- Programing Language + Security
  (theory)
- built-in security (Building Security in)

  - static analysis vs. run-time analysis

- type systems (type safety => security property (semantic property))
  syntaxic
- Semantics (Denotational)

  - Abstract interpretation

Robert Harper - Practical Foundations for Programming Languages
John Mitchell - Foundations for Programming Languages
Benjamin Pierce - Types and programing Languages

Induction and Inductive Definition

Judgment (J) — بیان یک گزاره یا مطلب از objها، خاصیتی را دارد. (نیاز به تعریف دارد)

- n nat
- $n = n_1 + n_2$ ⇐ judgment form
- a ast
- τ type
- $e : \tau$
- $e \Downarrow v$

judgment form ↓ $n = n_1 + n_2$

$3 = 2 + 1$ → instance of judgment form

اکثرا برای تعریف J از inference Rule ها استفاده می‌کنند.

Inference Rule

An inductive definition of a judgment form consists of a collection of Rules of the form

$$\frac{J_1 \; J_2 \; \ldots \; J_k}{J}$$

(top: مقدمه; bottom: نتیجه)

J_i : Premises

J : conclusion (consequent)

$k = 0 \Rightarrow$ axiom

$k \neq 0 \Rightarrow$ proper Rule

ex)

$$\frac{}{\text{zero nat}} \qquad \frac{a \text{ nat}}{\text{succ}(a) \text{ nat}}$$ ← inference Rule

the strongest judgment that is closed, or respects, the rules.

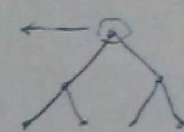
Closed: the rules are sufficient to show the validity of a judgment. that is, J holds if there is a way to obtain J using the given Rules

strongest: the rules are necessary in the

مثال دیگر) تعریف درخت

$$\frac{}{\text{empty tree}}$$

$$\Rightarrow \frac{\frac{}{\text{empty tree}} \quad \frac{}{\text{empty tree}}}{\text{node(empty; empty) tree}}$$



$$\frac{a_1 \text{ tree} \quad a_2 \text{ tree}}{\text{node}(a_1; a_2) \text{ tree}}$$

⇓

تعریف یک node (k) ها

$$\frac{k \quad k'}{\text{node}(k; k') \text{ tree}}$$

تعریف استقرایی برای درخت ⇒

⇒ مثل یک deducti... است.

a, b در natural number مانند ⇐ (a ≤ b nat)

مثال) روابط

$$\frac{}{\text{zero} \le \text{zero nat}} \qquad \frac{a \le b \text{ nat}}{\text{succ}(a) \le \text{succ}(b) \text{ nat}}$$

$$\frac{}{x \text{ exp}} \qquad \frac{e_1 \text{ exp} \quad e_2 \text{ exp}}{e_1 + e_2 \text{ exp}} \qquad \frac{e_1 \text{ exp} \quad e_2 \text{ exp}}{e_1 * e_2 \text{ exp}}$$

$$\frac{}{n \text{ exp}}$$

BNF ← تعریف e ::= n | x | e+e | e*e

اگر مدل این استنتاج ها یا اثبات ها متناهی نباشد. اثبات به صورت co-inductive تعریف می‌شود.

categorical

یعنی J ها بدون هیچ شرطی و قیدی تعریف شده‌اند.
(مثال: همه آنچه که تا الان تعریف کردیم)

categorical judgment J

برای اثبات اینکه J تعریف شده، خاصیتی را دارد، باید از استقرا استفاده کرد.

induction

(جلسه اول اسفند) Subject: فرمال نیمه‌رسمی
Date: ۹۳/۷/۶ (جلسه دوم)

Judgement

فرق Judgment و preposition این است که یک اثباتی دارد.

$\dfrac{J_1 \cdots J_n}{J}$ → Hilbert سیستم → Inference rule , rule skim

1) $\dfrac{}{\text{zero nat}}$ → Conclusion, Consequent, Axiom

2) $\dfrac{\text{a nat}}{\text{succ(a) nat}}$

قوانین ۱ و ۲، judgment a nat را تعریف می‌کند. تعریف natural برای همین است.

(nat: natural)

derivation:

$$\dfrac{\dfrac{\dfrac{}{\text{zero nat}}}{\text{succ (zero) nat}}}{\text{succ (succ(zero)) nat}}$$

→ یک الگوریتم که از طریق derivation مطرح شده است proof

تعریف iterated judgement ها:

(مثل تعریف nat، list را تعریف می‌کنیم)

1) $\dfrac{}{\text{nil list}}$

2) $\dfrac{\text{a nat} \quad \text{b list}}{\text{cons (a:b) list}}$

به این نوع تعریف، تعریف استقرائی گفته می‌شود. inductive

cons (succ (zero) : nil) list → <1>

می‌توان دو judgement همزمان تعریف کرد: (Simultaneous)

1) $\dfrac{}{\text{zero even}}$

2) $\dfrac{\text{a odd}}{\text{succ (a) even}}$ $\qquad$ $\dfrac{\text{a even}}{\text{succ (a) odd}}$

Subject
Date
(foundation، mitchell — کتاب)
صفحات ۸ از صفحه 27 به بعد

معروف‌ترین روش specification زبان‌ها، جبری روش است.

چون تعریف استقرائی است، پس روش اثبات هم استقرائی است.
(proof method is induction) (Inductive Definition)

Proof by Induction:

$\forall n.\ p(n) \qquad n \in \mathbb{N} = \{0, 1, 2, \dots\}$

$p(0),\ p(1),\ \dots$ اثبات‌ها به صورت تک تک عملی نیست.
proof
پس استقرا لازم می‌شود.

Natural Number Induction (Form 1):

$$\Big(p(0) \wedge \forall m \in \mathbb{N}.\ p(m) \rightarrow p(m+1)\Big) \Rightarrow \forall n \in \mathbb{N}.\ p(n)$$

$p(0)$ → base; $p(m)$ → hypothesis; $p(m+1)$ → step

Natural Number Induction (Form 2) (strong Induction) (Complete Ind.):

$$\Big(\forall m \in \mathbb{N}.\ \big(\forall i < m.\ p(i)\big) \rightarrow p(m)\Big) \Rightarrow \forall n \in \mathbb{N}.\ p(n)$$

فرم اول و دوم استقرا با یکدیگر معادل هستند.

Example –

$p(n) \overset{def}{=}$ if $n > 1$ then there exists prime numbers $p_1, \dots, p_k$ with $n = p_1 p_2 \cdots p_k$

$\forall n \in \mathbb{N}.\ p(n)$

if $A = \emptyset$ (مفهوم اساسی مهم)

$\forall x \in A.\ p(x) = true$

Subject: ...ن ...
Date: ~~۹۴/۷/۴~~

نطقش استفاده از نوع دوم استقراء بهتر است.

For every $m \in \mathbb{N}$, $\forall i < m . P(i) \to P(m)$

$Q(m)$

$m \leq 1 \to P(0), P(1)$

برقرار است چون مقدم برقرار نمی‌شود.

$P(\text{prime})$ از طرف اگر عدد اول باشد $Q(m)$ نیز بطور بدیهی برقرار است.

پس اعدادی که صفر یا یک یا اول نیستند بشکل زیر می‌توان نوشت:

$m = m_1 m_2 ; m_1, m_2 > 1$

$P(m_1) \Rightarrow m_1 = P_1 P_2 \cdots P_r$

$P(m_2) \Rightarrow m_2 = P_1' P_2' \cdots P_r'$

$\Rightarrow m = m_1 m_2 = P_1 \cdots P_r \cdot P_1' \cdots P_r'$

پس $Q(m)$ را اثبات کردیم. پس مسئله حل شد!

آیا برای اشیای غیر طبیعی می‌توان از استقرا استفاده کرد؟

$\forall a \in A . P(a)$

$A \neq \mathbb{N}$

می‌توان هر مجموعه‌ای باشد

اگر بتوان تابعی پیدا کرد که:

$f: A \to \mathbb{N}$

(یک به یک بودن و پوشا بودن مهم نیست)

$Q(n) \overset{\text{def}}{=} \forall a \in A . f(a) = n \to P(a)$

↓

خاصیت روی اعداد طبیعی

$\forall n \in \mathbb{N} . Q(n)$

که این را اثبات کنیم (با یکی از نوع‌های استقرا)

پس در واقع $\forall a \in A . P(a)$ اثبات خواهد شد.

۳/

Subject: فرمال پیشرفته
Date:

مثال:

$height$: Binary Trees $\rightarrow \mathbb{N}$

$Q(n) \stackrel{def}{=} \forall t \in \text{Trees}_{\text{Binary}} .\ height(t) = n \rightarrow leaves(t) \leq nodes(t) + 1$

$\forall n \in \mathbb{N} .\ Q(n)$

↓

این اثبات از طریق استقرای ریاضی قابل انجام است.

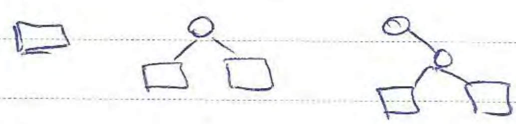
I. ابتدا باید تابع مناسب، احکام درخت‌ها باشیم به اعداد طبیعی تبدیل شوند.

II. اثبات: $\forall n \in \mathbb{N} .\ Q(n)$

انتخاب تابع مناسب به این دلیل است که بتوانیم II را اثبات کرد.

(جلسه سوم، ۹۳/۷/۸)

درخت دودویی:



ارتفاع 0 1 2

درخت Empty درختی است که یک برگ دارد، هر دو ارتفاع صفر دارند.

$height$: binary trees^{B} $\rightarrow \mathbb{N}$

چند حکم اشاره شده، انتخاب تابع برای درخت، اهمیت بسزایی دارد.

$P(t) \stackrel{def}{=}$ tree t has at most one more leaf than internal nodes.

می‌خواهیم این حکم را برای تمام درخت‌های دودویی اثبات کنیم:

$Q(n) \stackrel{def}{=} \forall t \in B .\ height(t) = n \rightarrow P(t)$

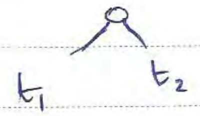
$\forall n \in \mathbb{N} .\ Q(n)$

Subject زبان شرحی
Date ۹۳/۷/۸

$Q(0)\ \forall t \in B.\ height(t) = 0 \rightarrow p(t)$

برای درخت Empty و درخت‌هایی که تک عضو دارند این برقرار است.

برای $m>0$ فرض می‌کنیم ثابت شده: وقتی ارتفاع کمتر از m است پس حتماً ⇐ Subtree خواهیم داشت P



$height(t_1),\ height(t_2) < m \rightarrow P(t_1),\ P(t_2)$

برقرار است.

$Leaves(t) = Leaves(t_1) + Leaves(t_2) \leq nodes(t_1) + 1 + nodes(t_2) + 1$

(the part $nodes(t_1) + 1 + nodes(t_2)$ is bracketed and labelled $node(t)$)

$= node(t) + 1$

پس خاصیت اثبات شد.
(با استفاده از استقرای قوی)

Induction on expressions and proofs:

$e ::= 0 \mid 1 \mid v \mid e + e \mid e \times e$

(arrows: "::=" is defined to be; "v" → Variables)

$P(e)$ اگر بخواهیم خاصیتی را برای عبارتی اثبات کنیم:

$Q(n) \overset{def}{=} \forall t \in trees.\ (height(t) = n \wedge t = parse(e)) \rightarrow P(e)$
$\forall e \in expressions.$

$\forall n \in N.\ Q(n)$

بنا به فرض

اگر عبارت قابل اثبات شود، خاصیت P برای همه آن‌ها برقرار خواهد بود.

برای 0 و 1 و v می‌توان مستقیماً اثبات کرد. پس می‌توانیم بجای آن عبارت‌ها ... تبدیل و ...
اثبات کردن آن‌ها را می‌توان مستقیم با اثبات استقرایی دانست.

Subject فرمال سیستمز
Date ۹۴/۷/۸

Structural Induction (Form 1):

To prove that p(e) is true for every expression e generated by some grammar, it is sufficient to prove p(e) for every atomic expressions, and, for any compound expression e with immediate subexpressions $e_1, \dots, e_k$, prove that if $p(e_i)$ for $i = 1, 2, \dots, k$ then p(e).

In our example: Base Case: p(0), p(1), p(v)

Induction step: $p(e_1) \wedge p(e_2) \longrightarrow p(e_1 + e_2)$

$p(e_1) \wedge p(e_2) \longrightarrow p(e_1 * e_2)$

مثال این جلسه، معادل است با مثال اسلایدها.

Example -

$p(e) \stackrel{def}{=}$ for any list $v_0, \dots, v_n$ of variables containing all the variables in e, there is a polynomial $c\, v_0^k v_1^k v_2^k \cdots v_n^k$ such that for all natural number values of $v_0, \dots, v_n$ greater than zero, the value of e is less than the value of the polynomial

P(0), P(1), P(v_i)

$0 < V_0 \dots V_n$, $1 < 2V_0 \dots V_n$, $V_i < 2V_0 \dots V_n$

Subject: فرمال سمنتیک

Date: ۹۳/۷/۸

برای Induction Step ها باید نشان دهیم:

$$p(e) \wedge p(e') \rightarrow p(e+e')$$

$$\left.\begin{array}{l} e < c\, v_0^{k} \cdots v_n^{k} \\ e' < c'\, v_0^{k'} \cdots v_n^{k'} \end{array}\right\} \Rightarrow e+e' < (c+c')\, v_0^{\max(k,k')} \cdots v_n^{\max(k,k')} \Rightarrow p(e+e')$$

برقرار است.

$$p(e) \wedge p(e') \rightarrow p(e \times e')$$

$$\Rightarrow e \times e' < (c \times c')\, v_0^{\max(k,k')} \cdots v_n^{\max(k,k')} \Rightarrow p(e \times e')$$

برقرار است.

نوعی استقرای طبیعی حالت خاصی از استقرای structural است؛

$$n ::= 0 \mid succ(n)$$

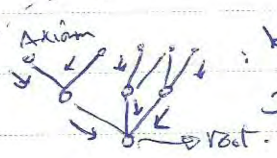
$\forall n \in N .\ p(n)$ ← در این جا استقرای طبیعی تعریف شده زیر است

Base $p(0)$

Step $\forall n.\ p(n) \rightarrow p(succ(n))$

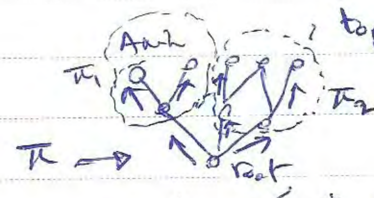
Induction on Proofs:

اثبات: forward یا bottom-up:



با چیزهای درست اثبات.

اثبات Backward یا top-down:



اگر اثبات را π در نظر بگیریم (که همان یک درخت است)، به درخت اثبات هم اثباتی است که می‌توان زیر درخت‌ها

یک زیردرخت را با استقرا اثبات کرد.

Subject

Date

Structural Induction on Proofs:

To prove that $P(\pi)$ is true for every proof π in some proof system, it is sufficient to show that P holds for every axiom of the proof system and then, assume that P holds for proofs, $\pi_1, \dots, \pi_k$, Prove that $P(\pi)$ for any proof that ends by extending one or more of the proof, $\pi_1, \dots, \pi_k$ with one inference rule.

Rule Induction:

$$\frac{J_1 \quad \cdots \quad J_k}{J} \qquad \frac{}{J_0}$$

این روابط، کلی‌ترین فرم تعریف‌ها است.

(مثل استقرا در اعداد طبیعی، ساختاری و ...)

$$P(J)$$

$$P(J_1) \wedge P(J_2) \wedge \cdots \wedge P(J_k) \rightarrow P(J)$$

مثال:

$$\frac{}{zero = zero \quad nat}$$

$$\frac{a = b \quad nat}{succ(a) = succ(b) \quad nat}$$

$P(zero = zero \quad nat)$

$P(a = b \; nat) \rightarrow P(succ(a) = succ(b) \; nat)$

$P(x = y \quad nat)$

تمرین: نمونه‌های مشابه بنویسید:

$$\frac{}{zero\ nat}$$

$$\frac{a\ nat}{succ(a)\ nat}$$

$P(a\ nat)$ اگر بخواهیم اثبات کنیم طبیعی است:

$P(zero\ nat)$ باید نشان دهیم

$P(a\ nat) \rightarrow P(succ(a)\ nat)$

$$\frac{}{empty\ tree}$$

$$\frac{a_1\ tree \quad a_2\ tree}{node(a_1; a_2)\ tree}$$

$P(a\ tree)$ برای اینکه چیزی را اثبات کنیم

$P(empty\ tree)$ باید نشان دهیم که:

$P(a_1\ tree) \wedge P(a_2\ tree) \rightarrow P(node(a_1; a_2)\ tree)$

$$\frac{}{nil\ list}$$

$$\frac{a\ nat \quad b\ list}{cons(a; b)\ list}$$

$P(a\ list)$ اگر بخواهیم خاصیتی را نشان دهیم:

$P(nil\ list)$ باید اثبات کنیم:

$a\ nat \wedge P(b\ list) \rightarrow P(cons(a; b)\ list)$

Subject Advanced PL
Date 93.7.8

$e ::= 0 \mid 1 \mid v \mid e+e \mid e*e$

Proof System

$$\frac{}{e \leq e} \text{(ref)}$$

$$\frac{}{0 \leq e} \text{(0≤)}$$

$$\frac{e \leq e' \quad e' \leq e''}{e \leq e''} \text{(trans)}$$

$$\frac{e_1 \leq e_2 \quad e_3 \leq e_4}{e_1 + e_3 \leq e_2 + e_4} \text{(+ mon)}$$

$$\frac{e_1 \leq e_2 \quad e_3 \leq e_4}{e_1 * e_3 \leq e_2 * e_4} \text{(* mon)}$$

$p(\pi) \overset{\text{def}}{=}$ if π is a proof of $e \leq e'$, then for all values of variables, the value of e is $\leq$ the value of e'.

