程式語言理論

導論：建立程式語言的模型

游書泓

2020 邏輯、語言與計算暑期研習營

為什麼研究程式語言？

程式語言建立抽象介面：高階語言

程式語言建立抽象的界面，並藉由略去無關細節，讓程式設計師直接描述並解決問題本身。在電腦系統中，人們已經為不同層次的抽象化設計了對應的程式語言。

```
int maximum(int a, int b) {
  if (a >= b)
    return a;
  return b*b - 4;
}
```

```
        .globl  maximum
maximum:
        mov     eax, edi
        cmp     edi, esi
        jge     .L1
        imul    esi, esi
        lea     eax, [rsi-4]
.L1:
        ret
```

```
define dso_local i32 @maximum(i32, i32) local_unnamed_addr #0 {
  %3 = icmp slt i32 %0, %1
  %4 = mul nsw i32 %1, %1
  %5 = add nsw i32 %4, -4
  %6 = select i1 %3, i32 %5, i32 %0
  ret i32 %6
}
```

程式語言建立抽象介面：各式語言特性

程式語言也包含對不同應用抽象化設計的語言特性與函式庫。

```
SELECT author FROM commit_list
  ORDER BY commit_time DESC
  WHERE ci_result='pass'
  LIMIT 10;
```

```
def seq(Gs):
  for G in Gs:
    yield from G()
def replicate(G, n):
  yield from seq(G for i in range(n))
```

```
struct parse_instruction_and_construct_ast_p :
  pegtl::sor<
    inst_p<ret_keyword_p, make_new<ret_i_t>, 0>, // return
    inst_p<pegtl::seq<TAO_PEGTL_KEYWORD("br"),   //br
                  sep_p, save_p<label_ap>, sep_p>,
           make_new<br_i_t>, 1, 0>,
    ...
```

常見描述、定義程式語言特性的方式各有限制

1. 用自然語言描述輔以範例。然而文字敘述無法操作、分析性質與研究語言構造的互動，並且面對複雜的構造有其極限。
2. 編譯成低階語言做為解釋。但可能引入不必要的細節或打破高階語言的抽象介面。

Python 3 documentation, 6.2.9. Yield expressions:

> *When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of the generator function. The execution starts when one of the generator's methods is called. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of* `expression_list` *to the generator's caller. By suspended, we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the yield expression were just another external call.*

程式語言理論使用數學工具及形式邏輯建立範式與框架

可操作的規格與工具： 藉由數學與形式邏輯工具，程式語言各個構造皆使用數學精確定義，甚至一部份存在對應的數學物件。

同樣的，編譯、最佳化、重構變換、動態/靜態分析與驗證等工具也能在此框架中以數學描述。我們並可以討論這些工具相對於程式語言的定義是否正確。

證明性質： 透過程式語言構造的數學定義，我們能證明語言的設計是否滿足所需的性質。

分析優缺點： 程式語言理論也包括分析語言的表達力強弱、探討語言構造之間的互動關係。一個例子是說明為何圖靈完備性並不等價於與程式語言的表達力。

怎麼分析程式語言？

問題：

- 程式語言表層語法對分析本身無關緊要
- 基於低層次抽象的描述過於瑣碎
- 自然語言的表述無法操作、計算

方法： 建立數學模型並以數學工具分析

目標：

- 簡潔描述程式語言中的程式
- 以數學模型精確定義程式的語義
- 形式的討論語言中不同部件的規格，例如型別系統
- 做到符號計算與推論

程式語言的數學模型

程式語言的數學模型

程式語言的定義常見分為語法、語義、可選的型別系統等部件。

1. **語法**：一個語言中，程式通常以歸納定義的**抽象語法樹**建模。抽象語法樹略去了諸如排版、分號或括弧等語法分析用的細節，僅保留語言構造本身的結構。
   程式語言模型討論的對象物件即所有抽象語法樹的集合。
2. **語義**：抽象而言，語義指定了程式（抽象語法樹）的「意思」。語義有三種常見形式，分別從不同角度將「意思」詮釋成可以操作、推理的定義。舉例來說，操作語義是定義於程式集合之上的轉移系統。
3. **型別系統**：型別系統為每一個子程式關聯了一些型別。整個程式關聯的型別並預測了其執行期間的行為。換句話說，型別系統相對於程式語言的語義是**健全** (*sound*) 的。

為什麼研究程式語言？

程式語言的數學模型

- 語法
- 語義
- 擴展算術語言：變數綁定及語法作用域
- 擴展算術語言：更多內建資料型態以及型別系統

結論

算術語言的語法：怎麼用可操作的方式建立程式的模型？

回顧

一個語言中，程式通常以歸納定義的**抽象語法樹**建模。抽象語法樹略去了諸如排版、分號或括弧等語法分析用的細節，僅保留語言構造本身的結構。

考慮一個處理基礎算術的程式語言：**算術語言**。算術語言中，一個程式即為一個由加法、乘法等二元運算和自然數組成的算式。此算式計算順序為由左向右。我們並規定算術語言的語法符合四則運算優先順序及左結合的慣例，同時允許自由的排版。

