De occam para o transputer: compilação via reescrita de termos

Renata Spencer Augusto Sampaio
Universidade Federal de Pernambuco
Departamento de Informática
Caixa Postal 7851
Cidade Universitária
CEP 50.740-540
Recife - PE
{rcsn, acas}@di.ufpe.br

Abstract

The aim of this work is the development of a prototype compiler from occam to the transputer, in such a way that the correctness of the compiler is ensured by construction. The method used reduces the source program, through a series of algebraic transformations, to a *normal form* which can be directly mapped into the instructions of the target machine. Using the term rewriting facilities provided by OBJ3, the correctness proof gives as a byproduct a compiler prototype, so that at the end of the process, a running prototype will be available. This compilation method was already available, and its application was illustrated for a very simple language; our purpose here is to extend the method to deal with realistic languages and machines, such as occam and the transputer.

keywords:

Projeto de compiladores; reescrita de termos; verificação mecânica.

1. Introdução

O objetivo deste trabalho é o projeto de um compilador que traduza um programa fonte em occam [1] para a linguagem dos transputers [2], garantindo a corretude da compilação.

A estratégia utilizada é reduzir um programa fonte arbitrário a uma forma normal na mesma linguagem (no nosso caso, occam). Pode-se pensar na forma normal como um interpretador no qual cada ação (modelada por um comando de atribuição) pode ser mapeada diretamente para uma única instrução da máquina destino (no caso, o transputer). Esta redução para a forma normal é feita através de uma série de transformações algébricas no programa fonte. Através das leis que definem uma semântica para occam [3], podemos provar a corretude de cada uma destas transformações, ou seja, podemos provar que elas preservam a semântica do programa original ou são um *refinamento* do programa original.

Com a utilização deste método, conseguimos reduzir a complexidade do processo de prova da corretude da compilação, pois podemos utilizar a semântica algébrica da linguagem fonte, não sendo necessário trabalhar com dois domínios semânticos distintos. Note que isto é possível exatamente porque um modelo abstrato para a máquina destino é definido por uma forma normal na própria linguagem occam.

Esta estratégia de compilação consiste de pelo menos três fases: simplificação de expressões, eliminação de estruturas de controle e geração de instruções.

Este método de compilação foi definido e mecanizado em [4, 5]. Naquele momento, o objetivo era a definição da metodologia, de forma que as linguagens fonte e destino utilizadas foram extremamente simples: a linguagem fonte é uma versão da linguagem de Dijkstra e a linguagem destino é a de uma máquina que inclui um único registrador de propósito geral e um conjunto mínimo de instruções.

A finalidade aqui é estender o método com a utilização de linguagens e máquinas reais, no caso, occam e o transputer, além de atender uma necessidade do projeto PROTEM-PISH (Projeto Integrado de Software e Hardware), cujo objetivo é definir um metodologia para *hardware/software codesign* com ênfase em corretude.

Para permitir a tradução dos componentes de software (programados em occam) para o transputer, desenvolvemos um protótipo de um compilador, usando as facilidade de reescrita de OBJ3 [6]. Resolvemos inicialmente trabalhar com um subconjunto da linguagem occam, implementando as três fases da compilação citadas acima, para só então estender a implementação para alcançar todos os construtores desta linguagem.

2. A Linguagem Fonte

A linguagem fonte a ser utilizada é um subconjunto da linguagem occam, composto pelos seguintes construtores:

skip	não faz nada
x,y := e,f	atribuição (possivelmente múltipla)
SEQ(p,q)	composição sequencial
IF (b p) **(not(b) q) FI**	condicional: se **b** então **p** senão **q**
WHILE b DO p OD	iteração: enquanto **b** faça **p**
VAR v . p	declara uma lista de variáveis **v** para uso no programa **p**

Numa segunda fase do nosso trabalho, esta linguagem será estendida para compreender os demais construtores suportados em occam, como por exemplo ALT (que permite não determinismo) e PAR (composição paralela).

Como mencionado acima, adotamos uma semântica para occam que está definida em termos de leis algébricas. Cada um dos construtores acima possui o seu próprio conjunto de leis algébricas. A seguir, apresentamos alguns exemplos de leis de occam. Observe que as variáveis escritas em negrito são na realidade listas e que o operador + concatena listas.

A primeira das leis estabelece que a atribuição à uma variável do seu próprio valor não tem efeito.