(+ mon) $\quad p(e_1 \leq e_2) \wedge p(e_3 \leq e_4) \rightarrow p(e_1 + e_3 \leq e_2 + e_4)$

$n_1 \leq n_2 \wedge n_3 \leq n_4 \rightarrow n_1 + n_3 \leq n_2 + n_4$

(given)

Subject

Date

$\frac{}{zero\ nat}$ $\qquad$ $\frac{}{zero = zero\ nat}$ ⊛

$\frac{a\ nat}{succ(a)\ nat}$ $\qquad$ $\frac{a = b\ nat}{succ(a) = succ(b)\ nat}$

Theorem - If a nat, then a=a nat. (prove)

$P(a\ nat) \overset{def}{=} a\ nat \longrightarrow a = a\ nat$

$P(zero\ nat) \overset{def}{=} zero\ nat \longrightarrow zero = zero\ nat$ ⊛ طبق

$P(a\ nat) \Rightarrow P(succ(a)\ nat)$

[P holds of a nat]

$P(succ(a)\ nat) \overset{def}{=} succ(a)\ nat \rightarrow succ(a) = succ(a)\ nat$

Take apart — Taking apart — Building up

a nat $\xrightarrow{I.H.}$ a = a nat

I.H.: Induction Hypothesis

Theorem

تعریف function:

$\frac{b\ nat}{sum(zero; b; b)}$ ⊛

$\frac{sum(a; b; c)}{sum(succ(a); b; succ(c))}$

قاعده rule:

$\frac{zero\ nat}{succ(zero)\ nat}$

$\frac{}{sum(zero; succ(zero); succ(zero))}$

$sum(succ(zero); succ(zero); succ(succ(zero)))$

negin

I 11

Subject

Date

Theorem – For every a nat and b nat, there exists a unique c nat such that sum(a;b;c).

1) Existence: If a nat and b nat, then there exists c nat such that sum(a;b;c).

2) Uniqueness: If a nat and b nat, c nat, c' nat, sum(a;b;c), and sum(a;b;c'), then c = c' nat.

Existence –

$P(a\ \text{nat}) \overset{\text{def}}{=} (a\ \text{nat} \wedge b\ \text{nat}) \rightarrow \exists c\ \text{nat}.\ \text{sum}(a;b;c)$.

$P(\text{zero nat}) \overset{\text{def}}{=} (\text{zero nat} \wedge b\ \text{nat}) \rightarrow \exists c\ \text{nat}.\ \text{sum}(\text{zero};b;c)$.

(*) ↓ def

sum(zero; b; b)

by choosing c to be b.

$P(a\ \text{nat}) \Rightarrow P(\text{succ}(a)\ \text{nat})$

$P(\text{succ}(a)\ \text{nat}) \overset{\text{def}}{=} (\text{succ}(a)\ \text{nat} \wedge b\ \text{nat}) \rightarrow \exists c\ \text{nat}.\ \text{sum}(\text{succ}(a);b;c)$

Taking apart ↓

$a\ \text{nat} \wedge b\ \text{nat} \xrightarrow{\text{I.H.}} \exists c'\ \text{nat}.\ \text{sum}(a;b;c')$

Building up ↑

by choosing c to be succ(c')

Uniqueness ...

negin

Subject

Date

Uniqueness: $Q(\mathrm{sum}(a;b;c)) \stackrel{\text{def}}{=} \mathrm{sum}(a;b;c) \wedge \mathrm{sum}(a;b;c') \rightarrow c = c'\ \mathrm{nat}$

(Exercise!)

Exercise – Consider the following rules which define the height of a binary tree as the judgement $\mathrm{hgt}(a;b)$.

$$\frac{}{\mathrm{hgt}(\mathrm{empty};\mathrm{zero})}$$

$$\frac{\mathrm{hgt}(a_1;b_1) \quad \mathrm{hgt}(a_2;b_2) \quad \max(b_1;b_2;b)}{\mathrm{hgt}(\mathrm{node}(a_1;a_2);\mathrm{succ}(b))}$$

Prove by tree induction that the judgement hgt has the mode $(\forall, \exists)$, with inputs binary trees and outputs being natural numbers.

For all trees a, exists b

Hypothetical Judgements:

Categorical Judgement

Hypothetical

Derivability:

For a given set of rules R, we define derivability, written

judgment

$J_1, J_2, \dots, J_n \vdash_R K$, where each J_i and K are categorical, to mean that we may derive K from expansion $R[J_1, \dots, J_n]$ of the rules R with the additional axioms

$$\frac{}{J_1} \quad \frac{}{J_2} \quad \cdots \quad \frac{}{J_n}$$

می‌توانیم با استفاده از J_1 تا J_n (اصول موضوعه) و Rule های R به K برسیم.
یعنی اگر J_1 تا J_n را به Rule های R اضافه کنیم، به K برسیم.

$$\begin{array}{c} \overline{J_1} \quad \overline{J_2} \quad \cdots \quad \overline{J_n} \\ \vdots \qquad\qquad \vdots \\ \hline K \end{array} \equiv \frac{J_1 \cdots J_n}{K} \text{ is derivable}$$

$\Gamma \vdash_R K$ — K is derived from $R[\Gamma]$

Example

$$a \text{ nat} \vdash_{\text{nat rules}} succ(succ(a)) \text{ nat}$$

$$\cfrac{\cfrac{\cfrac{}{a \text{ nat}}}{succ(a) \text{ nat}}}{succ(succ(a)) \text{ nat}}$$

برخی خواص دستگاه‌ها:

§ Uniformity:

این خواص را meta-rule هم می‌گویند.

If $\Gamma \vdash_R J$, then $\Gamma \vdash_{R \cup R'} J$

* سیستم های Non-Monotonic، یعنی سیستم‌هایی که با اضافه شدن Rule [illegible] هیچ [illegible] نیستند.

Structural

Reflexivity For any J and Γ, $\Gamma, J \vdash_R J$

Weakening $\Gamma \vdash_R J \Rightarrow \Gamma, K \vdash_R J$

(در Non-monotonic یعنی Weakening برقرار نیست)

Exchange $\Gamma_1, J_1, J_2, \Gamma_2 \vdash_R J \Rightarrow \Gamma_1, J_2, J_1, \Gamma_2 \vdash_R J$

Contraction $\Gamma, J, J \vdash_R K \Rightarrow \Gamma, J \vdash_R K$

(در منطق linear، همچنین Contraction برقرار نیست، یعنی Axiom ها consume (مصرف) می‌شوند)

Transitivity $\Gamma, K \vdash_R J \wedge \Gamma \vdash_R K \Rightarrow \Gamma \vdash_R J$

Admissibility:

$\Gamma \models_R J$ is weaker than $\Gamma \vdash_R J$

یعنی J از R قابل استنتاج است هرگاه با فرض Γ و قوانین R بتوان به دست آورد.

اگر Γ، derivable باشد، J هم derivable است. ← هرچند derivable نیست Γ را فرض کنند.

Derivability و Admissibility فرق دارند.

اگر Derivability برقرار باشد Admissibility برقرار است

$\Gamma \models_R J$ $\quad$ $\Gamma \vdash_R J$

ولی اگر Admissibility برقرار باشد لزوماً Derivability برقرار نیست

$$\frac{\begin{array}{c} R \\ \Gamma \\ \vdots \end{array}}{J} \qquad \frac{\begin{array}{c} \Gamma \\ \vdots \end{array}}{J}$$

مجموعه اثر قوانین R را ... : R

Subject

Date

$\vdash_{nat}$ succ(a) nat $\models$ a nat

$$\frac{}{\text{zero nat}} \qquad \frac{a\ \text{nat}}{\text{succ(a) nat}} \qquad \frac{}{\text{succ(junk) nat}}$$

(nat_junk)

succ(a) nat $\not\models_{nat\text{-}junk}$ a nat

succ(junk) nat $\not\models_{nat}$ junk nat

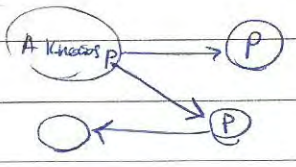
succ(junk) nat $\models_{nat}$ junk nat

Theorem - $\Gamma \vdash_R J \Rightarrow \Gamma \models_R J$.

$q \neq r$

q: A knows p

r: A knows that A knows p



Simple λ-Calculus :

1) $\dfrac{}{x:\tau \vdash x:\tau}$

2) $\dfrac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$

3) $\dfrac{\Gamma, x:\tau \vdash e : \tau'}{\Gamma \vdash \lambda x:\tau.e' : \tau \to \tau'}$

negin

(Gertsen) (MONAD)

Subject فرمال پیشرفته ⊢ : V-dash خوانده می‌شود

Date ۹۴/۷/۲۰

$$\Gamma \vdash_R J \Rightarrow$$ از روی Γ و با استفاده از R، J قابل derive است.

$$\Gamma \models_R J \Rightarrow$$ اگر Γ و R برقرار باشند J نیز قابل حصول است.

داشتیم:

$$\frac{J_1 \; J_2 \cdots J_n}{J}$$

حال:

Γ: global hypotheses

Γ_i: local hypotheses

$$\frac{\Gamma\Gamma_1 \vdash J_1 \quad \Gamma\Gamma_2 \vdash J_2 \cdots \Gamma\Gamma_k \vdash J_k}{\Gamma \vdash J}$$

مثال:

$$\frac{}{\Gamma, x:T \vdash x:T}$$

تعریف استقرایی $\Gamma \vdash e:T$

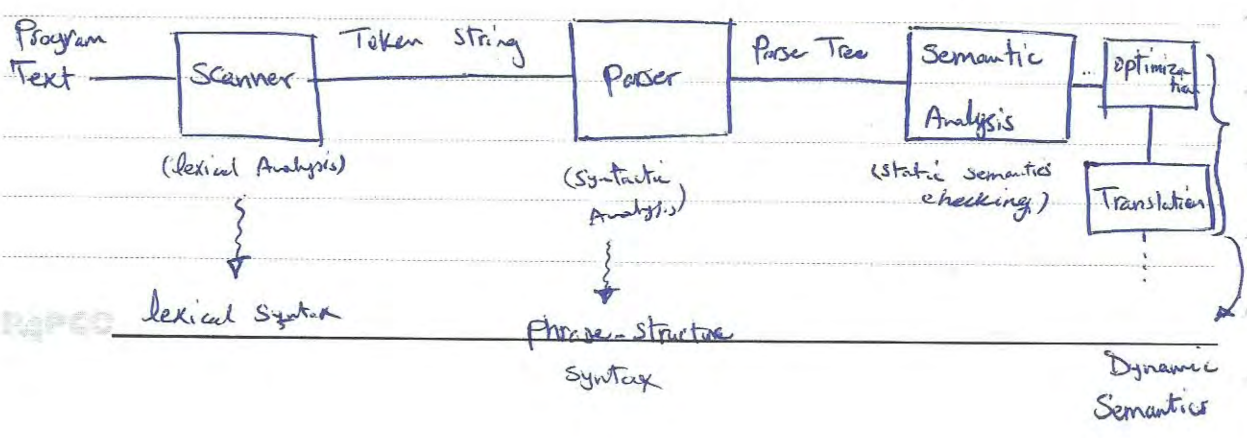
$$\frac{\Gamma, x:T_1 \vdash e:T_2}{\Gamma \vdash \lambda x:T_1 . e : T_1 \to T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_1 \to T_2}{\Gamma \vdash e_2 e_1 : T_2}$$

اگر بخواهیم خاصیتی را در این قسمت ثابت کنیم: (اثبات خاصیت استقرا)

$$P(\Gamma\Gamma_1 \vdash J_1) \wedge P(\Gamma\Gamma_2 \vdash J_2) \wedge \cdots \wedge P(\Gamma\Gamma_k \vdash J_k) \Rightarrow P(\Gamma \vdash J)$$

مرحله‌ای از زبان برنامه‌نویسی:



Subject
Date

Example -

	Abstract Syntax ↓ متن اصلی (token)	Concrete Syntax ↓ (physical) text متن خود برنامه
Type $\tau ::=$	num	num
	str	str
Exp $e ::=$	x	x
	num[n]	n
	str[s]	"s"
	plus[e_1; e_2]	e_1+e_2
	times(e_1; e_2)	$e_1 * e_2$
s to token متن ←	len(e)	|e|
را است کردن	let(e_1; $x.e_2$)	let x be e_1 in e_2
	cat(e_1; e_2)	e_1 ^ e_2

Type System:

$\Gamma \vdash e : \tau$ ⟶ Judgment

rules

1) $$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

2) $$\frac{}{\Gamma \vdash str[s] : str}$$

3) $$\frac{}{\Gamma \vdash num[n] : num}$$

binder مفهوم x، e_1 را در e_2 bind می‌کند.
متغیر x، bound است در e_1.

... در Context ها

4) $$\frac{\Gamma \vdash e_1 : num \quad \Gamma \vdash e_2 : num}{\Gamma \vdash plus(e_1; e_2) : num}$$

5) $$\frac{\Gamma \vdash e_1 : num \quad \Gamma \vdash e_2 : num}{\Gamma \vdash times(e_1; e_2) : num}$$

6) $$\frac{\Gamma \vdash e_1 : str \quad \Gamma \vdash e_2 : str}{\Gamma \vdash cat(e_1; e_2) : str}$$

7) $$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash let(e_1; x.e_2) : \tau_2}$$

هر متغیر در Typing Context فقط یکبار ظاهر می‌شود.

Subject: مدل سیستم

Date: ۹۳/۷/۲۲

مثال :

$let(num[1]; x. (let(num[2]; y. plus(x;y))))$

$let(num[1]; x. x)$

مثال

$$\frac{\vdash num[1] : num \qquad x:num \vdash x:num}{\vdash let(num[1]; x.x) : num}$$

7 قدم

Forward

① Backward

بدون هیچ Context ای

Theorem – (Unicity of Typing)

For every typing Context Γ and expressions e, there exist at most one τ such that $\Gamma \vdash e:\tau$.

روش اثبات، استقرا روی Rule است.

$$p(\Gamma \vdash e:\tau) \overset{def}{=} \forall \Gamma . \forall e . \exists^{\leq 1} \tau . \Gamma \vdash e:\tau .$$

اثبات برای Axiom ها:

$$P(\Gamma, x:\tau \vdash x:\tau) = \forall \Gamma . \forall x . \exists^{\leq 1} \tau . \Gamma, x:\tau \vdash x:\tau$$

$\vdots$

به طور بدیهی اثبات است.

چون در Γ ، x دیگری نداریم.

الف- روی Rule ها استقرا میزنیم یا روی Structure (بعدا)

Subject
Date

$P(e) \stackrel{def}{=} \forall \Gamma.\ \exists^{\leq 1} \tau.\ \Gamma \vdash e : \tau$

$P(num[n]) \stackrel{def}{=} \forall \Gamma.\ \exists^{\leq 1} \tau.\ \Gamma \vdash num[n] : \tau$

$\frac{}{num[n]\ exp} \qquad \frac{}{str(s)\ exp}$

$P(e_1) \wedge P(e_2) \Rightarrow P(plus(e_1; e_2))$

$\frac{e_1\ exp \quad e_2\ exp}{plus(e_1; e_2)\ exp}$

$\forall \Gamma.\ \exists^{\leq 1} \tau.\ plus(e_1; e_2) : \tau$

either $\tau = num$ or there is no τ.

Theorem - (Inversion for typing):

Suppose that $\Gamma \vdash e : \tau$. If $e = plus(e_1; e_2)$ then $\tau = num$, $\Gamma \vdash e_1 : num$ and $\Gamma \vdash e_2 : num$

Theorem - (weakening): ⟶ Structural

If $\Gamma \vdash e' : \tau'$, then $\Gamma, x : \tau \vdash e' : \tau'$ for any $x \notin dom(\Gamma)$ and any type τ.