Example (**具體語法** (concrete syntax))

```
(5 + 2)*3
```

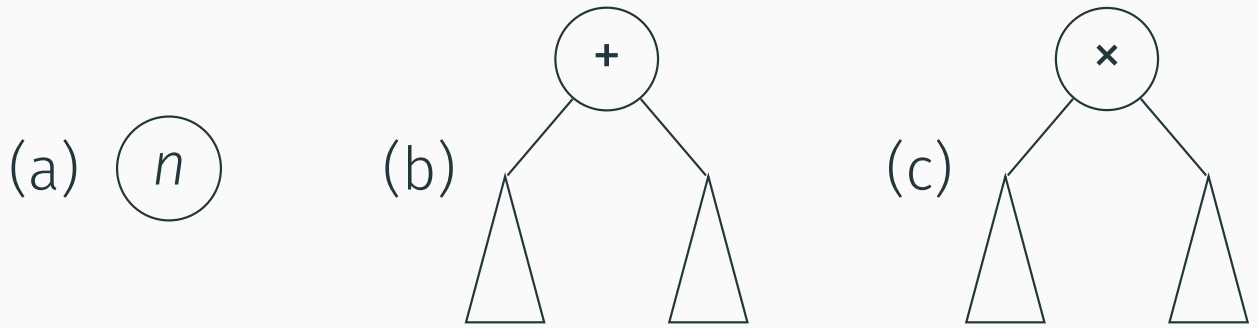
```
3 +  4 *
     5
```

```
2 * 3 * (5 * 7)
```

算術語言的語法：抽象語法樹

算術語言的抽象語法樹包含：

1. 一種 terminal node：代表自然數的節點 (a) （$n \in \mathbb{N}$）。
2. 兩種 inner nodes：代表二元運算 $+$、$*$ 的節點 (b) 與 (c)。



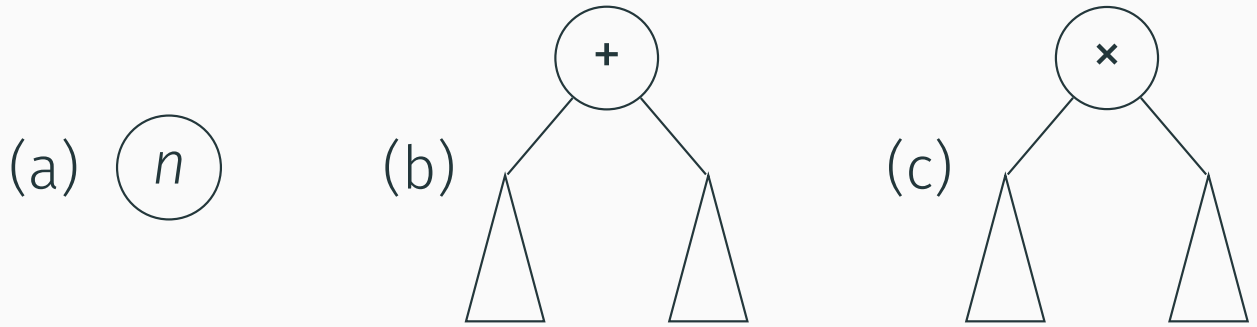
我們仍將以上的樹簡寫為數學算式，依照慣例省略括弧，但了解**實際操作的對象是簡寫代指的語法樹**，而非線性的符號串。

Example (抽象語法樹 (abstract syntax tree) 的簡寫)

$(5 + 2) \times 3$	$3 + (4 \times 5)$ 或 $3 + 4 \times 5$	$((2 \times 3) \times (5 \times 7))$ 或 $2 \times 3 \times (5 \times 7)$

簡單的 Haskell 實做

```
data Expr = ILit Integer                    -- (a)
  | Plus Expr Expr | Times Expr Expr        -- (b)(c)
```



Example (以 algebraic data type `Expr` 實作抽象語法樹)

```
Times (Plus (ILit 5) (ILit 2)) (ILit 3)
```

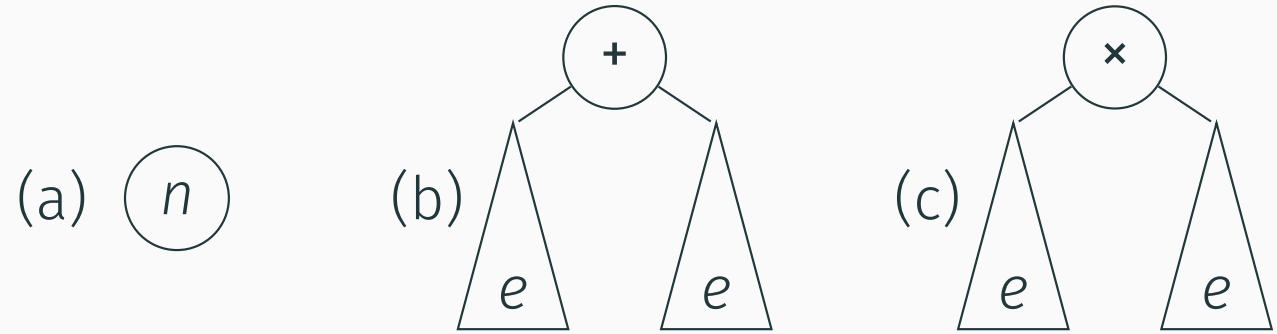
```
Plus (ILit 3) (Times (ILit 4) (ILit 5))
```

```
Times (Times (ILit 2) (ILit 3))
      (Times (ILit 5) (ILit 7))
```

