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

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

10 de Novembro de 2010

Conteúdo

1 Introdução

1.1 Motivação

O panorama (1)

- A área de Linguagens de Programação é uma das mais antigas de Ciência de Computação
- Permanece uma área ativa e vibrante
- Algumas tendências
  - A web renova interesse em projeto de linguagens: veja Java, por exemplo
  - Tipos são reconhecidas como tendo um papel importante em segurança
  - Análise de programas se torna um componente significante em Engenharia de Software

Projeto de linguagens na prática (2)

- Linguagens são adotadas para preencher uma necessidade: abstração
  - Permitir uma aplicação antes difícil ou impossível executar em uma nova plataforma
- Treinamento de programador tem custo muito alto
  - Linguagens com muitos usuários raramente são substituídas
  - Linguagens populares se tornam estagnadas
  - Novas aplicações e novas plataformas apresentam oportunidades para inovação

Tipos de Linguagens (3)

Imperativa Transformação de estado

Funcional Avaliação sem *side-effects*, funções são dados.

Lógico Declaração de regras lógicas

Orientada a objetos Encapsulamento, herança, polimorfismo.

Concorrentes ou distribuídas Distribuição, sincronização e comunicação

Orientadas a Aspectos aspectos, *pointcuts*

Exemplos (4)

Imperativas Fortran, Algol, Cobol, C, Pascal

Funcionais Lisp, Scheme, ML, Haskell, XPath

Lógicas Prolog, λ Prolog

Orientadas a objetos Smalltalk, Eiffel, Self, C++, Java

Concorrentes ou distribuídas Fortan90, HPF, CSP

Orientadas a Aspectos AspectJ

Por que tantas linguagens? (5)

- Muitas linguagens foram criadas para aplicações específicas
- Aplicações com necessidades distintas ou conflitantes
  - IA: computação simbólica (Lisp, Prolog)
  - Computação científica: alto desempenho (Fortran)
  - Negócios: geração de relatórios (Cobol)
  - Programação de sistemas: acesso ao baixo nível (C)
  - Customização: Scripts (Perl, Javascript)
  - Sistemas distribuídos: computação móvel (Java, C#)
  - Outras (LaTeX, ...)

1.2 Linguagens

Critérios desejáveis (6)

Queremos uma linguagem

- compreensível, com sintaxe e semântica simples,

- legível e fácil para escrever,
- poderosa e abstrata, com muitas funções,
- portável (independente da máquina)
- escalável, que suporte programação de sistemas grandes,
- eficiente na execução e compilação.

Critérios desejáveis (7)

- Estes objetivos são quase sempre conflitantes
  - Segurança e independência de máquina em geral custam eficiência
  - Sistemas de tipos restringem estilo de programação para obter garantias
  - Muitas funções podem prejudicar a compreensão
- Projeto de boas linguagens é uma tarefa difícil
- Não se chega a uma boa linguagem por acidente

Como projetar linguagens? Características (8)

- Tipos e estruturas de dados
- Operadores, predicados e expressões
- Controle
- Abstração: funções anônimas, tipos de dados abstratos (objetos), módulos, polimorfismo, reflexão, ...
- Tratamento dos erros
- Concorrência, paralelismo
- Entrada/saida
- Comentários
- Biblioteca padrão

Estudando características (9)

- Uma maneira de estudar um conceito ou construção é considerá-lo em uma linguagem pequena
  - experimentá-lo com o uso da linguagem
  - definir a linguagem rigorosamente: sintaxe (gramática), verificações estáticas (sistema de tipos), comportamento de execução (semântica)
  - estudar as propriedades da linguagem
  - relacionar a linguagem com outras
- Depois disso, a construção pode ser estudada no contexto de uma linguagem grande.

Semântica - por que? (10)

Dada uma linguagem de programação, há questões naturais tais como

- A sua implementação está correta?
- O que significa um programa escrito nessa linguagem?
- Um programa está correto em relação a sua especificação?

Uma semântica formal é um modelo matemático de uma linguagem.

- Ela serve como especificação precisa da linguagem
- Ela permite analisar e provar características de linguagens e de programas
- A atividade de definir uma semântica formal ajuda a revelar e corrigir problemas no projeto de uma linguagem

Semântica - quem precisa? (11)

- Semântica é necessária, mas também é "pesada" ?!
  - nem sempre o benefício em relação aos custos vale a pena
- A maioria dos programadores se dá por satisfeito com conhecimento menos preciso a aprendem por tentativa e erro

Semântica - quem precisa? (12)

Os que querem

- saber *exatamente* e sem *ambiguidades* o que um programa significa
- explicar uma linguagem para outros (inventor da linguagem)
- programar ferramentas de análise (coleta de informações sobre o código)
- programar ferramentas de transformação (compilador, interpretador, ferramentas para refatoração ...)
- provar que uma software está correto (sistemas críticos)

1.3 Como definir a semântica de uma linguagem?

Como descrever exatamente uma linguagem? Muitas vezes uma descrição textual não é sucinta e resulta em manuais volumosos e com muitas ambiguidades próprias da linguagem natural. Como melhorar a situação? Qual seria uma formalização adequada? Ao menos os seguintes critérios devem ser satisfeitos:

Sucinta, compreensível e não-ambígua A especificação de uma linguagem serve para a comunicação entre projetistas de linguagem, engenheiros de compiladores e programadores. Por isso, ele deve ser sucinta e acessível a humanos.

Flexível Preferencialmente uma única técnica de formalização para todos aspectos de linguagens e paradigmas diferentes.

Formal Acessível aos métodos formais, provas matemáticas, etc.

Semântica (13)

Temos três linhas predominantes para descrever semântica dinâmica:

Semântica operacional

Semântica denotational

Semântica axiomática

Semântica estática definida através de um **Sistema de Tipos**

Veremos semântica denotacional e semântica axiomática para uma linguagem imperativa simples que chamaremos de IMP. Semântica operacional e sistemas de tipos serão estudados através de diversas extensões de cálculo lambda tipado simples.

2 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).

2.1 A Linguagem L1

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

Sintaxe de L1 (14)

$$
\begin{array}{rcl}
e & ::= & n \mid b \mid e_1 \; op \; e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
& \mid & l\texttt{:=}\ e \mid \texttt{!}\ l \\
& \mid & \texttt{skip} \mid e_1\texttt{;}\ e_2 \\
& \mid & \texttt{while}\ e_1\ \texttt{do}\ e_2
\end{array}
$$

onde

$$
\begin{array}{rcl}
n & \in & \textit{conjunto de numerais inteiros} \\
b & \in & \{\texttt{true}, \texttt{false}\} \\
op & \in & \{+, \geq\} \\
l & \in & \textit{conjunto de endereços}
\end{array}
$$

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 expressões sem sentido tais como $10 + \texttt{false}$ e também $\texttt{while}\ 2 + 1\ \texttt{do}\ 20$, ou seja, nem todo elemento do conjunto definido pela gramática abstrata acima é uma expressão L1
- em L1 o programador tem acesso direto a endereços (mas não há aritmética com endereços).

