Semantica dei linguaggi di programmazione (Sintassi)

Luca Paolini
paolini@di.unito.it
Università di Torino
Dipartimento di Informatica

November 7, 2012

Funzione

Cos'è una funzione?

Siano X, Y insiemi.

- Una relazione da X in Y è un sottoinsieme dell'insieme $X \times Y$, i.e. il prodotto cartesiano tra insiemi.
- Una funzione da X in Y è una relazione da X in Y, totale (ovunque definita) e funzionale.
- Una relazione funzionale è anche chiamata funzione parziale.
- Il sottoinsieme di $X \times Y$ è l'estensione della relazione (funzione) considerata, i.e. la sua descrizione estensionale.
- Un programma che descrive una funzione è una descrizione intensionale della funzione.
- Una breve storia della nozione di funzione è in
  Carlo Marchini. Appunti del corso di Didattica della Matematica I. Technical report, Universita di Parma, 1999.
  `http://www.unipr.it/arpa/urdidmat/SSIS/Marchini/1%B0anno/Funzioni.pdf`
  `http://www.unipr.it/arpa/urdidmat/SSIS/Marchini/1%B0anno/Funzioni.pdf`

Entscheidungsproblem

In mathematics and computer science, the Entscheidungsproblem (German for 'decision problem') is a challenge posed by David Hilbert in 1928. The Entscheidungsproblem asks for an algorithm that takes as input a statement of a logic and answers "Yes" or "No" according to whether the statement is universally valid, i.e., valid in every structure satisfying the logic axioms. Mathematicians have studied algorithms and computation since ancient times, but the modern study of computability began around 1900.

David Hilbert was deeply interested in the foundations of mathematics. Hilbert [1904] proposed proving the consistency of arithmetic by what emerged [1928] as his finitist program.

Hilbert proposed using the finiteness of mathematical proofs in order to establish that contradictions could not be derived. This tended to reduce proofs to manipulation of finite strings of symbols devoid of intuitive meaning which stimulated the development of mechanical processes to accomplish this.
Church's Thesis (First Version) [1934]. A function is effectively calculable if and only if it is lambda-definable.

> Although Kleene was convinced by Church's first thesis, Gödel was not. Gödel rejected Church's first thesis as thoroughly unsatisfactory."

(More historical details in:
Robert Irving Soare. Turing oracle machines, online computing, and three displacements in computability theory. Technical report, University of Chicago, 2009.
`http://arxiv.org/pdf/math.lo/0209332`).

Algoritmo

".. and always come up with the right answer, so God will", Muhammad ibn Musa, Al-Khowarizmi (whose name is at the origin of the words "algorithm" and "algebra").

> Turing's definition of effective procedure follows from an analysis of how a human(!) computer proceeds when excuting an algorithm.

Analizziamo l'idea intuitiva di algoritmo. Un algoritmo viene descritto in un certo linguaggio, che può anche essere semplicemente l'italiano, così come usando i linguaggi nati appositamente per descrivere algoritmi quali i linguaggi di programmazione e vari sistemi formali (e.g. macchine di Turing). Indipendentemente dal problema che intende risolvere, ha delle caratteristiche comuni.

1. Un algoritmo è descritto come una sequenza finita di operazioni.
2. Esiste un agente di calcolo (tipicamente umano) che porta avanti il calcolo eseguendo le istruzioni dell'algoritmo.
3. L'agente di calcolo ha a disposizione una memoria dove vengono immagazzinati i risultati intermedi del calcolo (un foglio di carta).
4. Il calcolo avviene per passi discreti (ognuno terminante in tempo finito).
5. Il calcolo è deterministico (tipicamente, data una situazione esiste un unico modo di avanzare ma a volte basta qualcosa di più debole).
6. L'algoritmo puo essere applicato ad input di qualsiasi dimensione.
7. La quantita di memoria necessaria all'esecuzione dell'algoritmo non deve essere limitata (sebbene, a posteriori, deve essere sempre finita).
8. Deve esserci un limite finito alla complessità delle istruzioni eseguibili dal dispositivo.
9. Sono ammesse solo esecuzioni con un numero di passi finito, sebbene non limitato a priori.

Turing Machine

A Turing Machine (TM for short) M is $(Q, \Sigma, \Gamma, \delta, q_0, \boxdot, F)$ where

- Q is the (finite) set of internal states $\{q_i | i \in \mathbb{N}\}$
- Σ is the input alphabet, Γ is the finite set of symbols called tape alphabet (i.e. $\Sigma \cup \boxdot$)
- $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, S, R\})$ is the transition relation (if $\delta : (Q \times \Gamma) \to Q \times \Gamma \times \{L, S, R\})$ then the TM is deterministic)
- $\boxdot$ is the blank symbol.
- q_0 (is a member of Q) is the initial state
- F (is a subset of Q) is the set of final states[a]
- A tape is a pair of strings w_L and w_R such that $w_L \in \boxdot^\infty \Gamma^*$ and $w_R \in \Gamma^* \boxdot^\infty$.
- $h \in \Gamma$ is the head of the tape whenever is the righmost symbol of w_L.
- A configuration is a triple in $Q \times (\boxdot^\infty \Gamma^*) \times (\Gamma^* \boxdot^\infty)$.
- The initial configuration is $\langle q_0, \boxdot^\infty w, \boxdot^\infty \rangle$ where $w \in \Gamma^*$ is the input.
- An final configuration is $\langle q_F, \boxdot^\infty w, \boxdot^\infty \rangle$ where $q_F \in F$ and $w \in \Gamma^*$ is the output.
- A TM-computation is a (finite) sequence of configurations $c_0, \ldots, c_n$ such that, for all $i \in [0, n-1]$, if $c_i = \langle q^i, w_L^i : h, w_R^i \rangle$ then c_{i+1} is obtained by applying the transition rule in the straighforward way,
  i.e. by respecting the information of $\delta(q^i, h) = [q^{i+1}, h^{i+1}, m]$.

[a] one final state is sufficient

Esercizi

JFLAP provides a Turing-machine emulator,
see `http://www.youtube.com/watch?v=IkYhfk4X47c`.

At `http://www.jflap.org/` you can download the emulator and its tutorial.