算術語言的語法：形式定義

$$\begin{aligned} \text{Term} &\ni e ::= n \mid e + e \mid e \times e \\ \mathbb{N} &\ni n ::= 0 \mid 1 \mid 2 \mid \ldots \end{aligned}$$

抽象語法樹的算式簡寫常用 context-free grammar 歸納地定義。



Term 為所有抽象語法樹的集合。於 ::= 左側，Term $\ni e$ 表明符號 e 代稱 Term 中的語法樹。等號右側由 | 分隔的 productions 依序對應語法樹 e 的三種構造方法：(a) n (b) $e + e$ 或 (c) $e \times e$。

production 中的符號如 n、e 代表對應子樹的類別，例如 $e + e$ 意指 (b) 中兩側的子樹都必須是 Term 中的語法樹。

為什麼研究程式語言？

程式語言的數學模型

- 語法
- 語義
- 擴展算術語言：變數綁定及語法作用域
- 擴展算術語言：更多內建資料型態以及型別系統

結論

語義：怎麼用可操作的方式定義程式的「意思」？

回顧

抽象而言，語義指定了程式（抽象語法樹）的「意思」。語義有三種常見形式，分別從不同角度將「意思」詮釋成可以操作、推理的定義。舉例來說，操作語義是定義於程式集合之上的轉移系統。

從不同角度對「意思」作出的詮釋對應三種類型的語義：

1. **操作語義** (*operational semantics*)：
   程式怎麼執行？
2. **公理語義** (*axiomatic semantics*)：
   程式執行前後系統的預期狀態、性質是（以邏輯式表達）？
3. **指稱語義** (*denotational semantics*)：
   程式代表什麼意思？

語義

```
1  int maximum(int a, int b) {
2    if (a >= b)
3      return a;
4    return b;
5  }
```

- **程式怎麼執行？**
  maximum 被呼叫時將執行第 2 行。若 a >= b 是 true 則執行第 3 行回傳 a，否則執行第四行回傳 b。
- **程式執行前後系統的預期狀態、性質是？**
  maximum 接收兩個參數，並回傳其中的較大值。
- **程式代表什麼意思？**
  maximum(a,b) 計算數學上的 $\max(a, b)$ 函數。

語義

```
1  int factorial(int n) {
2    int prod = 1;
3    while (n != 0) { prod = prod * n
4                   ; n = n - 1; }
5    return prod;
6  }
```

- **程式怎麼執行？**（略）
- **程式執行前後系統的預期狀態、性質是？**
  設 `factorial` 輸入 `n` $:= N$。當 $N \geq 0$ 時，函式 `factorial` 會中止。其回傳值 `factorial(n)` $= N!$
- **程式代表什麼意思？**
  `factorial` 計算如右的函數：$f(n) = \begin{cases} n! & \text{if } n \geq 0 \\ \bot & \text{otherwise} \end{cases}$

算術語言的 operational semantics

結構操作語義 (*structural operational semantics*) 由 G. D. Plotkin 系統地整理，特性為

1. 以**轉移系統**的方式在語法樹集合上規範程式如何執行
2. 轉移系統本身以**語法導向**的原則**歸納**地定義。

將結構操作語義應用於算術語言時，我們引入幾個定義及概念：

- 一個語法樹集合的子集 $\text{Value} \subset \text{Term}$，包含所有最終的計算結果。集合中的元素 $v \in \text{Value}$ 稱為**值**。
- 一個定義於語法樹集合 Term 上的 binary relation $\to \subset \text{Term} \times \text{Term}$，描述算術語言的程式如何**化簡**。
- 一個程式的意思是其經由 $\to$ 多次化簡最終得到的值。對 $e \in \text{Term}$，若 $e = e_1 \to \cdots \to e_n = v$ 則 e 的意思是 v。

算術語言的 operational semantics

$$\begin{aligned} \text{Term} &\ni e ::= n \mid e + e \mid e \times e \\ \text{Value} &\ni v ::= n \end{aligned}$$

算術語言的值（最終計算結果）只有自然數一種可能，而化簡操作恰巧對應到算術中的計算、化簡動作。

- $(n_1 + n_2) \to n$ 其中 $n = n_1 + n_2$
- $(n_1 \times n_2) \to n$ 其中 $n = n_1 \times n_2$
- 計算化簡可能發生於程式中的一個子片段，如範例中劃線之處。因此 $\to$ 也擔負定位子片段、指定化簡順序的責任。

