

INF05516 - Semântica Formal N
Ciência da Computação
Universidade Federal do Rio Grande do Sul

Álvaro Moreira, Marcus Ritt
{afmoreira,mrpritt}@inf.ufrgs.br

21 de Junho de 2009

Conteúdo

1	Semântica Operacional e Sistemas de Tipos	5
1.1	A Linguagem L1	5
1.1.1	Semântica Operacional de L1	5
1.1.2	Sistema de Tipos para L1	9
1.1.3	Propriedades de L1	11
1.2	A Linguagem L2	13
1.2.1	Funções	14
1.2.2	Declarações	16
1.2.3	Propriedades de L2	17
1.3	A Linguagem L3	18
1.3.1	Sintaxe de L3	18
1.3.2	Semântica Operacional de L3	19
1.3.3	Sistema de Tipos para L3	22
1.4	Exceções	23
1.4.1	Ativando exceções	24
1.4.2	Tratamento de exceções	26
1.4.3	Tratamento de exceções com passagem de valores	27
1.5	Subtipos	28
1.6	Orientação a Objetos	32
		38

1 Semântica Operacional e Sistemas de Tipos

Vamos definir a semântica operacional de uma série de linguagens no estilo conhecido por *semântica operacional estrutural* chamado também de *semântica operacional small-step*.

O material dessas notas de aulas foi elaborado com base nas notas de aula de Peter Sewell, Universidade de Cambridge (parte sobre as linguagens L1, L2 e L3) e no livro *Types and Programming Languages* de Benjamin Pierce (parte sobre exceções, subtipos e orientação a objetos).

1.1 A Linguagem L1

Programas em L1 pertencem ao conjunto definido pela gramática abstrata abaixo:

Sintaxe de L1 (1)	
$e ::=$	$n \mid b \mid e_1 \text{ op } e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$ $\mid l := e \mid ! l$ $\mid \text{skip} \mid e_1; e_2$ $\mid \text{while } e_1 \text{ do } e_2$
onde	$b \in \{\text{true}, \text{false}\}$ $n \in \text{conjunto de numerais inteiros}$ $l \in \text{conjunto de endereços}$ $\text{op} \in \{+, \geq\}$

Observações:

- em L1 não há distinção entre comandos e expressões.
- note que, de acordo com a gramática abstrata acima, fazem parte do conjunto de árvores de sintaxe abstrata de L1 expressões sem sentido tais como $10 + \text{false}$ e também $\text{while } 2 + 1 \text{ do } 20$
- em L1 o programador tem acesso direto a endereços (mas não há aritmética com endereços).

1.1.1 Semântica Operacional de L1

Vamos definir a semântica operacional de L1 no estilo conhecido por *semântica operacional estrutural* chamado também de *semântica operacional small-step*.

Relação de transição entre configurações (2)

- Uma semântica operacional *small-step* é um *sistema de transição de estados*
- A relação de transição entre estados é chamada de \longrightarrow .
- Escrevemos

$$c \longrightarrow c'$$

para dizer que há uma transição do estado c para o estado c'

- A relação \longrightarrow^* é o fecho reflexivo e transitivo de \longrightarrow
- Escrevemos $c \not\rightarrow$ quando não existe c' tal que $c \longrightarrow c'$

Semântica Operacional de L1 - Configurações (3)

- Um estado, para L1, é um par $\langle e, \sigma \rangle$ onde e é uma expressão e σ é uma *memória*.
- Uma memória é um mapeamento **finito** de endereços para inteiros. Exemplo:

$$\{l_1 \mapsto 0, l_2 \mapsto 10\}$$

- Para memória σ acima, temos que $Dom(\sigma) = \{l_1, l_2\}$

Valores e Erros de Execução (4)

- Valores são expressões já completamente avaliadas
- Os valores v da linguagem L1 são os seguintes:

$$v ::= n \mid b \mid \text{skip}$$
- Se $\langle e, \sigma \rangle \not\rightarrow$ e a expressão e não é valor temos um *erro de execução*
- Após vermos as regras da semântica operacional de L1 veremos que, por exemplo
 - $\langle 2+\text{true}, \sigma \rangle \not\rightarrow$ e
 - $\langle l:=2, \sigma \rangle \not\rightarrow$, caso $l \notin Dom(\sigma)$

A relação de transição \longrightarrow é definida através de um conjunto de regras de inferência da forma

$$\frac{\text{premissa} \dots \text{premissa}}{\text{conclusao}}$$

Ajuda na compreensão das regras da semântica operacional *small step* lembrar que, tipicamente, para cada construção da gramática abstrata que não é valor, temos:

- uma ou mais regras de *reescrita* (ou redução) e
- uma ou mais regras de *computação*

As regras de reescrita especificam a ordem na qual uma expressão é avaliada e as regras de computação dizem, de fato, como uma determinada expressão efetua uma computação interessante.

Semântica Operacional de Operações Básicas (5)	
$\frac{\llbracket n \rrbracket = \llbracket n_1 + n_2 \rrbracket}{\langle n_1 + n_2, \sigma \rangle \longrightarrow \langle n, \sigma \rangle}$	(OP+)
$\frac{\llbracket b \rrbracket = \llbracket n_1 \geq n_2 \rrbracket}{\langle n_1 \geq n_2, \sigma \rangle \longrightarrow \langle b, \sigma \rangle}$	(OP≥)
$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle e_1 \text{ op } e_2, \sigma \rangle \longrightarrow \langle e'_1 \text{ op } e_2, \sigma' \rangle}$	(OP1)
$\frac{\langle e_2, \sigma \rangle \longrightarrow \langle e'_2, \sigma' \rangle}{\langle v \text{ op } e_2, \sigma \rangle \longrightarrow \langle v \text{ op } e'_2, \sigma' \rangle}$	(OP2)

As regras OP1 e OP2 são regras de reescrita de expressões, e as regras OP+ e OP≥ são regras de computação. Observe que as regras OP1 e OP2 especificam que a avaliação dos operandos é feita da esquerda para direita. Observe também o uso das meta-variáveis n_1 e n_2 nas regras OP+ e OP≥: as regras se aplicam caso os operandos de + e ≥ sejam ambos números inteiros, caso contrário temos um erro de execução.

Condicional (6)	
$\langle \text{if true then } e_2 \text{ else } e_3, \sigma \rangle \longrightarrow \langle e_2, \sigma \rangle$	(IF1)
$\langle \text{if false then } e_2 \text{ else } e_3, \sigma \rangle \longrightarrow \langle e_3, \sigma \rangle$	(IF2)

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, \sigma \rangle \longrightarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, \sigma' \rangle} \quad (\text{IF3})$$

Seqüência (7)

$$\langle \text{skip}; e_2, \sigma \rangle \longrightarrow \langle e_2, \sigma \rangle \quad (\text{SEQ1})$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle e_1; e_2, \sigma \rangle \longrightarrow \langle e'_1; e_2, \sigma' \rangle} \quad (\text{SEQ2})$$

Pela regra de reescrita SEQ2 acima, a avaliação seqüencial de duas expressões é feita da esquerda para direita. Pela regra SEQ1, quando o lado esquerdo estiver completamente reduzido para **skip**, a avaliação deve continuar com a expressão no lado direito da **;**. Note que aqui foi feita uma escolha arbitrária: a avaliação continua com a expressão no lado direito somente se a expressão do lado esquerdo do ponto e vírgula avalia para **skip**. Essa opção faz com que expressões sintaticamente bem formadas, como $5 + 4; l := 4$ por exemplo, levem a erro de execução, caso sejam avaliadas.