Esercizi:

- Progettare una macchina che confronta due interi.
- Progettare una macchina che somma due interi.
- Progettare una macchina che riconosce il linguaggio $\{0^{2^n} | n \in \mathbb{N}\}$

Kleene's functions

The class of partial recursive functions is the smallest class of functions containing the initial functions

$$Z(n) = 0,$$
$$S(n) = n + 1,$$
$$U_i^m(n_1, \ldots, n_m) = n_i \text{ for all } 1 \leq i \leq m$$

and closed under

- composition:
  if $g : \mathbb{N}^k \to \mathbb{N}$ and $h_1, \ldots, h_k : \mathbb{N}^m \to \mathbb{N}$ are partial recursive functions, then so is $g(h_1(n_1, \ldots, n_m), \ldots, h_k(n_1, \ldots, n_m))$;

- primitive recursion:
  if $g : \mathbb{N}^m \to \mathbb{N}$ and $h : \mathbb{N}^{m+2} \to \mathbb{N}$ are partial recursive functions, then so is $f : \mathbb{N}^{m+1} \to \mathbb{N}$ defined by
  $$f(0, n_1, \ldots, n_m) = g(n_1, \ldots, n_m),$$
  $$f(n+1, n_1, \ldots, n_m) = h(f(n, n_1, \ldots, n_m), n, n_1, \ldots, n_m);$$

- minimalization:
  if $g : \mathbb{N}^{m+1} \to \mathbb{N}$ is a partial recursive function,
  then so is $\min x[g(x, n_1, \ldots, n_m) = 0] : \mathbb{N}^m \to \mathbb{N}$ denoting the smallest $k \in \mathbb{N}$, if it exists, such that the equation
  $$g(k, n_1, \ldots, n_m) = 0$$
  is satisfied and $\forall k' \leq n$, $g(k', n_1, \ldots, n_m)$ is defined.

Primitive ricorsive

- Le funzioni di Kleene che non fanno uso della minimalizzazione sono totali, i.e. terminano in un numero di passi finito. Queste funzioni sono chiamate primitive ricorsive.
- Le stragrande maggioranza dei programmi che scriviamo corrispondono a funzioni primitive ricorsive.
- Non tutte le funzioni totali e calcolabili sono primitive ricorsive.
- Ad esempio, la funzione di Ackermann è una funzione calcolabile e totale che cresce più velocemente di qualsiasi funzione ricorsiva primitiva.

  La funzione di Ackermann $A(m, n)$ è definita per ricorrenza nel seguente modo:

  - se $m = 0$ allora $A(0, n)$ vale $n + 1$
  - se $m > 0$ ed $n = 0$ allora $A(m, 0) := A(m - 1, 1)$
  - se $m > 0$ ed $n > 0$ allora $A(m, n) := A(m - 1, A(m, n - 1))$
- More details about Ackermann functions can be found at `http://xlinux.nist.gov/dads/HTML/ackermann.html`.
- Tutte le funzioni di Kleene sono calcolate in maniera sequenziale, come anche quelle definite dalla macchina di Turing.

Osservazioni

- Le funzioni di Kleene mandano numeri naturali in numeri naturali.
- Le n-ple di naturali ed i naturali sono in biiezione.
- Le macchine di Turing permettono di calcolare esattamente le stesse funzioni caratterizzate da Kleene.
- C, C++, Java, Lisp, Prolog, ... sono Turing-completi.
- I computer sono Turing-completi.
- Il lambda-calcolo è Turing-completo.
- Le funzioni di Kleene sono definite attraverso l'introduzione di un semplice linguaggio di programmazione. Infatti, possiamo scrivere infiniti programmi di Kleene che calcolano la stessa funzione di Kleene.
- I "programmi di Kleene" e le "macchine di Turing" sono descrizioni intensionali delle funzioni di Kleene.
- I "programmi di Kleene" e le "macchine di Turing" sono descrizioni sequenziali delle funzioni di Kleene.
- Le macchine di Turing non sono facilmente componibili, mentre le programmi di Kleene dispongono di un naturale operatore adatto all'uopo.
- Gli operatori (composizione, minimalizzazione e ricorsione) non sono "programmi di Kleene".
- I programmi di Kleene implementano un passaggio dei parametri call-by-value.
- Moralmente, le funzioni (in matematica) sono call-by-value.
- I programmi call-by-value e call-by-name calcolano le stesse funzioni?
- Le funzioni (calcolabili, parziali) sequenziali sono piu di quelle parallele, o viceversa?

Esercizi

- Scrivere un Kleene-programma che confronta due interi.
- Scrivere un Kleene-programma che somma due interi.
- Scrivere un Kleene-programma restituisce 1 se l'input appartiene all'insieme $\{0^{2^n} | n \in \mathbb{N}\}$, e restituisce 0 altrimenti.

Referenze

Sulla calcolabilita potete consultare:

- Agostino Dovier and Roberto Giacobazzi. *Fondamenti dell'Informatica: Linguaggi Formali, Calcolabilità e Complessità*. Università di Udine e Verona, dispense edition, 2012. `http://www.dimi.uniud.it/~dovier/DID/dispensa.pdf`
- Nigel Cutland. *Computability*. Cambridge University Press, Cambridge, 1980. An introduction to recursive function theory
- Hartley Rogers, Jr. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, MA, second edition, 1987

Conclusioni

- La memoria di un comune calcolatore puo essere vista come una stringa finita binaria (se considerate i bit).
- La memoria di un comune calcolatore puo essere vista come una stringa finita 256-aria (se considerate i byte).
- Sia dato un alfabeto finito di k simboli. Possiamo pensare che le stringhe su questo alfabeto sono numeri naturali in base k. Ossia possiamo dare un ordine totale a tali simboli e usarli come cifre.
- I programmi sono memorizzabili in memoria, i.e. sono descrivibili come stringhe finite su un alfabeto finito (ossia sono codificabili sugli interi).
- L'esecuzione di un programma trasforma la memoria (che contiene il programma stesso e l'input) da una situazione iniziale ad una finale (se eventualmente termina).
- In definitiva i programmi descrivono funzioni parziali da numeri interi in numeri interi.

λ-notation

- A notation for expressions and equations was not available until the 17th century, when Francois Viète started to make systematic use of placeholders for parameters and abbreviations for the arithmetic operations. Until then, a simple expression such as 3x2 had to be described by spelling out it in words.
- Viète's notation for expressions was the main innovation in FORTRAN, the world's first high-level programming language (Backus 1953), thus liberating the programmer from writing out tedious sequences of assembly instructions.
- To clarify notation for mathematical functions we can adopt the lambda notation. In mathematics, function formation is sometimes written as an equation, $f(x) = 3x$, sometimes as a mapping $x \mapsto 3x$.
- Patently: $f(x, y) = 3x + y$ and $f(y, x) = 3y + x$ are the same function, $\int_a^b f(x)\,dx = \int_a^b f(z)\,dz$, $\forall n, \sum_{k=1}^{n} k = \sum_{j=1}^{n} j$
- Likewise, $f(x, z) = 3z + z$ and $f(z, y) = 3z + z$ are different, $\int_a^b f(y)\,dx \neq \int_a^b f(z)\,dz$, $\forall n, \sum_{k=1}^{n} n \neq \sum_{j=1}^{n} j$
- Above we can see 4 kinds of common binders.