Example

- $\underline{(5 + 2)} \times 3 \to \underline{7 \times 3} \to 21$
- $\underline{2 \times 3} \times (5 \times 7) \to 6 \times \underline{(5 \times 7)} \to \underline{6 \times 35} \to 210$

算術語言的 operational semantics

$$\begin{array}{rcll} \text{Term} & \ni & e & ::= & n \mid e + e \mid e \times e \\ \text{Value} & \ni & v & ::= & n \end{array}$$

我們能將 $\to$ 拆成**定位子片段**以及**定義實際化簡操作**兩部份。

實際參與化簡的算式如 $n_1 + n_2$、$n_1 \times n_2$ 等皆稱作 *redex*。令 binary relation $\mathbf{r} \subset \text{Term} \times \text{Term}$ 負責定義 redexes 的化簡，則

$$\begin{array}{ll} (n_1 + n_2)\,\mathbf{r}\,n & \text{for all } n_1, n_2, n \in \mathbb{N} \text{ s.t. } n = n_1 + n_2 \\ (n_1 \times n_2)\,\mathbf{r}\,n & \text{for all } n_1, n_2, n \in \mathbb{N} \text{ s.t. } n = n_1 \times n_2 \end{array}$$

至於在完整程式中定位子片段（即 redex），由於算術語言計算順序為由左至右，因此 $\to$ 應該額外滿足

- $\mathbf{r} \subset \to$，因算式本身可能是 redex
- $e_1 + e_2 \to e_1' + e_2$ 當 $e_1 \to e_1'$；　$v + e_2 \to v + e_2'$ 當 $e_2 \to e_2'$。
- $e_1 \times e_2 \to e_1' \times e_2$ 當 $e_1 \to e_1'$；　$v \times e_2 \to v \times e_2'$ 當 $e_2 \to e_2'$。

算術語言的 operational semantics

$$(n_1 + n_2)\mathbf{r}n \quad \text{for all } n_1, n_2, n \in \mathbb{N} \text{ s.t. } n = n_1 + n_2$$
$$(n_1 \times n_2)\mathbf{r}n \quad \text{for all } n_1, n_2, n \in \mathbb{N} \text{ s.t. } n = n_1 \times n_2$$

注意此處符號與語法定義的 production 意義相異。以下敘述中符號 e_1、e_2、v 皆指稱具體但未知、隸屬於指定類別的語法樹，因此重複的符號代指相同的語法樹。

- $\mathbf{r} \subset \to$
- $e_1 + e_2 \to e_1' + e_2$ 當 $e_1 \to e_1'$；$v + e_2 \to v + e_2'$ 當 $e_2 \to e_2'$。
- $e_1 \times e_2 \to e_1' \times e_2$ 當 $e_1 \to e_1'$；$v \times e_2 \to v \times e_2'$ 當 $e_2 \to e_2'$。

Example

$\underline{2 \times 3} \times (5 \times 7) \to$	by	$(e_1 \times e_2) \to (e_1' \times e_2) \wedge e_1 \to e_1'$
$6 \times \underline{(5 \times 7)} \to$	by	$(v \times e_2) \to (v \times e_2') \wedge e_2 \to e_2'$
$\underline{6 \times 35} \to$	by	$(n_1 \times n_2)\mathbf{r}n \wedge n = n_1 \times n_2$
210		

算術語言的 operational semantics

$$
\begin{aligned}
(n_1 \texttt{+} n_2)\,\mathsf{r}\,n &\quad \text{for all } n_1, n_2, n \in \mathbb{N} \text{ s.t. } n = n_1 + n_2 \\
(n_1 \texttt{*} n_2)\,\mathsf{r}\,n &\quad \text{for all } n_1, n_2, n \in \mathbb{N} \text{ s.t. } n = n_1 \times n_2
\end{aligned}
$$

我們用函數 `redex` 實作 relation $\mathsf{r} \subset \text{Term} \times \text{Term}$。

一般而言，對單一 $e \in \text{Term}$ 可能有複數的 $e' \in \text{Term}$ 滿足 $e \mathsf{r} e'$，
因此我們選擇用型別為 `Expr -> [Expr]` 的函數表達 r。

```
redex :: Expr -> [Expr]
redex (Plus (ILit n) (ILit m)) = [ILit (n + m)]
redex (Times(ILit n) (ILit m)) = [ILit (n * m)]
redex _                        = []
```

算術語言的 operational semantics

函數 reduce 實作了 $\rightarrow$。isValue 負責甄別出 Expr 中的值。

```
reduce :: Expr -> [Expr]
reduce e = redex e ++
  case e of
    ILit _       -> []
    Plus e1 e2   ->
      [ Plus e1' e2  | e1' <- reduce e1 ] ++
      [ Plus e1  e2' | isValue e1,
                       e2' <- reduce e2 ]
    Times e1 e2 ->
      [ Times e1' e2  | e1' <- reduce e1 ] ++
      [ Times e1  e2' | isValue e1,
                        e2' <- reduce e2 ]
```