- **(x + y := e + y) = (x := e)** **[identidade na atribuição]**

A próxima lei estabelece que composição de duas atribuições para uma mesma lista de variáveis é facilmente combinada em uma única atribuição:

- SEQ(**x** := **e**, **x** := **f**) = (**x** := f[**e**/**x**]), onde **f[e/x]** denota a substituição de todas as ocorrências de **x** por **e** em **f**. **[combinação de atribuições]**

A lei seguinte trata da eliminação de VAR, e determina que se uma variável declarada nunca é usada, sua declaração pode ser eliminada sem nenhum efeito. Nós intencionalmente omitimos o tipo das variáveis, porque as leis a seguir são válidas independentemente de um tipo particular para a variável "x".

- VAR x. P = P se x $\notin$ free(P), onde a condição x $\notin$ free(P) denota que a variável x não se encontra livre em P. **[VAR eliminação]**

A próxima lei trata da eliminação de atribuição no final do escopo de uma declaração.

- VAR **x**. (**x**,y:=**e**,**f**) = VAR **x**: (y:= **f**) **[atrib elim. ao final do escopo de uma declaração]**

E finalmente, a última lei a ser apresentada, representa o fato de que o SEQ executa o seu primeiro argumento até que este termine, para só então executar o resto da sequência.

- SEQ(P,**P**) = SEQ(P,SEQ(**P**)) **[SEQ assoc]**

3. OBJ3

OBJ3 é o último de uma série de sistemas OBJ [6], todos baseados em lógica equacional de primeira ordem. OBJ3 é uma linguagem declarativa de propósito geral, especialmente útil para especificação e prototipagem. Uma especificação em OBJ3 é uma coleção de módulos de duas espécies: teorias e objetos. Uma teoria tem semântica *loose*, no sentido que ela define uma variedade de modelos. Um objeto tem semântica *tight* ou *standard* e define, a menos de isomorfismo, um modelo específico - sua álgebra inicial.

Um módulo (objeto ou teoria) é a unidade de especificação. Um módulo compreende uma declaração de tipos (sorts) e operadores, e um conjunto de equações, os axiomas, que podem ser condicionais. As equações são consideradas regras de reescrita e uma computação consiste em reescrever termos, como na execução de um programa numa linguagem funcional usual.

OBJ3 provê uma notação elaborada para definir módulos. Como OBJ3 é baseado em *order sorted algebra*, tem-se a noção de *subsort*, que é extremamente conveniente na prática. Por exemplo, declarando-se os inteiros como um subsort de pontos flutuantes, nenhuma função de conversão é necessária para tranformar um número inteiro em um número de ponto flutuante. Uma sintaxe *mixfix* pode ser usada para definir operadores. Por exemplo, um operador que

implementa um comando condicional, pode ser da forma **if_then_else_** onde o símbolo **_** indica a posição dos operandos.

Operadores podem ter atributos descrevendo propriedades úteis, tais como associatividade e comutatividade.

Módulos podem ser parametrizados por teorias, que definem a estrutura e propriedades requeridas de um parâmetro real (módulo). A instanciação de um módulo genérico requer um mapeamento (*view*) das entidades na teoria (usada como parâmetro formal) para as entidades no parâmetro real.

OBJ3 também pode ser usado como um provador de teoremas, onde as computações são obtidas por reescrita de termos, que é um método de dedução amplamente difundido.

Em OBJ3, podemos definir teorias para representar a linguagem de programação que vamos usar. Nessas teorias, definidas em OBJ3, conseguimos representar os operadores da linguagem juntamente com o seu comportamento semântico (leis algébricas).

Um exemplo de teoria em OBJ3 pode ser dado pelo seguinte módulo, no qual definimos um subconjunto da linguagem de expressões de occam .

```
th EXPR is
    protecting INT .
    sort  IntVar IntExp .
    subsort  Int IntVar < IntExp .

    op _+_ : IntExp IntExp -> IntExp   [assoc, comm, idr:0] .
    op _-_ : IntExp IntExp -> IntExp   [assoc, comm, idr:0] .
    op _*_ : IntExp IntExp -> IntExp   [assoc, comm, idr:1] .
    op _/_ : IntExp IntExp -> IntExp   [assoc, comm, idr:1] .
    op _\_ : IntVar IntExp -> Bool .

    Var  x y  IntVar .
    Var  e f   IntExp .
    Var n  Int .
    Var v  Bool .

    eq x \ n = true .
    eq x \ y = (x =\= y) .
    eq x \ (e + f) = (x \ e) and (x \ f) .
    eq x \ (e - f) = (x \ e) and (x \ f) .
    eq x \ (e * f) = (x \ e) and (x \ f) .
        ...
endth
```

Na teoria acima inicialmente é importado o módulo `INT`, que especifica a álgebra dos inteiros e é pré-definido em OBJ3. Nessa importação utilizamos a palavra-chave `PROTECTING`, significando que a semântica do módulo importado não pode ser modificado pelo que o está importando. O tipo `Bool` (para a álgebra booleana) não precisa ser importado, pois sua inclusão é automática. Logo a seguir, declaramos dois tipos, `IntVar` e `IntExp` que representam respectivamente as variáveis e expressões inteiras.