2.1.1 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 estados (15)

- 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 $c \not\longrightarrow$ quando não existe c' tal que $c \longrightarrow c'$

Semântica Operacional de L1 - estados (16)

- 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:

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

- Para memória σ acima, temos que $Dom(\sigma) = \{l_1, l_2\}$ e $\sigma(l_1) = 0$ e $\sigma(l_2) = 10$

Valores e Erros de Execução (17)

- Valores são expressões já completamente avaliadas
- Os valores da linguagem L1 são dados pela seguinte gramática:

$$v ::= n \mid b \mid \texttt{skip}$$

- Se $\langle e,\ \sigma \rangle \not\longrightarrow$ 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
  - $\langle 2\texttt{+true},\ \sigma \rangle \not\longrightarrow$
  - $\langle l\texttt{:=}2,\ \sigma \rangle \not\longrightarrow$, caso $l \notin Dom(\sigma)$

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

$$\frac{premissa \ldots premissa}{conclusao}$$

Em uma semântica operacional *small step* não há regras para valores e, 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 as subexpressões de uma expressão são avaliadas e as regras de computação dizem, de fato, como uma determinada expressão efetua uma computação interessante.

Semântica Operacional de Operações Básicas (18)

$$\frac{[\![n]\!] = [\![n_1 + n_2]\!]}{\langle n_1 + n_2,\ \sigma\rangle \longrightarrow \langle n,\ \sigma\rangle} \quad (\text{OP}+)$$

$$\frac{[\![b]\!] = [\![n_1 \geq n_2]\!]}{\langle n_1 \geq n_2,\ \sigma\rangle \longrightarrow \langle b,\ \sigma\rangle} \quad (\text{OP}\geq)$$

$$\frac{\langle e_1,\ \sigma\rangle \longrightarrow \langle e_1',\ \sigma'\rangle}{\langle e_1\ op\ e_2,\ \sigma\rangle \longrightarrow \langle e_1'\ op\ e_2,\ \sigma'\rangle} \quad (\text{OP1})$$

$$\frac{\langle e_2,\ \sigma\rangle \longrightarrow \langle e_2',\ \sigma'\rangle}{\langle v\ op\ e_2,\ \sigma\rangle \longrightarrow \langle v\ op\ e_2',\ \sigma'\rangle} \quad (\text{OP2})$$

As regras OP1 e OP2 acima são regras de reescrita, e as regras OP+ e OP$\geq$ 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$\geq$:dessa forma as regras especificam que a computação de + e $\geq$ se dará somente nos casos em que ambos operandos forem números inteiros, caso contrário temos um erro de execução (expressão que não é valor mas para a qual não há regra de transição).

Condicional (19)

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

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

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

As regras IF1 e IF2 acima são regras de computação para o condicional, e a regra IF3 é uma regra de reescrita.

Seqüência (20)

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

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

Pela regra de reescrita SEQ2 acima, a avaliação seqüencial de duas expressões é feita da esquerda para direita. Pela regra SEQ1 (uma regra de computação), quando o lado esquerdo estiver completamente reduzido para `skip`, a avaliação deve continuar com a expressão no lado direito do ponto e vírgula.
Note que aqui foi feita uma escolha arbritrária no projeto da linguagem: a avaliação continua com a expressão no lado direito somente se a expressão do lado esquerdo do ponto e vírgula avalia para `skip`. Qualquer outra possibilidade leva a erro de execução.

Operações com a Memória (21)

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

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

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

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 $l \in Dom(\sigma)$).

Na regra ATR1 podemos observar outras escolhas feitas em relaçao a semântica da linguagem: a memória somente pode receber valores inteiros e 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 atribuido.

While (22)

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

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

2.1.2 Sistema de Tipos para L1

Observe que a gramática abstrata definida no slide (1) admite expressões cuja avaliação leva a erro de execução (um erro de execução aqui é representado pela impossibilidade de aplicar uma regra da semântica operacional para uma expressão que não é valor). Vamos agora ver um *sistema de tipos* para a linguagem L1. Este sistema de tipos especifica uma análise estática a ser feita sobre árvores de sintaxe abstrata. Somente expressões consideradas *bem tipadas* por essa análise 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 (23)

- o sistema de tipos de L1 consiste de um conjunto de regras de inferência

$$\frac{premissa \ldots premissa}{conclusao}$$

- Premissas e conclusão são da forma $\Delta \vdash e : T$, lido "a expressão e é do tipo T dadas as informações a cerca do tipo de endereços em Δ", onde:
  - Δ é um mapeamento de endereços para seus tipos (int ref)
  - e é uma expressão da linguagem, e
  - T é um tipo pertencente ao conjunto definido pela seguinte gramática

$$T \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit}$$

Note que em L1 a memória só pode receber valores inteiros, logo todos os endereços em L1 são do tipo int ref. Seguem abaixo as regras do sistema de tipos. Há uma regra para cada cláusula da gramática de expressões.

Valores e Operações Básicas (24)

$$\Delta \vdash n : \mathsf{int} \qquad (\textsc{Tint})$$

$$\Delta \vdash b : \mathsf{bool} \qquad (\textsc{Tbool})$$

$$\frac{\Delta \vdash e_1 : \mathsf{int} \qquad \Delta \vdash e_2 : \mathsf{int}}{\Delta \vdash e_1 + e_2 : \mathsf{int}} \qquad (\mathrm{T}+)$$

$$\frac{\Delta \vdash e_1 : \mathsf{int} \qquad \Delta \vdash e_2 : \mathsf{int}}{\Delta \vdash e_1 \geq e_2 : \mathsf{bool}} \qquad (\mathrm{T}\geq)$$

Observe pelas regras OP+ e OP≥ que os operandos de + e de ≥ devem ser do tipo inteiro. Por estas regras expressões tais como $4 + \texttt{true}$ e $\texttt{true} \geq \texttt{skip}$ são consideradas *mal tipadas*.

Condicional (25)

$$\frac{\Delta \vdash e_1 : \mathsf{bool} \qquad \Delta \vdash e_2 : T \qquad \Delta \vdash e_3 : T}{\Delta \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T} \qquad (\textsc{If})$$

- Pela regra acima, para o condicional ser bem tipado a expressão da parte `then` e a expressão da parte `else` devem ser do mesmo tipo
- Note que expressões tais como $\texttt{if}\ 5 + 3 \geq 2\ \texttt{then true else}\ 5$, de acordo com a semântica operacional, não levam a erro de execução. Mesmo assim, não são consideradas bem tipadas pela regra de tipo acima. Podemos dizer que o sistema de tipos está sendo muito conservador e recusando mais expressões do que deveria
- Isso acontece pois o sistema de tipos especifica uma análise *estática* feita sobre a árvore de sintaxe abstrata, ou seja sem saber se o resultado da avaliação do condicional virá da avaliação da subexpressão da parte `then` ou da subexpressão da parte `else`. Para pode concluir sobre o tipo de toda a expressão é preciso portanto, exigir que o tipo de ambas subexpressões seja o mesmo.