$P(\Gamma \vdash e' : \tau') \stackrel{def}{=} \Gamma \vdash e' : \tau' \rightarrow \Gamma, x : \tau \vdash e' : \tau'$.

Subject فرمال سیستم‌ها

Date ۹۴/۷/۲۲

$P(\Gamma \vdash e_1 : \tau_1) \wedge P(\Gamma, x:\tau_1 \vdash e_2 : \tau_2) \Rightarrow P(\Gamma \vdash let(e_1; x.e_2) : \tau_2)$

$P(\Gamma \vdash let(e_1 : x.e_2) : \tau_2) \overset{def}{=} \Gamma \vdash let(e_1 : x.e_2) : \tau_2 \rightarrow \Gamma, y:\tau' \vdash let(e_1; x.e_2) : \tau_2$

Taking apart

$\Gamma \vdash e_1 : \tau_1$ for some τ_1 — I.H. → $\Gamma, y:\tau' \vdash e_1 : \tau_1$

$\Gamma, x:\tau_1 \vdash e_2 : \tau_2$ — I.H. → $\Gamma, y:\tau', x:\tau_1 \vdash e_2 : \tau_2$

Building up

Theorem - (substitution):

If $\Gamma, x:\tau \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau'$

$P(\Gamma, x:\tau \vdash e' : \tau') \overset{def}{=} \Gamma, x:\tau \vdash e' : \tau' \wedge \Gamma \vdash e : \tau \rightarrow \Gamma \vdash [e/x]e' : \tau'$

$P(\Gamma, x:\tau \vdash e_1 : \tau_1) \wedge P(\Gamma, x:\tau_2^{y:\tau_1} \vdash e_2 : \tau_2) \Rightarrow$

$P(\Gamma, x:\tau \vdash let(e_1; y.e_2) : \tau_2)$

Taking apart , Building up

$P(\Gamma, x:\tau \vdash let(e_1; y.e_2) : \tau_2) \overset{def}{=} \Gamma, x:\tau \vdash let(e_1; y.e_2) : \tau_2 \wedge \Gamma \vdash e : \tau \rightarrow$

$\Gamma \vdash [e/x]\, let(e_1; y.e_2) : \tau_2$

Taking apart ↓ Hyp.

$\Gamma, x:\tau \vdash e_1 : \tau_1$ $\qquad \Gamma, x:\tau, y:\tau_1 \vdash e_2 : \tau_2$ $\qquad \Gamma \vdash e : \tau$

for some τ_1

↓ I.H. $\qquad$ ↓ I.H., Weakening

$\Gamma \vdash [e/x]\, e_1 : \tau_1$ $\qquad \Gamma, y:\tau_1 \vdash [e/x]\, e_2 : \tau_2$

↓ Building up

$\Gamma \vdash let([e/x]e_1; y.[e/x]e_2) : \tau_2$

با فرض‌های مربوط که x و y مستقل از هم هستند. $\Rightarrow \Gamma \vdash [e/x]\, let(e_1; y.e_2) : \tau_2$

باید به ازای rule ها ثابت شود.

اگر به یک term یا phrase از یک type system داده شده، type نسبت بدهیم، به آن term،

typable یا well-typed گفته می‌شود و در غیر این صورت ill-typed گفته می‌شود.

مثلا plus(num[1]; str[Alice])، ill-typed است.

طراحی یک مجموعه rule های typing، این اطمینان را می‌دهد به هر برنامه که خواهد داد اگر typable باشند.

Program → [type checker] → Well-typed

در زبان‌های برنامه‌نویسی، اگر عبارتی را اجرا کنیم و بیش از آن قابل محاسبه نباشد، آن عبارت normal form است.

مثلاً برای num[5]+str[Alice] در هنگام اجرای آن به مشکل برمی‌خوریم، لازم است این محاسبه get stuck evaluation است.

وظیفه type checker این است که عبارت قبل از اجرا statically تشخیص دهد به stuck نمی‌رسد.

می‌خواهیم از ترکیب static semantics و Dynamic Semantics، Type Safety را به دست بیاوریم.

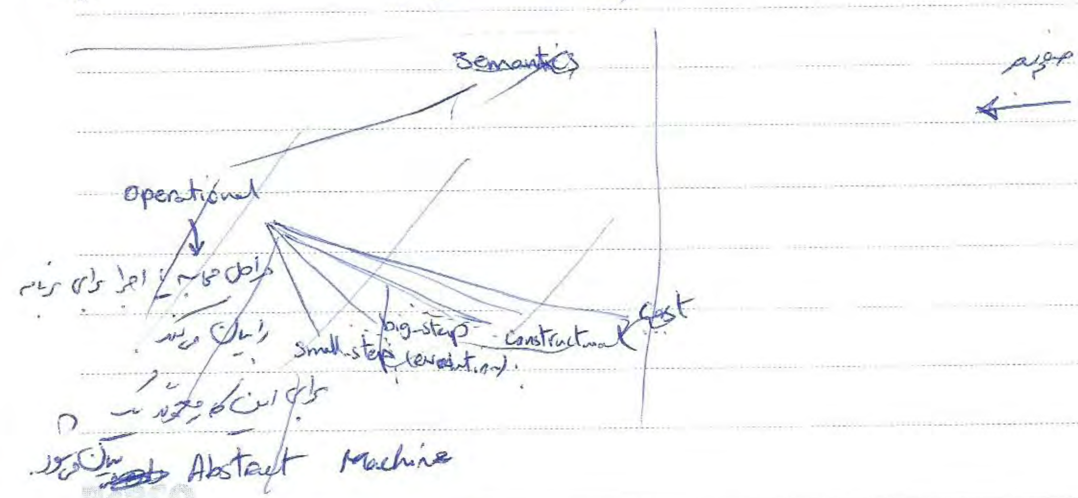
یعنی اگر یک برنامه well-typed شد، در زمان اجرا، اتفاق بدی رخ نخواهد داد.

Dynamic Semantics (Semantics):

زبان شامل سه جزء اصلی است: Syntax، Semantics و Pragmatics. در زبان‌ها مهم‌ترین کدام است؟

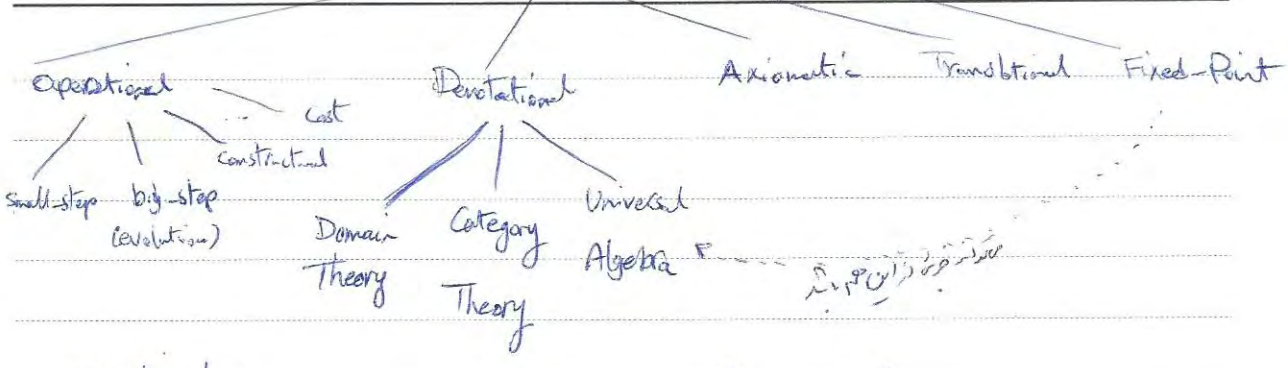
برای یک زبان Semantics (معنا شناخت) تعریف می‌شود. اما برنامه‌ها دارای meaning (معنا) هستند.

(language Semantics vs. Program Meaning)



Subject
Date

Semantics



operational:

(Abstract Machine

Denotational:

Axiomatic:

Fixed-point: Recursion

Structural Semantics:

$$\frac{}{num[n]\ \ Val}$$

$$\frac{}{str[s]\ \ Val}$$

judgment

(one-step evaluation)

$$\frac{n_1 + n_2 = n \quad nat}{plus(num[n_1]; num[n_2]) \longrightarrow num[n]} \quad (*)$$

Subject اصول پیشرفته

Date ۹۴/۷/۲۷

$$\frac{e_1 \rightarrow e_1'}{plus(e_1; e_2) \rightarrow plus(e_1'; e_2)}$$

$$\frac{e_1 \ val \qquad e_2 \rightarrow e_2'}{plus(e_1; e_2) \rightarrow plus(e_1; e_2')}$$

$$\frac{s_1 \hat{} s_2 = s \ str}{cat(str[s_1]; str[s_2]) \rightarrow str[s]} \quad (*)$$

$$\frac{e_1 \rightarrow e_1'}{cat(e_1; e_2) \rightarrow cat(e_1'; e_2)}$$

$$\frac{e_1 \ val \qquad e_2 \rightarrow e_2'}{cat(e_1; e_2) \rightarrow cat(e_1; e_2')}$$

$$\frac{e_1 \ val}{let(e_1; x.e_2) \rightarrow [e_1/x]e_2} \quad (*)$$

$$\frac{e_1 \rightarrow e_1'}{let(e_1; x.e_2) \rightarrow let(e_1'; x.e_2)}$$

کمتر محاسبه می‌شود، بعد استفاده کن

eager semantics ← همین اول که می‌رسد محاسبه را انجام بده یعنی اول مقدارش را حساب کن و بعد جایگذاری کن

lazy semantics ← اول e_1 را داریم در جای x جایگذاری کن بعد هرجا لازم شد حساب می‌کنیم

به تعداد x هایی که داریم باید محاسبه شود.

★ برای هر زبان مشخص semantics و given است.

Subject

Date ۹۳/۷/۲۷

- rule هایی که علامت (*) دارند، instruction transition نامیده می‌شود. (کار اصلی که برنامه انجام می‌دهد است)

- بقیه rule ها، search transition نامیده می‌شود. می‌گردند ببینند کدام rule اجرا شود.

✓ eliminatory form are inversed to introductory forms.

دو form در rule ها داریم ← introduction: بیان می‌کند که چه نوعی است؛ num[n] nat

← elimination: از type ها تعریف می‌کند که چگونه از آن استفاده شود.

(از دو type عملگر دیگر را می‌سازیم) (؟)

plus در num، num

آزمایش‌ها elimintory form همیشه با search rule ها شروع می‌شود.

□ : n

مثال:

let(plus(num[1]; num[2]); x . plus(plus(x; num[3]); num[4]))

$$\frac{1+2=3 \ \text{nat}}{\text{plus}(\text{num}[1];\text{num}[2]) \longrightarrow \text{num}[3]}$$

□ → let(num[3]; x . plus(plus(x; num(3]); num(4])) ⟶ plus(plus(num[3]; num(3)) num(4))

کاری که انجام دادیم برای اینکه ... تعریف‌ها ... است!

... هر مرحله ... می‌شود.

⟶ plus(num[6]; num[4]) ⟶ num[10]

Subject
Date

Theorem – (Determinacy): If $e \to e'$ and $e \to e''$, then e' is e''.

هیچ فرقی هم ندارد، حتی syntactically

(یعنی یک عبارت در هر گام فقط به یک عبارت دیگر ارزیابی می‌شود. (پس برای هر عبارت تنها یک عبارت نهایی ارزیابی خواهد شد)

$$P(e \to e') \overset{\text{def}}{=} (e \to e' \wedge e \to e'') \Rightarrow e' = e''$$

for all e'' ($\forall$)

از نظر درست $e' = e''$ ، یعنی از نظر Syntax این طور داشته باشد.

و در اینجا $e' = e''$ یعنی خودشان دقیقاً یکی است! حواستان باشد!

Evaluation Semantics

(Big-step)

$$\frac{}{num[n] \Downarrow num[n]}$$

ارزیابی می‌شود (در تعداد مراحل)

$$\frac{}{str[s] \Downarrow str[s]}$$

$$\frac{e_1 \Downarrow num[n_1] \quad e_2 \Downarrow num[n_2] \quad n_1 + n_2 = n \ nat}{plus(e_1; e_2) \Downarrow num[n]}$$

$$\frac{e_1 \Downarrow str[s_1] \quad e_2 \Downarrow str[s_2] \quad s_1 \hat{} s_2 = s \ str}{cat(e_1; e_2) \Downarrow str[s]}$$

$$\frac{[e_1/x]e_2 \Downarrow v}{let(e_1; x.e_2) \Downarrow v}$$

Definition – multi-step transition relation $\xrightarrow{*}$ is defined to be the reflexive transitive closure of small-step transition $\to$.

($\to$ ارتباط دو تا زوج expression است)

Subject

Date

$e \to e'$ ⟹ $e \to^* e'$ (تعدی)

$e \to^* e$ (reflexive)

$e \to^* e'$, $e' \to^* e''$ ⟹ $e \to^* e''$ (transitive)

Theorem _ for all closed expressions e, $e \to^* v$ iff $e \Downarrow v$.

(Structural Semantics) (value)

این theorem برای Semantics ساختاری برقرار است.

(Exercise)

اثبات این قضیه در کتاب Harper موجود است.

Subject درس زبان‌های برنامه‌سازی (؟)
Date ۹۳/۷/۲۹

Contextual semantics:

در زبان‌های دیگر برای بیان Semantics از این روش استفاده می‌شود.

در هر نوبت یک hole مشخص می‌شود که کدام قسمت محاسبه شود.

instruction step (instruction transition)
همه انتقال‌های اصلی (rule ها)

locating next instruction (evaluation context)
↓
بقیه rule ها هستند که محل انتقال بعدی را مشخص می‌کنند

$e_1 \leadsto e_2$

سه rule اصلی محاسبه (transition)

$$\frac{m+n=p \quad \text{nat}}{\text{plus}(\text{num}[m];\text{num}[n]) \leadsto \text{num}[p]}$$

$$\frac{s \hat{} t = u \quad \text{str}}{\text{cat}(\text{str}[s];\text{str}[t]) \leadsto \text{str}[u]}$$

$$\frac{e_1 \quad \text{val}}{\text{let}(e_1; x.e_2) \leadsto [e_1/x]e_2}$$

$e_1 \leadsto e_2$

redex — قسمت سمت چپ انتقال
reduce — (instruction transition دارد)
contraction — قسمت راست انتقال

Subject

Date

می‌خواهد evaluation Context را تعریف کنید :

$\mathcal{E}$ ectxt

$$\frac{}{\circ \text{ ectxt}}$$

hole ← $\circ$

$$\frac{\mathcal{E}_1 \text{ ectxt}}{\text{plus}(\mathcal{E}_1; e_2) \text{ ectxt}}$$

← کل عبارت حفره برای ما مشخص شده در $\mathcal{E}_1$ است.

$$\frac{e_1 \text{ val} \quad \mathcal{E}_2 \text{ ectxt}}{\text{plus}(e_1; \mathcal{E}_2) \text{ ectxt}}$$

$$\frac{\mathcal{E}_1 \text{ ectxt}}{\text{cat}(\mathcal{E}_1; e_2) \text{ ectxt}}$$

$$\frac{e_1 \text{ val} \quad \mathcal{E}_2 \text{ ectxt}}{\text{cat}(e_1; \mathcal{E}_2) \text{ ectxt}}$$

$$\frac{\mathcal{E}_1 \text{ ectxt}}{\text{let}(\mathcal{E}_1 ; x.e_2) \text{ ectxt}}$$

$e' = \mathcal{E}\{e\}$

تعریف.

The expression e' is the result of filling the hole in $\mathcal{E}$, with e.

$$\frac{}{e = \circ\{e\}}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\text{plus}(e_1; e_2) = \text{plus}(\mathcal{E}_1; e_2)\{e\}}$$

← آن حفره موجود در پیش $\mathcal{E}_1$ و e_2 ، e را بگذاریم، می‌شود e_1 و e_2 می‌شود.

Subject فرمال سیرنیه

Date ۹۳/۷/۲۹

$$\frac{e_1 \ \mathsf{val} \qquad e_2 = \mathcal{E}_2\{e\}}{\mathrm{plus}(e_1; e_2) = \mathrm{plus}(e_1; \mathcal{E}_2)\{e\}}$$

⋮

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\mathrm{let}(e_1; x.e_2) = \mathrm{let}(\mathcal{E}_1; x.e_2)\{e\}}$$

① Dynamic Semantics:

$$\frac{e = \mathcal{E}\{e_0\} \qquad e_0 \rightsquigarrow e_0' \qquad e' = \mathcal{E}\{e_0'\}}{e \mapsto_c e'}$$

که اگر در $\mathcal{E}$ و e_0 و e درست باشد (e نحوه قرار دادن e_0 در $\mathcal{E}$ باشد)

تنها rule، dynamic semantics خواهد شد.

evaluation context

$\rightarrow_c$ برای کاهش برای نامهای مختلف

می توان به شکل کوتاه‌تر نوشت:

$$\frac{e_0 \rightsquigarrow e_0'}{\mathcal{E}\{e_0\} \rightarrow_c \mathcal{E}\{e_0'\}}$$

Theorem – $e \rightarrow e'$ iff $e \rightarrow_c e'$.