Na cláusula `subsort`, definimos como os tipos se relacionam. Ou seja, podemos dizer quais tipos são subtipos de outros. No exemplo acima, os tipos `Int` (representando os inteiros) e `IntVar` (variáveis inteiras) são subtipos de `IntExp` (expressões inteiras).

O passo seguinte é definir os operadores que estarão disponíveis e a semântica destes, dada em termos de atributos e equações. Os atributos `assoc`, `comm` e `idr` representam as propriedades de associatividade, comutatividade e identidade respectivamente. A função do operador \ definido no módulo acima, é indicar se uma dada variável está presente ou não numa expressão. Podem ser também declaradas variáveis que vão ser utilizadas nas equações que descrevem o comportamento dos operadores.

O módulo seguinte exemplifica (de forma simplificada) como podem ser definidas as estruturas de controle da a linguagem occam.

```
th SINTAXE is
     protecting EXPR .
     sort Prog PairGC .

     define ListVar is LIST[Var] .
     define ListExp is LIST[Exp] .
     define ListProg is LIST[Prog] .
     define ListPairGC is LIST[PairGC] .

     subsort ListVar < ListExp .

     var e f ListVar .
     var x y ListExp .

     op skip : -> Prog .
     op _:=_ : ListVar ListExp -> Prog .
     op SEQ_ : ListProg -> Prog .
     op VAR_._ : ListVar Prog -> Prog .
     op __ : BoolExp Prog -> PairGC .
     op WHILE_DO_OD : BoolExp Prog -> Prog .
     op IF_FI : ListPairGC -> Prog .

     eq (x,y := e,y) = (x := e) .
        ...
endth
```

Os tipos Prog e PairGC representam programas e comandos com guarda, respectivamente. Em seguida são importadas várias instâncias do módulo LIST (que define listas de elementos de tipo arbitrário). O operador vírgula (,) representa a concatenação de duas listas.

Observe que os comandos de occam são definidos como operadores na teoria e suas leis como equações.

4. Forma Normal

Para representar o modelo de execução de um programa armazenado na nossa máquina destino (o transputer), usamos o seguinte programa em occam:

```
VAR v . SEQ (    P := s,
                 WHILE (s <= P < f) DO p OD    )
```

Esse programa pode ser considerado um modelo abstrato de um interpretador para a linguagem do transputer, da seguinte forma:

- a lista de variáveis v é a lista dos componentes do estado da máquina, como por exemplo os registradores. Eles são declarados como variáveis locais já que eles não existem no programa fonte e seu valor final não é relevante. Entre os componentes da lista v está o registrador P, que controla o fluxo de execução do programa (contador de programas);
- s representa a posição da primeira instrução a ser executada (o estado inicial);
- f representa o endereço seguinte ao da última instrução a ser executada;
- p é o programa a ser executado, sendo da forma:

```
IF    (P = s)      →    p1
      (P = s+1)    →    p2
      ...
      (P = f - 1)  →    pf-1    FI,
```

onde cada ramo do IF possui uma ação p_i que é executada se a condição correspondente for verdadeira.

O programa acima pode ser abreviado da seguinte forma:

Definição. (Forma Normal)

```
v[ s, p, f ] = VAR v . SEQ (    P := s,
                 WHILE s <= P <= f DO p OD    )
```

□

Para reduzir um programa arbitrário para a forma normal, é suficiente mostrar como cada comando primitivo pode ser escrito na forma normal e como cada operador da linguagem (quando aplicado a operandos na forma normal), possui resultados expressíveis na forma normal. Este processo de redução segue o princípio clássico de indução estrutural, como mostrado na seção 5.3.

5. Compilação

O processo de compilação é realizado em três etapas principais. São elas:

- Simplificação de Expressões;
- Eliminação de Estruturas de Controle;
- Geração de Instruções.

A primeira fase é a fase de simplificação de expressões, onde cada expressão é decomposta em expressões mais simples. Cada uma das expressões geradas durante esta fase, futuramente dará origem a uma instrução de máquina.

Na fase seguinte, de eliminação de estruturas de controle, tratamos das estruturas de controle (IF, WHILE, SEQ). Durante esta fase, todas as estruturas de controle são eliminadas, gerando como resultado um programa na forma normal, descrito na seção anterior.

Finalmente, na fase de geração de instruções, as expressões geradas são substituidas por instruções da máquina destino.

A seguir, apresentaremos subseções que detalham cada uma destas fases. Antes porém, apresentaremos um modelo da máquina para a qual se destina o código gerado.