λ-notation

- Binders are common also in modern programming languages supporting a structured programming paradigm.

For instance, in Pascal

```
function f( x : int) : int
begin
      f := 3 * x
end;
```

and in Lisp or Scheme:

```
(defun factorial (n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
(lambda (x) (* 3 x))
```

Referenze

- Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, January 1964
- Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I and Part II. *Communications of the ACM*, 8(2-3):89–101,158–165, 1965
- Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966

Content Landin's paper introduces the lambda calculus as a basis for defining a programming language; he dubs the language ISWIM. The paper presents the syntax of the language without regard to details and a mechanical evaluator, a so-called machine, that in a step-wise fashion produces an answer for the programs in the language. The goal is to use this novel language to explain other languages.

Ideas The paper contains many important ideas. First, it introduces the notion of abstract syntax, i.e., that only certain elements of a phrase matter and the rest can be ignored. Second, it is one of the first papers to spell out the meaning of a language in a concise mathematical manner. (Bauer, McCarthy, and Böhm have alternative proposals, though none of them survived.) Third, the paper presents the idea that one can use this novel language as a meta-language to explain other languages. Fourth, Landin adds imperative constructs to the lambda calculus, anticipating mostly functional programming à la Scheme and ML. (Burge (IBM) later implemented the machine in the 1970s but few noticed.)

- Corrado Böhm. The CUCH as a formal and description language. In Jr. T. B. Steel, editor, *Formal Language Description Languages for Computer Programming*, pages 179–197. North-Holland Co., 1966
- Corrado Böhm and W. Gross. Introduction to the CUCH. In E. R. Caianiello, editor, *Automata Theory*, pages 35–65. Academic Press, New York, 1966
- Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966
- Donald E. Knuth. Structured programming with go to statements. *ACM Computing Survey*, 6(4):261–301, December 1974

Linguaggi Moderni

- L'introduzione dei binders nei linguaggi di programmazione è ispirata dal lambda-calcolo, essi vengono introdotti con ALGOL e LISP. Il primo è un linguaggio imperativo (i.e. basato sui comandi, istruzioni per la macchina), mentre il secondo è un linguaggio funzionale (i.e. basato sulla descrizione di funzioni e privo di assegnmento).
- L'esecuzione di linguaggi con binders richiede una allocazione non statica della memoria (ma automatica, i.e. non stiamo parlando di allocazione on-demand). Vengono introdotti a tal fine i record di attivazione.
- I linguaggi precedenti a questa rivoluzione (Assembly, COBOL, FORTRAN) allocavano la memoria per le variabili in maniera statica.

Renaming

- I λ-termini sono definiti con tre sole regole di formazione sintattica:

$$M, N ::= x | \lambda x.M | MN$$

  dove x è una generica variabile, $\lambda x.M$ è chiamata λ-astrazione (M è il corpo dell'astrazione) e MN è chiamata applicazione.
- La lambda-astrazione è un binder!
  L'astrazione $\lambda x.M$ lega la variabile x nel corpo dell'astrazione,
  i.e. il corpo dell'astrazione è lo scope del binder.
- Le variabili libere di un termine sono definite come segue:

$$\begin{array}{lcl} \mathrm{FV}(x) & := & \{x\} \\ \mathrm{FV}(\lambda x.M) & := & \mathrm{FV}(M) - \{x\} \\ \mathrm{FV}(MN) & := & \mathrm{FV}(M) \cup \mathrm{FV}(N) \end{array}$$

- Le variabili non libere sono legate.
- Le variabili legate possono essere rinominate, purchè si eviti la cattura di variabili libere. Questa operazione è chiamata α-rinomina.
- Ad esempio, $\lambda x.xyy =_\alpha \lambda z.zyy$ ed anche $\lambda xy.xyy =_\alpha \lambda yx.yxx$, tuttavia $\lambda x.xyy \neq_\alpha \lambda y.yyy$ e $\lambda x.xyy \neq_\alpha \lambda x.xzz$.

λ-calculus

- Il lambda-calcolo è ottenuto aggiungendo alla sintassi appena definita una unica regola di calcolo, la β-regola:

$$(\lambda x.M)N \to_\beta M[N/x]$$

  dove $M[N/x]$ denota la sostituzione di N a tutte le occorrenze libere di x in N evitando la cattura delle variabili libere in N.
  La chiusura per contesti della β-regola è chiamata β-riduzione.

- La regola di calcolo è chiusa per contesti. La sua chiusura riflessiva e transitiva (i.e. zero o piu passi di riduzioni) è indicata con $\to_\beta^*$. Infine, la sua chiusura riflessiva, simmetrica e transitiva è indicata con $=_\beta$.

- La sostituzione è sempre possibile dopo aver rinominato le variabili legate di N, e.g. $(\lambda xy.xy)y \to_\beta \lambda z.yz$.

- Talvolta, La sintassi viene costruita considerando non solo delle variabili, ma anche delle costanti. Per esempio possiamo aggiungere numeri assieme ad i rispettivi comuni operatori su di essi $(1, 2, 3, ..., +, *, ...)$. Potremmo aggiungere stringhe e loro operatori, booleani e condizionali. Il linguaggio ISWIM di Landin è un esempio.

- Gli operatori diventano tali fornendo le rispettive regole di calcolo, e.g.

$$\mathtt{succ}\, 3 \to_{\mathsf{sum}} 4$$

$$\mathtt{if}\, \mathtt{tt} M N \to_{\mathsf{if}} M$$

- Queste estensioni del lambda-calcolo sono talvolta chiamati $\lambda\beta\delta$-calcoli. Il loro principale problema è che ci sono termini mal-formati che vorremmo evitare, e.g. $+ +$ `if` .