(بنابراین هر انتقالی در term طبق structural semantics انتقالی در contextual semantics مجاز است.)

(با توجه به معادل بودن contextual s. و structural s.)

Example –

$$\mathcal{E} = \mathrm{let}(\circ; x.\mathrm{plus}(\mathrm{plus}(x; \mathrm{num}[3]); \mathrm{num}[4]))$$

(*) $\mathrm{let}(\mathrm{plus}(\mathrm{num}[1]; \mathrm{num}[2]); x.\mathrm{plus}(\mathrm{plus} \ldots)) = \mathcal{E}\{\mathrm{plus}(\mathrm{num}[1]; \mathrm{num}[2])\}$

Subject
Date

$\varepsilon\{num[3]\}$

(*) $plus(num[1]; num[2]) \rightsquigarrow num[3])$ $let(num[3]; x.plus(\cdots)) = ?$

$let(plus(num[1]; num[2]); x.plus(\cdots)) \rightarrow_c let(num[3]; x.plus(\cdots))$

$\varepsilon\{e_0\}$ $\varepsilon\{e_0'\}$

مثال:

$e = plus(\underbrace{let(num[1]; x.x)}_{o(hole)}, num[2])$

ε

$let(num[1]; x.x) \rightsquigarrow num[1]$

$\varepsilon\{let(num[1]; x.x)\} \rightarrow_c \varepsilon\{num[1]\}$

$e \rightarrow_c plus(num[1], num[2])$

اثبات آنی: (استقرا) Theorem

$p(e = \varepsilon\{e_0\}) \overset{def}{=} (e = \varepsilon\{e_0\} \wedge e_0 \rightsquigarrow e_0' \wedge e' = \varepsilon\{e_0'\}) \Rightarrow e \rightarrow e'$

base step:

$p(e = o\{e_0\}) \overset{def}{=} (e = o\{e_0\} \wedge e_0 \rightsquigarrow e_0' \wedge e_0' = o\{e_0'\}) \Rightarrow e \rightarrow e'$

$\Rightarrow e \rightsquigarrow e' \Rightarrow e \rightarrow e'$ ✓

$p(plus(e_1; e_2) = plus(\varepsilon_1; e_2)\{e_0\}) \overset{def}{=} (plus(e_1; e_2) = plus(\varepsilon_1; e_2)\{e_0\} \wedge$

$e_0 \rightsquigarrow e_0' \wedge e' = plus(\varepsilon_1; e_2)\{e_0'\}) \Rightarrow$

Taking apart

$plus(e_1; e_2) \rightarrow e'$

$e_1 = \varepsilon\{e_0\} \wedge e_0 \rightsquigarrow e_0' \wedge \underbrace{e_1' = \varepsilon_1\{e_0'\}}_{assume}$

IH.

$\Rightarrow e_1 \rightarrow e_1' \Rightarrow plus(e_1; e_2) \rightarrow \underbrace{plus(e_1'; e_2)}_{e'}$

Subject: زبان پیشرفته
Date: ۹۳/۷/۲۹

باید برای تعمیم state ها نیز اثبات کرد. ضمناً طرف دیگر قضیه را نیز اثبات کرد.

✓ به عنوان تمرین، به Middleweight Java (MJ) مراجعه کنید و Semantics آن را بخوانید.

در مورد یک زبان مشخص است operational Semantics با Denotational Semantics معادل باشد
(اگر درست انجام شده باشد)

Type Safety:

- Static Semantics
- Dynamic Semantics

در کتاب pierce سعی شده که درباره مفهوم safety صحبت کرده است.

بعضی از زبان‌ها، safe نیستند و بعضی هستند. آنهایی که هستند، یا با روش static checking یا با روش
← قبل از اجرا
dynamic checking استفاده می‌کنند. C ← unsafe و ML ← safety نظیر
← در طول اجرا

زبانی که از Abstraction های خودش محافظت می‌کند، safe است. هر زبان از دید طراح تعریف برای
نوع داده‌ها operation های مجاز است. الگوریتم زبان حقیقی جز آنکه طراح مشخص کرده، انجام شود، safety از بین رفته است.

Type safety به طرزی بین Static Semantics است و Dynamic Semantics.

Subject

Date ۹۳/۸/۴

Behavior برنامه مهم‌ترین خواصی است که از Dynamic Semantics می‌توان به دست آورد. (اجرا)

Type Safety شامل دو ویژگی است:

Preservation: if $e:\tau$ and $e \to e'$, then $e':\tau$.

type برنامه در اثر اجرا حفظ می‌شود. (type دار بودن عبارت حفظ می‌شود)

Progress: if $e:\tau$ then either e val or there exists e' such that $e \to e'$.

یعنی هر دو این ویژگی با هم بیانگر این است که یک عبارت type دار بعد از اجرا نمی‌شود که به value نرسد. (stuck نمی‌شود)

این مفهوم type safety است.

(زبانی که در فصل ۴ معرفی شد)

Type safety of L{num str} states that it will never arise that a number is to be added to a string.

type system جلوی این اتفاق را می‌گیرد و اصلاً در تعریف قوانین type نمی‌شود که بخواهد یک عدد با یک رشته جمع شود.

اصطلاحات مطرح شده:

- safe
- type-safe
- strongly-typed

→ مرتبط به هم (هم‌معنی)

- get stuck
- well-typed (typable)
- ill-typed

→ مرتبط به هم (نه هم‌معنی)

Subject
Date

$\forall^c, \wedge, \forall^c$

Theorem (Preservation) – If $e:\tau$ and $e \to e'$, then $e':\tau$.

$$P(e \to e') \stackrel{def}{=} (e:\tau \wedge e \to e') \Rightarrow e':\tau$$

باید روی ساختار یکی از حکم‌ها استقرا بزنیم. D

Theorem (Progress) – If $e:\tau$, then either e val or there exists e' such that $e \to e'$.

Lemma (Canonical form) – If e val and $e:\tau$, then:

1) If $\tau = num$, then $e = num[n]$ for some number n.

2) If $\tau = str$, then $e = str[s]$ for some string s.

$$P(e:\tau) \stackrel{def}{=} e:\tau \Rightarrow e \text{ val} \vee \exists e'.\ e \to e'.$$

$$\frac{e_1 \text{ num} \qquad e_2 \text{ num}}{plus(e_1; e_2) \text{ num}}$$ شکل ارزش:

$$P(plus(e_1; e_2):num) \stackrel{def}{=} plus(e_1; e_2):num \Rightarrow plus(e_1; e_2) \text{ val} \vee \exists e'.\ plus(e_1; e_2) \to e'$$

Taking apart

$e_1:num \quad e_2:num$

I.H. → e_2 val $\vee \exists e_2'.\ e_2 \to e_2'$ (II)

I.H. → e_1 val $\vee \exists e_1'.\ e_1 \to e_1'$ (I)

Building up

(I), (II) → $plus(e_1; e_2) \to plus(e_1; e_2')$

(I), (III) → lemma → $e_1 = num[n_1]$, $e_2 = num[n_2]$ → $plus(e_1; e_2) \to num[n_1 + n_2]$; e

Subject

Date ۹۳/۸/۴

- Runtime-Error

فرض کنیم به زبان $L\{num\ str\}$ ، $div(e_1;e_2)$ را اضافه کنیم:

$$\frac{e_1\ num \qquad e_2\ num}{div(e_1;e_2) : num}$$

$\Rightarrow div(num[3]; num[0]) : num$

$div(num[3]; num[0])$ ✗ → get stuck   اینجا نمی‌دانیم

دو راه برای این مشکل است:

1. Enhance the type system so that no well-typed program may divide by zero.
2. Add dynamic checks so that division by zero signals an error as the out come of evaluation.

$$\frac{e_1\ val}{div(e_1; num[0])\ err}$$

$$\frac{e_1\ err}{plus(e_1;e_2)\ err}$$

$$\frac{e_1\ val \qquad e_2\ err}{plus(e_1;e_2)\ err}$$

Subject [illegible]
Date ۹۳/۸/۶

Theorem (progress with error) if $e : \tau$, then either e err or e val or $\exists e'. e \to e'$.

راه دیگر :

این‌ها به rule ها اضافه شود تا درست به progress برسیم .

$$\frac{}{error : \tau}$$

$$\frac{e_1 \; val}{div(e_1; num[0]) \to error}$$

$$\frac{}{plus(error; e_2) \to error}$$

$$\frac{e_1 \; val}{plus(e_1; error) \to error}$$

A well-typed program never goes wrong.

(؟) ⟶ stuck

[illegible] type-safety ، اگر semantics طوری طراحی شده است که بر اساس آن ، برنامه stuck می‌شود ، به

که برنامه ویژگی خوب مدنظر ما هست .

trapped ← در اجرا خطا رخ می‌دهد و مدیریت می‌شود .

خطاها

untrapped ← بعد از وقوع اشتباه خطاها ، برنامه متوقف می‌شود .

[illegible]

Subject
Date

مثال: برنامه C :

```c
int i;
int k=2;
int p[10];
  :
i = *(p+k);
```

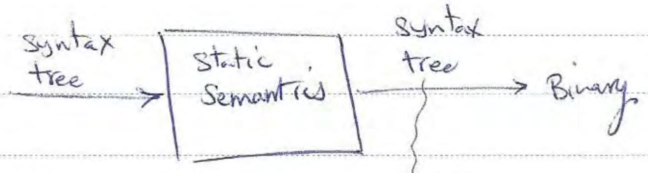
← چون آدرس به int اشاره می‌کند.

چون rule ای برای این جور برنامه، از نظر rule ها این برنامه stuck است، در حالی که اجرا انجام می‌شود ولی احتمالاً غلط و نادرست محاسبه انجام می‌شود. با این حال هم از نظر ما خطا است.

یا ریختن مقداری high در low ریختن، stuck نشده، ولی از نظر ما مشکل دارد.

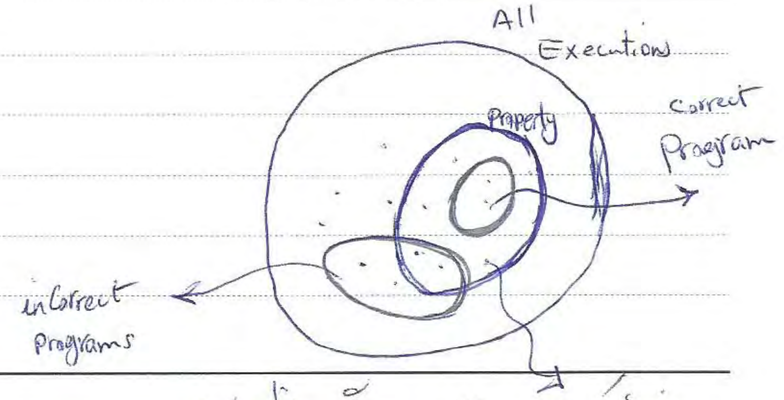
در اینجا نتوانستیم ویژگی مورد نظر را بیان کنیم. پس type safety این به معنای safety نیست.

درجه سختی و تعریف دقیق type safety دارد (بسته به مثل تعریف طراحی زبان دارد.)



ویژگی‌های خوب مد نظر ما را دارد. (چیزهایی که مورد نظر ما است)

درجه ممکن است برخی Conservative هستند؛
بعضی برنامه‌های سالم که قربانی می‌شوند.
Sound هستند؛ یعنی راه حل Sound and complete نیست.



برنامه‌هایی که در همه اجراهای آنها دارای ویژگی مورد نظر ما است.

Subject (mitchell, pierce کتاب مطالعه

Date ۹۳/۸/۶

Property ممکن است لزوماً با get stuck نشدن برآورده شود، هرچه اگر نشود، type safety،

property را برآورده می‌کند.

استفاده از Type System یکی از روش‌های Static Analysis است. روش‌های دیگر Abstract Interpretation

و Data Flow Analysis نیز قابل استفاده است.

البته استفاده از Type System صرفاً برای safety نیست، بلکه برای Documentation، Implementation و ...

نیز هست. البته به نظر، اصل کار Type System، همین safety است.

λ – calculus :

کوچک‌ترین زبان برنامه‌نویسی با قابلیت کامل (در حد تورینگ ماشین) است.

untyped:

lambda term

Syntax ⇒ $t ::= x \mid \lambda x.t \mid tt$

tt → Application

λx.t → Abstraction

این تعریف tiny، قدرت محاسباتی را می‌دهد. (همان قدرت ماشین تورینگ را دارد)

Turing، Church و Markov پیشنهاد کردند "ماشین"‌ای که — church در سال ۱۹۳۰، نوع تایپ را مطرح کرد.

مسئله Hilbert را به طور انتزاعی مطرح کند. هر کدام از این ماشین‌ها با یکدیگر معادل هستند،

بالاترین قدرت محاسباتی را دارند.

Subject

Date

Church-Turing Thesis : هر مسئله‌ای که در ماشین تورینگ و یا lambda حل شود. (این تز اثبات نشده ولی نقضی هم نشده است

(اثبات نداشته، شبیه مسئله‌ای که با ماشین تورینگ یا λ-calculus قابل حل نیست، با ماشین دیگری حل شود.)

✓ همه زبان‌های Functional تعمیمی از lambda هستند و در واقع پیاده‌سازی آن با کمی تعمیم.

untyped:

Semantics:

$$(\lambda x. t)\, t' \longrightarrow [x \mapsto t']\, t \qquad (\beta\text{-reduction})$$

یا

$$[t'/x]\, t$$

t' به جای x در t قرار می‌گیرد.

$$r u v = (r u) v$$

Application، از چپ انجام می‌شود.

Abstraction، بزرگترین جای ممکن به سمت راست گسترش می‌یابد.

$tru = \lambda t. \lambda f. t$

$fls = \lambda t. \lambda f. f$

$tru\ v\ w \longrightarrow \lambda f. v \longrightarrow v$

$fls\ v\ w \longrightarrow (\lambda f. f)\ w \longrightarrow w$

$test = \lambda b. \lambda m. \lambda n.\ b\ m\ n$

$test\ b\ v\ w \longrightarrow^{*} b\ v\ w$

$test\ tru\ v\ w \longrightarrow^{*} tru\ v\ w \longrightarrow v$

$test\ fls\ v\ w \longrightarrow^{*} fls\ v\ w \longrightarrow w$

$and = \lambda b_1. \lambda b_2.\ b_1\ b_2\ fls$

$or = \lambda b_1. \lambda b_2.\ b_1\ tru\ b_2$

Subject
Date

$(\lambda x.t)t' \Longleftarrow$ reducible expression $\leftarrow$ redex

Normal-Form $\leftarrow$ redex ست

$\lambda x.xy$ ، xx متن

Value $\leftarrow$ ست $\rightarrow$ Value

Abstraction ، lambda ، Value ست Value ، {Normal-Form از

$\lambda x.t$ val

diversion :

$(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow \dots$

$A = \{x \mid x \notin x\}$

$A \notin A \Longleftrightarrow A \in A$

untyped lambda calculus

$\neg = \lambda b.b$ fls tru

$r = \lambda x.\neg xx$

$rr = \neg(\lambda x.\neg xx)(\lambda x.\neg xx) = \neg rr$

Type theory Type System

Subject (The Theory of objects — ...)

Date

Featherweight Java

Model Language ← از یک زبان بزرگ شامل همه‌ی ویژگی‌ها، feature های کوچکتر مورد بررسی قرار می‌گیرد. مثل

Core Language ← که به عنوان هسته اصلی زبان‌های مختلف مطرح هستند. مثل λ، π، ...

$\lambda \rightarrow$ functional core

$\pi \rightarrow$ concurrency core

object-calculus → object-oriented core

معمولاً از زبان‌های هسته برای تعریف Model زبان استفاده می‌شود.

λ-calculus (انواع) → Typed / Untyped

نکته: Recursion چیزی است که هم در زبان‌ها (با حلقه) و هم در λ-calculus مطرح است.

در Untyped λ-calculus اینطور است که:

تابع partial و هم داشتن حلقه‌ی بی‌نهایت برای Turing-Complete بودن الزامی است.

λ-calculus خاصیت Normalization property ندارد، یعنی هر ترمی به normal term نمی‌رسد.

در λ اینطور نیست. Strong Normalization: یعنی هر برنامه به Value می‌رسد.

↓

نداشتن این ویژگی لازمه‌ی Turing Complete بودن است.

