INF05516 - Semântica formal N
Ciência da Computação - UFRGS
2006-2

Marcus Ritt
mrpritt@inf.ufrgs.br

13/09/2006

Extensões simples 2 / 30

Extensões simples

- Comparação de expressões booleanas

$$\frac{\gamma \vdash \mathbf{e_1} : \tau \quad \gamma \vdash \mathbf{e_2} : \tau}{\gamma \vdash \mathbf{e_1}\ \mathbf{op}\ \mathbf{e_2} : \mathrm{bool}}\ \text{t-ae com } \mathbf{op} \in \{=, <\}$$

- Expressões condicionais

$$e ::= \cdots \mid \texttt{if}\ \mathbf{e_1}\ \texttt{then}\ \mathbf{e_2}\ \texttt{else}\ \mathbf{e_3} \mid \cdots$$

Exercício!

Constantes 4 / 30

Sintaxe para constantes

- Suponhe que ao lado da declaração de um identificador

$$\mathbf{c} ::= \cdots \mid \texttt{let var x:=e in c} \mid \cdots$$

  permitimos a declaração de constantes

$$\mathbf{c} ::= \cdots \mid \texttt{let var x=e in c} \mid \cdots$$

- Vantagem/desvantagem?

Sistema de tipos

- O sistema de tipos tem que recusar programas como

```
let var x:=2 in
  let const x=3 in
    x:=x+1
```

  que tentam de atribuir uma constante.
- Portanto, o conjunto de tipos diferencia entre constantes e variáveis; além dos tipos

$$\tau \in T = \{\text{int}, \text{bool}\}$$

  vamos introduzir qualificadores

$$q \in Q = \{\text{var}, \text{const}\}$$

Ambiente de tipos

- O ambiente de tipos agora "lembre-se" do tipo junto com o qualificador:

$$\gamma \in \Gamma = [\text{Ident} \rightharpoonup T \times Q]$$

- com as relações
  - $\vdash_E \subseteq \Gamma \times \text{Exp} \times T$
  - $\vdash_C \subseteq \Gamma \times \text{Com} \times \{\text{void}\}$

Regras de tipos

- A leitura de variáveis ou constantes ignora a qualificação

$$\frac{\gamma(\mathbf{x}) = (\tau, q)}{\gamma \vdash_E x : \tau}\ \text{t-ident}$$

- Mas a atribuição é só possível no caso de variáveis

$$\frac{\gamma(\mathbf{x}) = (\tau, \text{var}) \qquad \gamma \vdash_E \mathbf{e} : \tau}{\gamma \vdash_C \mathbf{x} := \mathbf{e} : \text{void}}\ \text{t-assign}$$

Regras de tipos

- Para a chegacem de tipos de declarações, o ambiente de tipos vai ser aumentado com o tipo e o qualificador correspondente

$$\frac{\gamma \vdash_E \mathbf{e} : \tau \qquad \gamma[\mathbf{x} \mapsto (\tau, \text{var})] \vdash_E c : \text{void}}{\gamma \vdash_C \texttt{let var } \mathbf{x}\texttt{:=}\mathbf{e} \texttt{ in } \mathbf{c} : \text{void}}\ \text{t-letvar}$$

$$\frac{\gamma \vdash_E \mathbf{e} : \tau \qquad \gamma[\mathbf{x} \mapsto (\tau, \text{const})] \vdash_E c : \text{void}}{\gamma \vdash_C \texttt{let const } \mathbf{x}\texttt{=}\mathbf{e} \texttt{ in } \mathbf{c} : \text{void}}\ \text{t-letconst}$$

Exemplo

O exemplo do começo vai ser recusado mesmo?

Exemplo

```
let const x = 3 in
  let var x:=x+3 in
    let const y = 5 in
      x := x+y
```

Semântica operacional

- O estado que foi usado para variáiveis inteiros e booleanas serve ainda

$$\sigma \in \Sigma = [\mathrm{Ident} \rightharpoonup V]$$

- Até a regra para `let var` fica a mesma:

$$\frac{\mathbf{e}, \sigma \Downarrow v \qquad \mathbf{c}, \sigma[x \mapsto v] \Downarrow \sigma'}{\texttt{let var x:=e in c}, \sigma \Downarrow \sigma'[x \mapsto \sigma(x)]}\ \text{letvar}$$

- A regra para `let const` é identica com a regra de `let var`:

$$\frac{\mathbf{e}, \sigma \Downarrow v \qquad \mathbf{c}, \sigma[x \mapsto v] \Downarrow \sigma'}{\texttt{let const x=e in c}, \sigma \Downarrow \sigma'[x \mapsto \sigma(x)]}\ \text{letconst}$$

Variáveis e constantes 13 / 30

Semântica

- Tanto as constantes quanto as variáveis denotam valores no estado.
- Observe como na atribuição

```
x:=x+1
```

`x` denota um *endereçõ ou local na memôria* no lado esquerdo, mas um *valor* no lado direito.

