Uma Semântica de ações para SASL

Tópicos Avançados em Linguagens de Programação 1

Pós-Graduação em Ciência da Computação
Curso de Mestrado em Ciência da Computação
TALP1- Módulo I – Projeto
Prof. Hermano Perrelli de Moura
Aluno: José Airton Salzano Filho

Recife, 16 de Novembro de 1998

1 Introdução

SASL (St. Andrews Static Language) é uma linguagem funcional estrita e bastante simples que se baseia no λ-Calculus e possui poucos construtores de tipos. Ela foi concebida em 1976 pelo então Prof. Assistente David Turner para ensino de graduação na Universidade de St. Andrews na Escócia.

A linguagem que iremos considerar neste trabalho foi obtida da gramática original de SASL realizando algumas pequenas alterações e suprimindo as partes relacionadas a utilização de ZF, casamento de padrões, e dos tipos de dados real, caractere e String.

2 Descrição Informal de SASL

SASL (St. Andrews Static Language) é uma linguagem funcional pura e bastante simples que se baseia no λ-Calculus e possui poucos construtores de tipos. Ela foi concebida em 1976 pelo então Prof. Assistente David Turner para ensino de graduação na Universidade de St. Andrews na Escócia.

A linguagem que iremos considerar neste trabalho foi obtida da gramática original de SASL realizando algumas pequenas alterações e suprimindo as partes relacionadas a utilização de ZF, casamento de padrões, e dos tipos de dados real, caractere e String.

2.1 Estrutura de um Programa

A Estrutura de um programa em SASL é baseada em um conjunto de definições de funções e uma expressão a ser avaliada. O bloco de definições de funções é iniciado com ‘DEF’ e terminado com ‘END’. A expressão a ser avaliada é formada por qualquer expressão válida em SASL terminada com uma interrogação: ‘?’.

Sintaxe

<Program> ::= DEF <declr> END. <expr> ?

Semântica

Um programa SASL consiste em uma série de declaração de funções seguida de uma expressão a ser avaliada. A execução do programa consiste na avaliação da expressão a ser avaliada. A avaliação deverá considerar o ambiente contento as funções definidas no script do programa corrente.

Em SASL, Todas as palavras chave são maiúsculas: DEF, FUN, ENDFUN, END. Existe distinção entre os identificadores minúsculos e maiúsculos. Os tipos usam a notação currificada e não existem tuplas.

Exemplo

```
DEF
FUN fib s :: Int->Int;
  fib x = if (x<2)
  then 1
  else fib (x-2) + fib (x-1)
ENDFUN;
END.
fib 30?
```

2.2 Estrutura de Tipos

Os tipos em SASL usam a notação currificada e não existem tuplas. Tipos podem ser inferidos pelo compilador, assim, as definições explícitas podem ser omitidas. Entretanto, não estamos interessados em dar uma semântica para a inferência de tipos na linguagem, portanto consideraremos que os tipos estarão explicitamente declarados neste trabalho. SASL possui tipos básicos e estruturados. Os básicos são inteiros e booleanos, os estruturados, listas e funções.

Sintaxe

```
<type>      ::= <stype> | <rtype>
<rtype>     ::= <type>-><type>
<stype>     ::= List <type> | Int | Bool
```

Semântica

1. Tipos Básicos

Os Tipos básicos de SASL são: inteiro (Int) e lógico (Bool):

- Inteiros: Representam o conjunto de números inteiros.
- Lógicos: Representam os dois possíveis valores lógicos: TRUE e FALSE.

2. Tipos Estruturados

Existem dois tipos estruturados em SASL: Listas e funções

- Listas: Uma lista é uma seqüência de elementos de mesmo tipo, em que apenas pode ser acessado o primeiro elemento. É representado pela notação de tipo: List X, onde X é o tipo dos elementos que a compõem. Na criação de uma lista, utiliza-se o construtor de listas (:) que insere um elemento no início da lista. Para acessar o primeiro elemento da lista, utiliza-se o operador HD e para acessar a cauda da lista utiliza-se o operador TL. Em geral, as listas terminam numa lista vazia ([]), que indica o fim da lista.