Operações com a Memória (8)

$$\frac{l \in \text{Dom}(\sigma)}{\langle l := n, \sigma \rangle \longrightarrow \langle \text{skip}, \sigma[l \mapsto n] \rangle} \quad (\text{ATR1})$$

$$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle}{\langle l := e, \sigma \rangle \longrightarrow \langle l := e', \sigma' \rangle} \quad (\text{ATR2})$$

$$\frac{l \in \text{Dom}(\sigma) \quad \sigma(l) = n}{\langle ! l, \sigma \rangle \longrightarrow \langle n, \sigma \rangle} \quad (\text{DEREF})$$

Observe que as operações de atribuição e derreferência (regras ATR1 e DEREF) somente são executadas se o endereço l estiver mapeado na memória (formalizado pela premissa $l \in \text{Dom}(\sigma)$). Na regra ATR1 podemos observar outra escolha feita em relação a semântica da linguagem: uma atribuição é reduzida para o valor **skip**. Em algumas linguagens de programação o valor final de uma expressão de atribuição é mesmo do valor atribuído.

While (9)

$$\langle \text{while } e_1 \text{ do } e_2, \sigma \rangle \longrightarrow \langle \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else skip}, \sigma \rangle \text{ (WHILE)}$$

Note que a regra para o comando while não se encaixa na classificação das regras dada anteriormente (regras de reescrita e computação).

1.1.2 Sistema de Tipos para L1

Observe que a gramática para L1 admite expressões cuja execução leva erro (um erro aqui é representado pela impossibilidade de aplicar uma regra da semântica operacional para uma expressão que não é valor). Vamos ver um sistema de tipos que especifica uma análise a ser feita sobre expressões. Somente expressões *bem tipadas* serão avaliadas.

Um sistema de tipos deve ser definido em acordo com a semântica operacional, em outras palavras uma expressão só deve ser considerada bem tipada pelas regras de um sistema de tipos se a sua avaliação, pelas regras da semântica operacional, não levar a erro de execução. Essa propriedade fundamental é conhecida como *segurança* do sistema de tipos em relação a semântica operacional.

Tipos para L1 (10)

- O sistema de tipos para foi projetado considerando que a linguagem é estaticamente tipada, ou seja, que a verificação de tipos será feita em tempo de compilação percorrendo a árvore de sintaxe abstrata
- A linguagem de tipos é dada pela gramática abaixo onde `int ref` é o tipo associado a endereços

$$T ::= \text{int} \mid \text{bool} \mid \text{unit}$$

$$T_{loc} ::= \text{int ref}$$

- Γ é um mapeamento de endereços para `int ref`

Note que em L1 a memória só pode receber valores inteiros, logo todos os endereços em L1 são do tipo `int ref`.

Valores e Operações Básicas (11)

$$\Gamma \vdash n : \text{int} \quad (\text{TINT})$$

$$\Gamma \vdash b : \text{bool} \quad (\text{TBOOL})$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \quad (\text{T}+) \quad (12)$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \geq e_2 : \text{bool}} \quad (\text{T}\geq)$$

Observe pelas regras OP+ e OP \geq que os operandos de + e de \geq devem ser do tipo inteiro. Estas regras excluem expressões tais como $4 + \text{true}$ e $\text{true} \geq \text{skip}$.

Condicional (12)

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \quad (\text{TIF})$$

- Note que para o condicional ser bem tipado a expressão da parte **then** e a expressão da parte **else** devem ser do mesmo tipo
- Com essa regra expressões tais como **if** $5 + 3 \geq 2$ **then true** **else** 5 não são consideradas bem tipadas mesmo que não levem a erro de execução, de acordo com a semântica operacional

Operações com a memória (13)

$$\frac{\Gamma \vdash e : \text{int} \quad \Gamma(l) = \text{int ref}}{\Gamma \vdash l := e : \text{unit}} \quad (\text{TATR})$$

$$\frac{\Gamma(l) = \text{int ref}}{\Gamma \vdash !l : \text{int}} \quad (\text{TDEREF})$$

Seqüência e While (14)

$$\Gamma \vdash \text{skip} : \text{unit} \quad (\text{TSKIP})$$

$$\frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T} \quad (\text{TSEQ})$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}} \quad (\text{TWHILE})$$

Observe que o tipo `unit` é reservado para expressões que são avaliadas mais pelo seu efeito. O tipo `unit` possui somente um valor que é o **skip**.

1.1.3 Propriedades de L1

Propriedades (15)

O teorema abaixo expressa que a avaliação, **em um passo**, é determinística

Teorema 1 (Determinismo) Se $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ e se $\langle e, \sigma \rangle \longrightarrow \langle e'', \sigma'' \rangle$ então $\langle e', \sigma' \rangle = \langle e'', \sigma'' \rangle$.

Prova. Por indução na estrutura de e . ■

A partir do teorema acima concluímos que a avaliação de programas L1 é determinística.

Na seção anterior vimos que é fundamental que um sistema de tipos seja seguro em relação a semântica operacional da linguagem. A noção de segurança foi então explicada de maneira informal:

Segurança (16)

- Um sistema de tipos é seguro se a avaliação de expressões consideradas bem tipadas pelas suas regras não leva a **erro de execução** quando avaliadas de acordo com a semântica operacional
- Erro de execução ocorre quando temos uma configuração não final a qual nenhuma regra da semântica operacional se aplica
- de maneira **informal** podemos resumir essa noção de segurança através do seguinte *slogan*:

$$\text{Se } \Gamma \vdash e : T \text{ então } e \not\rightarrow^* \text{ erro}$$

A técnica de prova mais utilizada para provar que um sistema de tipos é seguro é conhecida como *segurança sintática*. Ela consiste em realizar basicamente duas provas:

- prova do *progresso de expressões bem tipadas*, ou seja, provar que, se uma expressão for bem tipada, e se ela não for um valor, sua avaliação pode progredir um passo pelas regras da semântica operacional. *Slogan*:

Se $\Gamma \vdash e : T$ então e é valor, ou existe e' tal que $e \longrightarrow e'$

- prova da *preservação, após um passo da avaliação, do tipo de uma expressão*, ou seja, se uma expressão bem tipada progride em um passo, a expressão resultante possui o mesmo tipo. *Slogan*:

Se $\Gamma \vdash e : T$ e $e \longrightarrow e'$ então $\Gamma \vdash e' : T$.

Note que ambas as provas são necessárias para provar segurança, ou seja:

$$\text{Segurança} = \text{Progresso} + \text{Preservação}$$

Provar somente *progresso* não é suficiente para provar segurança. É preciso provar que a expressão que resulta da progressão em um passo de uma expressão bem tipada também é bem tipada (ou seja, que a propriedade de ser bem tipado é preservada pela avaliação em um passo). Da mesma forma, provar somente *preservação* não é suficiente para provar segurança. É preciso provar que a expressão bem tipada que resulta da progressão em um passo da expressão original pode progredir (ou seja, é preciso provar progresso em um passo de expressões bem tipadas).