Omega

$\Omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x) \rightarrow \ldots$

divergence

مثال) تابع فاکتوریل

$f(0)=1$

$f(n) = n f(n')$ $n=n'+1$

این تابع، تعریف استقرایی است. چون uniquely مقدار می‌دهد.

$$f: n \mapsto \begin{cases} 1 & n=0 \\ n f(n') & n=n'+1 \end{cases} \quad \text{یا} \quad \begin{cases} 1 & n=0 \\ n f(n-1) & o.w. \end{cases}$$

functional

$F(f) = f'$

← این جا f نابرابر نیست

F یک functional است که یک تابع می‌گیرد و یک تابع برمی‌گرداند.

$$f': n \mapsto \begin{cases} 1 & n=0 \\ n f(n-1) & o.w. \end{cases}$$

در f' از f استفاده شده، نه f'. پس بازگشتی نیست.

مثال: $f: n \mapsto n+2$

$F(f) = ?$

$f'(0)=1$

$f'(1) = 2 = 1 * f(0)$

$f'(2) = 6$

$f'(3) = 12$

$f'(4) = 20$

$f'(5) = 30, \dots$

سوال اینجاست که کدام تابع است که اگر به F بدهیم، خودش را تحویل دهد؟

این همان $f = f'$ است. یعنی در مثال فوق چه تابع فاکتوریل خواهد بود.

$f: A \to A$

$f(x) = x$ x is a fixed point

مثلاً در $f(x) = x^2$، $x=0, 1, -1$ fixed point ها ی تابع x^2 است.

اگر راهی داشته باشیم که بتوانیم Fixed Point توابع را حساب کنیم، پس می‌توان در λ-calculus، Recursion داریم!

$$Y = \lambda f.(\lambda x. f(x\, x))(\lambda x. f(x\, x))$$

$$YF = (\lambda x. F(x\, x))(\lambda x. F(x\, x)) = F\big((\lambda x. F(x\, x))(\lambda x. F(x\, x))\big) = F(YF)$$

پس: YF is the fixed point of F

$fact = Y(\lambda f. \lambda n. \text{ if } n=0 \text{ then } 1 \text{ else } n * f(n-1))$

$fact\ 5 = 120$

پس fixed point فوق، همان تابع فاکتوریل است.

Call - by - value :

Value
$v ::= \lambda x.t$

$$\frac{}{(\lambda x.t)v \longrightarrow [v/x]t}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2}$$

$$\frac{t_2 \longrightarrow t_2'}{v t_2 \longrightarrow v t_2'}$$

تا اینجا Untyped λ - calculus بود.

Simple Typed λ - calculus :

$t ::= x \mid \lambda x : \tau_1 . t \mid tt$

$\tau ::= b \mid \tau \rightarrow \tau$

nat مثل base types

برای ساده شدن:

$\tau ::= nat \mid \tau \rightarrow \tau$

پس value ها را نیز با اعداد حساب کند و ترکیب.

مثلاً می‌توانیم داشته باشیم که:

$nat \rightarrow (nat \rightarrow nat)$ ← مثال: تابع جمع

$(nat \rightarrow nat) \rightarrow (nat \rightarrow nat)$ ← مثال: تابع F بگیرد تابع دیگر بسازد

$(nat \rightarrow nat) \rightarrow nat$ ← مثال: تابع MAX

→ نیز یک type است.

type System :

(Syntax - directed)

به ازای هر Syntax یک rule داریم در type systems

1) $$\frac{}{\Gamma, x:\tau_1 \vdash x : \tau_1}$$

2) $$\frac{\Gamma, x:\tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x:\tau_1 . t : \tau_1 \rightarrow \tau_2}$$

3) $$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t' : \tau_2}{\Gamma \vdash tt' : \tau_2}$$

مثال:

$\lambda x : nat . \lambda y : nat . x + y$

~~$x:nat, y:nat \vdash x + y : nat$~~

$$\frac{\dfrac{\dfrac{x{:}nat,\ y{:}nat \vdash x{:}nat \qquad x{:}nat,\ y{:}nat \vdash y{:}nat}{x{:}nat,\ y{:}nat \vdash x+y : nat}\ (\text{plus})}{x{:}nat \vdash \lambda y{:}nat.\ x+y : nat \rightarrow nat}\ (2)}{\vdash \lambda x{:}nat.\ \lambda y{:}nat.\ x+y : nat \rightarrow (nat \rightarrow nat)}\ (2)$$

(→ L{num, str} قاعده‌ی مربوط rule)

↓ بدون Context

طبق type system ای که مطرح شد نمی‌توان $(\lambda x:\tau .\ x\ x)(\lambda x:\tau .\ x\ x)$ را type پیدا کرد. چون طبق

سومین rule که در Application ، function که apply می‌شود باید به شکل $\tau_1 \longrightarrow \tau_2$ باشد و این نقض است،

پس حلقه‌ای وجود ندارد که بتوان در به دست آوردن type آن استفاده کرد.

(PCF: programming computable Functions)

پس زبان بدون type دارای Fixed point خوبی است و Semantics خاص خودش را دارد و از خوبیت خودش به دست می‌آید.

(Fixed point Combinator با نام معروف است)

((hungry function) recursive type را داشته باشد، $\mu X.\tau \rightarrow X$ را داشته باشد، type یعنی به (می‌تواند))

مطالعه: (فصل آخر کتاب bible (PCF)

(کتاب Pierce

فصل مربوطه به ... در Harper اخر فصل ۱۳۰

بحث Type Safety در Typed lambda calculus یک بحث مهمی است.

(در این جا مطرح می‌شود که عبارت Ω را بدون type اعلام می‌کند، و آن تاکید شده به عبارتی ...)

The Curry-Howard Correspondence:

(منطق ریاضی کلاسیک / منطق شهودگرایی)

Constructive or intuitionistic logic (شهودگرایی)

$p \vee \neg p \leftrightarrow T$ (Excluded Middle) → در منطق عادی، این p یا $\neg p$ درست است.

$\neg p \overset{def}{=} p \rightarrow \bot$

$p \vee q \overset{def}{=} (p \rightarrow \bot) \rightarrow q$ — اثبات همواره درست است.

اما در منطق شهودی شهودی که یا p یا $\neg p$ را اثبات کن و پس آن همواره درست است، و گرنه غلط است.

Curry-Howard ⇒ (نام)
Propositions are types
proofs are program

$p \rightarrow q$: اگر یک proof از p می‌دهی، به تو proof از q می‌دهم.

$p \rightarrow p \qquad (\lambda x:p.x) : p \rightarrow p$ (↳ proof)

$$\frac{x:p \vdash x:p}{\vdash \lambda x:p.x : p \rightarrow p}$$

$p \to q$ $(\lambda x : p . y) : p \to q$

$(p \to p) \to p$ → از نظر منطقی درست نیست

و از طرفی هم نمی‌توان proof ای برایش نوشت:

$\lambda x : p \to p . \; y : (p \to p) \to p$

Propositional Logic ≡ Simply-typed λ-calculus

First order Logic ≡ typed λ-calculus with dependent typing

که سوری داریم quantifier

second order Logic ≡ Universal Typing

که روی propositionها، یعنی quantifier روی propositionها داریم

⋮

(از photo نکات این جزوه نیست)

pfennig: استاد دانشگاه CMU

(هم در این زمینه کتاب دارد)

مثال: x داده‌ای که از type address است، یا از type mail address است یا email address. یعنی

داده از type sum، mail address یا email address است. (گسترش)

Extensions:

$\mathcal{A}$ base types

$A \in \mathcal{A}$

$\lambda x : A . x : A \to A$

'→' : type constructor

'→' is right associative

$A \to B \to C \equiv A \to (B \to C)$

$\lambda f : A \to A . \lambda x : A . f(f(x)) : (A \to A) \to (A \to A)$

Subject:
Year. Month. Date. ( )

The Unit Type:

زبان‌های pure functional، زبان‌هایی هستند که side effect ندارند.

(term) $t ::= \ldots$ → به مجموعه term ها
$\quad$ unit → اضافه شد

(value) $v ::= \ldots$
$\quad$ unit → اضافه می‌شود

(Type) $T ::= \ldots$
$\quad$ Unit → اضافه می‌شود.

Rule زیر اضافه می‌شود:

$$\frac{}{\Gamma \vdash \text{unit} : \text{Unit}}$$

New derived forms:

$$t_1 ; t_2 \stackrel{\text{def}}{=} (\lambda x : \text{Unit}.\, t_2)\ t_1$$

$$x \notin FV(t_2)$$

طبق Semantics، ابتدا t_1 evaluate می‌شود سپس apply می‌شود یعنی

t_2 evaluate می‌شود. مستقل از نتیجه t_1. چون abstraction یک Value است.

یعنی دقیقاً sequence $t_1 ; t_2$ انجام می‌شود. (side effect ندارد)

Theorem (Sequencing is a derived form):

Write λ^{E} (E for external language) for the simply typed λ-calculus with the Unit type, the sequencing construct and

Subject:

Year. Month. Date. ( )

the rules E_SEQ, E_SEQNEXT, and T_SEQ.

$$\text{(E\_SEQ)}\ \frac{t_1 \longrightarrow t_1'}{t_1; t_2 \longrightarrow t_1'; t_2}$$

$$\text{(E\_SEQNEXT)}\ \frac{}{unit; t_2 \longrightarrow t_2}$$

$$\text{(T\_SEQ)}\ \frac{\Gamma \vdash t_1 : Unit \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2}$$

and λ^I (I for internal language) for the simply typed λ-calculus with Unit only. Let $e \in \lambda^E \longrightarrow \lambda^I$ be the elaboration function that translates from λ^E to λ^I by replacing every occurrence $t_1; t_2$ with $(\lambda x : Unit. t_2)\, t_1$ (internalization) where x is chosen fresh each case. Now, for each term t of λ^E, we have

- $t \longrightarrow_E t'$ iff $e(t) \longrightarrow_I e(t')$
- $\Gamma \vdash^E t : T$ iff $\Gamma \vdash^I e(t) : T$

(derived form) ...

Wild Card ⟵ (Don't care) ...

$$\frac{\lambda x : A.\, t \qquad x \notin FV(t)}{\Longrightarrow \lambda \_ : A.\, t}$$

Wild Card

P4PCO

Subject: فرمال پیشرفته
Year ۱۳۹۳ Month ۸ Date ۲۰ (فصل ۹ و ۱۱ کتاب Pierce)

Ascription:

$t ::= \ldots$

t as T ⇝ t را از type T در نظر بگیر

New evaluations rules:

$v \text{ as } T \rightarrow v$

$$\frac{t_1 \rightarrow t_1'}{t_1 \text{ as } T \rightarrow t_1' \text{ as } T}$$

New typing rules:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T}$$

به term، type مطابق Ascription خود رسید
اگر سازگار نباشد، type error پیدا می‌شود.

به طور کلی، Ascription در برنامه‌نویسی از ابهام است.

در type ها دو دسته introduction و elimination operation مطرح می‌شود
→ تایپ های اولیه
که از روی تایپ‌ها، تایپ‌های دیگر را می‌سازیم

Pair:

$t ::= \ldots$ ⇝ هر جفت جدید ساخته شود

$\{t, t\}$ → curly brace
→ این سبک نمایش از ML آورده است

$\{t, t\}$
$t.1$ ⇝ Projection
$t.2$
تصور کردن روی مختصات زوج

51

Subject:

Year. Month. Date. ( )

$V ::= \ldots$

$\{V, V\}$

Value اگر به صورت زوج باشد را هم Value حساب کن.

$T ::= \ldots$ (type)

$T_1 \times T_2$

↳ cross

New evaluation rules:

$$\frac{}{\{v_1, v_2\}.1 \longrightarrow v_1}$$

$$\frac{}{\{v_1, v_2\}.2 \longrightarrow v_2}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.1 \longrightarrow t_1'.1}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.2 \longrightarrow t_1'.2}$$

$$\frac{t_2 \longrightarrow t_2'}{\{v_1, t_2\} \longrightarrow \{v_1, t_2'\}}$$

Subject:

Year. Month. Date. ( )

New Typing Rules:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1 \cdot 1 : T_{11}}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1 \cdot 2 : T_{12}}$$

$(\lambda x : Nat \times Nat \,.\, x \cdot 2)\ \{pred\ 4,\ pred\ 5\}$

call-by-value

$\longrightarrow (\lambda x : Nat \times Nat \,.\, x \cdot 2)\ \{3, pred\ 5\}$ — ein Value

$\longrightarrow (\lambda x : Nat \times Nat \,.\, x \cdot 2)\ \{3, 4\} \longrightarrow \{3,4\} \cdot 2 \longrightarrow 4$

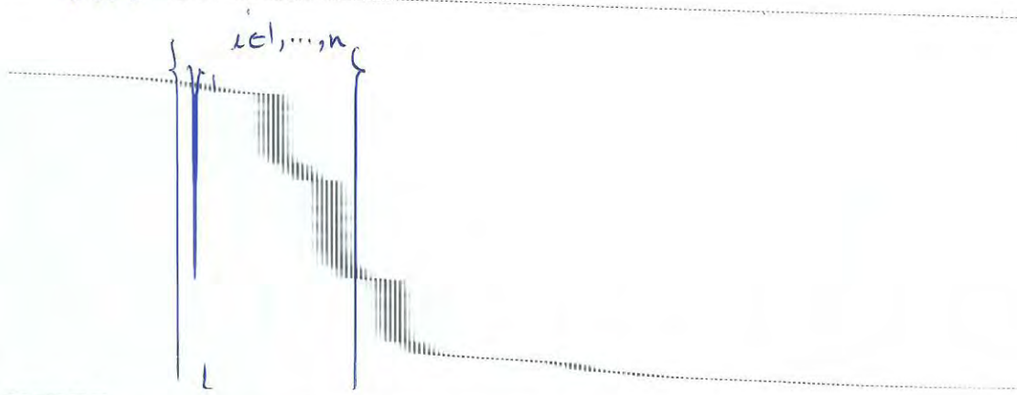
Tuples:

$t ::= \ldots$

$\{t_i^{\,i \in 1,\ldots,n}\}$

$t \cdot i$

$v ::= \ldots$



$T ::= \ldots$

$\{l_i : T_i^{\,i \in 1,\ldots,n}\}$

Evaluation rules:

$$\{l_i = v_i^{\,i \in 1,\ldots,n}\} \cdot l_j \longrightarrow v_j$$

Subject:

Year. Month. Date. ( )

$T_{..} = \ldots$

$\{T_i^{\,i\in 1,\ldots,n}\}$

Evaluation rules:

$$\frac{}{\{v_i^{\,i\in 1,\ldots,n}\}.j \longrightarrow v_j} \quad (1 \le j \le n \text{ نفرض})$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.i \longrightarrow t_1'.i}$$

$$\frac{t_j \longrightarrow t_j'}{\{v_i^{\,i\in 1,\ldots,j-1}, t_j, t_k^{\,k\in j+1,\ldots,n}\} \longrightarrow \{v_i^{\,i\in 1,\ldots,j-1}, t_j', t_k^{\,k\in j+1,\ldots,n}\}}$$

Typing rules

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i^{\,i\in 1,\ldots,n}\} : \{T_i^{\,i\in 1,\ldots,n}\}}$$

$$\frac{\Gamma \vdash t_1 : \{T_i^{\,i\in 1,\ldots,n}\}}{\Gamma \vdash t_1.j : T_j}$$

(مثال)

$\{1, 2, \text{true}\} : \{\text{Nat}, \text{Nat}, \text{Bool}\}$

Subject:

Year. Month. Date. ( )

Records:

فرق tuple و record در این است که در رکورد ... →

$t ::= \ldots$

$\{ l_i = t_i \;^{i \in 1, \ldots, n} \}$

$t.l$ → یعنی name، l چی میشه؟

$v ::= \ldots$

$\{ l_i = v_i \;^{i \in 1, \ldots, n} \}$

$T ::= \ldots$

$\{ l_i : T_i \;^{i \in 1, \ldots, n} \}$

Evaluation rules:

$$\frac{}{\{ l_i = v_i \;^{i \in 1, \ldots, n} \} . l_j \longrightarrow v_j}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 . l \longrightarrow t_1' . l}$$

$$\frac{t_j \longrightarrow t_j'}{\{ l_i = v_i \;^{i \in 1, \ldots, j-1}, l_j = t_j, l_k = v_k \;^{k \in j+1, \ldots, n} \} \longrightarrow \{ l_i = v_i \;^{i \in 1, \ldots, j-1}, l_j = t_j', l_k = v_k \;^{k \in j+1, \ldots, n} \}}$$

Typing rules:

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{\,i\in 1,\dots,n}\} : \{l_i : T_i^{\,i\in 1,\dots,n}\}}$$

$$\frac{\Gamma \vdash t_1 : \{l_i : T_i^{\,i\in 1,\dots,n}\}}{\Gamma \vdash t_1 . l_j : T_j}$$

Sums:

$A = \{a, b, c\}$

$B = \{b, d, e\}$

$A \cup B = \{a, b, c, d, e\}$

disjoint union ← $A \uplus B = \{a_A, b_A, c_A, b_B, d_B, e_B\}$

$A \uplus B = (A \times \{A\}) \cup (B \times \{B\})$

داده‌های درون این مجموعه یا از جنس داده‌ی A است و یا B . یعنی هر داده از این مجموعه از جنس sum A و B است

مثال) physical Addr = {firstlast: string, addr: string}

یک type است.

virtualAddr = {name: string, email: string}

Addr = physicalAddr + VirtualAddr

sum

Subject:

Year. Month. Date. ( )

getName = λa: Addr.

case a of

inl x ⇒ x.firstlast

| inr y ⇒ y.name

t ::= ...

inl t → inject left

inr t → inject right

case t of inl x ⇒ t | inr x ⇒ t

v ::= ...

inl v

inr v

T ::= ...