- Funções: Em SASL funções podem ser utilizadas como tipos de dados (funções de alta ordem), ou seja, podem ser passadas para ou retornadas como o resultado da avaliação de outras funções. Uma aplicação parcial de uma função é também considerada uma função.

Exemplo

```
FUN tamanho;
  Tamanho x = if x == [] then 0
      Else 1 + tamanho (TL x);
ENDFUN;
```

- O tipo inferido para a função acima é: List Int -> Int
- A Tradicionalmente referenciada função map para inteiros tem em SASL o seguinte tipo: map :: (Int->Int)->List Int->List Int : é o tipo da função map

2.3 Declarações

Declarações em SASL é uma série de zero, uma, ou mais definições de função. Zero, porque por exemplo um programa SASL, pode não conter nenhuma nova declaração em seu script, e simplesmente pretender avaliar uma expressão do tipo: 2 + 3 ?. Esta expressão não precisa da definição de nenhuma nova expressão para ser avaliada. Uma definição de função inclui o nome da função, o tipo e o corpo da função. O corpo da função é uma expressão que geralmente expressa uma relação entre os parâmetros da função. Importante salientar que a função constante, que é uma função sem parâmetros eqüivale a declarações do tipo: val x = 3, ou const Pi = 3,14, tais como encontramos em outras linguagens de programação, inclusive funcionais.

Sintaxe

```
<declr>       ::= <definition> | <definition> <declr>
<definition>  ::= FUN <name> :: <type>; <defs> ENDFUN;
<defs>        ::= <combname> = <expr>
<combname>    ::= <combname> <name> |  <name>
```

Semântica

Uma declaração em SASL é uma função. Uma função deve Ter nome, tipo e lei de formação, ou seja, uma expressão a partir da qual relacionamos elementos do domínio com suas respectivas imagens.

Exemplo

```
DEF
FUN suc :: Int->Int;
          suc  x = x + 1;
ENDFUN;
...
FUN double :: Int->Int
         double  x = x + x;
ENDFUN;
...
FUN unidade :: Int;
        unidade = 1;
ENDFUN;
...
```

- No exemplo acima temos três declarações: a da função suc, a da função double, e a da função constante unidade.

2.4 Expressões

O corpo de uma função declarada é uma expressão. Também é uma expressão em SASL o que devemos avaliar para termos o resultado de um programa, ou seja o resultado da execução de um programa SASL é o da avaliação de uma expressão, a expressão seguida de uma interrogação, que é armada pelo programador quando da elaboração do script funcional.

Sintaxe

```
<expr>          ::=IF <opexpr> THEN <expr> ELSE  <expr> | <opexpr>
<opexpr>        ::= <expression> | <comb>
<expression>    ::= <prefix> <opexpr> | <opexpr> <infix> <opexpr>
<comb>          ::= <comb> <simple> | <simple>
<simple>        ::= <name> | <constant> | (<expr>)
```

Semântica

Existem quatro tipos de expressões: as expressões simples, as criadas por combinações de termos e operadores, as que envolvem chamadas de função e as compostas por expressões condicionais, criadas por comandos IF-THEN-ELSE.

1. Expressões Simples

Expressões Simples são variáveis ou constantes:

- Constantes: são valores que possuem um tipo e não podem ser modificados no decorrer do programa. Existem quatro tipos de constantes

em SASL: inteiras (Int), booleanas (Bool), Listas ([]), e nomes de funções que são na verdade identificadores.
Exemplos: 1, 200, -2, TRUE, FALSE, [1,2,3].

- Variáveis: são identificadores compostos de letras e números, onde o primeiro caracter deve ser uma letra. Não existe limite quanto ao tamanho dos identificadores. Variáveis são usadas apenas como parâmetro para uma função e são usadas no contexto da mesma.
  Exemplos: x, y3453, par

2. Expressões Compostas

Expressões compostas são aquelas combinadas pelo uso de operadores. Esses operadores em SASL dividem-se em quatro tipos: aritméticos, relacionais, lógicos e operadores de Lista.