Observe que os *slogans* acima capturam a *essência* de progresso e preservação válida para qualquer linguagem de programação. Seguem abaixo as formulações precisas de progresso e preservação *específicas* para a linguagem L1.

Progresso e Preservação para L1 (18)

Teorema 2 (Progresso) Se $\Gamma \vdash e : T$ e $\text{Dom}(\Gamma) \subseteq \text{Dom}(\sigma)$ então (i) e é valor, ou (ii) existe $\langle e', \sigma' \rangle$ tal que $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$

Prova. Por indução na estrutura de e . ■

Teorema 3 (Preservação) Se $\Gamma \vdash e : T$ e $\text{Dom}(\Gamma) \subseteq \text{Dom}(\sigma)$ e $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ então $\Gamma \vdash e' : T$ e $\text{Dom}(\Gamma) \subseteq \text{Dom}(\sigma')$

Prova. Por indução na estrutura de e . ■

Estamos interessados em saber se os dois problemas abaixo são decidíveis, ou seja, se existem algoritmos que os resolvem.

Problemas algorítmicos (19)

- Problema da **Verificação de Tipos**: dados ambiente Γ , expressão e e tipo T , o julgamento de tipo $\Gamma \vdash e : T$ é derivável usando as regras do sistema de tipos?
- Problema da **Tipabilidade**: dados ambiente Γ e expressão e , encontrar tipo T tal que $\Gamma \vdash e : T$ é derivável de acordo com as regras do sistema de tipos

O problema da tipabilidade é mais difícil do que o problema da verificação de tipos para sistemas de tipos de linguagens de programação. Dependendo do sistema de tipos, resolver o problema da tipabilidade requer algoritmos de *inferência de tipos* muitas vezes complicados. No caso da linguagem L1 ambos os problemas são decidíveis e os algoritmos que os resolvem são muito simples.

Observe que, do ponto de vista prático, o problema da *verificação de tipos* para L1 não é interessante¹. Já o problema da *tipabilidade* é relevante na prática para L1: dado um programa L1 queremos saber se ele é ou não bem tipado e, ser for, queremos saber qual é o seu tipo .

Decidibilidade (20)

Teorema 4 (Decidibilidade da Tipabilidade) Dados ambiente Γ e expressão e , existe algoritmo que decide se existe tipo T tal que $\Gamma \vdash e : T$ é verdadeiro ou não.

Prova. A prova consiste em exibir um algoritmo (ver exercício abaixo) que tem como entrada um ambiente de tipo Γ e uma expressão e e, como saída, retorna ou um tipo T tal que $\Gamma \vdash e : T$ é derivável com o sistema de tipos para L1, ou um aviso de que a expressão e não é tipável em Γ quando, de fato, não for o caso. ■

Observação: Embora a diferença entre os dois problemas acima (da *verificação* de tipos e *tipabilidade*) seja clara, é bem comum se referir ao programa que os resolve como sendo o *verificador* de tipos para a linguagem.

1.2 A Linguagem L2

A linguagem L2 é uma extensão de L1 com funções, funções recursivas e declarações. Consideramos primeiro a inclusão de funções.

Sintaxe de L2 (21)

Primeiro, devemos estender a sintaxe de L1:

$$\begin{aligned} e &::= \dots \\ &| \text{fn } x:T \Rightarrow e \mid e_1 \ e_2 \mid x \\ \\ v &::= \dots \\ &| \text{fn } x:T \Rightarrow e \end{aligned}$$

¹fora alguns exercícios de verificação de tipos a serem feitos

Na gramática acima:

- x representa um elemento pertencente ao conjunto $Ident$ de identificadores
- $fn\ x:T \Rightarrow e$ é uma função (sem nome)
- $e_1\ e_2$ é a aplicação da expressão e_1 a expressão e_2
- os tipos da linguagem L2 são dados pela seguinte gramática:

$$\begin{aligned} T &::= \text{int} \mid \text{bool} \mid \text{unit} \mid T_1 \rightarrow T_2 \\ T_{loc} &::= \text{int ref} \end{aligned}$$

Sintaxe de L2 (22)

- Aplicação é associativa a esquerda, logo $e_1\ e_2\ e_3$ é o mesmo que $(e_1\ e_2)\ e_3$
- As setas em tipos função são associativas a direita, logo o tipo $T_1 \rightarrow T_2 \rightarrow T_3$ é o mesmo que $T_1 \rightarrow (T_2 \rightarrow T_3)$
- fn se estende o mais a direita possível, logo $fn\ x:\text{unit} \Rightarrow x;x$ é o mesmo que $fn\ x:\text{unit} \Rightarrow (x; x)$

1.2.1 Funções

Informalmente a semântica *call by value* de L2 pode se expressa da seguinte maneira: reduzimos o lado esquerdo da aplicação para uma função; reduzimos o lado direito para um valor; computamos a aplicação da função ao seu argumento.

Semântica Formal (23)

$$\langle (fn\ x:T \Rightarrow e)\ v, \sigma \rangle \longrightarrow \langle \{v/x\}e, \sigma \rangle \quad (\beta)$$

$$\frac{\langle e_2, \sigma \rangle \longrightarrow \langle e'_2, \sigma' \rangle}{\langle v\ e_2, \sigma \rangle \longrightarrow \langle v\ e'_2, \sigma' \rangle} \quad (\text{APP1})$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle e_1\ e_2, \sigma \rangle \longrightarrow \langle e'_1\ e_2, \sigma' \rangle} \quad (\text{APP2})$$

Substituição - Exemplos (24)

- A semântica da aplicação de função a argumento envolve substituir variável por valor no corpo da função
- a notação $\{v/x\}e$ representa a expressão que resulta da substituição de todas as **ocorrências livres** de x em e por v .
- Exemplos:

$\{3/x\}(x = x) \equiv (3 = 3)$
 $\{3/x\}((fn\ x : int \Rightarrow x + y)\ x) \equiv (fn\ x : int \Rightarrow x + y)3$
 $\{2/x\}(fn\ y : int \Rightarrow x + y) \equiv fn\ y : int \Rightarrow 2 + y$

Segue abaixo a definição da operação de substituição. Note que a condição associada a aplicação da substituição a funções garante que somente variáveis livres vão ser substituídas e que nenhuma variável livre ficará indevidamente ligada após a substituição.

$\{e/x\} x$	$= v$
$\{e/x\} y$	$= y \text{ (se } x \neq y)$
$\{e/x\} fn\ y : T \Rightarrow e'$	$= fn\ y : T \Rightarrow (\{e/x\}e') \text{ se } x \neq y \text{ e } y \notin fv(e)$
$\{e/x\} (e_1 e_2)$	$= (\{e/x\}e_1)(\{e/x\}e_2)$
$\{e/x\} n$	$= n$
$\{e/x\} (e_1 \text{ op } e_2)$	$= \{e/x\}e_1 \text{ op } \{e/x\}e_2$
$\{e/x\} (\text{if } e_1 \text{ then } e_2 \text{ else } e_3)$	$= \text{if } \{e/x\}e_1 \text{ then } \{e/x\}e_2 \text{ else } \{e/x\}e_3$
$\{e/x\} b$	$= b$
$\{e/x\} \text{skip}$	$= \text{skip}$
$\{e/x\} l := e'$	$= l := \{e/x\}e'$
$\{e/x\} !!$	$= !!$
$\{e/x\} (e_1; e_2)$	$= \{e/x\}e_1; \{e/x\}e_2$
$\{e/x\}(\text{while } e_1 \text{ do } e_2)$	$= \text{while } \{e/x\}e_1 \text{ do } \{e/x\}e_2$

Tipando Funções (25)

- Antes, em L1, Γ continha somente os tipos de endereços; em L2, Γ contém também os tipos de variáveis:
- Notação: se $x \notin dom(\Gamma)$, escrevemos $\Gamma, x : T$ para a função parcial que mapeia x para T , mas para outros casos é como Γ .