- Semânticamente, uma variável tem uma denotação que depende do contexto.

Identificadores, endereços e memôria

- Para resolver isso vamos separar o *ambiente de execução* e a *memôria*.
- Temos os valores armazenáveis na memôria

$$v \in V = Z \cup B$$

- Para separar os identificadores da memôria serve um conjunto de endereços $l \in \mathrm{End}$.
- O ambiente de execução contém a denotação dos nomes, que inclui os endereços

$$\rho \in \mathrm{AmbExec} = [\mathrm{Ident} \rightharpoonup V \cup \mathrm{End}]$$

- e a memôria contém os valores atuais das variáveis

$$\sigma \in \Sigma = [\mathrm{End} \rightharpoonup V]$$

Sistema de tipos

- O que mudo no sistema de tipos?

Semântica operacional

- Além do estado temos um ambiente de execução.
- Portanto, precisamos uma nova notação que inclui o ambiente de execução.

$$\rho \vdash \mathbf{c}, \sigma \Downarrow \sigma'$$

- Como só o estado vai ser modificado na execução, o resultado de um comando é um estado.
- O ambiente de execução nunca vai ser modificado, só aumentado.
- A notação $\rho \vdash$ mostra isso.

Semântica: Expressões

$$\frac{\rho(\mathbf{x}) = v}{\rho \vdash_E \mathbf{x}, \sigma \Downarrow v}\ \text{ident}$$

$$\frac{\rho(\mathbf{x}) = l}{\rho \vdash_E \mathbf{x}, \sigma \Downarrow \sigma(l)}\ \text{ident}$$

Semântica: Comandos

$$\frac{\rho \vdash_E \mathbf{e}, \sigma \Downarrow v}{\rho \vdash_C \mathbf{x}\texttt{:=}\mathbf{e}, \sigma \Downarrow \sigma[\rho(\mathbf{x}) \mapsto v]}\ \text{assign}$$

$$\frac{\rho \vdash \mathbf{e}, \sigma \Downarrow v \qquad l \notin \mathrm{dom}(\sigma) \qquad \rho[\mathbf{x} \mapsto l] \vdash \mathbf{c}, \sigma[l \mapsto v] \Downarrow \sigma'}{\rho \vdash \texttt{let var } \mathbf{x}\texttt{:=}\mathbf{e} \texttt{ in } \mathbf{c}, \sigma \Downarrow \sigma'}\ \text{letvar}$$

$$\frac{\rho \vdash \mathbf{e}, \sigma \Downarrow v \qquad \rho[\mathbf{x} \mapsto v]\mathbf{c}, \sigma \Downarrow \sigma'}{\texttt{let const } \mathbf{x}\texttt{=}\mathbf{e} \texttt{ in } \mathbf{c}, \sigma \Downarrow \sigma'}\ \text{letconst}$$

Discussão

- Com memôria separada, não restauramos o valor da variável local. Conseqûencias?
- As regras "ident" implementam um teste dinâmico de tipos.
- Eles refletem, que semânticamente valores e endereços são diferentes.
- Uma definição mais elegante e ortogonal seria de diferenciar entre valores e endereços explicitamente:

  $$\mathbf{e} ::= \cdots | \texttt{ref } \mathbf{e} | \texttt{!}\mathbf{e} | \cdots$$

  (Compare com OCaml!)
- O que acontece se permitimos endereços na memôria também?

Procedimentos

- Suponhe agora, que queremos aumentar IMP com procedimentos.
- Qual seria uma abordagem?
- Vamos aumentar os comandos com mais um tipo de declaração

$$\mathbf{c} ::= \cdots | \texttt{let proc } \mathbf{p}(\mathbf{x}{:}\tau)\texttt{=}\mathbf{c} \texttt{ in } \mathbf{c}' | \cdots$$

- e com a sintaxe para *chamar* os procedimentos

$$\mathbf{c} ::= \cdots | \mathbf{p}(\mathbf{e}) | \cdots$$

- Quais são os modificações necessárias?
- Porque declaramos o tipo do argumento?