T+T

Evaluation rules:

case inl(v₀) of inl x₁ → t₁ | inr x₂ → t₂

⟶ [x₁ ↦ v₀] t₁

case inr(v₀) of inl x₁ → t₁ | inr x₂ → t₂

⟶ [x₂ ↦ v₀] t₂

Subject:

Year. Month. Date. ( )

$$\frac{t_0 \longrightarrow t_0'}{\text{case } t_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \longrightarrow \text{case } t_0' \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2}$$

Sum type

$$\frac{t_1 \longrightarrow t_1'}{\text{inl } t_1 \longrightarrow \text{inl } t_1'}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{inr } t_1 \longrightarrow \text{inr } t_1'}$$

Typing rules:

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 : T_1 + T_2}$$

Ascription

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \text{inr } t_2 : T_1 + T_2}$$

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma, x_1 : T_1 \vdash t_1 : T \qquad \Gamma, x_2 : T_2 \vdash t_2 : T}{\Gamma \vdash \text{case } t_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 : T}$$

getName : Addr → String

$$\frac{}{\lambda x : \text{physicalAddr} \,.\, \text{inl } x}$$

inl as physicalAddr + VirtualAddr

Subject: Type Systems, Information Flow (A Type System for Secure Information Flow — Volpano and Smith, 1999)

Year. Month. Date.

Variants:

$t ::= \ldots$

$\langle l = t \rangle$ as T

Case t of $\langle l_i = x_i \rangle \Rightarrow t_i^{\,i \in 1,\ldots,n}$

$T ::= \ldots$

$\langle l_i : T_i^{\,i \in 1,\ldots,n} \rangle$

New Evaluation rules:

$$\frac{}{\text{Case } (\langle l_j = v_j \rangle \text{ as } T) \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{\,i \in 1,\ldots,n} \longrightarrow [x_j \mapsto v_j]\, t_j}$$

$$\frac{t_0 \longrightarrow t_0'}{\text{Case } t_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{\,i \in 1,\ldots n} \longrightarrow \text{Case } t_0' \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{\,i \in 1 \ldots n}}$$

$$\frac{t_i \longrightarrow t_i'}{\langle l_i = t_i \rangle \text{ as } T \longrightarrow \langle l_i = t_i' \rangle \text{ as } T}$$

Typing Rules:

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \langle l_j = t_j \rangle \text{ as } \langle l_i : T_i^{\,i \in 1 \ldots n} \rangle : \langle l_i : T_i^{\,i \in 1 \ldots n} \rangle}$$

Subject: فزول شم
Year. ۹۳ Month. ۸ Date. ۲۷

$$\frac{\Gamma \vdash t_0 : \langle l_i : T_i^{\ i\in 1\ldots n} \rangle \quad \text{for each } i \ \ \Gamma, x_i : T_i \vdash t_i : T}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{\ i\in 1\ldots n} : T}$$

مثال:

Addr = < physical: PhysicalAddr, virtual: virtualAddr >

physical: ⟶ فیلدها را برچسب

a = <physical = pa> as Addr

getName = λa : Addr. case a of

<physical = x> ⇒ x.firstlast
| <virtual = y> ⇒ y.name

getName : Addr ⟶ String

در ML:

datatype student = BS of name
| MS of name * school
| PhD of name * faculty

fun name (BS(n)) = n | name (MS(n,s)) = n |
name (PhD(n,f)) = n

⇒ val name = fn : student ⟶ name

Subject: [illegible]
Year. 93 Month. 8 Date 27 ( )

مثلاً (در ML) :

datatype tree = LEAF of int |

Node of tree * tree

fun inTree(x, LEAF(y)) = x = y |

inTree(x, Node(y, z)) = inTree(x, y) orelse inTree(x, z)

در مثال فوق هم Variant Typing داریم و هم Recursive Typing.

(مثال)

Type enumeration

می‌خواهیم type ای برای روزهای هفته داشته باشیم:

weekday = <monday : Unit, tuesday : Unit, wednesday : Unit, thursday : Unit, friday : Unit>

Case w of <monday = x> ⟹ ...
| <tuesday = x> ⟹ ...
| ⋮

Recursion :

در Typed λ-calculus :

$(\lambda x : T.\ x\ x)(\lambda x : T.\ x\ x)$

(صفحه 61)

$\begin{cases} x : T_1 \to T_2 \\ x : T_1 \end{cases} \to x = ?!$

پس نمی‌توانیم [illegible] آن را با Fixed point Combinator ها استفاده می‌کنیم، اگر recursion داریم از [illegible] ثبت کردیم!

Subject:

Year. Month. Date. ( )

Turing Complete ... Recursion ...

... PCF (Programming Computable Function) ... Plotkin ...

... type ... primitive ... fix point operator ...

مثل: iseven : Nat → Bool

ff = λie : Nat → Bool. λx : Nat.

if iszero x then true
else if iszero (pred x) then false
else ie (pred (pred x))

ff : (Nat → Bool) → (Nat → Bool)

iseven : fix ff

$t ::= \ldots$

fix t

New Evaluation Rules:

$$fix(\lambda x:T_1.\, t_2) \longrightarrow [x \mapsto (fix(\lambda x:T_1.\, t_2))]\, t_2$$

$$\frac{t_1 \longrightarrow t_1'}{fix\ t_1 \longrightarrow fix\ t_1'}$$

Typing:

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t_1 : T_1}$$

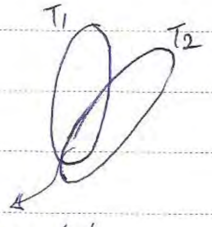
مثلاً برای ورودی، fix ff 2 جواب true می‌دهد.

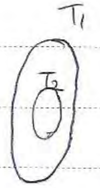
(در مورد PCF، شرح در Foundation fix point کتاب گفته شده است) (حل مسائل و مثال‌های Pierce)

تا اینجا Typed lambda calculus را به همراه گسترش‌های آن دیدیم. سپس می‌دانیم هر term یک type دارد

و progress و preservation را دارد.

Subtyping:





subsumption

$A \subseteq B \rightsquigarrow$ B subsumps A or A is a subset of B

رابطه Subsumption یک partial order است.

$T_3 <: T_1 \rightsquigarrow T_3$ is a subtype of T_1.

T_3 ، subtype ، T_1 است، اگر داده‌ای از T_3 را بدون هیچ نگرانی و خطر safe به جای T_1

به کار ببریم، هیچ مشکلی پیش نمی‌آید.

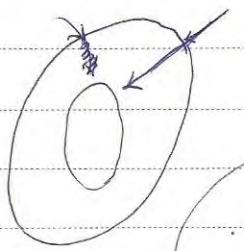
Subject:
Year. 99 Month. 9 Date. ۲ ( )

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$(\lambda r : \{x : Nat\}.\ r.x)\ \{x=0, y=1\}$

$\{x : Nat, y : Nat\}$

$\{x : Nat\}$ $\qquad$ $\{x : Nat, y : Nat\}$



$\{x : Nat, y : Nat\} <: \{x : Nat\}$

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$ ← Subsumption

The subtype relation ($<:$)

$T <: T$ reflexive

$$\frac{T_1 <: T_2 \qquad T_2 <: T_3}{T_1 <: T_3}$$ transitive

(preorder / partial order)

$T_1 <: T_2 \wedge T_2 <: T_1 \not\Rightarrow T_1 = T_2$

Subject:
Year 93 Month. 9 Date. 22 ( )

$$\frac{}{\{l_i : T_i^{\,i\in 1..n+k}\} <: \{l_i : T_i^{\,i\in 1..n}\}} \quad \text{(width subtyping)} \quad \text{(S\_width)}$$

↳ نظیر کلاس و زیرکلاس

$$\frac{\forall i.\ T_i <: T_i'}{\{l_i : T_i^{\,i\in 1..n}\} <: \{l_i : T_i'^{\,i\in 1..n}\}} \quad \text{(depth subtyping)} \quad \text{(S\_depth)}$$

هر کدام زیرنوع متناظر

$$\frac{\{K_i : T_i^{\,i\in 1..n}\} \in \pi\{l_i : T_i'^{\,i\in 1..n}\}}{\{K_i : T_i^{\,i\in 1..n}\} <: \{l_i : T_i'^{\,i\in 1..n}\}} \quad \text{(permutation subtyping)} \quad \text{(S\_perm)}$$

بدون تغییر نوع

$$\{x : Nat,\ y : Real,\ z : Nat \to Nat\} <: \{z : nat \to nat,\ x : Nat,\ y : Real\}$$

$$:>$$

$$\neq$$

$$\{x : \{a : Nat,\ b : Nat\},\ y : \{m : Nat\}\} <: \{x : \{a : Nat\}\}$$ (مثال)

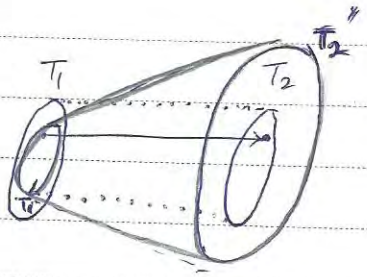
$$\frac{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat, b : Nat\}\}}{\{x : \{a : Nat\}, b : Nat\}, y\{m : Nat\}\} <: \{x : \{a : Nat, b : Nat\}\}} \text{(S\_width)}$$

$$\frac{\{a : Nat, b : Nat\} <: \{a : Nat\}}{\{x : \{a : Nat, b : Nat\}\} <: \{x : \{a : Nat\}\}} \text{(S\_width)} \quad \text{(S\_depth)}$$

$$\frac{}{\{x : \{a : Nat\}, b : Nat\}, y\{m : Nat\} <: \{x : \{a : Nat\}\}} \text{(S\_perm)}$$

Subject: فرمال Adv. Formalism
Year 93 Month 9 Date 2

Function Types :



$T_1 \to T_2$

$T_1' <: T_1$,
$T_2 <: T_2'$

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'} \quad (S\_Arrow)$$

$\mathbb{N} <: \mathbb{R} \qquad \mathbb{R} <: \mathbb{C}$

$\mathbb{R} \to \mathbb{R} <: \mathbb{N} \to \mathbb{C}$

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'} \quad (S\_Arrow)$$

$$\frac{}{S <: Top} \quad (S\_Top)$$

Simply typed λ-Calculus with subtyping :

$(\lambda_{<:})$

Subject: A Type-based Approach to program Security

Year. Month. Date. ( )

T ::= ...

| Top

Subtyping

S-Refl

S-Trans

S-Top

S-Arrow

Static Semantics (Typing Rules)

...

$$\frac{\Gamma \vdash e_1 : T_1 \quad T_1 <: T_2}{\Gamma \vdash e_1 : T_2} \text{ (T-Sub)}$$

Lemma (Inversion of subtyping relation):

$$T <: T_1 \to T_2 \Longrightarrow T = T_1' \to T_2' \wedge T_1 <: T_1' \wedge T_2' <: T_2$$

Lemma:

$$\Gamma \vdash \lambda x : T_1 . e : T_1' \to T_2'$$

$$\Longrightarrow T_1 <: T_1' \wedge \Gamma, x : T_1 \vdash e : T_2'$$

Proof: Induction on p ($p :: \Gamma \vdash \lambda x : T_1 . e : T_1' \to T_2'$)

Induction on typing rules

Subject:

Year. Month9 Date. ۲۰ ( )

$$\frac{\Gamma \vdash \lambda x:T_1.e:T \quad T < : T_1' \to T_2'}{\Gamma \vdash \lambda x:T_1.e:T_1' \to T_2'}$$

: قاعده Subsumption

حال اثبات:

$$p(\Gamma \vdash \lambda x:T_1.e:T) \wedge T < : T_1' \to T_2'$$

$$\Rightarrow p(\Gamma \vdash \lambda x:T_1.e:T_1' \to T_2')$$

$$\Gamma \vdash \lambda x:T_1.e:T_1' \to T_2$$

↓ Take apart

$$\Gamma \vdash \lambda x:T_1.e:T \wedge T < : T_1' \to T_2'$$

Inversion

$$T = T_1'' \to T_2''$$

$$T_1' < : T_1''$$

$$T_2'' < : T_2'$$

$$\Gamma \vdash \lambda x:T_1.e.T_1'' \to T_2'' \wedge T_1' < : T_1'' \wedge T_2'' < : T_2'$$

↓ I.H. Build up

$T_1'' < : T_1 \wedge \Gamma, x:T_1 \vdash e:T_2'' \wedge T_1' < : T_1'' \wedge T_2'' < : T_2'$ → T-Sub → $\Gamma, x:T_1 \vdash e:T_2'$

S-trans → $T_1' < : T_1$

(Co-variant (اصطلاح)) ...
Subject:
Year. ۹۳ Month. ۹ Date. ۴

(اصطلاح Co-variance و Contravariance مطالعه شود)

$$\frac{T <: T'}{\alpha(T') <: \alpha(T)} \qquad \frac{T <: T'}{\alpha(T) <: \alpha(T')}$$

operation, type constructor

نقش subclass یا function یا casting

(cell های mutable) (pierce ...)

اگر type، preserve می‌شود، Covariance (Same) است و اگر ... Contravariance (Reverse)

فصل ۱۵ ... ۱۸۵

مثال:

$$\frac{S_1 <: T_1}{Array\ S_1 <: Array\ T_1} \qquad \frac{S_1 <: T_1}{list\ S_1 <: list\ T_1}$$

(subtype relation ...)

$$\frac{T_1 <: S_1}{Sink\ S_1 <: Sink\ T_1}$$

Polymorphism:

یک term ... type ... .

$doubleNat = \lambda f: Nat \to Nat.\ \lambda x: Nat.\ f(f\ x)$

$doubleRcd = \lambda f: \{l: Bool\} \to \{l: Bool\}.\ \lambda x: \{l: Bool\}.\ f(f\ x)$

$doubleFun = \lambda f: (Nat \to Nat) \to (Nat \to Nat).\ \lambda x: Nat \to Nat.\ f(f\ x)$

یک راه این است که ... این ... . این است ... .

Subject:

Year. Month. Date. ( )

Abstraction Principle:

Each significant piece of functionality in a program should be implemented in just one place in the source code.

polymorphism

این اصل در مفهوم polymorphism هم اصول موفق است. یعنی subtyping و polymorphism در دو جا ابتدا بر تحقق هم ...

شبیه subclassing و subtyping را یکی در نظر می‌گیرد polymorphism در سیستم شده override ...

(در زبان‌ها type را تعریف کند)

polymorphism به شیوه‌های مختلف قابل انجام است:

- Subtyping ∈ Ad-hoc polymorphism

- Overloading ⟶ مثلاً sum برای int و real هر دو یک نام داریم. و برنامه مناسب را می‌خواند.
  ∈ Ad-hoc polymorphism

- Type Case ∈ Ad-hoc p. یعنی بر اساس type با آن برخورد شود

- Parametric Polymorphism ⟶ اگر اینکه داده‌ها type خاصی نداشته باشند
  - در زبان‌های کامپایلری اجرا وارد شده ولی قدیمی است (Java) ↳ Genericity (Generic Java)
  - اجازه می‌دهیم هر نوعی بگیرد ... generic

$double : \forall x . (X \to X) \to (X \to X)$

type متغیر

(Girard, Reynold)

Subject:
Year 99 Month 9 Date 4 ( )

System F : ... Girard ...

$t : \forall X . (X \to X) \to (X \to X)$

↳ Second-order logic

(proposition vs quantification)

... type ... type ...