Operações com a memória (26)

$$\frac{\Delta \vdash e : \mathsf{int} \qquad \Delta(l) = \mathsf{int\ ref}}{\Delta \vdash l\texttt{:=}e : \mathsf{unit}} \quad (\textsc{Tatr})$$

$$\frac{\Delta(l) = \mathsf{int\ ref}}{\Delta \vdash \texttt{!}\ l : \mathsf{int}} \quad (\textsc{Tderef})$$

Seqüência e While (27)

$$\Delta \vdash \texttt{skip} : \mathsf{unit} \quad (\textsc{Tskip})$$

$$\frac{\Delta \vdash e_1 : \mathsf{unit} \qquad \Delta \vdash e_2 : T}{\Delta \vdash e_1\texttt{;}e_2 : T} \quad (\textsc{Tseq})$$

$$\frac{\Delta \vdash e_1 : \mathsf{bool} \qquad \Delta \vdash e_2 : \mathsf{unit}}{\Delta \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 : \mathsf{unit}} \quad (\textsc{Twhile})$$

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`.

2.1.3 Propriedades de L1

Propriedades (28)

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 concluimos 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 (29)

- Um sistema de tipos é seguro se expressões consideradas bem tipadas pelas suas regras não levam a **erro de execução** quando avaliadas de acordo com as regras da semântica operacional
- Erro de execução o ocorre quando temos um estado $< e, \sigma >$ ao qual nenhuma regra da operacional se aplica
- De maneira **informal** podemos resumir essa noção de segurança através do seguinte *slogan*:

$$Se\ \Delta \vdash e : T\ então\ e \not\longrightarrow^*\quad erro$$

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:

Segurança Sintática (30)

- 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*:

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

- 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 da expressão original. *Slogan*:

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

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

$$Segurança = Progresso + Preservação$$

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 *especificas* para a linguagem L1.

Progresso e Preservação para L1 (31)

Teorema 2 (Progresso) Se $\Delta \vdash e : T$ e $Dom(\Delta) \subseteq Dom(\sigma)$ 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 $\Delta \vdash e : T$ e $Dom(\Delta) \subseteq Dom(\sigma)$ e $\langle e,\ \sigma \rangle \longrightarrow \langle e',\ \sigma' \rangle$ então $\Delta \vdash e' : T$ e $Dom(\Delta) \subseteq Dom(\sigma')$

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 algoritmicos (32)

- Problema da **Verificação de Tipos**: dados ambiente Δ, expressão e e tipo T, o julgamento de tipo $\Delta \vdash e : T$ é derivável usando as regras do sistema de tipos?
- Problema da **Tipabilidade**: dados ambiente Δ e expressão e, encontrar tipo T tal que $\Delta \vdash e : T$ 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 há algoritmos simples para ambos os problemas
Observe que, do ponto de vista prático, o problema da *verificação de tipos* para L1 não é interessante[1]. 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 .

Algoritmo de Inferência de Tipos (33)

Teorema 4 Dados ambiente Δ e expressão e, existe algoritmo que resolve o problema da tipabilidade para L1.

Prova. Exibir um algoritmo (ver exercício abaixo) que tem como entrada um ambiente de tipo Δ e uma expressão e e provar que o algoritmo: (i) termina sempre sua execução, e (ii) retorne um tipo T se e somente se $\Delta \vdash e : T$ é derivável de acordo com o sistema de tipos para L1 ■

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] fora alguns exercícios de verificação de tipos a serem feitos

2.2 A Linguagem L2

A linguagem L2 é uma extensão de L1 com funções, aplicação, variáveis, funções recursivas e declarações. Primeiro, devemos estender a sintaxe de L1:

Sintaxe de L2 (34)

$$
\begin{array}{lcl}
e & ::= & n \mid b \mid e_1\ op\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
 & \mid & l\texttt{:=}\ e \mid \texttt{!}\ l \\
 & \mid & \texttt{skip} \mid e_1\texttt{;}\ e_2 \\
 & \mid & \texttt{while}\ e_1\ \texttt{do}\ e_2 \\
(*) & \mid & fn\ x\texttt{:}T \Rightarrow e \mid e_1\ e_2 \mid x \\
(*) & \mid & \texttt{let}\ x\texttt{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end} \\
(*) & \mid & \texttt{let rec}\ f\texttt{:}T_1 \to T_2 = (fn\ y\texttt{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end} \\
 & & \\
v & ::= & n \mid b \mid \texttt{skip} \\
(*) & \mid & fn\ x\texttt{:}T \Rightarrow e
\end{array}
$$

Na gramática acima:

- x representa um elemento pertencente ao conjunto $Ident$ de identificadores
- $fn\ x\texttt{:}T \Rightarrow e$ é uma função (sem nome)
- $e_1\ e_2$ é a aplicação da expressão e_1 a expressão e_2
- $\texttt{let}\ x\texttt{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end}$ é uma construção que permite declarar identificadores e $\texttt{let rec}\ f\texttt{:}T_1 \to T_2 = (fn\ y\texttt{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end}$ permite a declaração de funções recursivas
- note que em L2 o programador deve escrever informação de tipo em programas

Linguagem de Tipos (35)

- os tipos da linguagem L2 são dados pela seguinte gramática:

$$T ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid T_1 \to T_2$$

- $T_1 \to T_2$ é o tipo de função cujo argumento é do tipo T_1 e cujo resultado é do tipo T_1

Sintaxe de L2 (36)

- 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 $T_1 \to T_2 \to T_3$ é o mesmo que $T_1 \to (T_2 \to T_3)$

- fn se estende o mais a direita possível, logo $fn\ x{:}\mathsf{unit} \Rightarrow x; x$ é o mesmo que $fn\ x{:}\mathsf{unit} \Rightarrow (x;\ x)$

2.2.1 Funções

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

Semântica Formal (37)

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

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

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

Substituição - Exemplos (38)

- 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:

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

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 substituidas e que nenhuma variável livre ficará indevidamente ligada após a substituição.

$$
\begin{array}{lcl}
\{e/x\}\ x & = & e \\
\{e/x\}\ y & = & y \ \ (se\ x \neq y) \\
\{e/x\}\ fn\ y : T \Rightarrow e' & = & fn\ z : T \Rightarrow \{e/x\}\{z/y\}e' \\
 & & se\ x \neq y \\
 & & z \notin\ fv(e) \\
 & & z \notin\ fv(e') \cup \{y\} \\
\{e/x\}\ (e_1\ e_2) & = & (\{e/x\}e_1)(\{e/x\}e_2) \\
\{e/x\}\ n & = & n \\
\{e/x\}\ (e_1\ \text{op}\ e_2) & = & \{e/x\}e1\ \text{op}\ \{e/x\}e2 \\
\{e/x\}\ (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) & = & \texttt{if}\ \{e/x\}e1\ \texttt{then}\ \{e/x\}e2\ \texttt{else}\ \{e/x\}e3 \\
\{e/x\}\ b & = & b \\
\{e/x\}\ \texttt{skip} & = & \texttt{skip} \\
\{e/x\}\ l := e' & = & l := \{e/x\}e' \\
\{e/x\}\ !l & = & !l \\
\{e/x\}\ (e1; e2) & = & \{e/x\}e1; \{e/x\}e2 \\
\{e/x\}(\texttt{while}\ e_1\ \texttt{do}\ e_2) & = & \texttt{while}\ \{e/x\}e_1\ \texttt{do}\ \{e/x\}e_2 \\
\{e/x\}(\texttt{let}\ y{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end}) & = & \texttt{let}\ z{:}T = \{e/x\}e1\ \texttt{in}\ \{e/x\}\{z/y\}e_2\ \texttt{end} \\
 & & se\ x \neq y \\
 & & z \notin\ fv(e) \\
 & & z \notin\ fv(e_2) \cup \{y\}
\end{array}
$$