關於可操作性

- 抽象語法樹的集合 Term 是**歸納地**建構的；任意一個程式 $e \in \text{Term}$ 只有 $e := n \in \mathbb{N}$、$e := e_1 + e_2$ 或 $e := e_1 \times e_2$ 等三種可能之一。
  由於 Term 是歸納定義的，對程式的分析與操作可以轉化為 $e \in \text{Term}$ 上的**遞迴定義**或**結構歸納法**。
- 雖然我們省略了 $\to$ 的形式定義，但 `reduce` 函數大略指出了 $\to$ 的歸納結構及操作方法。
  對於任意一步化簡操作 $e \to e'$ 而言，共有以下的可能：
  - $e \to e'$ 即為 $e\mathsf{r}e'$，而 $e\mathsf{r}e'$ 又會是 $(n_1 + n_2)\mathsf{r}(n_1 + n_2)$ 或 $(n_1 \times n_2)\mathsf{r}(n_1 \times n_2)$ 之一。
  - $e \to e'$ 即為 $e_1 + e_2 \to e_1' + e_2$，同時 $e_1 \to e_1'$ 。
  - $e \to e'$ 即為 $v + e_2 \to v + e_2'$，同時 $e_2 \to e_2'$ 。
  - 乘法的可能性。與加法類似。

小節

- 程式語言基本的數學模型包含**語法**及**語義**兩部份。
- 一個語言的語法用**抽象語法樹**建模。抽象語法樹的集合即為所有的程式。
- 語義中的結構操作語義為語法樹集合上的**轉移系統**，以值的集合搭配語法樹集合上的 binary relation 描述。

Side Note

結構操作語義中描述轉移的 relation 事實上可以用 inference rules 以及 derivations 正式地、結構地書寫。如此能清晰的表述該 relation 的歸納結構（諸如歸納假設等等）。

為什麼研究程式語言？

程式語言的數學模型

- 語法
- 語義
- 擴展算術語言：變數綁定及語法作用域
- 擴展算術語言：更多內建資料型態以及型別系統

結論

加入變數

$$\begin{array}{rcccl} \text{Term} & \ni & e & ::= & \ldots \mid x \mid \texttt{let}\, x = e\, \texttt{in}\, e \\ \text{Var} & \ni & x, y, z, \ldots & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ldots \end{array}$$

Var is a countable set of variables. It is left abstract.

我們在算術語言中引入程式語言中常見的變數構造。新的語言構造有變數參考以及建立新綁定的 let 算式兩種。

算式 $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ 將會先計算 e_1 的值，並在計算 e_2 時把 x 綁定為 e_1 計算的結果。

Example

$1 + \texttt{s}$

```
let z = 5 in
let w = z + 3 in
z × w
```

```
let x = 2 in
(let y = 5 in x × y) + x
```

Binding、Scopes 與 Substitution

$$\begin{array}{rcll} \text{Term} & \ni & e & ::= \quad \ldots \mid x \mid \texttt{let}\, x = e\, \texttt{in}\, e \\ \text{Var} & \ni & x, y, z, \ldots & ::= \quad \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ldots \end{array}$$

Var is a countable set of variables. It is left abstract.

令符號 $e[e'/x]$ 代表 *capture-avoiding substitution*，即在保持 let 綁定結構的情況下把 e 中所有未綁定的變數 x 都代換成 e'。

算式 e 中未綁定的變數稱為 e 當中的**自由變數** (*free variable*)。

let 算式能透過 substitution 定義。敘述「計算 e 時將變數 x 綁定到值 v」能翻譯為 $e[v/x]$，如此我們間接地描述了綁定的作用。

Example

$$\begin{array}{rcl} (\texttt{x} + \underline{\texttt{y}} + 5)[8/\texttt{y}] & = & \texttt{x} + \underline{8} + 5 \\ (\underline{\texttt{z}} \times (\texttt{let}\, \texttt{z} = 2\, \texttt{in}\, \texttt{z} + 3))[1/\texttt{z}] & = & \underline{1} \times (\texttt{let}\, \texttt{z} = 2\, \texttt{in}\, \texttt{z} + 3) \end{array}$$

計算 let 算式

$$\begin{array}{rcll} \text{Term} & \ni & e & ::= \quad \ldots \mid x \mid \texttt{let}\, x = e\, \texttt{in}\, e \\ \text{Var} & \ni & x, y, z, \ldots & ::= \quad \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ldots \end{array}$$

Var is a countable set of variables. It is left abstract.