5.1 Máquina Destino

A máquina destino para a qual o compilador produzirá código é o transputer. A seguir, descreveremos um modelo abstrato para o mesmo, composto pelos seguintes componentes:

- P — Um registrador sequencial, que representa o Contador de Programas
- A, B, C — Três registradores de uso geral
- M — Memória para armazenamento de variáveis (RAM)
- m — Memória para armazenamento de instruções

A idéia é representar os componentes da máquinas como variáveis dos programas e projetar as instruções como atribuições que atualizem o estado da máquina.

As variáveis que representam os registradores P, A, B, C são variáveis do tipo inteiro. A variável P deverá receber atribuições do tipo inteiro, porque representa endereços na memória ROM. As variáveis A, B e C também serão consideradas como inteiros, respeitando a política de tipos de occam (não devemos esquecer que apesar de A, B e C representarem registradores de uso geral, eles nada mais são, no nosso modelo, do que variáveis de um programa em occam). Para representarmos atribuições de variáveis booleanas definiremos funções auxiliares (*code* e *decode*) que realizem o mapeamento dos valores booleanos para inteiros.

As instruções da nossa máquina (um subconjunto das instruções do transputer) estão definidas abaixo:

- ldl (x.int) = (A, B, C, P := x.int, A, B, P+1) — (load x.int em A)
- stl (x.int) = (A, B, x.int, P := B, C, A, P+1) — (store A em x.int)
- ldl (x.bool) = (A,B,C, P := code(x.b),A,B, P+1) — (load code(x..bool) em A)
- stl (x.bool) = (A, B, x.bool, P := B, C, decode(A), P+1) — (store decode(A) em x.bool)
- add = (A,B,C, P := (A + B), C, arb, P+1) — (armazena A + B em A)
- sub = (A, B, C, P := (A - B), C, arb, P+1) — (armazena A - B em A)
- mull = (A,B,C, P := (A * B), C, arb, P+1) — (armazena A * B em A)
- div = (A, B, C, P := (A / B), C, arb, P+1) — (armazena A / B em A)
- gt = (A,B,C, P := (code(B > A), C, arb, P+1)) — (armazena code(B > A) em A)
- or = (A,B,C, P := (((A + B) - (A * B)), C, arb, P+1)) — (armazena A or B em A)
- and = (A,B,C, P := ((A * B), C, arb, P+1)) — (armazena A and B em A)
- not = (A,B,C, P := ((1 - A),B,C, P+1)) — (armazena not(A) em A)

- j(n) = (A, B, C, P) := (arb, arb, arb, n) (jump para o endereço n)
- cj(n) = if (A = false) then (A, B, C, P := arb, arb, arb, P+1)
  else (A, B, C, P := arb, arb, arb, n) (jump condicional para o endereço n)

O argumento destas instruções, na prática, são posições de memória. Aqui, por simplicidade, representamos diretamente por variáveis.

Por uma questão de clareza, representamos aqui as variáveis do tipo inteiro por x.int e as variáveis do tipo booleano por x.bool. A constante n nas instruções de jump representam um valor inteiro.

5.2 Simplificação de Expressões

Nesta fase, procuramos simplificar expressões complexas. A saída desta fase é um programa onde cada atribuição é simples, no sentido de que equivale a uma instrução de máquina. Por exemplo:

- A, B, C := x.int, A, B
- A, B, x.int := B, C, A

...

Cada uma das atribuições acima está fortemente relacionada a uma instrução de máquina. Por exemplo a primeira instrução, eventualmente se tornará um load (ldl). A segunda um store (stl), e assim por diante. A introdução da variável P representando o contador de programas só é feita na fase seguinte.

Eventualmente, outras variáveis temporárias podem surgir, quando trabalhamos com a simplificação de expressões. Mostraremos, em seguida, como trabalhar com as expressões contendo as variáveis A, B e C e as novas variáveis que eventualmente surjam.

A seguir, apresentamos as regras para a simplificação de expressões. Para ilustrar como as regras podem ser verificadas, apresentamos a prova de uma regra simples, a regra de atribuição. A prova utiliza as leis algébricas ilustradas na seção 2. A prova das demais regras é similar. Esta regra mostra como são introduzidas as variáveis A, B e C a partir de uma atribuição simples:

Regra 1: **(Atribuição)**
x := e =
VAR A, B, C : SEQ (A, B, C := e, A, B ; A, B, x := B, C, A) if (A \ e) and (A \ x)

Prova:

VAR A, B, C: SEQ (A, B, C := e, A, B ; A, B, x := B, C, A)
[pela lei de **identidade na atribuição**:]
= VAR A, B, C : SEQ (A, B, C, x := e, A, B, x ; A, B, C, x := B, C, C, A)
[pela lei de **combinação de atribuições**:]
= VAR A, B, C : (A, B, C, x := A, B, B, e)
[pela lei **atrib elim. ao final do escopo de declaração**]
= VAR A, B, C : (x := e)
[pela lei **VAR elim.**]
= x := e

Regra 2: **(Operadores Aritméticos)**
A, B, C := e aop f, A, B =
VAR t: SEQ (A, B, C := e, A, B ; A, B, t := B, C, A ; A, B, C := f, A, B ;
A, B, C := t, A, B ; A, B, C := A aop B, C, arb if A \ (e aop f)

Regra 3: **(Atribuição de Booleano)**
x.b := e.b =
VAR A, B, C : SEQ (A, B, C := code(e.b), A, B ; A, B, x.b := B, C, decode(A))
if (A \ e.b) and (A \ x.b)

Regra 4: **(Operador Relacional <)**
A,B,C := code(e.i < f.i),A,B =
VAR t : SEQ (A,B,C := e.i,A,B ; A,B,t := B,C,A ; A,B,C := f.i,A,B ; A,B,C := t,A,B ;
A,B,C := code(B > A) ,C,arb) if A \ (e.i < f.i)

Regra 5: **(Operador Relacional >)**
A,B,C := code(e.i > f.i),A,B =
VAR t : SEQ (A,B,C := f.i,A,B ; A,B,t := B,C,A ; A,B,C := e.i,A,B ; A,B,C := t,A,B ;
A,B,C := code(B > A) ,C,arb) if A \ (e.i > f.i)

Regra 6: **(Operador Booleano or)**
A,B,C := code(e.b or f.b),A,B =
VAR t : SEQ (A,B,C := code(f.b),A,B ; A,B,t := B,C,A ; A,B,C := code(e.b),A,B ;
A,B,C := t,A,B ; A,B,C := ((A + B) - (A * B)), C, arb) if A \ (e.b or f.b)

Regra 7: **(Operador Booleano and)**
A,B,C := code(e.b and f.b),A,B =
VAR t : SEQ(A,B,C := code(f.b),A,B ; A,B,t := B,C,A ; A,B,C := code(e.b),A,B ;
A,B,C := t,A,B ; A,B,C := (A * B), C, arb)) if (A \ (e.b and f.b)

Regra 8: **(Operador Booleano not)**
A,B,C := code(not(e.b)),A,B =
VAR t :SEQ(A,B,C := code(e.b),A,B ; A,B,C := (1 - A),B,C) if A \ (not(e.b))

Para exemplificar como estas regras estão definidas em OBJ3, apresentamos a seguir um trecho do módulo em que elas estão definidas.

```
th EXPRESSOES is
     extending SINTAXE .

     var x.i y.i : IntVar .
     var e.i f.i : IntExp .
     var x.b y.b : BoolVar .
     var e.b f.b : BoolExp .
     var n : Nat .
     var v : Bool .
```

```
    ops A B C t: -> IntVar .
    op arb : -> IntExp .

    eq code(true)  = 1 .
    eq code(false) = 0 .
    eq decode(1) = true .
    eq decode(0) = false .

    cq (x.i := e.i) = (VAR A,B,C . (SEQ((A,B,C :=
e.i,A,B),(A,B,x.i := B,C,A)))) if (A \ e.i) and (A \ x.i) .

    cq (A,B,C := (e.i + f.i),A,B) = VAR t . SEQ((A,B,C :=
e.i,A,B), (A,B,t := B,C,A), (A,B,C := f.i,A,B), (A,B,C := t,A,B),
(A,B,C := (A + B) ,C,arb)) if (A \ (e.i + f.i)) .

    cq (x.b := e.b) = (VAR A,B,C . (SEQ((A,B,C :=
(code(e.b)),A,B),(A,B,x.b := B,C,(decode(A)))))) if (A \ e.b) and
(A \ x.b) .

    cq (A,B,C := code(e.b or f.b),A,B) = VAR t . SEQ((A,B,C :=
code(f.b),A,B), (A,B,t := B,C,A), (A,B,C := code(e.b),A,B),
(A,B,C := t,A,B), (A,B,C := ((A + B) - (A * B)), C, arb)) if (A \
(e.b or f.b)) .
                ...
endth
```

5.3 Eliminação de Controle

Nesta fase, procuramos escrever as estruturas de controle na Forma Normal, lembrando que estamos representando o contador de programas por uma variável P. Dessa forma, controlamos o fluxo de controle dos programas pela atribuição de valores à variável P. Ou seja, se incrementarmos o valor da variável P, estaremos apontando para a instrução seguinte.