Na definição acima $fv(e)$ é o conjunto de variáveis livres da expressão e (variáveis que ocorrem mas não são declaradas em e).

$$
\begin{array}{lcl}
fv(x) & = & \{x\} \\
fv(fn\ y : T \Rightarrow e) & = & fv(e) - \{y\} \\
fv(e_1 e_2) & = & fv(e_1) \cup fv(e_2) \\
fv(n) & = & \{\ \} \\
fv(e1\ \text{op}\ e2) & = & fv(e1) \cup fv(e2) \\
\ldots & \ldots & \ldots
\end{array}
$$

Como exercício, termine a definição de fv iniciada acima.

Tipando Funções (39)

- O ambiente Δ mapeia somente os tipos de endereços; em L2, Γ é um mapeamento de identificadores para seus tipos
- Notação: escrevemos $\Gamma, x : T$ para a função parcial que mapeia identificador x para T, mas para outros identificadores diferentes de x o mapemento é de acordo com Γ
- Γ é uma representaçao simplificada de uma tabela de símbolos (ver disciplina de Compiladores)

$$\frac{\Gamma(x) = T}{\Gamma; \Delta \vdash x : T} \qquad (\textsc{Tvar})$$

$$\frac{\Gamma, x : T; \Delta \vdash e : T'}{\Gamma; \Delta \vdash fn\ x : T \Rightarrow e : T \to T'} \qquad (\textsc{Tfn})$$

$$\frac{\Gamma;\Delta \vdash e_1 : T \to T' \qquad \Gamma;\Delta \vdash e_2 : T}{\Gamma;\Delta \vdash e_1\ e_2 : T'} \quad (\textsc{Tapp})$$

2.2.2 Declarações

Definições Locais (40)

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

$$e ::= \cdots \mid \texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end}$$

Pode ser considerada como simples açúcar sintático:

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

Exemplo: 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:

$$\texttt{let}\ f{:}\mathsf{int} \to \mathsf{int} = fn\ x{:}\mathsf{int} \Rightarrow x + 1\ \texttt{in} \quad f\ 10\ \texttt{end}$$

Regras de tipagem e regras de redução (41)

$$\frac{\Gamma;\Delta \vdash e_1 : T \qquad \Gamma, x : T;\Delta,\vdash e_2 : T'}{\Gamma;\Delta \vdash \texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end} : T'} \quad (\textsc{Tlet})$$

$$\langle \texttt{let}\ x{:}T = v\ \texttt{in}\ e_2\ \texttt{end},\ \sigma\rangle \quad \longrightarrow \quad \langle \{v/x\}e_2,\ \sigma\rangle \quad (\textsc{let1})$$

$$\frac{\langle e_1,\ \sigma\rangle \quad \longrightarrow \quad \langle e_1',\ \sigma'\rangle}{\langle \texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end},\ \sigma\rangle \quad \longrightarrow \quad \langle \texttt{let}\ x{:}T = e_1'\ \texttt{in}\ e_2\ \texttt{end},\ \sigma'\rangle} \quad (\textsc{let1})$$

Funções recursivas (42)

$$e ::= \ldots \mid \texttt{let rec}\ f{:}T_1 \to T_2 = (fn\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end}$$

$$\frac{\Gamma, f : T_1 \to T_2, y : T_1;;\Delta \vdash e_1 : T_2 \qquad \Gamma, f : T_1 \to T_2;\Delta,\vdash e_2 : T}{\Gamma;\Delta \vdash \texttt{let rec}\ f{:}T_1 \to T_2 = (fn\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end} : T} \quad (\textsc{Tletrec})$$

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 (43)

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

Mais açucar sintático (44)

- $e1; e2$ pode ser codificado como $(fn\ y\texttt{:unit} \Rightarrow e2)\ e1$ onde $y \notin fv(e_2)$
- `while` $e1$ `do` $e2$ 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 $y \notin fv(e1) \cup fv(e2)$.

2.2.3 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 (45)

Teorema 5 (Progresso) Se e é fechado e $\Gamma; \Delta \vdash e : T$ e $Dom(\Delta) \subseteq Dom(\sigma)$ 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 $\Delta \vdash e : T$, $Dom(\Gamma; \Delta) \subseteq Dom(\sigma)$ e $\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle$ então $\Gamma; \Delta \vdash e' : T$ e $Dom(\Gamma; \Delta) \subseteq dom(\sigma')$.

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

2.3 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 de qualquer tipo.

2.3.1 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 (46)

$$
\begin{array}{lcl}
e & ::= & n \mid b \mid e_1\ op\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
(*) & \mid & e_1\texttt{:=}\ e_2 \mid \texttt{!}\ e \mid \mathsf{ref}\ e \mid \boxed{\mathsf{l}} \\
 & \mid & \texttt{skip} \mid e_1\texttt{;}\ e_2 \\
 & \mid & \texttt{while}\ e_1\ \texttt{do}\ e_2 \\
 & \mid & fn\ x\texttt{:}T \Rightarrow e \mid e_1\ e_2 \mid x \\
 & \mid & \texttt{let}\ x\texttt{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end} \\
 & \mid & \texttt{let rec}\ f\texttt{:}T_1 \to T_2 = (fn\ y\texttt{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end} \\
(*) & \mid & (e_1, e_2) \mid \sharp 1\ e \mid \sharp 2\ e \\
(*) & \mid & \{lab_1 = e_1, \ldots lab_n = e_n\} \mid \sharp lab\ e \qquad n \geq 0
\end{array}
$$

onde

$$
\begin{array}{lcl}
n & \in & \textit{conjunto de numerais inteiros} \\
b & \in & \{\texttt{true}, \texttt{false}\} \\
op & \in & \{+, \geq\} \\
l & \in & \textit{conjunto de endereços} \\
lab & \in & \textit{conjunto de rótulos}
\end{array}
$$

Novas construções de L3 (47)

A linguagem L3 difere de L2 no seguinte:

- a memória pode conter valores de qualquer tipo
- programador não tem mais acesso direto a endereços (note que na gramática acima contudo ainda aparecem endereços e agora eles são expressões!!)
- e_1 `:=` e_2 ao invés de l `:=` e_2 - a atribuição adquire o mesmo *status* 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
- ref e - esta construção é utilizada para alocar um endereço de memória contendo o valor da expressão e

Novas estruturas de dados (48)

- L3 possui também pares ordenados e registros com quantidade variável de campos identificados por rótulos
- operações $\sharp 1\ e$, $\sharp 2\ e$ para projeção do primeiro e do segundo componente de um par ordenado e
- $\sharp lab\ e$ para projeção do componente de rótulo lab do registro e