- Operadores Aritméticos: Existem cinco operadores que podem ser aplicados apenas para expressões do tipo inteiro (Int). São eles: + (adição), - (subtração), * (multiplicação), / (divisão), % (resto da divisão).

- Operadores Relacionais: Existem seis operadores de comparação, que podem ser aplicados apenas para expressões do tipo Inteiro (Int). São eles: > (maior), >= (maior ou igual), < (menor), <= (menor ou igual), == (igual), = (diferente).

- Operadores Lógicos: Existem quatro operadores que podem ser aplicados apenas para expressões do tipo Bool (lógico). São eles: AND (e lógico), OR (ou lógico), XOR (ou lógico exclusivo), NOT (negação).

  Exemplo:

  ```
  DEF
  FUN equal;
  Equal a b = a AND b OR ( (NOT A) AND (NOT B) );
  ENDFUN;
  END.
  Equal TRUE FALSE
  ```

- Operadores de Lista: Existem operadores para construção de uma lista, bem aqueles que servem para selecionar os elementos armazenados na mesma. São eles: :(cons) (construtor de lista), HD (seletor da cabeça da lista), TL (seletor da cauda da lista).

Exemplo:

```
DEF
FUN concat;
   Concat a b = if (a==[])
              then b
              else (HD a) : (concat (TL a) b);
ENDFUN;
FUN gl;
   gl n = if (n==0)
      then []
      else n: (gl (n-1));

ENDFUN
END.
Concat (gl 200) (gl 300) ?
```

Precedência dos Operadores:

A Precedência entre os operadores é na ordem: :, OR, AND, ==, =, >, <, >=, <=, +, -, *, /, %, NOT.

3. Expressões Condicionais

A sintaxe do comando IF-THEN-ELSE é a seguinte:

- IF expressão1 THEN expressão2 ELSE expressão3

Semântica:

- expressão1 deve ser do tipo Bool e expressão2 e expressão3 devem ser do mesmo tipo.
- Caso expressão1 seja TRUE, o retorno deste comando é expressão2, caso contrário retorna expressão3.
- Os comandos IF-THEN-ELSE podem ser aninhados.
- O ELSE é obrigatório no comando.

Exemplo:

```
DEF
FUN maior Int->Int->Bool
   maior a b = if (a>b)
          then TRUE
          else FALSE
ENDFUN;
END.
maior 10 5 ?
```

4. Funções

Funções são definidas da seguinte forma, já que são declarações:

```
FUN F1 [::tipo];
    F1 [parâmetros] = expressão.
ENDFUN;
```

Sendo que:

- O tipo é da forma tipo -> tipo -> ... ->tipo, onde o último tipo especifica o tipo de retorno e os demais especificam os tipos dos parâmetros, na ordem indicada. Se o tipo não for especificado, normalmente o compilador cuidará de inferir o tipo apropriado, usando o tipo mais geral, sempre que possível, permitindo polimorfismo, entretanto, como referido anteriormente, neste trabalho consideramos que tipos são explicitamente indicados.
- Parâmetros: especificam os parâmetros passados para a função.

Exemplo

```
DEF
FUN maior Int->Int->Bool
   maior a b = if (a>b)
            then TRUE
            else FALSE
ENDFUN;
END.
maior 10 5 ?
```

3 Sintaxe Abstrata de SASL

<Program>	::= DEF <declr> END. <expr> ?
<declr>	::= <definition> | <definition> <declr>
<definition>	::= FUN <name> :: <type>; <defs> ENDFUN;
<type>	::= <stype> | <rtype>
<rtype>	::= <type>-><type>
<stipe>	::= List <type> | Int | Bool
<defs>	::= <combname> = <expr>
<combname>	::= <combname> <name> | <name>
<expr>	::= IF <opexpr> THEN <expr> ELSE <expr> | <opexpr>
<opexpr>	::= <expression> | <comb>
<expression>	::= <prefix> <opexpr> | <opexpr> <infix> <opexpr>
<comb>	::= <comb> <simple> | <simple>
<simple>	::= <name> | <constant> | (<expr>)
<constant>	::= <Intnumber> | <Boolconst> | []
<prefix>	::= NOT | HD | TL
<infix>	::= + | ... | > | ... | EXOR
<Intnumber>	::= <digit>+
<name>	::= <char><digit_or_char>*

BNF de SASL

4 Entidades Semânticas

Nesta seção definiremos entidades semânticas utilizadas na descrição da semântica de ações de SASL.