更正式的說，我們如以下方式擴展 binary relations r、$\to$ 來描述 let 算式與變數的語義：

- $(\texttt{let}\, x = v\, \texttt{in}\, e)\mathsf{r}(e[v/x])$
- $\texttt{let}\, x = e_1\, \texttt{in}\, e_2 \to \texttt{let}\, x = e_1'\, \texttt{in}\, e_2$ 當 $e_1 \to e_1'$

Example

$$\underline{\texttt{let x} = 2\, \texttt{in}\, (\texttt{let y} = 3 + \texttt{x}\, \texttt{in}\, \texttt{x} \times \texttt{y}) + \texttt{x}} \to$$
$$(\texttt{let y} = \underline{3 + 2}\, \texttt{in}\, 2 \times \texttt{y}) + 2 \to$$
$$\underline{(\texttt{let y} = 5\, \texttt{in}\, 2 \times \texttt{y})} + 2 \to$$
$$\underline{(2 \times 5)} + 2 \to \underline{10 + 2} \to 12$$

為什麼研究程式語言？

程式語言的數學模型

- 語法
- 語義
- 擴展算術語言：變數綁定及語法作用域
- 擴展算術語言：更多內建資料型態以及型別系統

結論

加入其他資料型態

$$
\begin{array}{rcll}
\text{Term} & \ni & e & ::= \quad \ldots \mid b \mid e = e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{Value} & \ni & v & ::= \quad \ldots \mid b \\
\mathbb{B} & \ni & b & ::= \quad \texttt{true} \mid \texttt{false}
\end{array}
$$

我們往算術語言中引進 boolean 作為新的原生資料類別。與 boolean 相關的新語言構造包含

- boolean 值 $b \in \mathbb{B} = \{\texttt{true}, \texttt{false}\}$
- 判定兩個算式計算結果是否相等的 = 運算子
- 有分支、選擇功能的 if 算式

我們並擴展最終計算結果——Value 的定義，納入 boolean 值。

Example (程式片段)

```
if z = 0 then 0 else z × 3
```

boolean 值與 if 算式的計算

注意由於 $v \in \text{Value}$ 擴展了，原先 $\to$ 中的規則也有受到影響。

- $(v_1 = v_2)\texttt{rtrue}$，如果 $v_1 = v_2 \in \mathbb{N}$ 或 $v_1 = v_2 \in \mathbb{B}$
  $(v_1 = v_2)\texttt{rfalse}$，其他情況
- $e_1 = e_2 \to e_1' = e_2$ 當 $e_1 \to e_1'$ ； $v = e_2 \to v = e_2'$ 當 $e_2 \to e_2'$
- $(\texttt{if true then } e_1 \texttt{ else } e_2)\texttt{r}e_1$
  $(\texttt{if false then } e_1 \texttt{ else } e_2)\texttt{r}e_2$
- $\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \to \texttt{if } e' \texttt{ then } e_1 \texttt{ else } e_2$ 當 $e \to e'$

Example

$\texttt{if (if } \underline{5 = 3} \texttt{ then false else true) then } 3 + 4 \texttt{ else } 0 \to$
$\texttt{if } \underline{\texttt{(if false then false else true)}} \texttt{ then } 3 + 4 \texttt{ else } 0 \to$
$\underline{\texttt{if true then } 3 + 4 \texttt{ else } 0} \to \underline{3 + 4} \to 12$

$(5 \times \underline{(8 = \texttt{true})}) = 0 \to (5 \times \texttt{false}) = 0 \not\to (\textit{stuck!})$

型別與型別系統

進一步延伸模型，納入**型別**的概念。型別是什麼？有什麼作用？

```
bool has_solution(int a, long b, int c) {
  long AC = 4 * (long)a * (long)c;
  return b*b >= AC;
}
```

對型別字面上的理解很容易得到下列不完整的印象：

- 型別代表儲存一個變數所需的記憶體空間
- 型別代表一個值在記憶體中儲存的形式
- 型別代表一個變數被允許的操作

型別與型別系統

進一步延伸模型，納入**型別**的概念。型別是什麼？有什麼作用？

J.H. Morris, *Lambda-Calculus Models of Programming Languages*, PhD dissertation, pp. 12, 1968:

> *"We tend to understand these subjects pragmatically. When a programner thinks of recursion, he thinks of push-down stacks and other aspects of how recursion "works". Similarly, types and type declarations are often described as communications to a compiler to aid it in allocating storage, etc.*
> *The thesis of this dissertation, then, is that these aspects of programming languages can be given an intuitively reasonable semantic interpretation."*

型別與型別系統

進一步延伸模型，納入**型別**的概念。型別是什麼？有什麼作用？

B.C. Pierce, *Types and Programming Languages*, pp. 1-13:

> *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."*

型別以及對應的型別系統

- 是**基於語法**的**靜態**系統
- 對於**程式的每一個片段**，預測、分類其**執行結果**。
- 因為預測了執行結果，所以能提前避免各式錯誤

算術語言的型別系統

$$\begin{array}{rcll} \mathit{Type} & \ni & \tau & ::= \quad \mathbb{N} \mid \mathbb{B} \\ \text{Context} & \ni & \Gamma & ::= \quad (x_1 : \tau_1), \ldots, (x_n : \tau_n) \end{array}$$

算術語言的型別系統是一個自然演繹風格的**證明演算系統**，對每個程式片段做出**判斷** (*judgment*)，並分類為適當的型別，最終整合所有判斷來證明完整程式的執行結果。