As regras a seguir constituem uma prova (por indução estrutural) da redução de um programa arbitrário à forma normal: uma regra de redução é apresentada para cada comando básico e para cada operador da linguagem.

O construtor **skip** não faz nada, de forma que pode ser representado pela seguinte expressão na forma Normal:

skip = P[s, false skip, s]

A equação seguinte mostra a representação da atribuição na forma normal.

x := e = P[s, (P = s) (x, P := e, P+1), s + 1]

A equação seguinte permite a eliminação do operador de composição seqüencial:

SEQ(P[s, e.bool p, f], P[f, f.bool q, j]) =
P[s, (e.bool or f.bool) IF (e.bool p
f.bool q) ,j]

Note que os argumento utilizados pela regra do SEQ são programas que, por hipótese de indução, estão na forma normal e o resultado também é um programa na forma normal.

Representando o condicional, temos a seguinte equação:

```
IF    e.bool       P[s, f.bool p, j]
      not(e.bool)  P[j, g.bool q, f]   =
P[s,  (P = s)            IF   (e.bool        P:=s+1
                              not(e.bool)  P := j+1),
      (P = j)            P := f,
      (s + 1< P < j)     p,
      (j + 1 < P < f)    q ,f]
```

Note que nesta equação é gerada uma instrução extra para decidir qual das cláusulas deve ser executada.

A equação seguinte lida com iteração:

```
WHILE   e.bool    DO   P[s, f.bool p, f]      =
P[s,    (P = s)   IF   (e.bool        P := P + 1,
                       not(e.bool)   P := f + 1),
        (f.bool        p),
        P = f          P := s, f+1]
```

Nesta equação é gerada uma instrução extra para decidir se o corpo do loop deve ser executado.

A prova de corretude destas transformações é feita a partir das leis algébricas de occam, como ilustrado na seção anterior. O módulo seguinte mostra a codificação destas transformações em OBJ3.

```
th CONTROLE is
     including TAG .
     vars  e s f j s0 j0 : IntExp .
     var x lv : ListVar .
     var p q : Prog .
     var e.b f.b g.b c : BoolExp .
     op P : -> IntVar .

     *** Forma Normal:
     eq (VAR lv . SEQ((P := s),
                 (WHILE((s <= P) and (P < f)) DO(IF lpgc FI)OD)))
                                          = lv[s,lpgc,f].

     [skip] eq {s} skip = P[s,false skip,s] .
     [att] eq {s} x:=e = P[s, (P = s) (x,P:=e,(P + 1)), s + 1] .
     [seq] eq SEQ((P[s, e.b p, f]), (P[f, f.b q, j]))
          = P[s, (e.b or f.b)  (IF (e.b p), (f.b q) FI), j] .
     [if] cq {s} IF (e.b P[s0, f.b p, j]),
               (not(e.b) P[j0, g.b q, f])  FI =
               P[s, (((P = s) (IF (e.b (P := s + 1)),
                                  (not(e.b) (P := j + 1)) FI)),
                     ((P = j) (P := f)),
                     ((((s + 1) <= P) and (P < j)) p),
                     ((((j + 1) <= P) and (P < f)) q))
                    ,f] if ((s0 == s + 1) and (j0 == j + 1)) .
     [while] cq {s} WHILE e.b DO (P[s0, f.b p, f]) OD =
```

```
      P[s,(((P = s) (IF(e.b (P := P + 1)),
                        not(e.b) (P := f + 1)) FI)),
                       (f.b p),
                       ((P = f) (P:=s))), f + 1] if s0 == s + 1 .
                                  ...
endth
```

No módulo acima temos a inclusão de um módulo TAG. Neste módulo, está definida uma função auxiliar, que nós chamamos de tag e que rotula cada uma das instruções com um número inteiro. Desta forma, conseguimos guardar a seqüência em que as instruções devem ser executadas na forma normal.

5.4 Geração de Instruções

Durante esta fase, as atribuições geradas no corpo da forma normal são substituídas pelos nomes que representam as instruções do transputer.

