSTRUCTOR e definida de modo que a partir da segunda versão é possível escrever o compilador da linguagem na própria linguagem.

A concepção da linguagem foi precedida da elaboração de um modelo para ambientes computacionais, dentro do qual existem quatro tipos de ambientes computacionais: seqüencial, concorrente, parelelo e distribuído. A seguir é mostrado como a linguagem descreve cada ambiente.

2.1 AMBIENTE SEQUENCIAL

A parte algoritmica seqüencial foi bastante influenciada pelas linguagens PASCAL e EDISON |HAN 81|, possuindo porém um menor número de comandos. A declaração de tipos e similar a da linguagem EDISON, pois facilita a compilação. Também foram adicionados alguns aprimoramentos em relação ao PASCAL.

2.1.1 ESTRUTURA DE DADOS

Na definição da estrutura de dados, só é permitida a declaração de tipos novos definidos pelo usuário, que não sejam padrão do sistema. Além disso, o tipo ARRAY e definido pelo usuário especificando, ao invés do número de dimensões que possui, o tipo de indexação usado e o tipo do elemento básico que o compõe. Os limites são estabelecidos na declara-

ção de variáveis. Portanto o tipo ARRAY funciona como uma espécie de tipo parametrizado. Desta maneira, mantendo as mesmas regras de definição de parâmetros do PASCAL, e possível fazer com que as procedures e functions sejam aplicadas a arrays de qualquer limite ou tamanho.

Exemplos:

```
TYPE  typeline(INT)CHAR;
VAR   line|1..80|:typeline;
      impline|1..132|:typeline;
PROCEDURE manipula (VAR buf:typeline; VAL tam:INT);
VAR  l:INT;
  .
  .
ENDPROC;
      .
      .
manipula(line,80);
manipula(impline,132);
```

Também e permitido que os limites de variáveis tipo ARRAY declaradas dentro de procedures ou functions possam ser especificados por expressões contendo constantes e parâmetros passados "by-value".

Exemplos:

```
PROCEDURE proc (VAL a,b:INT);
VAR line |a..b+2| : typeline;
```

2.1.2 COMANDOS SEQUENCIAIS

Os comandos seqüenciais podem ser subdivididos em comandos simples e comandos compostos. Os comandos simples são os de atribuição, de avaliação de construtores (EVAL), chamada de procedure e o comando de saída de comando estruturado (OUT). Os comandos estruturados são o de alternação (IF) e o de repetição (WHILE). Cada um destes comandos pode conter várias guardas seguidas de comandos. Cada um destes grupos de comandos só é executado quando a guarda estiver aberta, ou seja, for verdadeira.

2.1.3 PROCEDURE E FUNCTIONS

Quando houver uma procedure passada como parâmetro, os parâmetros da mesma devem ser também especificados na declaração de parâmetros da procedure.

Exemplos:

```
PROCEDURE bisect(FUNCTION f(VAL x:FLOAT):FLOAT;VAL l,h:FLOAT);
```

É permitido o aninhamento de procedures e functions, e ainda é possível a declaração particionada de procedures. Isto é feito especificando um cabeçalho que antecede uma posterior declaração completa da procedure.

Como o modelo em que se baseia a linguagem permite a utilização de memória compartilhada como meio de comunicação, considerou-se interessante dar liberdade a linguagem para descrever procedures reentráveis. Nos contextos de ambientes paralelos e concorrentes , isto pode permitir uma boa eficiência de uso de memória, além de fornecer recursos adicionais.

Outra facilidade oferecida pela linguagem e a de execução de rotinas em biblioteca. Isto implicitamente implementa o conceito de overlay sobre a pilha de dados do processo.

Exemplo:

LIB PROCEDURE Spascal(VAL source, arqlist,code:indentifier)|SPASCAL|;

2.2 AMBIENTE CONCORRENTE

A linguagem CONSTRUCTOR e fortemente baseada na linguagem SR |AND 82| na parte concorrente. Basicamente foram incluídos alguns aprimoramentos em relação a linguagem SR. principalmente quanto a uma maior disciplina no acesso a recursos de baixo nível, e o fornecimento de estruturas que facilitem uma programação mais clara e uma compilação mais eficiente.

2.2.1 COMANDOS CONCORRENTES

O conceito de operação permite a linguagem descrever mecanismos das mais variadas características semânticas de uma maneira concisa e clara. De acordo com o modelo de ambiente computacional, as operações consistem em mensagens com uma determinada semântica associada ao envio e ao recebimento das mesmas. Além disso, a liberdade de descrever a semântica das operacões confere a linguagem a habilidade de implementar eficientemente qualquer mecanismo de sincronização desejado. Os processos, também descritos pela linguagem, são vistos como um recurso disponível para programar a semântica do mecanismo das operações.

Basicamente os comandos concorrentes atuam sobre o ambiente como um todo e sobre operações. O comando que atua sobre todo o ambiente e o ABORT, que interrompe a execução de todos os processos do ambiente. Os comandos sobre operações se subdividem entre aqueles que definem operações e aqueles que invocam operações. As operações são definidas pelo comando SELECT, dentro de um processo. Os parâmetros formais de uma operação são os mesmos de uma procedure.