Regras de tipo para funções, aplicações e variáveis (26)

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad (\text{T}_{\text{VAR}})$$

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash fn\ x : T \Rightarrow e : T \rightarrow T'} \quad (\text{T}_{\text{FN}})$$

$$\frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'} \quad (\text{TAPP})$$

1.2.2 Declarações

Definições Locais (27)

Para facilitar a leitura, nomeando definições e restringindo o escopo, é adicionada a seguinte construção:

$$e ::= \dots \mid \text{let } x:T = e_1 \text{ in } e_2 \text{ end}$$

Pode ser considerada como simples açúcar sintático:

$$\text{let } x:T = e_1 \text{ in } e_2 \text{ end} \equiv (fn\ x:T \Rightarrow e_2)\ e_1$$

A expressão abaixo declara o identificador de nome f associado a uma função que soma um ao seu argumento. Esta função é aplicada a 10 na parte **in** da expressão:

$$\text{let } f:\text{int} \rightarrow \text{int} = fn\ x:\text{int} \Rightarrow x + 1 \text{ in } f\ 10 \text{ end}$$

Regras de tipagem e regras de redução (28)

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma, x : T \vdash e_2 : T'}{\Gamma \vdash \text{let } x:T = e_1 \text{ in } e_2 \text{ end} : T'} \quad (\text{TLET})$$

$$\langle \text{let } x:T = v \text{ in } e_2 \text{ end}, \sigma \rangle \longrightarrow \langle \{v/x\}e_2, \sigma \rangle \quad (\text{LET1})$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle \text{let } x:T = e_1 \text{ in } e_2 \text{ end}, \sigma \rangle \longrightarrow \langle \text{let } x:T = e'_1 \text{ in } e_2 \text{ end}, \sigma' \rangle} \quad (\text{LET1})$$

Funções recursivas (29)

$$e ::= \dots \mid \text{let rec } f:T_1 \rightarrow T_2 = (fn\ y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}$$

$$\frac{\Gamma, f : T_1 \rightarrow T_2, y : T_1 \vdash e_1 : T_2 \quad \Gamma, f : T_1 \rightarrow T_2 \vdash e_2 : T}{\Gamma \vdash \text{let rec } f:T_1 \rightarrow T_2 = (fn\ y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end} : T} \quad (\text{TLETREC})$$

Segue abaixo a definição da função fatorial e a sua chamada para calcular o fatorial de 5 (neste exemplo supomos que operadores para igualdade, multiplicação e para subtração foram adicionados a linguagem):

```
let rec fat : int -> int =
  (fn y:int => if y = 0 then 1 else y * fat (y-1))
in fat 5
end
```

Semântica de funções recursivas (30)

$$\begin{aligned} & \langle \text{let rec } f:T_1 \rightarrow T_2 = (fn\ y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}, \sigma \rangle \\ & \quad \longrightarrow \quad (LETREC) \\ & \langle \{(fn\ y:T_1 \Rightarrow \text{let rec } f:T_1 \rightarrow T_2 = (fn\ y:T_1 \Rightarrow e_1) \text{ in } e_1 \text{ end})/f\}e_2, \sigma \rangle \end{aligned}$$

Mais açúcar sintático (31)

- $e_1; e_2$ poderia ser codificar como $(fn\ y:unit \Rightarrow e_2) e_1$ onde $y \notin fv(e_2)$
- **while** e_1 **do** e_2 poderia ser codificado como:


```
let rec w:unit -> unit =
  fn y:unit => if e1 then (e2; (w skip)) else skip
in
  w skip
end
```

 para um novo w e $y \notin fv(e_1) \cup fv(e_2)$.

1.2.3 Propriedades de L2

Assim como L1, a linguagem L2 é determinística e o enunciado do teorema que afirma essa propriedade é o mesmo. O enunciado das propriedades de preservação e progresso na essência são os mesmos mas neles assumimos que as expressões em consideração são expressões fechadas, ou seja, expressões onde não há variáveis não declaradas.

Progresso e Preservação (32)

Teorema 5 (Progresso) Se e é fechado e $\Gamma \vdash e : T$ e $Dom(\Gamma) \subseteq Dom(\sigma)$ então e é um valor ou existe $e'; \sigma'$ tal que $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ e e' é fechado.

Teorema 6 (Preservação) Se e é fechado e $\Gamma \vdash e : T$, $Dom(\Gamma) \subseteq Dom(\sigma)$ e $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ então $\Gamma \vdash e' : T$ e $Dom(\Gamma) \subseteq Dom(\sigma')$.

Todos os resultados sobre os problemas da tipabilidade e da verificação de tipos de L1 permanecem os mesmos para L2.

1.3 A Linguagem L3

A linguagem L3 é uma extensão da linguagem L2 com pares ordenados e registros. São também definidas construções para criar referências e as operações de atribuição e derreferência são generalizadas para operarem com expressões.

1.3.1 Sintaxe de L3

Programas em L3 pertencem ao conjunto definido pela gramática abstrata abaixo (as linhas marcadas com (*) indicam o que mudou em relação a L2):

Sintaxe de L3 (33)	
e	$::= n \mid b \mid e_1 \text{ op } e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$
(*)	$\mid e_1 := e_2 \mid ! e \mid \text{ref } e \mid l$
	$\mid \text{skip} \mid e_1; e_2$
	$\mid \text{while } e_1 \text{ do } e_2$
	$\mid \text{fn } x:T \Rightarrow e \mid e_1 \ e_2 \mid x$
	$\mid \text{let } x:T = e_1 \text{ in } e_2 \text{ end}$
	$\mid \text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}$
(*)	$\mid (e_1, e_2) \mid \#1 \ e \mid \#2 \ e$
(*)	$\mid \{lab_1 = e_1, \dots, lab_k = e_k\} \mid \#lab \ e$
onde	
	$b \in \{\text{true}, \text{false}\}$
	$n \in \text{conjunto de numerais inteiros}$
	$l \in \text{conjunto de endereços}$
	$op \in \{+, \geq\}$
	$lab \in \text{conjunto de rótulos}$

Novas construções de L3 (34)	
A linguagem L3 difere de L2 nas seguintes construções:	
•	$e_1 := e_2$ ao invés de $l := e_2$ - a atribuição adquire o mesmo <i>status</i> das operações $+$ e \geq , ou seja opera sobre duas expressões
•	$! e$ ao invés de $! l$ - a operação de acesso a memória opera com uma expressão
•	$\text{ref } e$ - esta construção é utilizada para alocar um endereço de memória contendo o valor da expressão e
•	a memória pode conter valores de qualquer tipo

Novas construções (35)