Tipos para L3 (49)

- Note também **três novos tipos**: para pares ordenados, registros e endereços
- o tipo T ref é o tipo de expressões ref e

$$
\begin{array}{rcll}
T & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} & \\
& \mid & T_1 \to T_2 & \\
& \mid & T_1 * T_2 & \\
& \mid & \{lab_1 : T_1, \ldots lab_n : T_n\} & n \geq 0 \\
& \mid & T\ \mathsf{ref} &
\end{array}
$$

2.3.2 Semântica Operacional de L3

Valores de L3 (50)

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

$$
\begin{array}{rcll}
v & ::= & n \mid b \mid \texttt{skip} \mid fn\ x{:}T \Rightarrow e & \\
& \mid & (v_1, v_2) & \\
& \mid & \{lab_1 = v_1, \ldots lab_n = v_n\} & n \geq 0 \\
& \mid & \boxed{\mathsf{l}} &
\end{array}
$$

Pares e Projeção (51)

$$
\frac{\langle e_2,\ \sigma\rangle \longrightarrow \langle e_2',\ \sigma'\rangle}{\langle (v, e_2),\ \sigma\rangle \longrightarrow \langle (v, e_2'),\ \sigma'\rangle} \quad (\textsc{par}1)
$$

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

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

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

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

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

Registros e Projeção (52)

$$\frac{\langle e_j,\ \sigma\rangle \quad\longrightarrow\quad \langle e_j',\ \sigma'\rangle}{\begin{array}{c}\langle \{lab_i = v_i^{i\in 1\ldots j-1}, lab_j = e_j, lab_k = e_k^{k\in j+1\ldots n}\},\ \sigma\rangle \\ \longrightarrow \\ \langle \{lab_i = v_i^{i\in 1\ldots j-1}, lab_j = e_j', lab_k = e_k^{k\in j+1\ldots n}\},\ \sigma'\rangle\end{array}} \qquad (\text{RCD1})$$

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

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

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

Operações com Memória (53)

$$\frac{l \notin Dom(\sigma)}{\langle \mathsf{ref}\ v,\ \sigma\rangle \longrightarrow \langle l,\ \sigma[l \mapsto v]\rangle} \quad (\textsc{ref}1)$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle}{\langle \mathsf{ref}\ e,\ \sigma\rangle \longrightarrow \langle \mathsf{ref}\ e',\ \sigma'\rangle} \quad (\textsc{ref}2)$$

$$\frac{l \in Dom(\sigma) \quad \sigma(l) = v}{\langle \texttt{!}\ l,\ \sigma\rangle \longrightarrow \langle v,\ \sigma\rangle} \quad (\textsc{deref}1)$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma\rangle}{\langle \texttt{!}\ e,\ \sigma\rangle \longrightarrow \langle \texttt{!}\ e',\ \sigma\rangle} \quad (\textsc{deref}2)$$

Operações com Memória - cont. (54)

$$\frac{l \in Dom(\sigma)}{\langle l \texttt{:=}\ v,\ \sigma\rangle \longrightarrow \langle \texttt{skip},\ \sigma[l \mapsto v]\rangle} \quad (\textsc{atr}1)$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle}{\langle l \texttt{:=}\ e,\ \sigma\rangle \longrightarrow \langle l \texttt{:=}\ e',\ \sigma'\rangle} \quad (\textsc{atr}2)$$

$$\frac{\langle e_1,\ \sigma\rangle \longrightarrow \langle e_1',\ \sigma'\rangle}{\langle e_1 \texttt{:=}\ e_2,\ \sigma\rangle \longrightarrow \langle e_1' \texttt{:=}\ e_2,\ \sigma'\rangle} \quad (\textsc{atr}3)$$

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 ref e deve ser novo (na regra REF1 acima isso é especificado pela premissa $l \notin Dom(\sigma)$).

2.3.3 Sistema de Tipos para L3

Para especificar regras de tipos para árvores de sintaxe abstrata correspondentes a programas fonte escritos na linguagem L3 não é necessário o ambiente Δ em julgamentos de tipos $\Gamma; \Delta \vdash e : T$. Isso porque em programas fonte não é possivel escrever expressões com endereços. Entretanto, como a técnica de prova de segurança consite em verificar se expressões que aparecem em etapas intermediárias da avalação também são bem tipadas, e endereços podem aparecer em etapas intermediárias é necessário prever Δ e regra de tipo para endereços.

Regras de Tipo para Pares em L3 (55)

$$\frac{\Gamma; \Delta \vdash e_1 : T_1 \qquad \Gamma; \Delta \vdash e_2 : T_2}{\Gamma; \Delta \vdash (e_1, e_2) : T_1 * \ T_2} \quad (\textsc{Tpar})$$

$$\frac{\Gamma; \Delta \vdash e : T_1 * T_2}{\Gamma; \Delta \vdash \sharp 1 \ e : T_1} \quad (\textsc{Tprj1})$$

$$\frac{\Gamma; \Delta \vdash e : T_1 * T_2}{\Gamma; \Delta \vdash \sharp 2 \ e : T_2} \quad (\textsc{prj2})$$

- 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 (56)

$$\frac{\Gamma; \Delta \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma; \Delta \vdash e_n : T_n}{\Gamma; \Delta \vdash \{lab_1 = e_1, \ldots lab_n = e_n\} : \{lab_1 : T_1, \ldots lab_n : T_n\}} \quad (\textsc{Trcd})$$

$$\frac{\Gamma; \Delta \vdash e : \{lab_1 : T_1, \ldots lab_n : T_n\}}{\Gamma; \Delta \vdash \sharp lab_i \ e : T_i} \quad (\textsc{Tprj})$$

Sistema de tipos conservador (57)

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

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

- Outro exemplo de programa que não leva a erro de execução, mas também não é considerado bem tipado:

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

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

Regras de Tipos para Operações com Memória (58)

$$\frac{\Gamma; \Delta \vdash e_1 : T\ \mathsf{ref} \qquad \Gamma; \Delta \vdash e_2 : T}{\Gamma; \Delta \vdash e_1 \texttt{:=} e_2 : \mathsf{unit}} \quad (\textsc{Tatr})$$

$$\frac{\Gamma; \Delta \vdash e : T\ \mathsf{ref}}{\Gamma; \Delta \vdash \texttt{!}\ e : T} \quad (\textsc{Tderef})$$

$$\frac{\Gamma; \Delta \vdash e : T}{\Gamma; \Delta \vdash \mathsf{ref}\ e : T\ \mathsf{ref}} \quad (\textsc{Tref})$$

$$\frac{\Delta(l) = T\ \mathsf{ref}}{\Gamma; \Delta \vdash l : T\ \mathsf{ref}} \quad (\textsc{Tl})$$

2.4 Exceções

Exceções (59)

- várias situações nas quais uma função precisa sinalizar, para quem a chamou, que não poderá realizar a sua tarefa por alguma razão
  - algum cálculo involve 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 (60)

- 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 (61)

- 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 exceções

2.4.1 Ativando exceções

Ativando exceções (62)

- 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
- Simplesmente adicionamos a expressão `raise` a gramática abstrata de L3

$$e \quad ::= \quad \ldots \quad | \quad \texttt{raise}$$

Semântica Operacional (63)

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

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

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

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

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

Seqüência (64)

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

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

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

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

Semântica Operacional (65)

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

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

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

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

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

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

Semântica Operacional (66)

- 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 $\langle v\ \texttt{raise},\ \sigma\rangle \rightarrow \langle \texttt{raise},\ \sigma\rangle$ e não $\langle e\ \texttt{raise},\ \sigma\rangle \rightarrow \langle \texttt{raise},\ \sigma\rangle$ (pense como ficaria a avaliação de uma expressão e `raise` onde a avaliação de e entra em loop)

Tipos (67)

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

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

- agora deixa de ser verdade que toda expressão possui somente um único tipo !