O módulo em OBJ3 é dado por:

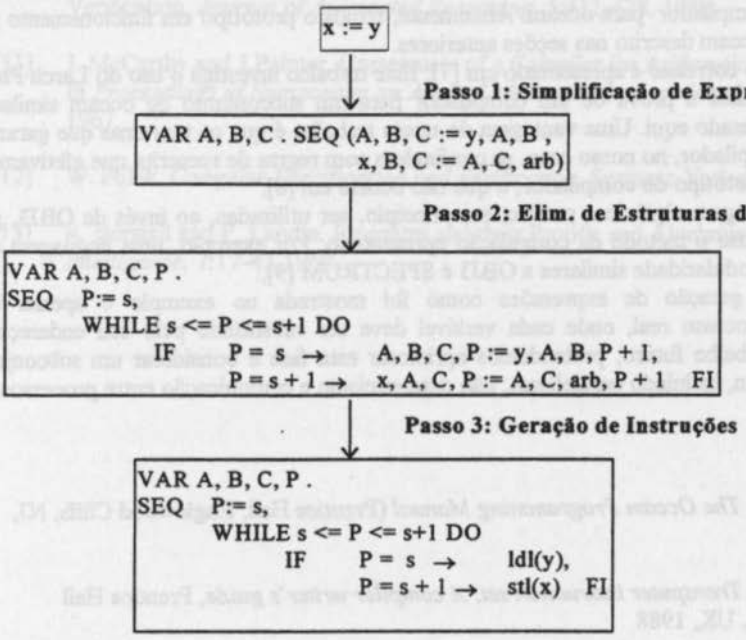
```
th INSTRUCOES
    including expressoes
    ops ldl  stl cj j : IntExp -> Prog    .
    ops ldl stl: BoolExp -> Prog .
    ops add sub mull div gt or and or and not : -> Prog.
    var x.int : IntExp .
    var x.bool : BoolExp .
    var n : Nat .

    eq (A, B, C, P := x.int, A, B, P+1) = ldl(x.int) .
    eq (A, B, x.int, P := B, C, A, P+1) = stl(x.int) .
    eq (A,B,C, P := code(x.b),A,B, P+1) = ldl(x.bool) .
    eq (A,B,x.bool, P := B,C,decode(A), P+1) = stl(x.bool) .
    eq (A,B,C, P := (A + B), C, arb, P+1) = add .
    eq (A, B, C, P := (A - B), C, arb, P+1) = sub .
    eq (A,B,C, P := (A * B), C, arb, P+1) = mull .
    eq (A, B, C, P := (A / B), C, arb, P+1) = div .
    eq (A,B,C, P := (code(B > A), C, arb, P+1)) = gt .
    eq (A,B,C,P := (((A + B) - (A * B)), C, arb, P+1) ) = or .
    eq (A,B,C, P := ((A * B), C, arb, P+1) ) = and    .
    eq (A,B,C, P := ((1 - A),B,C, P+1) ) = not    .
    eq P := n = j(n) .
    eq IF((A=1) (P := n)),(not(A=1) (P := P+1))FI = cj(n) .
endth
```

6. Um Exemplo Simples de Compilação

Um exemplo bastante simples de compilação é a compilação da atribuição x := y, onde x e y são variáveis inteiras.

Observe que a saída da terceira etapa, ainda é um programa em occam, mas que pode ser mapeado diretamente em instruções do transputer. Em particular, o código gerado neste caso é constituído apenas de duas instruções: um *load* e um *store*.

Um exemplo mais interessante de compilação usando o protótipo pode ser encontrado no Apêndice.



7. Conclusão

Uma grande variedade de abordagens tem sido sugerida para lidar com o problema de corretude de compiladores. As abordagens diferem tanto com relação ao estilo semântico adotado (operacional, denotacional, algébrico, gramática de atributos, ...) para definir linguagens fonte e destino, quanto com relação à definição de corretude associada à computação em si.

Há uma extensa literatura (em constante expansão) sobre o tópico e não há espaço aqui para uma revisão bibliográfica. Sugerimos a leitura de [4, 10] para um *survey* sobre o assunto. Alguns trabalhos se tornaram clássicos como representantes de cada um dos estilos semânticos:

[11] é baseado em semântica operacional, [12] usa semântica denotacional e [13] originou a abordagem algébrica.

A principal inovação do método aqui descrito é a caracterização do processo de compilação em um arcabouço uniforme de uma linguagem procedural (no caso, occam) com suas leis algébricas. Isto foi possível definindo-se um modelo abstrato (forma normal) para a máquina destino na própria linguagem fonte. Nas abordagens tradicionais para prova de compiladores, há usualmente uma função de codificação/decodificação entre as semânticas das linguagens fonte e destino, que adiciona complexidade ao processo.

Com o uso das propriedade de redução de OBJ3, conseguimos simular o funcionamento de cada uma das etapas de compilação. Como resultado do nosso trabalho, desenvolvemos um protótipo de um compilador para occam. Atualmente, temos o protótipo em funcionamento para o subconjunto de occam descrito nas seções anteriores.