此系統的型別有 $\mathbb{N}$ 與 $\mathbb{B}$ 二種，做出的 judgment 形式為

$$\Gamma \vdash e : \tau$$

意指程式片段 e 在 Γ 的環境下型別為 τ （符號「$\vdash$」、「:」是名字的一部分）。各個的 judgments 透過如下示意的 *inference rules* 組織並形成成 *derivation*。

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma_n \vdash e_n : \tau_n}{\Gamma \vdash e : \tau}\ [\textsc{Name}]$$

我們暫時延後環境 Γ 的解釋；Γ 與變數相關。

插曲：自然演繹 (natural deduction)

自然演繹沒有限定 judgment 的格式。在 $\Gamma \vdash e : \tau$ 之外，抽象語法樹、$\to$ 與 r 皆能用相同方法定義。而架構上 judgments 都是透過 inference rules 進行組織的：

$$\frac{J_1 \quad \cdots \quad J_n}{J}\ [\textsc{Name}]$$

在此 inference rule 中，$J_1, \ldots, J_n$ 是其**前提** (*premise*)，J 是其**結論** (*conclusion*)。透過重複應用 inference rules，judgments 會組合成樹狀的 derivation：

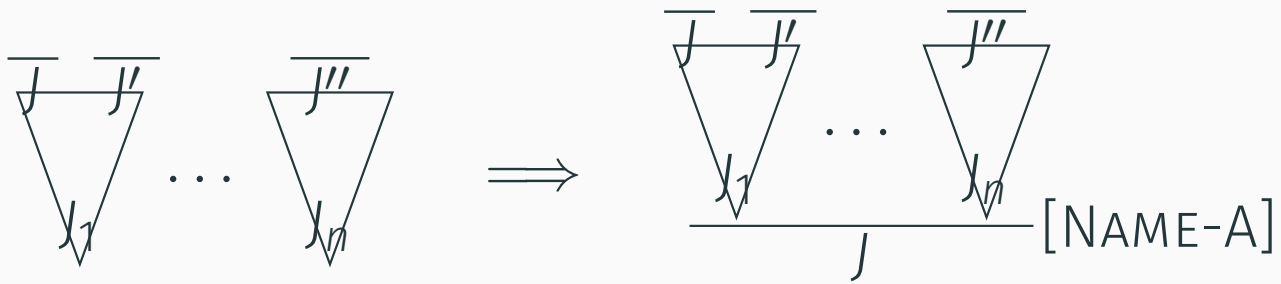
$$\dfrac{\dfrac{\vdots}{J_{1,1}} \quad \cdots \quad \dfrac{}{J_{1,m_1}} \qquad \dfrac{}{J_{n,1,1}} \quad \cdots \quad \dfrac{\vdots}{J_{n,1,k_{n1}}} \qquad \dfrac{}{\cdots \quad J_{n,m_n}}}{\dfrac{J_1 \quad \cdots \qquad\qquad \cdots \quad J_n}{J}}$$

derivation 起始的 judgment 必須沒有前提，即橫線上方為空。

插曲：自然演繹 (natural deduction)

$$\frac{J_1 \quad \dots \quad J_n}{J}\,[\textsc{Name-A}] \qquad \frac{\Gamma \vdash e_1 : \mathbb{N} \qquad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash e_1 + e_2 : \mathbb{N}}\,[\textsc{T-Plus}]$$

inference rule 捕捉了**單步邏輯推導**背後的直覺。當前提 J_1,...,J_n 皆有 derivations 時，[Name-A] 表明我們能據此判斷 J 成立，同時藉由 [Name-A] 將 $J_1, \dots, J_n$ 的 derivations 組合作成 J 的 derivation。



應用於型別系統時，[T-Plus] 描述如何判斷片段 $e_1 + e_2$ 是否具有型別 $\mathbb{N}$。若存在 derivations 證明 e_1、e_2 在環境 Γ 中皆有型別 $\mathbb{N}$，[T-Plus] 便能推導出 $e_1 + e_2$ 在 Γ 下同樣具有型別 $\mathbb{N}$。

算術語言的型別系統：Inference Rules

$$\frac{}{\Gamma \vdash n : \mathbb{N}}\ [\text{T-Nat}]$$

$$\frac{\Gamma \vdash e_1 : \mathbb{N} \qquad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash e_1 + e_2 : \mathbb{N}}\ [\text{T-Plus}]$$

$$\frac{\Gamma \vdash e_1 : \mathbb{N} \qquad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash e_1 \times e_2 : \mathbb{N}}\ [\text{T-Times}]$$

Example

$$\dfrac{\dfrac{\dfrac{}{\vdash 1 : \mathbb{N}}\ [\text{T-Nat}] \qquad \dfrac{}{\vdash 2 : \mathbb{N}}\ [\text{T-Nat}]}{\vdash 1 + 2 : \mathbb{N}}\ [\text{T-Plus}] \qquad \dfrac{}{\vdash 7 : \mathbb{N}}\ [\text{T-Nat}]}{\vdash (1 + 2) \times 7 : \mathbb{N}}\ [\text{T-Times}]$$