O comando SELECT é composto de uma ou várias guardas. Cada guarda e especificada pelo nome da operação definida pela mesma, e opcionalmente, por uma parte de sincronização e outra de escalonamento. Após cada guarda, segue-se o conjunto de comandos da operação. Portanto, um mesmo comando SELECT pode definir várias operações. Quando um processo executa um comando SELECT, fica suspenso até que uma das guardas deste comando seja aberta.

Quando há uma parte de sincronização, esta é especificada pela cláusula WHEN seguida de uma expressão booleana. Assim uma guarda e considerada aberta quando existe uma invocação pendente para a operação que define, e a condição especificada pela expressão booleana é verdadeira.

No caso de haver a parte de escalonamento, esta é especificada pela cláusula BY seguida de uma ordenção no atendimento das invocações, quando existe mais de uma invocação pendente. Aquela invocação para a qual o valor da expressão aritmética for menor, será a primeira a ser atendida. Naturalmente os parâmetros da operação podem entrar nesta expressão aritmética.

Exemplos:

```
SELECT read(RES v:item;i:INT) DO
       CALL  Startread;
       chave, v:=true, database |i|;
       SEND Endread;
ALSO Write(VAL v:item;i=ITEM) WHEN chave DO
       CALL  Startwrite;
       database|i|:=v;
       SEND Endwrite;
ANDSL;
SELECT do-io(c,...) BY ABS(c-position) DO
    position:=c;
ENDSL;
```

As operações podem ser invocadas por comandos CALL ou SEND, seguido do nome da operação com os parâmetros reais da chamada. No caso do SEND, o processo que invoca a operação não espera que esta termine a sua execução para continuar. Já no caso do comando CALL, o processo aguarda a confirmação de que a execução da operação terminou, antes de continuar.

Assim o comando CALL correspondende a uma chamada síncrona de operação, e o comando SEND correspondente a uma chamada assíncrona de operação.

Com o comando CONC é possível invocar várias operações no modo CALL paralelamente. Este comando é seguido de uma lista de chamada de operações com seus parâmetros reais. A execução do processo só irá continuar após todas as operações desta lista terem terminado as suas execuções.

Exemplos:

```
SEND put(c);
CALL startwrite;
CALL read(c);
CONC starwrite, read(c) ENDC;
```

2.2.2 ESTRUTURA DE DADOS

Para dar liberdade ao programador de controlar dinamicamente o contexto de operações que acessa, é implementado pela linguagem o conceito de capability de operações (CAP). Isto é interessante para implementar privacidade em sistemas operacionais.

Exemplo:

```
TYPE CAP io(input(RES buffer:line);
            output(VAL buffer:line)):
VAR leitesc:io; buf:line;


 leitesc.input:=read;          "read write são operações"
 leitesc.output:=write;        "pre-definidas"
CALL leitesc.input(buf);
CALL leitesc.output(buf);
```

2.2.3 MÓDULOS

O conceito de módulos do CONSTRUCTOR e uma extensão da idéia de Wirth |WIR 77|, onde os módulos isolavam partes do programa dependentes do tempo e de sincronização. No caso do CONSTRUCTOR, os módulos isolam características funcionais e sintáticas diferentes. Isto permite ao compilador detectar alguns erros de concepção de programa em tempo de compilação, aumentando a segurança dos programas escritos na linguagem.

Na linguagem CONSTRUCTOR existem três tipos de módulos: tipo OBJECT, que especifica os processos do ambiente, tipo INTHARD, que especifica uma interface de hardware, e tipo INTCOMP, que define um interface computacional. Cada um destes módulos apresentam construções sintáticas particulares, conforme a função a que se destinam. Não serão descritas as características sintáticas de cada módulo neste trabalho, apenas serão apresentados alguns exemplos explicativos.

O módulo tipo INTHARD descreve como o ambiente computacional irá "enchergar" a parte do hardware utilizada pelo sistema. Neste módulo são descritas as posições físicas de memória, as portas de I/O da máquina, as interrupções, e a forma de acesso pelo programa a estes recursos de baixo nível. A seguir é apresentado um exemplo de interface hardware de uma console descrito em CONSTRUCTOR:

```
INTHARD  intcons;
COUPLE console;
TYPE ARRAY registers(INT)BOOL;
VAR tks[0..7]:registers AT @177530; "reg. status keyboard"
    tps[0..7]:registers AT @177540; "reg. status printer"
    tkb:CHAR AT @17750; "Keyboard data register"
    tpb:CHAR AT @17751; "printer data register"
ACCESS Kint(tkb) INTERRUPT PORT @60;
       print INTERRUPT PORT @64;
ACTUATE startk; tsk[6]:=true; ENDACT;
ACTUATE startp; tps[6]:=true; ENDACT;
ACTUATE sckk; tks[6]:=false; ENDACT;
ACTUATE ackp; tps 6 :=false; ENDACT;
ACTUATE put(VAL c:CHAR); tpb:=c; ENDACT;
ENDHARD;
```

O módulo tipo INTCOMP (ou MSINTCOMP no caso de interface mestre) especifica a maneira como dois ambientes computacionais (programas) se intercomunicam. Especificam as operações importadas e exportadas, as operações importadas e exportadas, as eventuais exceções que possam ser atendidas, e a forma de controle da intercomunicação. A seguir é apresentado um exemplo do interface aplicado para a carga de programas por parte de um ambiente mestre (normalmente um sistema operacional), e o uso de primitivas por parte de um ambiente escravo (que pode também conter vários processos). Assim este interface funciona como um prefixo para o ambiente escravo, de uma maneira similar ao sistema SOLO de Brinch Hansen [HAN 77].