Osservazioni

- Alcuni termini del lambda calcolo hanno un nome convenzionale:

$$
\begin{array}{ll}
I = \lambda x.x & (identity) \\
\mathbb{1} = \lambda xy.xy & \\
\Delta = \lambda x.xx & (duplicator) \\
T = \lambda x.xxx & \\
\Omega = \Delta\Delta & (loop\_forever) \\
S = \lambda xyz.xz(yz) & \\
K = \lambda xy.x & \\
O = \lambda xy.y &
\end{array}
$$

- Diciamo che un termine ha forma normale ogniqualvolta non pou essere β-ridotto (ossia non contiene β-redessi).
- Rozzamente possiamo considerare le forme normale come i risultati della nostra computazione. Tra i termini prima definiti abbiamo solo forme normali, eccetto uno: da tale termine non ha forme normale, o se preferita cicla indefinitamente.
- Il lambda calcolo puro (senza costanti) è Turing-completo.
- Alcuni ulteriori termini dotati di nome sono gli operatori di punto fisso. Sia data una funzione f. Chiamiamo punto fisso di f ogni suo argomento x tale che $f(x) = x$.
- Il lambda calcolo contiene infiniti termini che calcolano il punto fisso delle lambda-funzioni.
- Il più famoso operatore di punto fisso è

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

  Lemma $YM =_\beta M(YM)$, per ogni λ-termine M.

- Un altro operatore di punto fisso è

$$T = (\lambda xy.y(xxy)(\lambda xy.y(xxy)$$

  Lemma $ZM \to_\beta M(ZM)$, per ogni λ-termine M.

- Gli operatori di punto fisso non hanno forma normale.

Referenze

- Una breve e ragionevole introduzione al lambda-calcolo.

  Achim Jung. A short introduction to the lambda calculus. Technical report, School of Computer Science, The University of Birmingham, 2004.
  http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf

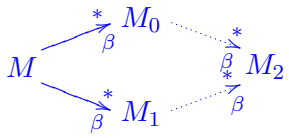
- Una analisi storica.

  J. Roger Hindley and Felice Cardone. History of lambda-calculus and combinatory logic. In D. Gabbay and J. Woods, editors, *Handbook of the History of Logic*, pages 723–817. Elsevier, 2009. http://www.di.unito.it/~felice/Cardone-pubs/lambdacomb.pdf

Fare gli esercizi a pagina 8 di
http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf.

Theorems

Confluence or Church-Rosser, If a term M can be reduced (in several steps) to terms M_0 and M_1, then there exists a term M_2 to which both M_0 and M_1 can be reduced (in several steps).



Corollary. Every λ-term has at most one normal form.

Standardization. To state and prove this theorem requires many formal tools, so we state just its main consequence.

Corollary. If a term has β-normal form then we can obtain it, by reducing at every step the leftmost redex (counting symbols on the left of the considered redex).

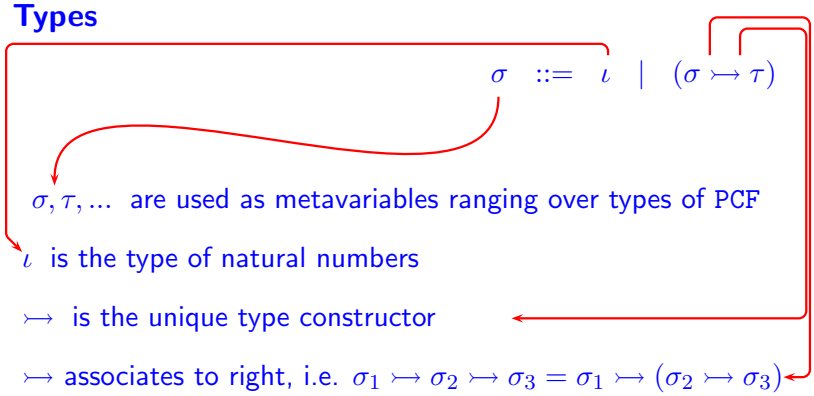
Separability (Böhm). If two term M, N are different $\beta\eta$-normal forms then there exists a context $C[.]$ and two different variables x, y such that $C[M] \to_\beta^* x$ and $C[N] \to_\beta^* y$.

- Contexts are defined by $M, N ::= [.]|x|\lambda x.M|MN$.
- $C[M]$ denotes the replacement of M to all occurrences of $[.]$ in $C[.]$ *allowing the capture of free variable in* M.
- If M is a term and x is a variable not in M then $M =_\eta \lambda x.Mx$.

  It is easy to prove that η-equivalence is the minimal extension of $=_\beta$ that makes the calculus extensional, i.e. in the sense that

$$\frac{\forall M \in \mathsf{Var}, \quad Mx = Nx}{M = N} \ (ext)$$

Types



Exercise 1

It is easy to see that all types are of the shape

$$\tau_1 \rightarrowtail \ldots \rightarrowtail \tau_n \rightarrowtail \iota,$$

for some types $\tau_1, \ldots, \tau_n$ where $n \geq 0$.

Types remarks

The type ι is called the ground type
and types $\sigma \rightarrowtail \tau$ are called arrow types.

In literature, often a type constant o representing booleans is added to type-syntax, for an explicit treatment of truth-values.

An introduction to Simple Typed λ-calculus can be found in J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1997.

Sometimes, we will write $\mathtt{M} : \sigma$ in order to say that σ is the type of $\mathtt{M}$.

A typed λ-calculus

$$\mathtt{M}^\sigma \quad ::= \quad \mathtt{x}^\sigma \quad | \quad \mathtt{const}^\sigma \quad | \quad (\lambda \mathtt{x}^\mu.\mathtt{N}^\tau)^{\mu\rightarrowtail\tau} \quad | \quad (\mathtt{P}^{\tau\rightarrowtail\sigma} Q^\tau)^\sigma$$

$\mathtt{M}^\sigma, \mathtt{N}^\sigma, \mathtt{P}^\sigma, \mathtt{Q}^\sigma, \ldots$ are metavariables ranging over typed terms

Var^σ is the set of variables of type σ and $\mathtt{x}^\sigma \in \mathrm{Var}^\sigma$
while $\mathtt{const}^\sigma$ is a metavariable for constant symbols

$(\lambda \mathtt{x}^\mu.\mathtt{N}^\tau)^{\mu\rightarrowtail\tau}$ is an abstraction,
and $(\mathtt{P}^{\tau\rightarrowtail\sigma}\mathtt{Q}^\tau)^\sigma$ is an application

Sometimes parentheses are omitted,
by respecting the following disambiguating conventions

- application associates to the left
- application binds more tightly than abstraction

Types of variables/constants/subterms/terms will be omitted when they are clear from the context or uninteresting

Exercise 2

Given types of all variables of a term $\mathtt{M}$,
there is a unique σ such that $\mathtt{M}^\sigma$.

A typed λ-calculus

If $\lambda \mathtt{x}^\sigma.\mathtt{M}^\tau$ is an abstraction then the variable $\mathtt{x}^\sigma$ is the formal parameter and $\mathtt{M}^\tau$ is the body of the abstraction.