Remark. 此處與操作語義類似，符號 e_1、e_2 等指稱任意具體的語法樹。重複的符號代指相同的語法樹。

[T-Nat]、[T-Plus] 與 [T-Times] 皆為樣板。他們界定如何對特定外型的算式（$e_1 + e_2$、$e_1 \times e_2$）進行證明推導，能任意實例化。

算術語言的型別系統：Inference Rules

$$\frac{}{\Gamma \vdash b : \mathbb{B}}\ [\text{T-Bool}] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 = e_2 : \mathbb{B}}\ [\text{T-Equ}]$$

$$\frac{\Gamma \vdash e_1 : \mathbb{B} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ [\text{T-If}]$$

$$\mathit{Type} \quad \ni \quad \tau \quad ::= \quad \mathbb{N} \mid \mathbb{B}$$

Example

$$\dfrac{\dfrac{\dfrac{}{\vdash 7 : \mathbb{N}} \quad \dfrac{}{\vdash 0 : \mathbb{N}}}{\vdash 7 = 0 : \mathbb{B}} \quad \dfrac{}{\vdash \texttt{false} : \mathbb{B}} \quad \dfrac{}{\vdash \texttt{true} : \mathbb{B}}}{\vdash \texttt{if}\ 7 = 0\ \texttt{then false else true} : \mathbb{B}}$$

由於無法預先知道 if 算式的選擇，[T-If] 要求 if 算式的兩個分支型別相同，以此確認 if 算式的運算結果同樣為固定的型別。

在 Judgment 中加入假設：Contexts

型別系統對**每一個程式片段**都預測一個型別，那麼：

- 程式片段 `b*b >= AC` 的型別是什麼？

```
bool has_solution(int a, long b, int c) {
  long AC = 4 * (long)a * (long)c;
  return b*b >= AC;
}
```

- 怎麼準確的指定 `z = 0` 算式的型別？

```
let z = 3 + 5 in
if z = 0 then false else true
```

在型別 judgment $\Gamma \vdash e : \tau$ 當中，$\Gamma := (x_1 : \tau_1), \ldots, (x_n : \tau_n)$ 為程式片段 e 提供了其可見的變數綁定上下文。

算術語言的型別系統：Inference Rules

$$\frac{}{\Gamma \vdash x : \tau}\ (x{:}\tau \in \Gamma)\quad [\text{T-Var}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, (x : \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2}\ [\text{T-Let}]$$

$$\text{Context} \quad \ni \quad \Gamma \quad ::= \quad (x_1 : \tau_1), \ldots, (x_n : \tau_n)$$

對於變數參考，[T-Var] 直接從型別上下文 Γ 中獲取變數的型別。對於 let 算式，[T-Let] 將其動態語義轉譯成推導規則。

考慮算式 $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$。此算式將計算 e_1 的值，並將 x 綁定到 e_1 的計算結果後繼續計算 e_2。

若 e_1 在環境 Γ 下有型別 τ_1，那麼 e_1 的計算結果也將是型別為 τ_1 的值。因此將 x 的型別指定為 τ_1 的環境下，e_2 具有的型別即為 let 算式的型別。

算術語言的型別系統：Inference Rules

$$\frac{}{\Gamma \vdash x : \tau}\ (x : \tau \in \Gamma)\quad [\text{T-Var}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, (x : \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2}\ [\text{T-Let}]$$

Example

令 $e_{\text{not}} := (\texttt{if z = 0 then false else true})$ 且令 D 代表

$$\frac{\dfrac{}{\texttt{z} : \mathbb{N} \vdash \texttt{z} : \mathbb{N}}\ [\text{T-Var}] \qquad \dfrac{}{\texttt{z} : \mathbb{N} \vdash 0 : \mathbb{N}}}{\texttt{z} : \mathbb{N} \vdash \texttt{z = 0} : \mathbb{B}}$$

則

$$\frac{\dfrac{}{\vdash 7 : \mathbb{N}} \qquad \dfrac{D \qquad \dfrac{}{\texttt{z} : \mathbb{N} \vdash \texttt{false} : \mathbb{B}} \qquad \dfrac{}{\texttt{z} : \mathbb{N} \vdash \texttt{true} : \mathbb{B}}}{\texttt{z} : \mathbb{N} \vdash e_{\text{not}} : \mathbb{B}}\ [\text{T-Let}]}{\vdash \texttt{let z = 7 in}\, e_{\text{not}} : \mathbb{B}}$$

結論

程式語言理論研究的框架

數學模型

我們以抽象語法樹及結構操作語義建立了程式語言的數學模型。此數學模型形式化的定義了程式的語法及語義，因此允許對程式本身的計算推理。

我們並介紹了如何擴展對象語言，加入變數相關的語法構造及新的資料型別。在展示過的語法構造之外，相同的框架能容納更多複雜的特性，包含函數、遞迴（自我指涉）、複合型資料型態、控制流結構、控制算子等等。

最後，我們概述了最簡單的靜態分析：型別系統。

證明與分析

由於我們介紹的數學模型皆為歸納定義的，數學模型本身的性質（因此也是程式語言本身的性質）能用結構歸納法加以證明。