No ambiente mestre o interface teria a forma:

```
INTCOMP programa;
IMPORT read(REF c:CHAR) /CALL/;
       write(VAL c:CHAR) /SEND/;
       etc...
COUPLE loader;
    .
    .
ENDCOMP;
```

No ambiente escravo teria a forma:

```
MSINTCOMP prefix;
EXPORT read(REF c:CHAR) /CALL/;
       write(VAL c:CHAR) /SEND/;
       etc...
ENDCOMP;
```

Um módulo OBJECT define um ou mais processos concorrentes que podem se comunicar entre si ou com o ambiente externo ao módulo por meio de operações. As operações que são visíveis ao ambiente externo ao módulo são descritas no início do mesmo, com regras de escopo bem estabelecidas.

Um módulo OBJECT pode conter outros módulo OBJECTs e até definir outros ambientes compu

tacionais. Exemplificar tal estrutura seria muito extenso para este trabalho, isto pode ser encontrado em |KEG 84a, KEG 84b|.

2.3 AMBIENTES PARALELOS

Em ambientes paralelos, a parte algoritmica e a mesma que a do ambiente concorrente. O problema novo que surge é a necessidade descrever a distribuição dos módulos na arquitetura da máquina hospedeira, que nestes casos deve possuir mais de um processador. Assim os elementos do ambiente devem ser identificados (ELEMENT), a conexão entre tais elementos (topologia) deve ser descrita (CONNECT), e deve ser de designada a localização dos módulos lógicos (LOCALIZE). No ambiente paralelo os elementos podem ser processadores (PROCESSOR) ou memória compartilhada (SHAREDMEM).

Exemplo:

```
CONSTRUCT HOST(parelelo);
ELEMENT PROCESSOR P1(1);
        PROCESSOR P2(2);
        SHAREDMEM mem(@17000);
CONNECT (P1,mem); (P2,mem);
LOCALIZE (modi,P1); (mod2,P2); (Compr,mem);
```

2.4 AMBIENTES DISTRIBUÍDOS

No caso de ambientes distribuídos, a situação é similar a dos ambientes paralelos, a única diferença na descrição e que os elementos são nodos. A indicação de que a descrição a seguir é de uma computação em ambiente distribuído, e feita pela cláusula NET.

Exemplo:

```
CONSTRUCT NET(rede);
ELEMENT NODE N1(1);
        NODE N2(2);
        NODE N3(3);
CONNECT (N1,N2); (N2,N3); (N3,N1);   "topologia em anel"
LOCALIZE (modi,N1); (mod2,N2); (mod3,N3);
```

3. CONCLUSÃO

Em todo o projeto houve a preocupação com a implementação da linguagem. Assim, por exemplo, foi especificado um símbolo diferente para indicação de fim para cada tipo de estrutura diferente, pois isto facilita a recuperação de erros no parsing. Além disso, o tratamento de entrada e saída pode ser totalmente feito em CONSTRUCTOR, o que irá aliviar grandemente a implementação do kernel do sistema. Também foi eliminada a possibilidade de criação e destruição dinâmica de processos dentro de um programa CONSTRUCTOR, isto confere uma complexidade desnecessária ao sistema, e com vantagens duvidosas.

O projeto do sistema CONSTRUCTOR e totalmente baseado em conceitos comuns e correntes em ciência da computação, e introduz um novo estilo de programação. De uma maneira simplificada, uma descrição de uma computação em CONSTRUCTOR e equivalente a uma descrição de um sistema computacional físico (computador) mais a descrição de um programa (algoritmo mais estrutura de dados).

BIBLIOGRAFIA

|AND 82| ANDREWS, G.R. The distributed programming language SR - mechanisms, design, and implementation. Software - Practice & Experience, 12(8):719-53, 1982.

|HAN 77| HANSEN, P.B. The architecture of concurrent programs. Englewood Cliffs, Prentice-Hall, 1977.

|HAN 81| _____. Edison - a multiprocessor language. Software - Practice & Experience, 11(4):325-61, 1981.

|KEG 84a| KEGLEVICH DE BUZIN, P.F.W. Uma abordagem para a concepção e implementação de sistemas distribuídos. Porto Alegre, PGCC da UFRGS, 1984. (Dissertação de mestrado).

|KEG 84b| _____. Constructor: projeto de uma ferramenta para descrição de computações. Prêmio SERPRO de monografia âmbito nacional, primeiro lugar. Nov. 1984.

|WIR 77| WIRTH, N. A programming language for modular multipromming. Software - Practice & Experience, 7:3-35, 1977.