A term of the shape $(\lambda \mathtt{x}^\sigma.\mathtt{M}^\tau)\mathtt{N}^\sigma$ is a redex, informally it is a "program" such that $\mathtt{N}^\sigma$ is the argument of the function $\lambda \mathtt{x}^\sigma.\mathtt{M}^\tau$.

The language without constants is said pure.

Terms of the language $\mathtt{M} ::= \mathtt{x} \mid \mathtt{const} \mid (\lambda \mathtt{x}.\mathtt{N}) \mid (\mathtt{PQ})$ are called untyped.
By forgetting all types of our terms we obtain an untyped term,
is the converse true?

In literature, often PCF is presented by using a type assignment system.

Example 3

$\mathtt{PQR}$ should be parsed as $(\mathtt{PQ})\mathtt{R}$, while $\lambda \mathtt{x}^\sigma.\mathtt{MN}$ should be parsed as $\lambda \mathtt{x}^\sigma.(\mathtt{MN})$.

Silently, we will avoid the use of the same name for two variables with different type.
Moreover, $\lambda \mathtt{x}^\sigma \mathtt{y}^\tau \mathtt{z}^\mu.\mathtt{M}$ will be used as an abbreviation for $\lambda \mathtt{x}^\sigma.\lambda \mathtt{y}^\tau.\lambda \mathtt{z}^\mu.\mathtt{M}$.

Examples of Constants

- $\tilde{0}, \tilde{1}, \tilde{2}, \ldots\ldots$ are examples of constants that we can add to our language, all with type ι
- $\mathtt{succ}$, $\mathtt{pred}$ having type $\iota \rightarrowtail \iota$ are further examples
- $\mathtt{or}$ having type $\iota \rightarrowtail \iota \rightarrowtail \iota$ is a further example
- $\mathtt{if}$ having type $\iota \rightarrowtail \iota \rightarrowtail \iota \rightarrowtail \iota$ is a further example

Note that the meaning of constant is not defined!

We can suggest the evaluation of the language by a evaluation relation $\Downarrow_N$ between terms, for example:

$$\frac{}{\tilde{n} \Downarrow_N \tilde{n}}\ (num)$$

$$\frac{\mathtt{M} \Downarrow_N \tilde{n}}{\mathtt{succ}\,\mathtt{M} \Downarrow_N \widetilde{n+1}}\ (\mathtt{succ})$$

$$\frac{\mathtt{M} \Downarrow_N \widetilde{n+1}}{\mathtt{pred}\,\mathtt{M} \Downarrow_N \tilde{n}}\ (\mathtt{pred}) \qquad \frac{\mathtt{M} \Downarrow_N \tilde{0}}{\mathtt{pred}\,\mathtt{M} \Downarrow_N ???}\ (\mathtt{pred})$$

$$\frac{\mathtt{M}_0 \Downarrow_N \tilde{0} \quad \mathtt{M}_1 \Downarrow_N \tilde{n}}{\mathtt{or}\,\mathtt{M}_0\,\mathtt{M}_1 \Downarrow_N \tilde{0}}\ (\mathtt{or}) \qquad \frac{\mathtt{M}_0 \Downarrow_N \tilde{n} \quad \mathtt{M}_1 \Downarrow_N \tilde{0}}{\mathtt{or}\,\mathtt{M}_0\,\mathtt{M}_1 \Downarrow_N \tilde{0}}\ (\mathtt{or})$$

$$\frac{\mathtt{M}_0 \Downarrow_N \widetilde{n+1} \quad \mathtt{M}_1 \Downarrow_N \widetilde{m+1}}{\mathtt{or}\,\mathtt{M}_0\,\mathtt{M}_1 \Downarrow_N \tilde{1}}\ (\mathtt{or})$$

$$\frac{\mathtt{M}_0 \Downarrow_N \tilde{0} \quad \mathtt{M}_1 \Downarrow_N \tilde{n}}{\mathtt{if}\ \mathtt{M}_0\,\mathtt{M}_1\,\mathtt{M}_2 \Downarrow_N \tilde{n}}\ (\mathtt{0if}) \qquad \frac{\mathtt{M}_0 \Downarrow_N \widetilde{k+1} \quad \mathtt{M}_2 \Downarrow_N \tilde{n}}{\mathtt{if}\ \mathtt{M}_0\,\mathtt{M}_1\,\mathtt{M}_2 \Downarrow_N \tilde{n}}\ (\mathtt{1if})$$

Examples of Constants

Three acceptable solutions for the meaning of `pred`:

- No rule for the case $\mathtt{M} \Downarrow_N \tilde{0}$
- An equivalent more explicit solution is the use of the rule

$$\frac{\mathtt{M} \Downarrow_N \tilde{0} \qquad \mathtt{pred\,M} \Downarrow_N \mathtt{N}}{\mathtt{pred\,M} \Downarrow_N \mathtt{N}}\ (\mathtt{pred})$$

- A last, different solution is the use of the rule

$$\frac{\mathtt{M} \Downarrow_N \tilde{0}}{\mathtt{pred\,M} \Downarrow_N \tilde{0}}\ (\mathtt{pred})$$

The first two choices make "partial" the behaviour of `pred`, while the third makes it "total". For laziness, we choose the first solution!

Programming Examples

A (useless) program computing the sum of natural numbers can be wrote in Scheme as follows:

```
(define (sum x y)
           (+ x y))
```

where + is a built-in operator. The previous code can be curryfied (elimination of n-ary argument functions) in the following way

```
(define sum
        (lambda (x)
           (lambda (y)
                (+ x y))))
```

Note that sum is simply a name for the following function

```
(lambda (x)
    (lambda (y)
         (+ x y)))
```

Programming Examples

Note also that the formal parameter of the program can be renamed without changing its meaning

```
(lambda (n)
    (lambda (m)
        (+ n m)))
```

But clashes (name collisions) must be avoided,
in fact the following program is different from the previous one

```
(lambda (n)
    (lambda (n)
        (+ n n)))
```


A Programming Examples

A beautiful introduction to Scheme (a dialect of LISP) and functional programming can be found in Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, 1985.
`http://mitpress.mit.edu/sicp/full-text/book/book.html`

Free Variables

Definition 4

The set of free variables of a term $\mathtt{M}$, denoted by $\mathrm{FV}(\mathtt{M})$, is inductively defined as follows:

- $\mathtt{M} = \mathtt{x}^\sigma$ implies $\mathrm{FV}(\mathtt{M}) = \{\mathtt{x}^\sigma\}$,
- $\mathtt{M} = \mathtt{const}^\sigma$ implies $\mathrm{FV}(\mathtt{M}) = \emptyset$,
- $\mathtt{M} = \lambda\mathtt{x}^\sigma.\mathtt{M}'$ implies $\mathrm{FV}(\mathtt{M}) = \mathrm{FV}(\mathtt{M}') - \{\mathtt{x}^\sigma\}$,
- $\mathtt{M} = \mathtt{PQ}$ implies $\mathrm{FV}(\mathtt{M}) = \mathrm{FV}(\mathtt{P}) \cup \mathrm{FV}(\mathtt{Q})$.