2.4.2 Tratamento de exceções

Tratamento de exceções (68)

- 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 (69)

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

$$\begin{array}{rcl} e & ::= & \ldots \\ & & \texttt{raise} \\ & & \texttt{try}\ e_1\ \texttt{with}\ e_2 \end{array}$$

- `try` e_1 `with` e_2, quando avaliado, retorna o valor da avaliação de e_1. Se essa avaliação ativar uma exceção o tratador e_2 é avaliado

Semântica Operacional (70)

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

$$\langle \texttt{try}\ v_1\ \texttt{with}\ e_2,\ \sigma\rangle \rightarrow \langle v_1,\ \sigma\rangle \qquad (\textsc{try}1)$$

$$\langle \texttt{try raise with}\ e_2,\ \sigma\rangle \rightarrow \langle e_2,\ \sigma\rangle \qquad (\textsc{try}2)$$

$$\frac{\langle e_1,\ \sigma\rangle \rightarrow \langle e_1',\ \sigma'\rangle}{\langle \texttt{try}\ e_1\ \texttt{with}\ e_2,\ \sigma\rangle \rightarrow \langle \texttt{try}\ e_1'\ \texttt{with}\ e_2,\ \sigma'\rangle} \qquad (\textsc{try}3)$$

Regras de Tipos (71)

- `raise`, como antes, continua tendo seu tipo definido pelo contexto
- Note que a regra de tipo para `try` e_1 `with` e_2 requer que e_1 e e_2 tenham o mesmo tipo
- Nas regras a seguir será omitido o ambiente Δ

$$\Gamma \vdash \texttt{raise} : T \qquad (\textsc{Trs})$$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : T} \qquad (\textsc{Ttry})$$

2.4.3 Tratamento de exceções com passagem de valores

Tratamento de exceções com passagem de valores (72)

- 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 (73)

Sintaxe

$$
\begin{array}{rcl}
e & ::= & \ldots \\
 & & \texttt{raise}\ e \\
 & & \texttt{try}\ e_1\ \texttt{with}\ e_2
\end{array}
$$

Regras de tipo

$$\frac{\Gamma \vdash e_1 : \mathsf{int}}{\Gamma \vdash \texttt{raise}\ e_1 : T} \quad (\textsc{Trs-v})$$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : \mathsf{int} \to T}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : T} \quad (\textsc{Ttry-v})$$

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 extendida 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.

2.5 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 (74)

- 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 (75)

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

Subtipos - Motivação (76)

- um dos objetivos de um sistema de tipos é não permitir a avalição/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 (77)

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

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'} \quad \text{(T-App)}$$

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

Subtipos - Motivação (78)

- A função $fn\ r{:}\{x : \mathsf{int}\} \Rightarrow \sharp x\ r$ da expressão $(fn\ r{:}\{x : \mathsf{int}\} \Rightarrow \sharp x\ r)\ \ \{x = 0, y = 1\}$ 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 $\{x : \mathsf{int}, y; \mathsf{int}\}$ para uma função que espera argumento do tipo $\{x : \mathsf{int}\}$
- com subtipos, expressões como a do exemplo acima passam a ser bem tipadas

Subtipos (79)

- 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 $\{x : \mathsf{int}, y : \mathsf{int}\}$ pode ser usado em um contexto que espera registro do tipo $\{x : \mathsf{int}\}$, ou seja

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

A Relação de Subtipo (80)

- 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 propriedades da relação $<:$

$$S \ <: \ S \qquad \text{(S-Refl)}$$

$$\frac{S \ <: U \qquad U <: T}{S <: T} \qquad \text{(S-Trans)}$$

A

Subtipos e Registros (81)

$$\{lab_i : T_i^{i \in 1 \ldots n+k}\} \ \ <: \ \ \{lab_i : T_i^{i \in 1 \ldots n}\} \qquad \text{(S-RcdWidth)}$$

$$\frac{S_i <: T_i \quad \textit{para cada } i}{\{lab_i : S^{i\in 1\ldots n}\} \quad <: \quad \{lab_i : T^{i\in 1\ldots n}\}} \qquad \text{(S-RcdDepth)}$$

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

Subtipos (82)

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

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

- Note que essa é a única regra de tipo para expressões que é adicionada ao sistema de tipos
- Com subtipos é possível tipar expressões antes consideradas mal tipadas, como por exemplo

$$(fn\ r{:}\{x : \mathsf{int}\} \Rightarrow \sharp x\ r) \quad \{x = 0, y = 1\}$$

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

Subtipos e Funções (83)

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

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

Subtipos e Funções (84)

- contexto espera função do tipo $T_1 \to T_2$, 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 (85)

- para função do tipo $S_1 \to S_2$ ser usada no lugar de uma do tipo $T_1 \to T_2$
  - $S_2 <: T_2$, 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 (86)

A regra fica portanto:

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

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).

2.6 Orientação a Objetos

A referência para essa seção é o capítulo 18 do livro *Types and Programming Language* de Benjamin Pierce (MIT Press, 2002)

OO e Cálculo λ (87)

- 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 (88)

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 (89)

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

Objetos III (90)

$$
\begin{array}{l}
c \equiv \texttt{let}\ \ x = \mathsf{ref}\ 1\ \ \texttt{in} \\
\quad \{\texttt{get} = \ \ fn\ \_{:}\mathsf{unit} \Rightarrow\ \ !x, \\
\quad\ \ \texttt{inc} = \ \ fn\ \_{:}\mathsf{unit} \Rightarrow\ \ x := !x + 1\}
\end{array}
$$

$$
\triangleright\ c :\ \ \{\texttt{get} : \mathsf{unit} \to \mathsf{int}, \texttt{inc} : \mathsf{unit} \to \mathsf{unit}\}
$$

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

Objetos IV (91)

No que segue usaremos () no lugar da expressão `skip` do tipo unit e $e.lab$ no lugar de $\sharp\, lab\ e$

$$
\begin{array}{l}
c.\texttt{inc}\ () \\
\triangleright\ () :\ \ \mathsf{unit}
\end{array}
$$

$$
\begin{array}{l}
c.\texttt{get}\ () \\
\triangleright\ 2 : \mathsf{int}
\end{array}
$$

$$
\begin{array}{l}
c.\texttt{inc}\ ();\ \ c.\texttt{inc}\ ();\ \ c.\texttt{get}\ () \\
\triangleright\ 4 :\ \ \mathsf{int}
\end{array}
$$