(First-order logic $\leftrightarrow$ dependent type)

(Propositional logic $\leftrightarrow$ λ-calculus)

- Type Abstraction ... ($\lambda x . x + x$) ... term Abstraction : λ-calculus

$\lambda X . e$ ⇝ Type Abstraction

$e [T]$ ⇝ Type Application

Semantics Rule :

$(\lambda X . e) [T] \longrightarrow [T/X] e$ (E_TApp TABS)

$(\lambda x : T . e)\ v \longrightarrow [v/x] e$ (E-App ABS)

$id = \lambda X . \lambda x : X . x$ ⇝ identity generic

$id [Nat] = \lambda x : Nat . x$

$id [Nat \to Nat] = \lambda x : Nat \to Nat . x$

Subject: فرمال سیستم

Year. ۹۳ Month. ۹ Date. ۶ ( )

$id: \forall X.\ X \to X$: id type این است

Typing:

1) $\dfrac{\Gamma, X \vdash e.T}{\Gamma \vdash \lambda X.e : \forall X.T}$ (T-TABS) Typing - Type Abstraction

↓

Universal Typing

2) $\dfrac{\Gamma \vdash e : \forall X.T_1}{\Gamma \vdash e[T_2] : [T_2/X]T_1}$ (T-TApp) Typing - Type Application

$id = \lambda X.\lambda x:X.x$ → Universal id

$\dfrac{\dfrac{\dfrac{}{x:X \vdash x:X}\ (ref)}{X \vdash \lambda x:X.x : X \to X}\ (T\text{-}ABS)}{\vdash \lambda X.\lambda x:X.x : \forall X.X \to X}$

$\dfrac{\vdash \lambda X.\lambda x:X.x : \forall X.X \to X}{\vdash id[Nat] : [Nat/X](X \to X)}\ id$

↓

$= Nat \to Nat$

$e ::= \ldots$: پس

$\mid \lambda X.e$

$\mid e[T]$

Context

$\Gamma ::= \emptyset$
$\quad \mid \Gamma, x : T$
$\quad \mid \Gamma, X$

$v ::= \lambda x : T . e$
$\quad \mid \lambda X . e$

$T ::= \ldots$
$\quad \mid \forall X . T$ → Universal Type

قواعد تیپ: Typing اضافه می‌شود؟

Dynamic Semantics:

این قواعد به Dynamic Semantics اضافه می‌شود؟:

$$\frac{e_1 \to e_1'}{e_1 [T] \to e_1' [T]}$$

→ قبل از اعمال، باید ساده شود؟

$$\frac{}{(\lambda X . e)[T] \to [T/X] e}$$

(اول ساده‌سازی، سپس جایگذاری)

این زبان همین خواص را دارد:

Theorem — (Preservation):

$$(\Gamma \vdash e : T \wedge e \rightarrow e') \Rightarrow \Gamma \vdash e' : T$$

Theorem — (Progress):

$$e : T \Rightarrow e : \text{val} \ \vee \ \exists e'.\ e \rightarrow e'$$

Theorem — (Normalization)

Well-typed System F expressions are normalizing.

حتماً به مقدار (value) می‌رسد و خاتمه پیدا می‌کند.

چون fix ندارد، پس حلقه بی‌نهایت ندارد (Recursion ندارد)

(اثبات خواص فوق به عنوان تمرین)

(تمرین دیگر: تفاوت Parametric Polymorphism، Adhoc Polymorphism و ... در زبان‌های دیگر)

$$id\,[\forall Y. Y \rightarrow Y] = (\lambda x : \forall Y. Y \rightarrow Y.\ x) :$$

$$(\forall Y. Y \rightarrow Y) \rightarrow (\forall Y. Y \rightarrow Y)$$

اگر محدودیتی روی استفاده Universal Type ها در Universal Type ها نگذاریم، impredicative می‌شود.

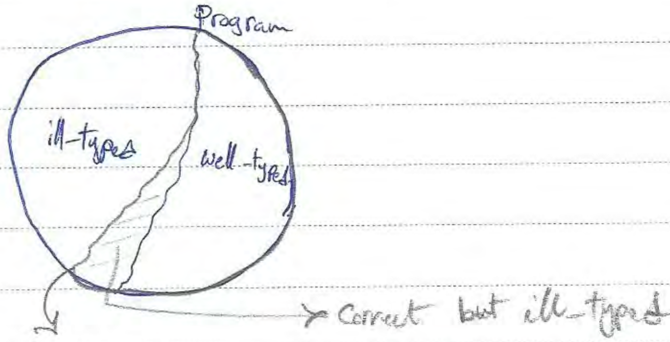
اگر محدودیت را بگذاریم، predicative است.

(استفاده در سیستم امنیتی ... (Access Control

Subject: Advanced Formal (parse)
Year 93 Month 9 Date 9

Type Reconstruction:

کاربردها: از شر نوشتن type راحت بشیم. (type anotation) Programmer (روش Polymorphism است) ← منظور این کار در ML است.



بدلیل conservative بودن type system (ذات type system)
قسمت هاشور خورده هیچ وقت ... ممکن است ... شود.

(type system) ها Sound هستند ولی complete نیست. البته conservative هم هست)

با anotate کردن type ها توسط برنامه‌نویس، باعث بزرگ‌تر شدن بخش هاشور خورده شکل فوق می‌شود. پس

ممکن است type system برنامه‌ای را که type reconstruction خودش تشخیص میدهد well-typed بوده را

ill-type تشخیص دهد. یعنی ممکن است بخش برنامه‌های well typed بزرگ‌تر شود.

با بسته شدن feature های Type Reconstruction Undecided می‌شود (حلقه کمتر ... تصمیم‌گیری)

محوریت این کار با الگوریتم‌های Unification است.

← (تغییر نام به طوری که یک معادله برقرار شود)

Undecidability هم به همین الگوریتم‌های Unification برمی‌گردد.

Type Reconstruction می‌خواهد عدم Type دار شدن یک برنامه فقط ناشی از Conservative بودن باشد و استفاده از
برنامه‌نویس را ... نداشته باشیم.

75/

Subject: فرمال متد

Year. ۹۴ Month. ۹ Date. ۱۱ ( )

$2x = 3y - 2$ راه حل این معادله

Unification (یکی) (substitution) $x \mapsto \frac{3y-2}{2}$

σ : type variables $\longrightarrow$ types

(type substitution)

$\sigma\tau \longrightarrow$ ترکیب σ و τ است (ابتدا τ و سپس σ)

$\sigma = [x \mapsto Bool]$

$\sigma(X \to X) = Bool \to Bool$

$\sigma[X \mapsto T, Y \mapsto U]$

Definition - A type substitution σ is a finite mapping from variables to types. $dom(\sigma)$ is the set of type variables appearing in the left-hand side and $range(\sigma)$ for the set of types appearing in the right-hand side. ($dom(\sigma) \cap range(\sigma) \neq \emptyset$ necessarily)

مثال)

$\sigma = [X \mapsto Bool, \; Y \mapsto X \to X]$

- $\sigma(X) = \begin{cases} \tau & \text{if } (X \to \tau) \in \sigma \\ X & \text{if } X \notin dom(\sigma) \end{cases}$

- $\sigma(Nat) = Nat, \; \sigma(Bool) = Bool$

- $\sigma(\tau_1 \to \tau_2) = \sigma\tau_1 \to \sigma\tau_2$

- $\sigma(\underbrace{x_1 : \tau_1, x_2 : \tau_2, \dots, x_n : \tau_n}_{\Gamma}) = (x_1 : \sigma\tau_1, x_2 : \sigma\tau_2, \dots, x_n : \sigma\tau_n)$

- $\sigma \circ \gamma = \begin{cases} X \mapsto \sigma(\tau) & (X \mapsto \tau) \in \gamma \\ X \mapsto \tau & (X \mapsto \tau) \in \sigma \text{ and } X \notin dom(\gamma) \end{cases}$

Theorem (preservation under substitution) —

$$\Gamma \vdash e : \tau \Rightarrow \sigma\Gamma \vdash \sigma e : \sigma\tau$$

به هر کدام متغیر type ای دارد، apply می‌شود.

مثال:

داریم:

$y : X \vdash \lambda x : X \,.\, x + y : X \to X$

$\sigma = [X \to Nat]$

$y : Nat \vdash \lambda x : Nat \,.\, x + y : Nat \to Nat$

Subject:
Year. Month. Date. ( )

بر این اساس دو سوال قابل طرح است:

1. Are all substitution instances of a well-typed?

$$\forall \sigma . \exists \tau . \sigma\Gamma \vdash \sigma e : \tau \quad (?)$$

2. Is some substitution instance of a well-typed?

$$\exists \sigma . \exists \tau . \sigma\Gamma \vdash \sigma e : \tau \quad (?)$$

Example -

$$\left(\lambda f : X \to X . \lambda a : X . f(f\ a)\right) : (X \to X) \to (X \to X)$$

$\sigma = [X \mapsto U]$ برای هر جانشینی، type دارد. (سوال اول درست نیست)

$$g = \lambda f : Y . \lambda a : X . f(f\ a)$$

با هر جانشینی این type دار نمی‌شود.

$\sigma_1 = [Y \mapsto Nat \to Nat, X \mapsto Nat]$ $\quad \sigma_1 g : (Nat \to Nat) \to (Nat \to Nat)$

$\sigma_2 = [Y \mapsto X \to X]$ ⟿ اگر در domain باشد، یعنی خودش یعنی تابع

$\sigma_2 g : (X \to X) \to (X \to X)$ ⟶ (بیشترین کلی) این principle type است یعنی کلیه type ها را دارد.

P4PCO

مثال در زبان ML:

- fun length (nil) = 0

| length (x :: xs) = 1 + length (x));

▷ val length = fn : 'a list → int

← principle type است.

- fun f ({a = x, b = y, c = z}) = {d = y, e = x}

▷ val f = fn : {a: 'a, b: 'b, c: 'c} → {d: 'b, e: 'a}

↓ type کلی

← این type، principle type است. بزرگ‌ترین type ممکن!

Type Reconstruction یعنی پیدا کردن یک substitution که عبارت داده شده یک type پیدا کند.

Definition – let Γ be a context and e an expression (term).

A solution for (Γ, e) is a pair (σ, T) such that $\sigma\Gamma \vdash \sigma e : T$.

(ترکیب substitution و یک type یک solution است)

Example – let $\Gamma = f : X, a : Y$
$e = f\, a$

جواب‌ها: $([X \mapsto Y \to Nat], Nat)$

$([X \mapsto Y \to Z], Z)$ ⟶ principle type می‌تواند

$([X \mapsto Y \to Z, Z \to Nat], Z)$

Subject:
Year. 93 Month. 9 Date. 11

Constraint – Based Typing (CBT):

(Typing Reconstruction روش دیگر)

Γ, e

↓

Algorithm

A set of constraints (equations between type expression) that must be satisfied by any solution for (Γ, e).

$\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2$

$\Gamma \vdash e_1 e_2 : ?$

$\tau_1 = \tau_2 \rightarrow \tau_{12}$

$e_1 e_2 : \tau_{12}$ $\qquad$ $\boxed{\tau_1 = \tau_2 \rightarrow X}$

CBT شیوه‌ای Typing است ولی طوری طراحی شده با متغیرهای type ای هم کار کنند.

Definition – A constraint set C is a set of equations $\{\tau_i' = \tau_i\}^{i \in 1 \ldots n}$.

A substitution σ is said to unify an equation $\tau' = \tau$ if the substitution instances $\sigma\tau'$ and $\sigma\tau$ are identical.

We say that σ unifier (or satisfier) C if it unifies every equations in C.

Definition – The constraint typing relation $\Gamma \vdash e : \tau \mid_{\chi} C$ is defined by the following rules:

(کای) → مجموعه متغیرهای fresh variables است.

علاوه بر متغیرهای عبارت، متغیرهای type را هم در نظر بگیر

expression e has type τ under assumptions Γ whenever constraint C are satisfied.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \mid_{\emptyset} \{\}} \quad (\text{CT-Var})$$

$\{\}$ → هیچ شرطی

$\emptyset$ → هیچ fresh variable ای نمی‌خواهد.

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2 \mid_{\chi} C}{\Gamma \vdash \lambda x : \tau_1 . e_2 : \tau_1 \to \tau_2 \mid_{\chi} C} \quad (\text{CT-ABS})$$

$$\Gamma \vdash e_1 : \tau_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid_{\chi_2} C_2 \qquad \chi_1 \cap \chi_2 = \chi_1 \cap FV(\tau_2) = \chi_2 \cap FV(\tau_1) = \emptyset$$

$$X \notin \chi_1, \chi_2, \tau_1, \tau_2, C_1, C_2, \Gamma, e_1, \text{ or } e_2$$

$$C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to X\}$$ (→ Function type)

$$\frac{\cdots}{\Gamma \vdash e_1 e_2 : X \mid_{\chi_1 \cup \chi_2 \cup \{X\}} C'} \quad (\text{CT-APP})$$

Subject: (تهیه از زنگ)

Year. Month. Date. ( )

$$\frac{}{\Gamma \vdash 0 : Nat \mid_{\emptyset} \{\}} \quad (CT\_Zero)$$

$$\frac{\Gamma \vdash e_1 : \tau \mid_{\chi} C \qquad C' = C \cup \{\tau = Nat\}}{\Gamma \vdash \underset{(pred\ e_1)}{succ\ e_1} : Nat \mid_{\chi} C'}$$

$$\vdots$$

$$\Gamma \vdash e_1 : \tau_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid_{\chi_2} C_2 \qquad \Gamma \vdash e_3 : \tau_3 \mid_{\chi_3} C_3$$

χ_1, χ_2, χ_3 nonoverlapping

$$C' = C_1 \cup C_2 \cup C_3 \cup \{\tau_2 = \tau_3, \tau_1 = Bool\}$$

$$\frac{}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_2 \mid^{C'}_{\chi_1 \cup \chi_2 \cup \chi_3}} \quad (CT\text{-}IF)$$

$$\vdots (??)$$

$$\frac{}{\vdash \lambda x:X.\ \lambda y:Y.\ \lambda z:Z.\ (xz)(yz) : \tau \mid_{\chi} C} \quad (\cdots)$$

for some τ, χ, C

$$\frac{x : X \to Y \in \{x : X \to Y\}}{x : X \to Y \vdash x : X \to Y \mid \emptyset} \text{(CT-Var)} \qquad \frac{}{x : X \to Y \vdash 0 : Nat \mid \emptyset} \text{(CT-Zero)}$$

$$\phi \cap \phi = \phi \cap FV(Nat) = \phi \cap FV(X \to Y) = \phi$$

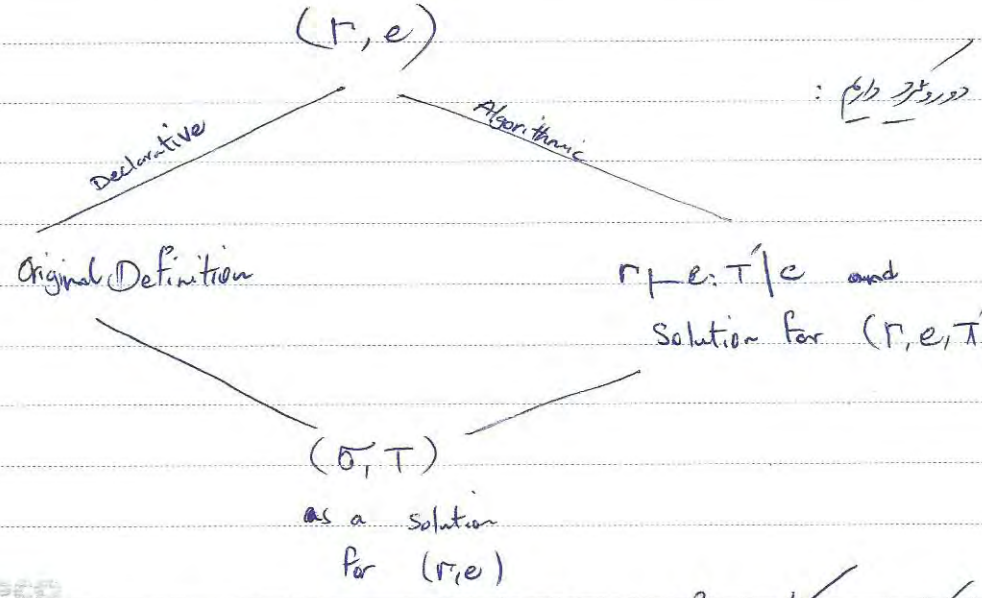
$$Z \notin \phi, \phi, X \to Y, Nat, \phi, \phi, x : X \to Y, x, 0$$

$$C' = \phi \cup \phi \cup \{X \to Y = Nat \to Z\}$$

$$\frac{x : X \to Y \vdash x\,0 : Z \mid \{X \to Y = Nat \to Z\}}{\{Z\}} \text{(CT-App)}$$

$$\frac{\vdash \lambda x : X \to Y.\ x\,0 : (X \to Y) \to Z \mid \{X \to Y = Nat \to Z\}}{\{Z\}} \text{(CT-Abs)}$$

Definition - Suppose that $\Gamma \vdash e : T \mid C$. A solution for (Γ, e, T, C) is a pair (σ, T') such that σ satisfies C and $\sigma T = T'$.