A variable is bound in M when it is not free in M.
Note that the λ-abstraction is the only binder of our language.

A term $\mathtt{M}$ is closed if and only if $\mathrm{FV}(\mathtt{M}) = \emptyset$,
otherwise $\mathtt{M}$ is said to be open.

Substitution

$\mathtt{M}^\tau[\mathtt{N}^\sigma/\mathtt{x}^\sigma]$ denotes the capture-free substitution of all free occurrences of $\mathtt{x}^\sigma$ in $\mathtt{M}^\tau$ by $\mathtt{N}^\sigma$. More formally,

Definition 5

The substitution $\mathtt{M}[\mathtt{N}/\mathtt{x}]$ is a term α-equivalent to the the context-substitution $\mathtt{P}\{\mathtt{N}/\mathtt{x}\}$ term where $\mathtt{P}$ is a term α-equivalent to $\mathtt{M}$ such that no bound variable of $\mathtt{P}$ is in $\mathrm{FV}(\mathtt{N})$.

The notion of substitution is crucial in order to formalize the main evaluation rule of our language, the β-reduction.

Substitution

The Substitution of $\mathtt{M}$ to $\mathtt{x}$ is defined only when $\mathtt{M}$ and $\mathtt{x}$ have the same type.

Example 6

- $(\lambda\mathtt{x}^\sigma\mathtt{y}^\tau.\mathtt{uxy})[\mathtt{z}^\tau/\mathtt{x}^\tau] = \lambda\mathtt{x}^\sigma\mathtt{y}^\tau.\mathtt{uxy}$
- $(\lambda\mathtt{x}^\sigma\mathtt{y}^\tau.\mathtt{uxz})[\mathtt{y}^\tau/\mathtt{z}^\tau] = \lambda\mathtt{x}^\sigma\mathtt{w}^\tau.\mathtt{uxy}$
- $((\lambda\mathtt{x}^\sigma\mathtt{y}^\tau.\mathtt{uxy})\mathtt{x})[\mathtt{z}^\tau/\mathtt{x}^\tau] = (\lambda\mathtt{w}^\sigma\mathtt{y}^\tau.\mathtt{uwy})\mathtt{z}$

Parameter Passing Mechanism

Exercise 7

It is easy to check that every term $\mathtt{M}^\sigma$ has the following shape:

$$\lambda \mathtt{x}_1 ... \mathtt{x}_n . \zeta \mathtt{M}_1 ... \mathtt{M}_m \qquad (\mathtt{n}, \mathtt{m} \geq 0),$$

where $\mathtt{M}_i \in \Lambda$ are the arguments of $\mathtt{M}^\sigma$ $(1 \leq i \leq m)$
and ζ is the head of $\mathtt{M}^\sigma$.

Note that ζ is

- either a variable,
- or a constant,
- or an application of the shape $(\lambda \mathtt{z}.\mathtt{P})\mathtt{Q}$ called head-redex.

The main evaluation relation $\Downarrow_N$ of our language will be

$$\frac{\mathtt{P}[\mathtt{Q}/\mathtt{x}]\mathtt{M}_1 ... \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}}{(\lambda \mathtt{x}^\sigma.\mathtt{P})\mathtt{Q}\mathtt{M}_1 ... \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}} \ (head)$$

The rule above implements a call-by-name parameter passing mechanism, since the arguments of abstractions are substituted without being evaluated.

Parameter Passing Mechanism

The rule $(head)$ presented before is often replaced by

$$\frac{\mathtt{M} \Downarrow_N \lambda \mathtt{x}.\mathtt{P} \qquad \mathtt{P}[\mathtt{N}/\mathtt{x}] \Downarrow_N \tilde{\mathtt{n}}}{\mathtt{M}\mathtt{N} \Downarrow_N \tilde{\mathtt{n}}} \ (cbn)$$

and some further rules for the evaluation of non-ground terms.

Usually PCF is presented by giving a type assignment system for an untyped language. The two presentations are equivalent in our perspective.

A different parameter passing mechanism is the call-by-value one

> Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975

Syntax of PCF

PCF formalize the core of
a typed sequential functional programming language.

- Types $\sigma ::= \iota \mid (\sigma \rightarrowtail \tau)$
- Terms $\mathtt{M}^{\sigma} ::= \mathtt{x}^{\sigma} \mid (\lambda \mathtt{x}^{\mu}.\mathtt{N}^{\tau})^{\mu \rightarrowtail \tau} \mid (\mathtt{P}^{\tau \rightarrowtail \sigma} Q^{\tau})^{\sigma} \mid \mathtt{Y}_{\sigma} \mid \mathtt{if} \mid \mathtt{succ} \mid \mathtt{pred} \mid \tilde{\mathtt{n}}$

For each type $(\sigma \rightarrowtail \sigma) \rightarrowtail \sigma$, the constant $\mathtt{Y}_{\sigma}$ is added,
endowed with the evaluation rule:

$$\frac{\mathtt{P}(\mathtt{Y}_{\sigma}\mathtt{P})\mathtt{M}_1 \ldots \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}}{\mathtt{Y}_{\sigma}\mathtt{P}\mathtt{M}_1 \ldots \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}}\ (\mathtt{Y})$$

A closed term of ground type is called program.

Syntax of PCF

PCF has been introduced in

> Gordon D. Plotkin. LCF considerd as a programming language. *Theoretical Computer Science*, 5:225–255, 1977

inspired by the language LCF of Scott

> Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1–2):411–440, 6 December 1993. A Collection of Contributions in Honour of Corrado Böhm on the Occasion of his 70th Birthday. This paper widely circulated in unpublished form since 1969

Note that types of terms/subterms are always superscript (as exponents). Often constants are wrote without its types, since it is implicit.
However, be careful to the index-type of $\mathtt{Y}_{\sigma}$: there are infinite constants $\mathtt{Y}_{\sigma}$ and σ is the minimum information needed to recovery the the type of a specific instance, namely $(\sigma \rightarrowtail \sigma) \rightarrowtail \sigma$.