- L3 possui também pares ordenados e registros com quantidade variável de campos identificados por rótulos
- operações $\#1\ e$, $\#2\ e$ para projeção do primeiro e do segundo componente de um par ordenado e
- $\#lab\ e$ para projeção do componente de rótulo lab do registro e
- Note que, de acordo com a gramática acima, endereços l são expressões
- Endereços contudo **não** são acessados diretamente pelo programador, ou seja não podem aparecer em programas!!
- Todo endereço deve ser criado através da construção $\text{ref } e$
- Como em $L3$ a memória pode conter valores de qualquer tipo, temos o tipo T ref ao invés do tipo int ref para endereços de memória

Tipos para L3 (36)

- Note também **três novos tipos**: para pares ordenados, registros e endereços

$$\begin{aligned}
 T &::= \text{int} \mid \text{bool} \mid \text{unit} \mid T_1 \rightarrow T_2 \\
 &\quad \mid T_1 * T_2 \\
 &\quad \mid \{lab_1 : T_1, \dots, lab_n : T_n\} \\
 &\quad \mid T \text{ ref}
 \end{aligned}$$

1.3.2 Semântica Operacional de L3

Valores de L3 (37)

- lembrando que valores são as expressões já completamente avaliadas
- pares e registros cujos componentes estão completamente avaliados são também valores

$$\begin{aligned}
 v &::= n \mid b \mid \text{skip} \mid fn\ x:T \Rightarrow e \\
 &\quad \mid (v_1, v_2) \\
 &\quad \mid \{lab_1 = v_1, \dots, lab_n = v_n\} \\
 &\quad \mid l
 \end{aligned}$$

Pares e Projeção (38)

$$\frac{\langle e_2, \sigma \rangle \longrightarrow \langle e'_2, \sigma' \rangle}{\langle (v, e_2), \sigma \rangle \longrightarrow \langle (v, e'_2), \sigma' \rangle} \quad (\text{PAR1})$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle (e_1, e_2), \sigma \rangle \longrightarrow \langle (e'_1, e_2), \sigma' \rangle} \quad (\text{PAR2})$$

$$\langle \#1 (v_1, v_2), \sigma \rangle \longrightarrow \langle v_1, \sigma \rangle \quad (\text{PRJ1})$$

$$\langle \#2 (v_1, v_2), \sigma \rangle \longrightarrow \langle v_2, \sigma \rangle \quad (\text{PRJ2})$$

$$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle}{\langle \#1 e, \sigma \rangle \longrightarrow \langle \#1 e', \sigma' \rangle} \quad (\text{PRJ3})$$

$$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle}{\langle \#2 e, \sigma \rangle \longrightarrow \langle \#2 e', \sigma' \rangle} \quad (\text{PRJ4})$$

Registros e Projeção (39)

$$\frac{\langle e_j, \sigma \rangle \longrightarrow \langle e'_j, \sigma' \rangle}{\langle \{lab_i = v_i^{i \in 1 \dots j-1}, lab_j = e_j, lab_k = e_k^{k \in j+1 \dots n}\}, \sigma \rangle \longrightarrow \langle \{lab_i = v_i^{i \in 1 \dots j-1}, lab_j = e'_j, lab_k = e_k^{k \in j+1 \dots n}\}, \sigma' \rangle} \quad (\text{RCD1})$$

$$\langle \#lab_i \{lab_1 = v_1, \dots lab_n = v_n\}, \sigma \rangle \longrightarrow \langle v_i, \sigma \rangle \quad (\text{RCD2})$$

$$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle}{\langle \#lab_i e, \sigma \rangle \longrightarrow \langle \#lab_i e', \sigma' \rangle} \quad (\text{RCD3})$$

Um par (e_1, e_2) é na verdade açúcar sintático para o registro $\{\#1 = e_1, \#2 = e_2\}$ com dois componentes e_1, e_2 identificados pelos rótulos de nome 1 e 2 respectivamente.

Memória (40)	
$\frac{l \notin Dom(\sigma)}{\langle \text{ref } v, \sigma \rangle \longrightarrow \langle l, \sigma[l \mapsto v] \rangle}$	(REF1)
$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle}{\langle \text{ref } e, \sigma \rangle \longrightarrow \langle \text{ref } e', \sigma' \rangle}$	(REF2)
$\frac{l \in Dom(\sigma) \quad \sigma(l) = v}{\langle ! l, \sigma \rangle \longrightarrow \langle v, \sigma \rangle}$	(DEREF1)
$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle}{\langle ! e, \sigma \rangle \longrightarrow \langle ! e', \sigma' \rangle}$	(DEREF2)

Memória - cont. (41)	
$\frac{l \in Dom(\sigma)}{\langle l := v, \sigma \rangle \longrightarrow \langle \text{skip}, \sigma[l \mapsto v] \rangle}$	(ATR1)
$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle}{\langle l := e, \sigma \rangle \longrightarrow \langle l := e', \sigma' \rangle}$	(ATR2)
$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle e_1 := e_2, \sigma \rangle \longrightarrow \langle e'_1 := e_2, \sigma' \rangle}$	(ATR3)

Conforme já foi dito anteriormente a memória agora pode conter qualquer valor (e não somente inteiros como em L1 e em L2). Observe também que o endereço l criado por $\text{ref } e$ deve ser novo (na regra REF1 acima isso é especificado pela premissa $l \notin Dom(\sigma)$).

1.3.3 Sistema de Tipos para L3

Regras de Tipo para Pares em L3 (42)

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 * T_2} \quad (\text{TPAR})$$

$$\frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#1 e : T_1} \quad (\text{TPRJ1})$$

$$\frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#2 e : T_2} \quad (\text{PRJ2})$$

- note que os pares $(e_1, (e_2, e_3))$ e $((e_1, e_2), e_3)$ são de tipos diferentes

Regras de Tipos para Registros (43)

$$\frac{\Gamma \vdash e_1 : T_1 \quad \dots \quad \Gamma \vdash e_n : T_n}{\Gamma \vdash \{lab_1 = e_1, \dots lab_k = e_k\} : \{lab_1 : T_1, \dots lab_n : T_n\}} \quad (\text{TRCD})$$

$$\frac{\Gamma \vdash e : \{lab_1 : T_1, \dots lab_n : T_n\}}{\Gamma \vdash \#lab_i e : T_i} \quad (\text{TPRJ})$$

Sistema de tipos conservador (44)

- Note que $\{A : \text{bool}, B : \text{int}\}$ e $\{B : \text{int}, A : \text{bool}\}$ são tipos diferentes. Logo o programa abaixo, embora não leve a erro de execução, não é considerado bem tipado

$(fn x : \{B : \text{int}, A : \text{bool}\} \Rightarrow \text{if } \#A \text{ then } \#B \text{ else } 3) \{A = \text{true}, B = 10\}$

- O programa abaixo não leva a erro de execução, mas também não é considerado bem tipado:

$(fn x : \{A : \text{bool}\} \Rightarrow \text{if } \#A \text{ then } 2 \text{ else } 3) \{A = \text{true}, B = 10\}$

- *subtipos* flexibilizam o sistema de tipos e permitem a tipagem de casos como os acima

Regras de Tipos para Operações com Memória (45)

$$\frac{\Gamma \vdash e_1 : T \text{ ref} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \text{unit}} \quad (\text{TATR})$$

$$\frac{\Gamma \vdash e : T \text{ ref}}{\Gamma \vdash ! e : T} \quad (\text{TDEREF})$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref } e : T \text{ ref}} \quad (\text{TREF})$$

$$\frac{\Gamma(l) = T \text{ ref}}{\Gamma \vdash l : T \text{ ref}} \quad (\text{TL})$$

1.4 Exceções

Exceções (46)

- várias situações nas quais uma função precisa sinalizar, para quem a chamou, que não poderá realizar a sua tarefa por alguma razão
 - algum cálculo envolve divisão por zero ou overflow
 - chave de busca ausente em um dicionário
 - índice de array está fora dos limites
 - arquivo não foi encontrado ou não pode ser aberto
 - falta de memória suficiente
 - usuário "matou" um processo

Exceções (47)

- algumas dessas condições excepcionais podem ser sinalizadas fazendo com que a função retorne um registro variante.
- o chamador da função então trata da exceção sinalizada

- isso mantém o fluxo de controle normal de execução mas força o programador a colocar, em cada trecho de código que chama a função, código para tratar o evento excepcional
- melhor seria centralizar o tratamento de eventos excepcionais!