Subject:
Year. Month. Date. ( )

1. Every solution for (Γ, e, T', C) is also a solution for (Γ, e). → Soundness

2. Every solution for (Γ, e) can be extended to a solution for (Γ, e, T', C) → Completeness

متغیرهایی که در مسیر Type Reconstruction ایجاد شد.

Unification:

Definition – A substitution σ is less specific (or more general) than σ', written $\sigma \sqsubseteq \sigma'$, if $\sigma' = \gamma \circ \sigma$ for some γ.

(square subset)

(بیشتر مشخص.) σ' بیشتر جایگزین می‌کند. چون متغیرهایی که σ آزاد گذاشته، ممکن است در دامنه γ
باشد و جایگزین شود. اگر در دامنه γ نباشد، پس هیچ تصمیمی به
آن متغیر گرفته را، تصمیم می‌گیرد (آزاد متغیر)

Definition – A principal unifier (or sometime most general unifier) (اصلی) (principle: اصل) for a constraint set C is a substitution σ that satisfies C and such that $\sigma \sqsubseteq \sigma'$ for every substitution σ' satisfying C.

Example – $\{X = Nat,\ Y = X \to X\}$

$\sigma = [X \mapsto Nat,\ Y \mapsto Nat \to Nat]$ ⟶ جواب unifier نیز

the most general unifier

PAPCO

Subject:
Year. ۹۴ Month. ۹ Date. ۱۴ ( )

Recursive Typing

Example — $\{Y = Nat \to Y\}$ X

- Unification ... $\sigma = [Y \mapsto \mu X.Nat \to X]$:

— $\{X \to Y = Y \to Z,\ Z = U \to W\}$

$\sigma = [X \mapsto U \to W,\ Y \mapsto U \to W,\ Z \mapsto U \to W]$

$\sigma' = [X \mapsto Nat \to Nat,\ Y \mapsto Nat \to Nat,\ Z \mapsto Nat \to Nat,$
$U \mapsto Nat,\ W \mapsto Nat]$

$\sigma'' = [U \mapsto Nat,\ W \mapsto Nat] \circ \sigma$ ... :

Unifier ... Principal ... σ is more general ...

Unify (C) = if $C = \emptyset$, then [ ]

else let $\{T' = T\} \cup C' = C$ in

if $T' = T$ (...)

then unify (C')

else if $T' = X$ and $X \notin FV(T)$

then unify $([X \mapsto T]C') \circ [X \mapsto T]$

else if $T = X$ and $X \notin FV(T')$

then unify $([X \mapsto T']C') \circ [X \mapsto T']$

Subject:
Year. 9۴ Month. 9 Date. ۱۷ ( )

else if $T' = T_1' \to T_2'$ and $T = T_1 \to T_2$
then unify $(C' \cup \{T_1' = T, T_2' = T\})$

else fail

Theorem – The algorithm unify always terminates, failing when given a nonunifiable constraint set as input and otherwise returning a principal unifier.

Example –

$$C = \{ X \to Y = Y \to Z, Z = U \to W \}$$

(1) $Z = U \to W$ ⟹ unify $(\{X \to Y = Y \to (U \to W)\}) \circ [Z \mapsto U \to W]$

(2) ⟹ unify $(\{X \to Y = Y \to (U \to W), X = Y, Y = U \to W\}) \circ [Z \mapsto U \to W]$

(3) $X = Y$ ⟹ ——— $\circ [X \mapsto Y] \circ [$ ——— $]$

⟹ unify $(\{Y \to Y = Y \to (U \to W), Y = U \to W\}) \circ [X \mapsto Y] \circ [Z \mapsto U \to W]$

⟹ unify $(\{(U \to W) \to (U \to W) = (U \to W) \to (U \to W)\}) \circ [Y \mapsto U \to W] \circ [X \mapsto Y] \circ [Z \mapsto U \to W]$

⟹ $[] \circ [Y \mapsto U \to W] \circ [X \mapsto Y] \circ [Z \mapsto U \to W]$

⟹ $\{Z \mapsto U \to W, X \mapsto U \to W, Y \mapsto U \to W\}$

Subject:
Year. ۹۳ Month ۹ Date ۱۴

یعنی پس از به دست آوردن Constraint ها (دسته معادلات Constraint ها!) از الگوریتم استفاده می‌کنیم و

Substitution (یک Solution) را به دست می‌آوریم.

مثال: A Type-based approach to program security :

Semantics : $e \rightarrow e'$

برنامه:

```
y:=1
x:=2
x:=y+x
```

($y=\bot$, $x=\bot$) → ($y=1$, $x=\bot$) → ($y=1$, $x=2$) → ($y=1$, $x=3$)

```
y=1;
in x;
y:=x+2;
```

این برنامه بی‌شمار Computation می‌تواند داشته باشد.

($y=\bot$, $x=\bot$) → ($y=1$, $x=1$) → ($y=3$, $x=1$)

⋮

($y=\bot$, $x=\bot$) → ($y=1$, $x=5$) → ($y=7$, $x=5$)

⋮

پس یک program مجموعه همه اجراهای ممکن است.

P = The set of all possible computations

مثلاً برنامه فوق این خاصیت را که هر چیز کمتر از ۵ باشد را لو می‌دهد.

87

Subject:
Year. Month. Date. ( )

مسئله آیا برنامه خاصیت این را دارد که اگر با state ای شروع شد که در آن $y = \bot$ باشد به خاتمه برسد.

$$P \subseteq \Psi = \Sigma^{\infty}$$

The set of all computations (اجراها Universal)

مجموعه‌ی state های Σ است

Σ^{∞} : مجموعه همه دنباله‌های state های متناهی و نامتناهی

پس یک برنامه زیرمجموعه‌ای از Σ^{∞} است. $P \subseteq \Sigma^{\infty}$

یک خاصیت (property) یعنی Π زیرمجموعه Σ^{∞} است یعنی هر یک از اجراهای است که آن property را دارا می‌باشد.

مثال)

$$\Pi = \left\{ \left\langle \begin{pmatrix} x=\bot \\ y=\bot \end{pmatrix}, \begin{pmatrix} x=1 \\ y=2 \end{pmatrix} \right\rangle, \left\langle \begin{pmatrix} y=\bot \\ x=\bot \end{pmatrix}, \begin{pmatrix} y=1 \\ x=1 \end{pmatrix} \right\rangle, \left\langle \begin{pmatrix} y=1 \\ z=1 \end{pmatrix}, \begin{pmatrix} z=5 \\ y=2 \end{pmatrix} \right\rangle, \dots \right\}$$

$y < 10$

$y = \bot$

و مثلاً اجرای زیر عضو این مجموعه نیست:

$$\begin{pmatrix} y=\bot \\ x=\bot \end{pmatrix} \rightarrow \begin{pmatrix} y=1 \\ x=11 \end{pmatrix} \rightarrow \begin{pmatrix} y=13 \\ x=11 \end{pmatrix}$$

پس اگر قرار باشد برنامه یک property را satisfy کند باید $P \subseteq \Pi$ شود.

program property

property enforcement :

(Program + Enforcement Mechanism)
↓
satisfies the property

مکانیزم های enforcement گوناگون است.

← مکانیزم هایی که monitor میکنند state ها را، فقط Safety را برآورده میکنند.

در صورت لزوم، اجرا را قطع میکنند

Enforcement Mechanism
- → Static → Conservative ⇒ در صورتی که شرایط مورد نظر را نداشته باشد، اجرای برنامه را با اجرای خاص بدل میکند.
  - ← مثل Abstract Interpretation, type System
- → Dynamic (Run-time) Data Flow Analysis, Model checking

مثلاً type system ها اگر جلوی برنامه را بگیرد، اجرای برنامه را با اجرای خاصی که شرایط را برآورده میکند

مثل Runtime ها، انواع monitor ها هستند. مثل:

Execution Monitors, (in-lined monitors)

↓
خودم این در سیستم عامل مطرح شد.

چیزی هم به نام Program Rewriting وجود دارد.

← مانیتورهایی که در کد برنامه است. سمت برنامه هستند
سمت OS

در monitor، action های برنامه چک میشود.

Subject:
Year. Month. Date. ( )

روش دیگر برای enforcement، Rewriting است. (Code Instrumentation)

↓

یک برنامه می‌گیرد، آن را تغییر می‌دهد، برنامه را تحویل می‌دهد

این روش Static است. چون قبل از اجرای برنامه، روی سورس برنامه کار می‌کند.

سخت

برنامه‌ها را طوری تغییر می‌دهند که چک کردن برنامه‌ها را در Runtime انجام دهند. این روش Runtime:

(ترجیحاً Run-time است)

کد تغییر یافته که در زمان اجرا monitor می‌کند. شرایط چک کردن در Runtime را فراهم می‌کند.

monitor ها هر خاصیتی را نمی‌توانند چک کنند در حالیکه Rewriter ها می‌توانند [illegible]

کدام روش enforcement، کدام property ها را می‌توانند بررسی کنند؟

روش‌های Static، فقط [illegible] decidable را بررسی می‌کنند.

ضعیف‌ترین مانیتور، می‌تواند Safety ها را enforce کند.

که فقط مشاهده و قطع اجرا ← (بدون محدودیت‌های Computability)
(Termination Monitor)

به کارگیری enforcement باید برای هر برنامه‌ای کار کند.

(قوی‌ترین مانیتور شناخته‌شده ... !)

PAPCO

Subject: (آزادی اسنایدر)

Year. ۹۳ Month. ۹ Date. ۲۳ ( )

دیدیم که : $\Pi \subseteq \Sigma^\infty$ ← property ها ویژگی مطلوب خود را براساس تک اجرا تعریف می‌کنند.

حال تعریف می‌کنیم : $\Lambda \subseteq 2^{\Sigma^\infty}$ ← مجموعه‌ای از مجموعه اجراها

= The set of all subsets of Σ^∞

← این مجموعه می‌تواند ویژگی‌های را طلب کند.

چه موقع یک برنامه Λ را برآورده می‌کند؟ ← $p \in \Lambda$

← مجموعه اجرا

مجموعه‌ای از مجموعه اجراها

policy

property یک policy خاص است.

$$\Pi \approx \Lambda = 2^{\Psi}$$

(معادل)

مثال) $\Lambda = \{A \mid \forall x \in A. \exists y \in A. xRy\}$ ← با یک تک اجرا قابل بیان نیستند

اجرا   اجرا $x, y \in \Sigma^\infty$

$\Pi = \{x \in \Sigma^\infty \mid \dots\}$ (رابطه)

معروف‌ترین مثال این حوزه (Information Flow Policy)

شرح مثال کلاسیک

← عموماً decidable نیستند

[رابطه بین Formal Method ها و Security در تمام مقاله است، بینید]

آیا مکانیزم‌های enforcement می‌توانند Λ ها را enforce کنند؟! جواب این سوال منفی است.

(Soundness and Transparency)

← تعاریف مختلف دارد ← اجراهای سالم

... ورودی و خروجی معادل هم باشد یا دقیقاً یکی باشد. (اجراهای ناسالم را درست کرده و ...)

(Corrective) سر جلسه بیان

خاصیتی که اجراهای تکی آن خاصیت را برآورده نکند، اینها را monitor نمی‌تواند enforce کند

↓

<>

در این مقاله برای Noninterference یک Type System معرفی کرده.

که با low I/O های یکسان و I/O high های متفاوت،

تفاوتی نباشد. (اگر ورودی و خروجی‌های high را یک کنیم، Google(?) است)

یعنی مشاهده‌گر public کاری به مقادیر private ندارد.

with all program inputs and outputs classified at various security levels, the property basically states that a program output, classified at some level, can never change as a result of modifying <u>only</u> inputs classified at high levels.

مثال) سطح امنیت نسبت به سلسله مراتب.

(Nondeducability)

Top secret
↓
Secret
↓
Classified
↓
unclassified

هر چیزی از این سطح‌بندی را در نظر بگیریم، اگر بالایی‌ها

تغییر کند (یا این‌ها تغییر کنند)، سطوح پایین

مقادیرش تغییر نکند. مقادیر مشاهده‌شده در سطوح پایین با تغییر مقادیر سطوح بالا تغییر نکند.

Subject:
Year 93 Month 9 Date 25 ( )

A core imperative language

A type system so that every typable program definitly satisfies P.
Policy

Undecidable است. Type System decidable است. Soundness و Completeness

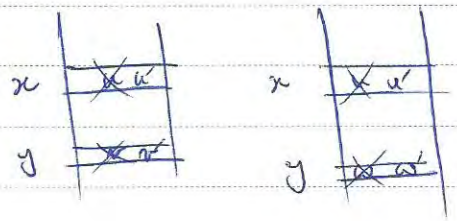
proc P(inout x : low, inout y : high)

output / input

P(u : low, v : high)

actual parameter

P(u : low, w : high)



Subject:
Year. Month. Date. ( )

Types:

τ types — سطح امنیت را با type ها نشان می‌دهیم:
→ Security level

π types → expressions & commands

ρ types → phrase
→ ترکیبی از expression و command

$\leq$ یک ترتیب جزئی lattice

(partial order)

low $\leq$ high → read شرط برای

trusted $\leq$ untrusted → write شرط برای

$\subseteq$ subtype relation (مثال: ۲ خوب ۱)

low ها می‌توانند جای high ها قرار بگیرند، چون اگر high در جایی قابل تحمل است پس low را هم می‌توان گذاشت.

τ cmd : every assignment in the command is to a variable whose security level is τ or higher.

(یک متغیر از نوع τ)

τ var : a variable that stores information whose security type is τ or lower.

τ acc : write_only variables → procedure سطر خروجی out

P4PCO

Subject: Advanced Formal
Year. ۹۳ Month. ۹ Date ۲۵

The subtype relation is contravariant in both command and acceptor types.

$$\frac{\tau \leq \tau'}{\tau' \ cmd \subseteq \tau \ cmd}$$

$$\frac{\tau \leq \tau'}{\tau' \ acc \subseteq \tau \ acc}$$

↳ write only

یکی از انواع نشتی (leakage)، Explicit Flow است. مثل:

$l := h$ متغیر مقدار high را در low ها قرار دهیم.

$$\frac{\gamma \vdash e : \tau \ acc \qquad \gamma \vdash e' : \tau}{\gamma \vdash e := e' : \tau \ cmd}$$

← Context ها

l: low var, h: high var $\in \gamma$

$l := h$ طبق rule قبلی و عبارت زیر typable خواهد بود.

$h := l$ → طبق rule قبلی با subtyping، typable می‌شود.
چون high acc $\subseteq$ low acc

برای این هم low cmd و هم high cmd را می‌توان derive کرد.

95/

while $h > 0$ **do**

$l := l + 1;$

$h := h - 1;$

در برنامه فوق Implicit Flow است چون با این افزایش l می‌فهمیم که h از صفر بزرگتر است.

$$\frac{\gamma \vdash e : \tau \quad \gamma \vdash c : \tau\ cmd}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ cmd}$$

پس طبق rule های فوق برنامه بالا typable نمی‌شود. (subtyping هم کمک نمی‌کند)

Flow ها انواع مختلفی دارند.

Type Inference:

proc (**in** x, **out** y)

letvar $a := x$ **in**

letvar $b := 0$ **in**

while $a > 0$ **do**

$b := b + 1;$

$a := a - 1;$

$y := b$

$\forall \alpha, \beta$ with $\alpha \leq \beta$. β proc(α, β acc).

high proc(low, high acc) : شده قابل استفاده شد

حتی این inference مشخص میشود که چه policyهایی میتواند این برنامه meet کند.

اگر چون type ها را مشخص میکردیم، یعنی میخواستیم مشخص کنیم چه policyهایی را مدنظر داریم.

البته بعدا مشخص میشود که تایپ بالایی میتواند $\forall \beta$. β proc(β, β acc) باشد چون subtyping کند.

(کار دیگری که این آقایان با هم در زمینه بررسی ویژه کارهای برنامه)
مشخص کردن شبیه FABEL

سینتکس : Syntax

(phrase) $p ::= e \mid c$

(expr) $e ::= x \mid n \mid l \mid e + e' \mid e - e' \mid e = e' \mid e < e' \mid$ ادامه پایین

↓ id, ↓ reference — قرار است جزء I/O قرار بدیم

ادامه بالا → proc(in x_1, inout x_2, out x_3) c
مقدار اولیه location هاش ورودی
مقدار خروجی location هاش خروجی

بار سنگین letvar، location ها را آزاد میکنیم

(comm) $c ::= e := e' \mid c ; c' \mid e(e_1, e_2, e_3) \mid$
while e do c | if e then c else c' |
letvar $x := e$ in c | letproc x(in x_1, inout x_2, out x_3) c in c'