For example, $(\mathtt{Y}_{\sigma}\mathtt{M}^{\sigma \rightarrowtail \sigma})$ has type σ, for all σ.

The evaluation of PCF can be presented by introducing a strategy on some rewriting rules. These rules satisfy Confluence and Standardization, see

> Gérard Berry and Jean-Jacques Lévy. A survey of some syntactic results in the lambda-calculus. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 552–566. Springer-Verlag, 1979

Overview

- I linguaggi macchina erano univocamente determinati dall'hardaware su cui erano eseguiti, i.e. la loro semantica era precisata dal progetto della macchina.
- Il desiderio di portabilità e maneggevolezza ha tuttavia portato all'introduzione di linguaggi di altro livello.
- I linguaggi assembly erano praticamente uno-uno con quelli macchina, ma permettevano al programmatore di astrarre dal posizionamento delle locazioni di memoria.
- Fortran e COBOL permettevano l'astrazione dei comandi.
  Nonostante la loro semplicita implementativa sulle architetture hardware esistenti, questi linguaggi presentavano gia una serie di problemi rispetto alla portabilità.
- La grande rivoluzione avviene con l'introduzione dei binders, i.e. Algol 60 ed il modello a record di attivazione.
  In realtà anche con il LISP che però all'epoca viene considerato un giocattolo puramente speculativo con nessuna vera prospettiva nell'informatica applicata.
- Il problema della semantica univoca di questi linguaggi si è manifestato anzitutto nella portabilità dei programmi su architetture differenti.
- La semantica deve permettere ad i programmatori di utilizzare il linguaggio.
- La semantica deve supportare i programmatori di compilatori dando loro indicazioni chiare:
  - sia sul comportamento che i costrutti del linguaggio deve fornire
  - sia sulla macchina virtuale che deve essere implementata sulla macchina fisica per poter fornire le feature dei linguaggi di alto livello (ricorsione, eccezioni, private-public metodi e campi, ...).
- Supporto alla verifica del software, per esempio attraverso strumenti per lo studio dell'equivalenza (estensionale) tra programmi sintatticamente (intensionalmente) differenti
- Supporto al confronto tra (interi) linguaggi (non singoli programmi) sintatticamente difficili da confrontare
- Possibilita di valutare i linguaggi nella loro interezza, individuando loro possibili estensioni non esprimibili per mezzo dei costrutti disponibili nel linguaggio (tipicamente nuovi operatori di parallelismo e/o meccanismi in grado di fornire informazioni intensionali sul comportamento del codice in esecuzione come le eccezzioni).
- Strachey è stato l'ideatore di CPL (un predecessore di C) ed è stato il principale promotore della sviluppo della semantica denotazionale: interpretazione in strutture matematiche indipendenti dai linguaggi che trascurando i dettagli sintattici, rendano piu agevole il confronto e lo sviluppo dei linguaggi di programmazione.
- Linguaggi Tipati e categorie: iso di Lawvere-Lambek e CCC
- Proprieta Denotazionali: continuita, stabilita, stabilita-forte, CDS e algoritmi sequenziali
- Semantica dei giochi: semantica basata sulla descrizione dell'interazione con il contesto.
- Caratterizzazione della sequenzialita, della calcolabilita higher-type ...

Riassumendo

- Semantica esemplificativa (informale): adatta ai programmatori, si pensi alla presentazione nei manuali e testi introduttivi ai linguaggi.
- Semantica operazionale: adatta agli implementatori di compilatori, esistono una serie di testi introduttivi.
  In effetti la semantica operazionale puo essere formalizzata a vari livelli tra astrazione e concretezza.
- Semantica denotazionale: dovrebbe supportare la verifica del software, ma permette anche di confrontare linguaggi molto differenti nella sintassi e di studiare la higher-order computability.

Operational Evaluation of PCF

Let $\Downarrow_N$ be the evaluation relation associating a program $\mathtt{M}$ to a numeral $\tilde{\mathtt{n}}$ whenever a judgment of the shape $\mathtt{M} \Downarrow_N \tilde{\mathtt{n}}$ can be proved by rules of the formal system:

$$\frac{\mathtt{P}[\mathtt{Q}/\mathtt{x}]\mathtt{M}_1 \ldots \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}}{(\lambda \mathtt{x}^{\sigma}.\mathtt{P})\mathtt{Q}\mathtt{M}_1 \ldots \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}}\ (head) \qquad \frac{\mathtt{P}(\mathtt{Y}_{\sigma}\mathtt{P})\mathtt{M}_1 \ldots \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}}{\mathtt{Y}_{\sigma}\mathtt{P}\mathtt{M}_1 \ldots \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}}\ (\mathtt{Y})$$

$$\frac{\mathtt{M}_0 \Downarrow_N \tilde{0} \quad \mathtt{M}_1 \Downarrow_N \tilde{\mathtt{n}}}{\mathtt{if}\ \mathtt{M}_0\, \mathtt{M}_1\, \mathtt{M}_2 \Downarrow_N \tilde{\mathtt{n}}}\ (\mathtt{0if}) \qquad \frac{\mathtt{M}_0 \Downarrow_N \widetilde{\mathtt{k+1}} \quad \mathtt{M}_2 \Downarrow_N \tilde{\mathtt{n}}}{\mathtt{if}\ \mathtt{M}_0\, \mathtt{M}_1\, \mathtt{M}_2 \Downarrow_N \tilde{\mathtt{n}}}\ (\mathtt{1if})$$

$$\frac{\mathtt{M} \Downarrow_N \widetilde{\mathtt{n+1}}}{\mathtt{pred}\,\mathtt{M} \Downarrow_N \tilde{\mathtt{n}}}\ (\mathtt{pred}) \qquad \frac{\mathtt{M} \Downarrow_N \tilde{\mathtt{n}}}{\mathtt{succ}\,\mathtt{M} \Downarrow_N \widetilde{\mathtt{n+1}}}\ (\mathtt{succ}) \qquad \frac{}{\tilde{\mathtt{n}} \Downarrow_N \tilde{\mathtt{n}}}\ (num)$$

If there is a numeral $\tilde{\mathtt{n}}$ such that $\mathtt{M} \Downarrow_N \tilde{\mathtt{n}}$ then we write $\mathtt{M} \Downarrow_N$, otherwise we write $\mathtt{M} \Uparrow_e$.

Operational Semantics of PCF

Summarizing, the relation $\Downarrow_N$ implements a call-by-name parameter passing mechanism, since the arguments of abstractions are substituted without being evaluated. It implements a weak (or lazy) evaluation strategy, since no evaluation is done under λ-abstractions.