Exceções (48)

- vamos considerar três casos
 1. um evento excepcional simplesmente aborta a execução do program
 2. um evento excepcional transfere controle para um tratador de exceção
 3. idem ao anterior mas passando informações adicionais para o tratador

No que segue vamos considerar uma extensão da linguagem L3 com extensões.

1.4.1 Ativando exceções

Ativando exceções (49)

- Vamos considerar uma extensão de L3 com a forma mais simples possível para sinalizar exceções: expressão **raise**
- programa deve ser abortado produzindo **raise**
- não há tratamento

Sintaxe (50)

Simplesmente adicionamos a expressão **raise** a gramática abstrata de L3

$$e ::= \dots \mid \text{raise}$$

Semântica Operacional (51)

- Temos que adicionar uma série de regras para propagar **raise**
- Como exemplo, segue abaixo o conjunto de regras para avaliação do condicional:

$$\langle \text{if true then } e_2 \text{ else } e_3, \sigma \rangle \longrightarrow \langle e_2, \sigma \rangle \quad (\text{IF1})$$

$$\langle \text{if false then } e_2 \text{ else } e_3, \sigma \rangle \longrightarrow \langle e_3, \sigma \rangle \quad (\text{IF2})$$

$$\langle \text{if raise then } e_2 \text{ else } e_3, \sigma \rangle \longrightarrow \langle \text{raise}, \sigma \rangle \quad (\text{IFRS})$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, \sigma \rangle \longrightarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, \sigma' \rangle} \quad (\text{IF3})$$

Seqüência (52)

- Outro exemplo: a adição, ao conjunto de regras para seqüência, de uma regra para propagar **raise**:

$$\langle \text{skip}; e_2, \sigma \rangle \longrightarrow \langle e_2, \sigma \rangle \quad (\text{SEQ1})$$

$$\langle \text{raise}; e_2, \sigma \rangle \longrightarrow \langle \text{raise}, \sigma \rangle \quad (\text{SEQRS})$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle e_1; e_2, \sigma \rangle \longrightarrow \langle e'_1; e_2, \sigma' \rangle} \quad (\text{SEQ2})$$

Semântica Operacional (53)

Adição das regras APPRS e FNRS para propagação de **raise** em aplicações:

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma' \rangle}{\langle e_1 \ e_2, \sigma \rangle \longrightarrow \langle e'_1 \ e_2, \sigma' \rangle} \quad (\text{APP1})$$

$$\frac{\langle e_2, \sigma \rangle \longrightarrow \langle e'_2, \sigma' \rangle}{\langle v \ e_2, \sigma \rangle \longrightarrow \langle v \ e'_2, \sigma' \rangle} \quad (\text{APP2})$$

$$\langle \text{raise } e_2, \sigma \rangle \rightarrow \langle \text{raise}, \sigma \rangle \quad (\text{APPRS})$$

$$\langle (fn \ x:T \Rightarrow e) \ v, \sigma \rangle \longrightarrow \langle \{v/x\}e, \sigma \rangle \quad (\beta)$$

$$\langle v \ \text{raise}, \sigma \rangle \longrightarrow \langle \text{raise}, \sigma \rangle \quad (\text{FNRS})$$

Semântica Operacional (54)

- Note que **raise** é uma expressão já completamente avaliada, mas ela **não** pode ser considerada como sendo um valor da linguagem
- Caso considerássemos **raise** como sendo um valor, a linguagem deixaria de ser determinística
- Por que a regra $\langle v \text{ raise}, \sigma \rangle \rightarrow \langle \text{raise}, \sigma \rangle$ e não $\langle e \text{ raise}, \sigma \rangle \rightarrow \langle \text{raise}, \sigma \rangle$ (pense como ficaria a avaliação de um termo $e \text{ raise}$ onde a avaliação de e entra em loop)

Tipos (55)

- observe que **raise** pode ter qualquer tipo!!
- em $(fn x:bool \Rightarrow x) \text{ raise}$, o termo **raise** deverá ter tipo `bool`
- em $(fn x:bool \Rightarrow x) (\text{raise true})$, o termo **raise** deverá ter tipo `bool \rightarrow bool`
- a regra de tipo para **raise** fica:

$$\Gamma \vdash \text{raise} : T \quad (\text{TRS})$$

- agora deixa de ser verdade que todo termo possui somente um único tipo !

1.4.2 Tratamento de exceções

Tratamento de exceções (56)

- as regras de avaliação para **raise** podem ser vistas como sendo regras que desfazem uma pilha de chamadas, descartando funções pendentes até que **raise** se propague para o nível de cima (primeiro chamador)
- em implementações reais é exatamente isso que acontece
- mas também é possível colocar na pilha de chamadas *tratadores de exceções*
- quando uma exceção é ativada começa o desempilhamento descartando funções. Se um tratador de exceção é encontrado na pilha o controle é transferido para ele.

Tratamento de exceções (57)

- A sintaxe de L3 é estendida com as seguintes expressões:

$$e ::= \dots$$

$$\text{raise}$$

$$\text{try } e_1 \text{ with } e_2$$

- $\text{try } e_1 \text{ with } e_2$, quando avaliado, retorna o valor da avaliação de e_1 . Se essa avaliação ativar uma exceção e_2 é avaliado

Semântica Operacional (58)

- todas as regras anteriores da extensão de L3 com **raise** são mantidas e as seguintes regras são acrescentadas

$$\langle \text{try } v_1 \text{ with } e_2, \sigma \rangle \rightarrow \langle v_1, \sigma \rangle \quad (\text{TRY1})$$

$$\langle \text{try raise with } e_2, \sigma \rangle \rightarrow \langle e_2, \sigma \rangle \quad (\text{TRY2})$$

$$\frac{\langle e_1, \sigma \rangle \rightarrow \langle e'_1, \sigma' \rangle}{\langle \text{try } e_1 \text{ with } e_2, \sigma \rangle \rightarrow \langle \text{try } e'_1 \text{ with } e_2, \sigma' \rangle} \quad (\text{TRY3})$$

Regras de Tipos (59)

- **raise**, como antes, continua tendo seu tipo definido pelo contexto
- Note que a regra de tipo para $\text{try } e_1 \text{ with } e_2$ requer que e_1 e e_2 tenham o mesmo tipo

$$\Gamma \vdash \text{raise} : T \quad (\text{TRS})$$

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{try } e_1 \text{ with } e_2 : T} \quad (\text{TTRY})$$