4.1 Valores (Values)

Intnumber = integer

Boolconst = truth-value

value = Intnumber | Boolconst

4.2 Bindings

letter = char

token = string of (letter, (letter | digit)*)

bindable = value | declr | cell

4.3 Funcitons

declr = function

function = abstraction

fun-argument = value

fun-result = value

5 Funções Semânticas

Nesta seção introduziremos as funções que dão a semântica de um programa funcional em SASL em termos de ações.

5.1 Programa (Program)

run

Função Semântica: run _

- run _ :: Program -> action

1. run [<P: Program> ::= DEF <D: declr> END. <E: expr> ?] =
   | elaborate D
   hence
   | eval E

5.2 Declarações (declr)

elaborate

Função Semântica: elaborate _

- elaborate _ :: declr -> action

<declr> ::= <definition> | <definition> <declr>
<definition> ::= FUN <name> :: <type>; <defs> ENDFUN;
<defs> ::= <combname> = <expr>
<combname> ::= <combname> <name> | <name>

1. elaborate [D: definition] =
   elaborate-fun D

2. elaborate [D1: definition D2: declr]
    | elaborate-fun D1
    before
    | elaborate D2

elaborate-fun

Função Semântica: elaborate-fun _

- elaborate-fun _ :: definition -> action

<declr> ::= <definition> | <definition> <declr>
<definition> ::= FUN <name> :: <type>; <defs> ENDFUN;

1. elaborate-fun[FUN <N: name> :: <type>; <D: defs> ENDFUN;] =
    recursively bind token N to closure abstraction
    | elaborate-def D

elaborate-def

Função Semântica: elaborate-def _

- elaborate-def _ :: defs -> action

<defs> ::= <combname> = <expr>
<combname> ::= <combname> <name> | <name>

1. elaborate-def [C: combname = E: expr] =
    | furthermore elaborate-fun-formals combname
    hence
    | elavuate E then give the fun-result

elaborate-fun-formals

Função Semântica: elaborate-fun-formals _

- elaborate-fun-formals _ :: combname -> action

<defs> ::= <combname> = <expr>
<combname> ::= <combname> <name> | <name>

1. elaborate-fun-formals [] = complete

2. elaborate-fun-formals [N: name] =
    | bind token-of N to the fun-argument

3. elaborate-fun-formals [N: name C:combname] =
    | bind token-of N to the fun-argument
    and then
    | elavuate-fun-formals C

5.3 Expressão (expr)

eval

Função Semântica: eval _

- eval _ :: expr -> action

<Program> ::= DEF <declr> END. <expr> ?
<expr> ::=IF <opexpr> THEN <expr> ELSE <expr> | <opexpr>
<opexpr> ::= <expression> | <comb>
<expression> ::= <prefix> <opexpr> | <opexpr> <infix> <opexpr>
<comb> ::= <comb> <simple> | <simple>
<simple> ::= <name> | <constant> | (<expr>)

1. eval [IF E1: opexpr THEN E2: expr ELSE E3: expr] =
    | eval-opexpr E1
    then
    || eval E2
    | else
    || eval E3

2. eval [E1: opexpr] =
    eval-opexpr E1

eval-opexpr

Função Semântica: eval-opexpr _

- eval-opexpr _ :: opexpr -> action

<opexpr> ::= <expression> | <comb>
<expression> ::= <prefix> <opexpr> | <opexpr> <infix> <opexpr>
<comb> ::= <comb> <simple> | <simple>
<simple> ::= <name> | <constant> | (<expr>)