Objetos V (92)

Vamos dar um nome ao tipo do registro:

$$Counter \equiv \{\texttt{get} : \mathsf{unit} \to \mathsf{int}, \texttt{inc} : \mathsf{unit} \to \mathsf{unit}\}$$

$$\texttt{inc3} \equiv fn\ c{:}\,Counter \Rightarrow (c.\texttt{inc}\ ();\ c.\texttt{inc}\ ();\ c.\texttt{inc}\ ())$$
$$\triangleright\ \texttt{inc3} : Counter \to \mathsf{unit}$$

$$\texttt{inc3}\ c;\ c.\texttt{get}\ ()$$
$$\triangleright\ 7 : \mathsf{int}$$

Geradores de objetos (93)

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

$$newCounter \equiv$$
$$fn\_ : \mathsf{unit} \Rightarrow$$
$$\texttt{let}\ x = \mathsf{ref}\ 1\ \texttt{in}$$
$$\{\texttt{get} = fn\ \_{:}\mathsf{unit} \Rightarrow\ !x,$$
$$\texttt{inc} = fn\ \_{:}\mathsf{unit} \Rightarrow\ x := !x + 1\}$$

$$\triangleright\ newCounter : \mathsf{unit} \to Counter$$

Geradores de objetos (94)

$\texttt{let}\ \ c = newCounter()\ \ \texttt{in}\ldots$

$$c.\texttt{inc}\ ()$$
$$\triangleright\ () :\ \mathsf{unit}$$

$$c.\texttt{get}\ ()$$
$$\triangleright\ 2 : \mathsf{int}$$

$$c.\texttt{inc}\ ();\ c.\texttt{inc}\ ();\ c.\texttt{get}\ ()$$
$$\triangleright\ 4 :\ \mathsf{int}$$

Subtipos I (95)

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

$$ResetCounter = \{\texttt{get} : \mathsf{unit} \to \mathsf{int}, \texttt{inc} : \mathsf{unit} \to \mathsf{unit},$$
$$\texttt{reset} : \mathsf{unit} \to \mathsf{unit}\}$$

$$newResetCounter =$$
$$fn\_ : \mathsf{unit} \Rightarrow \ \texttt{let}\ x = \mathsf{ref}\ 1\ \texttt{in}$$
$$\{\texttt{get} = fn\ \_{:}\mathsf{unit} \Rightarrow\ !x,$$
$$\texttt{inc} = fn\ \_{:}\mathsf{unit} \Rightarrow\ x := !x + 1$$
$$\texttt{reset} = fn\ \_{:}\mathsf{unit} \Rightarrow\ x := 1\}$$
$$\triangleright\ newResetCounter : \mathsf{unit} \to resetCounter$$

Subtipos II (96)

- 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*:

$$rc = \ newResetCounter\ ()$$
$$\triangleright\ rc : ResetCounter$$

$$\texttt{inc3}\ rc;\ rc.\texttt{reset}\ ();\ \texttt{inc3}\ rc;\ rc.\texttt{get}\ ()$$
$$\triangleright\ 4 : \mathsf{int}$$

Agrupando variáveis de instância I (97)

- até agora o estado dos objetos é constituido 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 (98)

$$c = \ \texttt{let}\ r = \{x = \mathsf{ref}\ 1\}\ \texttt{in}$$
$$\{\texttt{get} = fn\ \_ : \mathsf{unit}\ \Rightarrow !(r.x),$$
$$\texttt{inc} = fn\ \_ : \mathsf{unit}\ \Rightarrow\ r.x := !(r.x) + 1\}$$
$$\triangleright\ c : Counter$$

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

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

Classes simples I (99)

$$\begin{array}{l} counterClass = \\ \quad fn\ r : CounterRep\ \Rightarrow \\ \quad\quad \{\mathtt{get} : fn\ \_ : \mathsf{unit} \Rightarrow\ !(r.x), \\ \quad\quad\ \mathtt{inc} : fn\ \_ : \mathsf{unit} \Rightarrow\ r.x\ :=\ !(r.x) + 1\} \end{array}$$

$$\triangleright\ counterClass : CounterRep \to Counter$$

Criando objetos (100)

$$\begin{array}{l} newCounter = \\ \quad fn\_ : \mathsf{unit}\ \Rightarrow\ \ \mathtt{let}\ r = \{x = \mathsf{ref}\ 1\}\ \mathtt{in} \\ \quad\quad\quad\quad\quad\quad counterClass\ r \end{array}$$

$$newCounter : \mathsf{unit} \to Counter$$

Classes simples III (101)

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

$$\begin{array}{l} resetCounterClass = \\ \quad fn\ r : CounterRep\ \Rightarrow \\ \quad\ \mathtt{let\ super} =\ \ counterClass\ r\ \mathtt{in} \\ \quad\quad \{\mathtt{get} = \mathtt{super.get} \\ \quad\quad\ \mathtt{inc} = \mathtt{super.inc} \\ \quad\quad\ \mathtt{reset} = fn\ \_ : \mathsf{unit} \Rightarrow\ r.x\ :=\ 1\} \end{array}$$

$$\triangleright\ resetCounterClass : CounterRep \to ResetCounter$$

Criando objetos (102)

$$
\begin{array}{l}
newResetCounter = \\
\quad fn\ \_ : \mathsf{unit}\ \Rightarrow\ \ \texttt{let}\ r = \{x = \mathsf{ref}\ 1\}\ \ \texttt{in} \\
\qquad\qquad\qquad\qquad\ \ resetCounterClass\ r \\
\\
\triangleright\ newResetCounter : \mathsf{unit} \to ResetCounter
\end{array}
$$

Adicionando variáveis de instância (103)

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

Adicionando variáveis de instância (104)

$$
\begin{array}{l}
backupCounter = \{\texttt{get} : \mathsf{unit} \to \mathsf{int}, \texttt{inc} : \mathsf{unit} \to \mathsf{unit}, \\
\qquad\qquad \texttt{reset} : \mathsf{unit} \to \mathsf{unit}, \texttt{backup} : \mathsf{unit} \to \mathsf{unit}\} \\
\\
backupCounterRep = \{x : \mathsf{int\ ref}\ , b : \mathsf{int\ ref}\} \\
\\
backupCounterClass = fn\ r :\ \ backupCounterRep \Rightarrow \\
\qquad\qquad\qquad\qquad \texttt{let super} =\ \ resetCounterClass\ r\ \texttt{in} \\
\qquad\qquad\qquad\qquad\ \ \{\texttt{get} = \texttt{super.get} \\
\qquad\qquad\qquad\qquad\ \ \texttt{inc} = \texttt{super.inc} \\
\qquad\qquad\qquad\qquad\ \ \texttt{reset} = fn\ \_ : \mathsf{unit} \Rightarrow r.x := !(r.b) \\
\qquad\qquad\qquad\qquad\ \ \texttt{backup} = fn\ \_ : \mathsf{unit} \Rightarrow r.b := !(r.x)\} \\
\\
\triangleright\ backupCounterClass : backupCounterRep \to backupCounter
\end{array}
$$