1.4.3 Tratamento de exceções com passagem de valores

Tratamento de exceções com passagem de valores (60)

- No momento em que uma exceção é ativada, um valor pode ser passado como argumento para o seu tratador

- Esse valor pode ser uma informação sobre a causa exata da exceção, ou pode ser um valor que ajude o tratador
- Dessa forma, a expressão associada ao tratador deve ser de um tipo função

Regras de tipos para exceções com valores (61)

Sintaxe

$e ::= \dots$
 $\text{raise } e$
 $\text{try } e_1 \text{ with } e_2$

Regras de tipo

$$\frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash \text{raise } e_1 : T} \quad (\text{TRS-V})$$

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : \text{int} \rightarrow T}{\Gamma \vdash \text{try } e_1 \text{ with } e_2 : T} \quad (\text{TTRY-V})$$

Observe que as regras de tipo TRS-V e TTRY-V especificam que o valor a ser passado para o tratador de exceções na linguagem L3 estendida com exceções deve ser `int`. Esse valor inteiro pode ser interpretado como um código indicando a causa do erro. O tratador tipicamente é organizado na forma de um *case* que identifica a causa da exceção e dá a ela um tratamento adequado.

1.5 Subtipos

Polimorfismo significa (literalmente) ter múltiplas formas. Uma construção que pode assumir diferentes tipos, de acordo com o contexto, é dita polimórfica.

Tipos de Polimorfismo (62)

- Existem três formas de polimorfismo em linguagens modernas:
 - *polimorfismo paramétrico* - uma função pode ser aplicada a qualquer argumento cujo tipo *casa* com uma expressão de tipos envolvendo variáveis de tipos
 - *polimorfismo ad-hoc* - outro termo para *overloading*, no qual duas ou mais implementações com tipos diferentes são referenciadas pelo mesmo nome
 - *polimorfismo baseado em subtipos* - relações entre tipos permitem uma expressão ter mais do que um tipo

Aqui vamos ver com mais detalhes uma extensão do sistema de tipos da linguagem L3 de tal forma a admitir subtipos. Mais adiante vamos usar essa extensão de L3 com subtipos e programar em L3 utilizando um estilo orientado a objetos.

Subtipos (63)

- encontrados em linguagens orientadas a objetos
- considerados fundamentais nesse paradigma
- aqui veremos só seus aspectos essenciais através de
 - registros e
 - funções

Subtipos - Motivação (64)

- um dos objetivos de um sistema de tipos é não permitir a avaliação/geração de código de programas que levem a *determinados* erros de execução
- obviamente não queremos que um sistema de tipos exclua também termos cuja avaliação nunca levará a erro de execução
- mas como estamos considerando verificação estática de tipos (ou seja *em tempo de compilação*) certos programas são considerados mal tipados mesmo que suas avaliações sejam bem comportadas

Subtipos - Motivação (65)

O termo $(fn\ r:\{x:\text{int}\} \Rightarrow \#x\ r)\ \{x=0, y=1\}$ não é bem tipado devido a seguinte regra para aplicação

$$\frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'} \quad (\text{T-APP})$$

Porém sua avaliação não fica presa em nenhum momento de acordo com as regras da semântica operacional

Subtipos - Motivação (66)

- A função $fn\ r:\{x:\text{int}\} \Rightarrow \#x\ r$ da expressão $(fn\ r:\{x:\text{int}\} \Rightarrow \#x\ r)\ \{x=0, y=1\}$ só tem um exigência quanto ao seu argumento: ele deve ser um registro que possua um campo de nome x e do tipo int

- Sempre será seguro passar um argumento do tipo $\{x : \text{int}, y : \text{int}\}$ para uma função que espera argumento do tipo $\{x : \text{int}\}$
- com subtipos, expressões como a do exemplo acima passam a ser bem tipadas

Subtipos (67)

- dizemos que o tipo S é subtipo do tipo T , escrito $S <: T$, quando expressão do tipo S pode ser utilizada, com segurança, em algum contexto onde expressão do tipo T é esperada
- registro do tipo $\{x : \text{int}, y : \text{int}\}$ pode ser usado em um contexto que espera registro do tipo $\{x : \text{int}\}$, ou seja

$$\{x; \text{int}, y : \text{int}\} <: \{x : \text{int}\}$$

A Relação de Subtipo (68)

- até agora não vimos como determinar quando um tipo é subtipo de outro
- vamos ver como um tipo registro é subtipo de outro tipo registro
- e como um tipo função é subtipo de outro
- antes disso, eis duas características da relação $<:$

$$S <: S \quad (\text{S-REFL})$$

$$\frac{S <: U \quad U <: T}{S <: T} \quad (\text{S-TRANS})$$

A

Subtipos e Registros (69)

$$\{lab_i : T_i^{i \in 1 \dots n+k}\} <: \{lab_i : T_i^{i \in 1 \dots n}\} \quad (\text{S-RCDWIDTH})$$

$$\frac{S_i <: T_i \text{ para cada } i}{\{lab_i : S_i^{i \in 1 \dots n}\} <: \{lab_i : T_i^{i \in 1 \dots n}\}} \quad (\text{S-RCDDEPTH})$$

$$\frac{\{k_j : S_j^{j \in 1 \dots n}\} \text{ e permutação de } \{l_i : T_i^{i \in 1 \dots n}\}}{\{k_j : S_j^{j \in 1 \dots n}\} <: \{l_i : T_i^{i \in 1 \dots n}\}} \quad (\text{S-RCDPERM})$$

Subtipos (70)

- a conexão entre a relação de subtipo $<:$ e o sistema de tipos é dada pela seguinte regra:

$$\frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T} \quad (\text{T-SUB})$$

- Com subtipos é possível tipar expressões antes consideradas mal tipadas, como por exemplo

$$(fn \ r : \{x : \text{int}\} \Rightarrow \#x \ r) \ \{x = 0, y = 1\}$$

A seguir veremos que podemos definir reação de subtipos com outros tipos também, tais como tipos função e pares ordenados.

Subtipos e Funções (71)

- Considere um contexto que espera função do tipo $T_1 \rightarrow T_2$, como saber se uma função do tipo $S_1 \rightarrow S_2$ pode ser usada com segurança ?
- ou seja, como deve ser definida a relação $<:$ para tipos função?

$$\frac{??}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{S-ARROW})$$

Subtipos e Funções (72)

- contexto espera função do tipo $T_1 \rightarrow T_2$, isso quer dizer que:
 - função deve receber expressão do tipo T_1 como argumento
 - função, caso aplicada, retornará termo do tipo T_2 ao contexto em que foi chamada

Subtipos e Funções (73)

- para função do tipo $S_1 \rightarrow S_2$ ser usada no lugar de uma do tipo $T_1 \rightarrow T_2$
 - $S_2 <: T_2$, ou seja contexto espera que resultado seja do tipo T_2 mas vir algo do tipo S_2 , e
 - $T_1 <: S_1$, argumento que virá continuará sendo do tipo T_1 , mas a função estará a espera de argumento do tipo S_1

Subtipos e funções (74)

A regra fica portanto:

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{S-ARROW})$$

Note que para completar a extensão da linguagem L3 com subtipos falta ainda definir a relação de subtipagem envolvendo outros tipos de L3 (tipos pares ordenados e tipos referência).