1. eval-opexpr [E: expression] =

| eval-expression E

2. eval-opexpr [C: comb] =
   eval-comb C

eval-expression

Função Semântica: eval-expression _

- eval-expression _ :: expression -> action

<opexpr> ::= <expression> | <comb>
<expression> ::= <prefix> <opexpr> | <opexpr> <infix> <opexpr>
<comb> ::= <comb> <simple> | <simple>
<simple> ::= <name> | <constant> | (<expr>)

1. eval-expression [P: prefix E: opexpr] =
   | eval-opexpr E
   then
   | apply-prefix-operator P

2. eval-expression [E1: Opexpr I: infix E2: Opexpr] =
   | | eval E1 then give the value label #1
   | and
   | | eval E2 then give the value label #2
   then
   | apply-operator I

eval-comb

Função Semântica: eval-comb _

- eval-comb _ :: comb -> action

<opexpr> ::= <expression> | <comb>
<expression> ::= <prefix> <opexpr> | <opexpr> <infix> <opexpr>
<comb> ::= <comb> <simple> | <simple>
<simple> ::= <name> | <constant> | (<expr>)

1. eval-comb [S: simple] =
   | eval-simple S

2. eval-comb [C: comb S: simple] =
   | eval-simple S
   then
   | enact (the function bound to token-of C)

eval-simple

Função Semântica: eval-simple _

- eval-simple _ :: simple -> action

<opexpr> ::= <expression> | <comb>
<expression> ::= <prefix> <opexpr> | <opexpr> <infix> <opexpr>
<comb> ::= <comb> <simple> | <simple>
<simple> ::= <name> | <constant> | (<expr>)

1. eval-simple [N: name] =
   | give the value bound to token-of N

2. eval-simple [C: constant] =
   eval-constant C

3. eval-simple [(E: expr)]
   eval E

eval-constant

Função Semântica: eval-constant _

- eval-constant _ :: constant -> action

<expression> ::= <prefix> <opexpr> | <opexpr> <infix> <opexpr>
<comb> ::= <comb> <simple> | <simple>
<simple> ::= <name> | <constant> | (<expr>)

1. eval-constant [I: Intnumber] =
   | give valuation-of I

2. eval-constant [B: Boolconst] =
   | give boolvaluation-of B

apply-prefix-operator

Função Semântica: apply-prefix-operator _

- apply-prefix-operator _ :: prefix -> action

1. apply-prefix-operator [NOT] = give not (the truth value)

2. apply-prefix-operator [HD] = give head-of (the list)

3. apply-prefix-operator [TL] = give tail-of (the list)

apply-operator

Função Semântica: apply-operator _

- apply-operator _ :: infix -> action

1. apply-operator [+] = give sum (the integer #1, the integer #2)

2. apply-operator [-] = give difference (the integer #1, the integer #2)

3. apply-operator [*] = give product (the integer #1, the integer #2)

4. apply-operator [/] = give integer-quotient (the integer #1, the integer #2)

5. apply-operator [%] = give difference (the integer #1, product(integer-quotient (the integer #1, the integer #2),the integer #2))

6. apply-operator [>] = give is-greater-than (the integer #1, the integer #2)

7. apply-operator [>=] = give either (is-greater-than (the integer#1, the integer#2), the integer #1 is the integer #2)

8. apply-operator [<] = give is-less-than (the integer #1, the integer #2)

9. apply-operator [<=] = give either (is-less-than (the integer #1, the integer #2), the integer #1 is the integer #2)

10. apply-operator [==] = give (the primitive-value #1 is the primitive-value #2)

11. apply-operator [=] = give not (the primitive-value #1 is the primitive-value #2)

12. apply-operator [AND] = give both (the truth-value #1, the truth-value #2)

13. apply-operator [OR] = give either (the truth-value #1, the truth-value #2)

14. apply-operator [XOR] = give either(both (not (the truth-value #1), the truth-value #2), both (the truth-value #1, not (the truth-value #2)))

6 Sintaxe Léxica

Esta seção inclui o conjunto de tokens válidos, reconhecidos e corretamente classificados em SASL ,de acordo com uma definição léxica apresentada.