Sistema de tipos

- Um procedimento precisa mais um qualificador, logo

$$q \in Q = \{\mathrm{var}, \mathrm{const}, \mathrm{proc}\}$$

- Com isso, o ambiente de tipos fica

$$\gamma \in \Gamma = [\mathrm{Ident} \rightharpoonup T \times Q]$$

- com as mesmas relações
  - $\vdash_E \subseteq \Gamma \times \mathrm{Exp} \times T$
  - $\vdash_C \subseteq \Gamma \times \mathrm{Com} \times \{\mathrm{void}\}$

Regras de tipo

- Chamada de procedimentos

$$\frac{\gamma \vdash_E \mathbf{e} : \tau \qquad \gamma(\mathbf{p}) = (\tau, \text{proc})}{\gamma \vdash_C \mathbf{p(e)} : \text{void}} \text{ t-call}$$

- Declaração de procedimentos

$$\frac{\gamma[\mathbf{x} \mapsto (\tau, \text{const})] \vdash_C \mathbf{c} : \text{void} \qquad \gamma[\mathbf{p} \mapsto (\tau, \text{proc})] \vdash_C \mathbf{c}' : \text{void}}{\gamma \vdash_C \texttt{let proc } \mathbf{p}\texttt{(}\mathbf{x}\texttt{:}\tau\texttt{)=}\mathbf{c} \texttt{ in } \mathbf{c}' : \text{void}} \text{ t-letproc}$$

- Observações:
  - Não é permitido de atribuir valores para os argumentos.
  - O ambiente de tipos foi "abusado" no caso de procedimentos para guardar o tipo de argumento.

Semântica operacional

- Na execução de uma chamada de procedimento precisamos saber
  - o identificador do argumento
  - o comando para executar
- Portanto o ambiente de execução é

$$\rho \in \text{AmbExec} = [\text{Ident} \rightharpoonup V \cup \text{End} \cup (\text{Ident} \times \text{Com})]$$

Semântica operacional

$$\frac{\mathbf{e},\sigma \Downarrow v \qquad \rho(\mathbf{p}) = (\mathbf{x},\mathbf{c}) \qquad \rho[\mathbf{x} \mapsto v] \vdash \mathbf{c},\sigma \Downarrow \sigma'}{\rho \vdash \mathbf{p(e)},\sigma \Downarrow \sigma'}\ \text{call}$$

$$\frac{\rho[\mathbf{p} \mapsto (\mathbf{x},\mathbf{c})] \vdash \mathbf{c}',\sigma \Downarrow \sigma'}{\rho \vdash \texttt{let proc } \mathbf{p(x:}\tau\mathbf{)=c} \texttt{ in } \mathbf{c}',\sigma \Downarrow \sigma'}\ \text{letproc}$$

- Observações:
  - O argumento é avaliados antes de chamar o procedimento.

Exemplo

Escopos estáticos ou dinâmicos?

```
let var x:=1 in
  let proc p(y:int)=(x:=x+1) in
    let var x:=2 in
      p(2)
```

Escopos dinâmicos tem desvantagem graves

- Variáveis locais são visíveis em qualquer escopo dinâmico.
- O entendimento do programa fica mais complicado.

> Dnymic binding [...] is widely and unquivocally regarded as a design error.

Um erro grave

- Também, temos um erro grave nossa solução:

```
let var x:=3 in
  let proc p(y:int)=(x:=x+1) in
    let var x:=true in
      p(2)
```

- O programa acima é bem tipado?
- O que acontece no tempo da execução? Porque?

Escopos estáticos

- Para escopos estáticos, precisamos acesso para o ambiente de execução *no tempo da declaração*.
- Portante, no caso do procedimentos temos que usar *closures*

$$\text{Closure} = \text{Ident} \times \text{AmbExec} \times \text{Com}$$

que quarda esse ambiente.

- O novo ambiente de execução é

$$\rho \in \text{AmbExec} = [\text{Ident} \rightharpoonup V \cup \text{End} \cup \text{Closure}]$$

Semântica operacional

$$\frac{\mathbf{e}, \sigma \Downarrow v \qquad \rho(\mathbf{p}) = (\mathbf{x}, \rho', \mathbf{c}) \qquad \rho'[\mathbf{x} \mapsto v] \vdash \mathbf{c}, \sigma \Downarrow \sigma'}{\rho \vdash \mathbf{p(e)}, \sigma \Downarrow \sigma'}\ \text{call}$$

$$\frac{\rho[\mathbf{p} \mapsto (\mathbf{x}, \rho, \mathbf{c})] \vdash \mathbf{c}, \sigma \Downarrow \sigma'}{\rho \vdash \texttt{let proc } \mathbf{p}(\mathbf{x}{:}\tau)\texttt{=}\mathbf{c} \texttt{ in } \mathbf{c}', \sigma \Downarrow \sigma'}\ \text{letproc}$$