1.6 Orientação a Objetos

OO e Cálculo λ (75)

- objetivo: **compreender** características complexas de linguagens OO pela aproximação com construções de mais baixo nível
- podemos pensar que objetos e classes são formas derivadas definidas em termos de construções mais simples
- a linguagem de nível mais baixo será a linguagem L3, que é na verdade cálculo λ tipado com funções, records, referências e subtipos
- esse mapeamento pode se definido formalmente
- aqui veremos coleção de idiomas do cálculo λ simulando objetos e classes semelhantes aos de Java

Objetos I (76)

Que aspectos de OO serão descritos?

- objeto como uma estrutura de dados encapsulando estado interno

- acesso a esse estado via uma coleção de métodos
- estado interno representado por variáveis de instância que são compartilhadas pelos métodos e inacessíveis pelo resto do programa

Objetos II (77)

- objetos representando contadores
- cada objeto
 - possui um número e
 - fornece 2 métodos `get` e `inc` para obter e incrementar o número respectivamente

Objetos III (78)

```

c ≡ let x = ref 1 in
    {get = fn _:unit => !x,
     inc = fn _:unit => x := !x + 1}

▷ c : {get : unit → int, inc : unit → unit}

```

- objeto *c* (é um registro!!)
- estado interno *x*
- métodos `get` e `inc`

Objetos IV (79)

No que segue usaremos `()` no lugar da expressão `unit` e *e.lab* no lugar de `! lab` e

```

c.inc ()
▷ () : unit

c.get ()
▷ 2 : int

c.inc (); c.inc (); c.get ()
▷ 4 : int

```

Objetos V (80)

Vamos dar um nome ao tipo do registro:

```
Counter ≡ {get : unit → int, inc : unit → unit}

inc3 ≡ fn c:Counter ⇒ (c.inc (); c.inc (); c.inc ())
▷ inc3 : Counter → unit

inc3 c; c.get ()
▷ 7 : int
```

Geradores de objetos (81)

Função que gera um novo (objeto) contador cada vez que é chamada:

```
newCounter ≡
  fn _ : unit ⇒
    let x = ref 1 in
    {get = fn _:unit ⇒ !x,
     inc = fn _:unit ⇒ x := !x + 1}

▷ newCounter : unit → Counter
```

Geradores de objetos (82)

```
let c = newCounter() in...

c.inc ()
▷ () : unit

c.get ()
▷ 2 : int

c.inc (); c.inc (); c.get ()
▷ 4 : int
```

Subtipos I (83)

Tipo (classe) *ResetCounter* e função *newResetCounter*

```
ResetCounter = {get : unit → int, inc : unit → unit,  
                 reset : unit → unit}
```

```
newResetCounter =  
  fn _ : unit ⇒ let x = ref 1 in  
    {get = fn _ : unit ⇒ !x,  
     inc = fn _ : unit ⇒ x := !x + 1  
     reset = fn _ : unit ⇒ x := 1}  
▷ newResetCounter : unit → ResetCounter
```

Subtipos II (84)

- temos que *ResetCounter* <: *Counter* (por que?)
- códigos clientes que usam *Counter* podem usar também *ResetCounter*
- **inc3** espera tipo *Counter* mas pode ser usado com tipo *ResetCounter*:

```
rc = newResetCounter ()  
▷ rc : ResetCounter  
  
inc3 rc; rc.reset (); inc3 rc; rc.get ()  
▷ 4 : int
```

Agrupando variáveis de instância I (85)

- até agora o estado dos objetos é constituído de somente uma referência a memória
- objetos mais interessantes têm mais do que uma variável de instância
- nos exemplos que seguem vai ser conveniente poder tratar as variáveis de instância como uma única entidade agrupando-as em um registro

Agrupando variáveis de instância II (86)

```
c = let r = {x = ref 1} in  
  {get = fn _ : unit ⇒ !(r.x),  
   inc = fn _ : unit ⇒ r.x := !(r.x) + 1}  
▷ c : Counter
```

O tipo desse record com as variáveis de instância é chamado de tipo de representação. *CounterRep* é só um nome para um tipo

$$\text{CounterRep} = \{x : \text{int ref}\}$$

Classes simples I (87)

```
counterClass =  
  fn r : CounterRep ⇒  
    {get : fn _ : unit ⇒ !(r.x),  
     inc : fn _ : unit ⇒ r.x := !(r.x) + 1}  
  
▷ counterClass : CounterRep → Counter
```

Criando objetos (88)

```
newCounter =  
  fn _ : unit ⇒ let r = {x = ref 1} in  
                 counterClass r  
  
newCounter : unit → Counter
```

Classes simples III (89)

- Os métodos de *counterClass* podem ser **reusados** para definir novas classes chamadas subclasses
- Por exemplo, podemos definir uma classe de contadores com reset

```
resetCounterClass =  
  fn r : CounterRep ⇒  
    let super = counterClass r in  
    {get = super.get  
     inc = super.inc  
     reset = fn _ : unit ⇒ r.x := 1}  
  
▷ resetCounterClass : CounterRep → ResetCounter
```

Criando objetos (90)

```

newResetCounter =
  fn _ : unit ⇒ let r = {x = ref 1} in
    resetCounterClass r

▷ newResetCounter : unit → ResetCounter

```

Adicionando variáveis de instância (91)

- definir *backupCounter* reutilizando *resetCounter*
 - adicionando variável de instância
 - adicionando método **backup** e
 - redefinindo **reset**

Adicionando variáveis de instância (92)

```

backupCounter = {get : unit → int, inc : unit → unit,
  reset : unit → unit, backup : unit → unit}

backupCounterRep = {x : int ref, b : int ref}

backupCounterClass = fn r : backupCounterRep ⇒
  let super = resetCounterClass r in
    {get = super.get
    inc = super.inc
    reset = fn _ : unit ⇒ r.x :=!(r.b)
    backup = fn _ : unit ⇒ r.b :=!(r.x)}

▷ backupCounterClass : backupCounterRep → backupCounter

```

Overriding e subtipos (93)

- Duas coisas interessantes na definição anterior:
 - o objeto pai **super** possui método **reset** mas ele foi sobrescrito com nova definição
 - note o uso de subtipos: *resetCounterClass* foi definida para *counterRep* mas recebe *backupCounterRep*

Criando objetos (94)

```
newBackupCounter =  
  fn _ : unit ⇒ let r = {x = ref 1, b = ref 1} in  
    backupCounterClass r
```

▷ *newBackupCounter* : *unit* → *backupCounter*

Chamando métodos de superclasse (95)

- até aqui **super** foi usada para copiar funcionalidade de superclasses para subclasses
- podemos também usar **super** no corpo da definição de métodos. Supor que queiramos definir uma variação de *backupCounter* de tal forma que toda chamada a **inc** seja precedida de **backup**

```
funnyBackupCounterClass = fn r : backupCounterRep ⇒  
  let super = backupCounterClass r in  
    {get = super.get  
     inc = fn _ : unit ⇒ super.backup(); super.inc()  
     reset = super.reset  
     backup = super.backup}
```

▷ *funnyBackupCounterClass* : *backupCounterRep* → *backupCounter*