Gramática:

name = letter | name letter | name digit

Intnumber = digit | IntNumber digit

digit = 0 | 1 | ... | 9

letter = a | b | ... | z | A | B| .... | Z

6.1 Nomes (name)

- token-of _ :: name -> token

token-of[var] = var

6.2 Valores (value)

- valuation-of _ :: IntNumber -> integer

valuation-of[0] = 0

valuation-of[1] = 1

valuation-of[2] = 2

valuation-of[3] = 3

valuation-of[4] = 4

valuation-of[5] = 5

valuation-of[6] = 6

valuation-of[7] = 7

valuation-of[8] = 8

valuation-of[9] = 9

valuation-of[ND] = sum (procuct(valuation-of N, 10), valuation-of D)

- boolvaluation-of _ :: Boolconst -> truthvalue

boolvaluation-of[True] =True

boolvaluation-of[False] = False

7 Uma Semântica para Extensões de SASL

Uma questão a considerar é como dar uma semântica para extensões de SASL. Notadamente para isso precisamos ampliar a gramática da linguagem de forma que esta passe a compreender as novas características introduzidas.

Uma linguagem funcional como SASL pode ser enriquecida de várias formas para que seu poder de expressão sintático seja incrementado. Nesta seção faremos uma descrição informal de como poderíamos acrescentar definições de declarações locais em SASL.

7.1 Declarações Locais

Uma Declaração local é uma declaração cujo contexto fica restrito ao escopo do corpo da função em que foi declarada. Declarações locais em linguagens funcionais são geralmente cláusulas do tipo Let: LET ... IN ... END, ou uma cláusula Where: dobro ... WHERE dobro = 2.

SASL poderia comportar uma declaração local do tipo Let, extendendo sua BNF da seguinte forma:

<expr> ::= ... | LET <expr> IN <expr> END

7.2 Semântica

Uma declaração local pode ser encarada apenas como um açucaramento sintático da linguagem. Se for observado que variáveis locais definidas são únicas, isto é, não conflitam com identificadores de outras variáveis previamente definidas, podemos considerar que a declaração local é uma declaração como outra qualquer da linguagem, sem nenhuma distinção semântica, acrescentada também a restrição de que referências a esta variável só são validas no escopo da definição da declaração local. Desta forma podemos considerar que do ponto de vista semântico não há novidades na extensão de SASL com declarações locais, sendo aproveitada a semântica previamente definada para declarações de uma forma geral, como apresentada neste trabalho.

7.3 λ-Lifting

Uma alternativa para a implementação de declarações locais sem alteração na semântica da linguagem é a utilização de transformação de programas para a introdução desta nova característica da linguagem. Uma alternativa de implementação de declarações locais em uma linguagem funcional como SASL é a utilização do λ-Lifting. O λ-Lifting consiste na elavação de declarações locais ao escopo da função definida mais externamente através do acréscimo de tantos parâmetros à função quantas forem as variáveis declaradas localmente.

Uma análise mais aprofundada que o λ-Lifting é desvantajoso, enquanto alternativa de implementação de declarações locais, uma vez que tende a aumentar demasiadamente a aridade das funções, o que o torna, em geral, ineficiente. Por outro lado, demonstra que através de transformação de programas, que podem ser provadas corretas, é possível extendermos a linguagem ser termos que definir uma semântica para estas extensões, uma vez que podemos considera a linguagem objeto, resultante das transformações a qual já possui toda semântica devidamente definida.

8 Conclusão

Neste trabalho a linguagem funcional SASL e definimos uma semântica de ações para um subconjunto significativo da linguagem. Inicialmente descrevemos informalmente a linguagem, em seguida iniciamos a descrição de uma semântica de ações para SASL. Apresentamos as entidades semânticas e as funções semânticas que a linguagem em termos de ações. Vimos a sintaxe léxica da linguagem e tecemos considerações acerca do implemento de extensões da linguagem estudada através de uma descrição informal de uma semântica para estas extensões.