Overriding e subtipos (105)

- 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 (106)

$$
\begin{array}{l}
newBackupCounter = \\
\quad fn\ \_ : \mathsf{unit}\ \Rightarrow\ \texttt{let}\ r = \{x = \mathsf{ref}\ 1, b = \mathsf{ref}\ 1\}\ \texttt{in} \\
\qquad\qquad\qquad\qquad backupCounterClass\ r
\end{array}
$$

$$\triangleright\ newBackupCounter : \mathsf{unit} \to backupCounter$$

Chamando métodos de superclasse (107)

- 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`

$$
\begin{array}{l}
funnyBackupCounterClass = fn\ r :\ backupCounterRep \Rightarrow \\
\qquad \texttt{let super} = \ backupCounterClass\ r\ \texttt{in} \\
\qquad\quad \{\texttt{get} = \texttt{super.get} \\
\qquad\quad\ \texttt{inc} = fn\ \_ : \mathsf{unit} \Rightarrow \texttt{super.backup}(); \texttt{super.inc}() \\
\qquad\quad\ \texttt{reset} = \texttt{super.reset} \\
\qquad\quad\ \texttt{backup} = \texttt{super.backup}\}
\end{array}
$$

$$\triangleright\ funnyBackupCounterClass : backupCounterRep \to backupCounter$$

2.7 Concorrência

Introdução I (108)

- Até aqui vimos semântica estática e dinâmica para programas sequenciais e sem interação
- Várias formas de especificar computação não sequencial e vários modelos para interação
- Linguagens *não sequenciais*:
  - Concorrentes
  - Paralelas
  - Distribuídas
- Várias tentativas de classificação
- Não há um consenso quanto a vocabulário

Introdução II (109)

- Cálculo lambda tipado simples (com recursão) é largamente aceito como sendo a linguagem núcleo para o estudo de diversas características de linguagens de programação seqüencias
- Captura a essência dessas linguagens
- Para linguagens não sequenciais não há um cálculo universalmente aceito que capture as características essencias de não sequencialidade e de interação
- Talvez porque não haja um consenso sobre quais são essas características

Introdução III (110)

- Calculus of Communicating Systems (CCS) de Robin Milner talvez seja um bom candidato
- Vários cálculos foram desenvolvidos tendo CCS como inspiração
- Cálculo π, por exemplo, é uma extensão de CCS (também de R. Milner) que permite representar mobilidade
- Cálculo Join, por exemplo, expressa distribuição
- etc, etc

Introdução IV (111)

- Aqui, vamos extender o cálculo lambda tipado (na verdade L1) com primitivas para
  - criação de threads em paralelo
  - comunicaçao assíncrona entre threads via memória compartilhada
  - sincronizaçao através de semáforos
- semântica operacional *small step*
- sistema de tipos

A Linguagem I (112)

L1 é extendida com operador binário de composição concorrente $\|$

$$\begin{array}{rcl} e & ::= & n \mid b \mid e_1 \ op \ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ & \mid & l\texttt{:=}\ e \mid \texttt{!}\ l \\ & \mid & \texttt{skip} \mid e_1\texttt{;}\ e_2 \\ & \mid & \texttt{while}\ e_1\ \texttt{do}\ e_2 \\ & \mid & e_1 \parallel e_2 \end{array}$$

Regras de Tipo (113)

- $e_1 \parallel e_2$ avaliação concorrente das expressões/threads e_1 e e_2
- Decisão sobre o projeto da linguagem: expressões/threads e_1 e e_2 são imperativas, ou seja, são avaliadas pelo seu efeito

$$\frac{\Gamma \vdash e_1 : \mathsf{unit} \qquad \Gamma \vdash e_2 : \mathsf{unit}}{\Gamma \vdash e_1 \| e_2 : \mathsf{unit}}$$

Semântica Operacional (114)

- Threads são executados de forma não determinística podendo intercalar passos da sua execução
- Passos dados pelas regras de computação são atômicos
- Comunicação entre os threads é feita através de memória compartilhada
- Regras da semântica operacional:

$$\frac{\langle e_1,\ \sigma\rangle \longrightarrow \langle e_1,\ \sigma'\rangle}{\langle e_1 \| e_2,\ \sigma\rangle \longrightarrow \langle e_1' \| e_2,\ \sigma'\rangle}$$

$$\frac{\langle e_2,\ \sigma\rangle \longrightarrow \langle e_2',\ \sigma'\rangle}{\langle e_1 \| e_2,\ \sigma\rangle \longrightarrow \langle e_1 \| e_2',\ \sigma'\rangle}$$

Exercício (115)

Construir todos os traces possíveis para a avaliação das seguintes configurações:

- $\langle l := 1 \parallel l := 2,\ \{l \mapsto 0\}\rangle$
- $\langle l := 1+!l \parallel l := 7+!l,\ \{l \mapsto 0\}\rangle$

Sincronização (116)

- Várias primitivas/construções para sincronização foram propostas
- Aqui veremos *semáforos binários*
  - $m, m_1, \ldots \in$ *conjunto de nomes de semáforos*
  - configurações da semântica operacinal tem a forma $\langle e,\ \sigma,\ M\rangle$ onde M mapeia nomes de semáforos para um valor booleano
  - sintaxe:

$$e \quad ::= \quad \ldots \texttt{lock}\ m \mid \texttt{unlock}\ m$$

Regras de Tipo e Semântica Operacional (117)

- Operações com semáforos são avaliadas pelo efeito
- Regras de tipo:

$$\Gamma \vdash \texttt{lock} : \mathsf{unit}$$

$$\Gamma \vdash \texttt{unlock} : \mathsf{unit}$$

- Semântica operacional:

$$\langle \texttt{lock}\ m, \sigma,\ M\rangle \longrightarrow \langle \texttt{skip}, \sigma,\ M[m \mapsto \texttt{true}]\rangle \ \textit{se}\ M(m) = \texttt{false}$$

$$\langle \texttt{unlock}\ m, \sigma,\ M\rangle \longrightarrow \langle \texttt{skip}, \sigma,\ M[m \mapsto \texttt{false}]\rangle$$

Exercício (118)

1. Construir todos os traces possíveis para a avaliação da expressão abaixo em uma configuração onde M mapeia m para falso e σ é tal que $\sigma(l) = 0$

$$\begin{array}{l}(\texttt{lock}\ m;\ l := 1+\ !l;\ \texttt{unlock}\ m)\\ \parallel\\ (\texttt{lock}\ m;\ l := 7+\ !l;\ \texttt{unlock}\ m)\end{array}$$

2. Mostre um trace que leva o programa abaixo a entrar em *deadlock*

$$
\begin{array}{l}
(\texttt{lock}\ m_1;\ \texttt{lock}\ m_2;\ l_1 := !l_2;\ \texttt{unlock}\ m_2;\ \texttt{unlock}\ m_1) \\
\quad \| \\
(\texttt{lock}\ m_2;\ \texttt{lock}\ m_1;\ l_2 := !l_1;\ \texttt{unlock}\ m_1;\ \texttt{unlock}\ m_2)
\end{array}
$$