Um trabalho correlato é apresentado em [7]. Este trabalho investiga o uso do Larch Prover (LP) [8] para verificar a prova de um compilador para um subconjunto de occam similar ao subconjunto considerado aqui. Uma vantagem do nosso trabalho é que os teoremas que garantem a corretude do compilador, no nosso caso, se confundem com regras de reescrita que efetivamente servem como um protótipo do compilador, o que não ocorre em [8].

Outras linguagem algébricas podem, em princípio, ser utilizadas, ao invés de OBJ3, para formalizar e mecanizar o método de compilação apresentado. Por exemplo, uma linguagem com características de modularidade similares a OBJ3 é SPECTRUM [9].

A fase de geração de expressões como foi mostrada no exemplo é apenas uma simplificação do processo real, onde cada variável deve ser substituida pelo seu endereço na memória. Como trabalho futuro, pretendemos aprimorar esta fase e considerar um subconjunto mais amplo de occam, incluindo paralelismo, não determinismo e comunicação entre processos.

Referências

[1] INMOS Ltd, *The Occam Programming Manual* (Prentice Hall, Englewood Cliffs, NJ, 1984)

[2] INMOS Ltd, *Transputer instruction set: A compiler writer's guide*, Prentice Hall Internacional, UK, 1988

[3] A.W. Roscoe and C.A.R. Hoare. The Laws of Occam *Programming.Theoretical Computer Science*, 60:177-299, 1988.

[4] A. Sampaio. *An Algebraic Approach to Compiler Design*. Phd thesis, Oxford University Computing Laboratory, 1993.

[5] C. A. R. Hoare, He Jifeng and A. Sampaio. Normal Form Approach to a Compiler Design. *Acta Informática*, 30:701-739, 1993.

[6] J. Goguen *et al.* Introducing OBJ. Technical report, SRI Internacional, 1993. To appear.

[7] E. A. Scott. Automated Proof of the Correctness of a Compiling Specification. In *Proceedings of Third International Conference on Algebraic Methodology and Software Technology, Workshop in Computing Series, Springer-Verlag*, 1993.

[8] S. F. Garland and J. V. Guttag. An Overview of LP, The Larch Prover. In N. Dershowitz, editor, *Proceedings of the Third Internacional Conference on Rewriting Techniques and Applications* (LNCS 355), pages 137-155. Springer-Verlag, 1989.

[9] M. Broy *et al.* The requiriment and Design Specification Language Spectrum: An Informal Introduction. Institut Für Informatik, Techische Universitat Müncher, TUM--I9140, 1992.

[10] W. R. Bevier, W. A. Hunt, J. S. Moore, and W. D. Young. An Approach to Systems Verification. *Journal of Automated Reasoning*, 5:411-428, 1989.

[11] J. McCarthy and J.Painter. Correctness of a Compiler for Arithmetic Expressions. In *Proceedings of Sumposium on Applied Mathematics*. American Mathematical Society, 1967.

[12] W. Polak. *Compiler Specification and Verification*. Springer-Verlag (LNCS 124), 1991.

[13] R. Burstall and P. Landin. Programs and their Proofs: and Algebraic Approach. *Machine Intelligence*, 7:17-43,1969.

Apêndice

O exemplo de compilação abaixo representa um programa que calcula o fatorial e foi gerado usando o protótipo.

```
ops E R : -> IntVar .
=======================================
red tag(SEQ((R := 1),WHILE E > 0 DO SEQ((R := (R * E)),
                                        (E := (E - 1))) OD))
rewrites: 1879

result Prog:

VAR A,B,C,t1 . (P[0,((P = 0)  ldl(1)),
                   ((P = 1)  stl(R)),
                   ((P = 2)  ldl(0)),
                   ((P = 3)  stl(t1)),
                   ((P = 4)  ldl(E)),
                   ((P = 5)  ldl(t1)),
                   ((P = 6)  gt),
                   ((P = 7)  cj(26)),
                   ((P = 8)  ldl(E)),
                   ((P = 9)  stl(t1)),
                   ((P = 10) ldl(R)),
                   ((P = 11) ldl(t1)),
                   ((P = 12) mul),
                   ((P = 13) stl(R)),
                   ((P = 14) ldl(1)),
                   ((P = 15) stl(t1)),
                   ((P = 16) ldl(E)),
                   ((P = 17) ldl(t1)),
                   ((P = 18) sub),
                   ((P = 19) stl(E)),
                   ((P = 20) ldl(0)),
                   ((P = 21) stl(t1)),
                   ((P = 22) ldl(E)),
                   ((P = 23) ldl(t1)),
                   ((P = 24) gt),
                   ((P = 25) j(7)),    26])
```

Neste exemplo, t_1 é uma variável inteira temporária gerada durante o processo de compilação. O índice é gerado automáticamente, como um artifício para que tenhamos diferentes variáveis, quando necessário.