$\Downarrow_N$ is the abstract evaluation machine of our programming language.

Clearly, $\widetilde{\mathtt{n+1}}$ is simple metanotation for a integer constants different from $\tilde{0}$.

Exercise 8

- Show that $(\lambda \mathtt{x}^{\iota}\mathtt{y}^{\iota}.\mathtt{succ}\,\mathtt{x}^{\iota})\tilde{9}\tilde{1} \Downarrow_N \tilde{10}$.
- Show that $(\lambda \mathtt{x}^{\iota}.\mathtt{pred}\,\mathtt{x}^{\iota})\tilde{0} \Uparrow_e$, $\mathtt{Y}_{\iota}(\lambda \mathtt{x}^{\iota}.\mathtt{x}^{\iota}) \Uparrow_e$ and $\mathtt{Y}_{\iota}\mathtt{succ} \Uparrow_e$.

Recursive Programming

For instance, in order to calculate the factorial,
we can start by writing its recursive definition:

$$\begin{array}{lcl} \mathtt{f}^{\iota\to\iota}\,\mathtt{x}^{\iota} & = & \mathtt{if}\ \ \mathtt{x}^{\iota}\ \ \tilde{1}\ \ (\mathtt{mult}\,\mathtt{x}^{\iota}(\mathtt{f}^{\iota\to\iota}(\mathtt{pred}\,\mathtt{x}^{\iota}))) \\ \mathtt{f}^{\iota\to\iota} & = & \lambda\mathtt{x}^{\iota}.\mathtt{if}\ \ \mathtt{x}^{\iota}\ \ \tilde{1}\ \ (\mathtt{mult}\,\mathtt{x}^{\iota}(\mathtt{f}^{\iota\to\iota}(\mathtt{pred}\,\mathtt{x}^{\iota}))) \\ \mathtt{f}^{\iota\to\iota} & = & (\lambda\mathtt{t}.\lambda\mathtt{x}^{\iota}.\mathtt{if}\ \ \mathtt{x}^{\iota}\ \ \tilde{1}\ \ (\mathtt{mult}\,\mathtt{x}^{\iota}(\mathtt{t}(\mathtt{pred}\,\mathtt{x}^{\iota}))))\,\mathtt{f}^{\iota\to\iota} \end{array}$$

Note that $\mathtt{F}^{(\iota\to\iota)\to\iota\to\iota}$ is not recursive.

The fixpoint of $\mathtt{F}^{(\iota\to\iota)\to\iota\to\iota}$ is the desired program,
thus $\mathtt{Y}_{\iota\to\iota}\mathtt{F}^{(\iota\to\iota)}$ is a program calculating the factorial

Exercise 9

Prove that $(\mathtt{Y}_{\iota\to\iota}\mathtt{F}^{(\iota\to\iota)\to\iota\to\iota})\,\tilde{\mathtt{n}} \Downarrow_{\mathtt{N}} \tilde{\mathtt{m}}$
whenever $m \in \mathbb{N}$ is the factorial of $n \in \mathbb{N}$.

A program computing the factorial of a natural number can be wrote in Scheme as follows:

```
(define fact (lambda (x)
                (if (n=0)
                    1
                    (* n  fact (- n 1)))))
```


Recursive Programming

Exercise 10

Write a PCF-term M calculating the sum. Yet, give operational evaluations of a constant calculating the sum.

Exercise 11

Write a PCF-term M calculating the multiplication. Yet, give operational evaluations of a constant calculating the multiplication, with both "sequential" and "parallel" flavour! (Hint: a multiplication for zero give back zero?). Can both be simulated in PCF?

Exercise 12

Write a PCF-term M calculating the Fibonacci of an integer. Recall that $\mathtt{Fib}(0) = \mathtt{Fib}(1) = 1$ and if $\mathtt{n} > 1$ then $\mathtt{Fib}(\mathtt{n}) = \mathtt{Fib}(\mathtt{n}-1) + \mathtt{Fib}(\mathtt{n}-2)$.

Operational Theories

Definition 13

Let $\mathtt{M}^\iota$ be a term with at most a free variable (eventually there are multiple occurrences of it in $\mathtt{M}^\iota$), i.e. $\mathrm{FV}(\mathtt{M}^\iota) = \{\mathtt{x}^\tau\}$. The term $\mathtt{M}^\iota$ is a τ-context and noted $\mathtt{C}[\tau]$.

Moreover $\mathtt{C}[\mathtt{N}^\tau]$ denotes the term $\mathtt{M}^\iota\{\mathtt{N}^\tau/\mathtt{x}^\tau\}$ obtained by context-substitution of $\mathtt{N}^\tau$ to all free occurrences of $\mathtt{x}^\tau$.

Definition 14

Suppose $\mathtt{M}^\sigma, \mathtt{N}^\sigma$ be terms.

1. $\mathtt{M} \lessapprox_\sigma \mathtt{N}$ whenever
   if $C[\mathtt{M}] \Downarrow_N \tilde{\mathtt{n}}$ for some numeral $\tilde{\mathtt{n}}$ then $C[\mathtt{N}] \Downarrow_N \tilde{\mathtt{n}}$,
   for all contexts $\mathtt{C}[\sigma]$ s.t. $\mathrm{FV}(C[\mathtt{M}^\sigma]) = \mathrm{FV}(C[\mathtt{N}^\sigma]) = \emptyset$.
2. $\mathtt{M} \approx_\sigma \mathtt{N}$ if and only if $\mathtt{M} \lessapprox_\sigma \mathtt{N}$ and $\mathtt{N} \lessapprox_\sigma \mathtt{M}$.

Operational Theories

Usually, contexts are introduced in different way equivalent to that above, by presenting a grammar for them (essentially the same grammar of the language extended with a new constant denoting a "hole" in a term).

Contexts are not closed under α-equivalence,
so they are not considered up to α-equivalence.

Exercise 15

Check that, if $\mathtt{C}[\sigma]$ is a context and $\mathrm{FV}(C[\mathtt{M}^\sigma]) = \emptyset$ then $C[\mathtt{M}]$ is a program (i.e. a ground closed term).

$\lessapprox_\sigma$ ($\approx_\sigma$) is a preorder relation between terms having the same type σ of terms, so sometimes we will write simply $\lessapprox$ ($\approx$).

It is easy to check that $\approx$ is a congruence relation, i.e. an equivalence relation closed under contexts. Sometimes $\approx$ is called observational or contextual equivalence.

Syntactic Sugar

It will be useful to name some terms. In particular, Ω_σ will denote the term defined by induction on σ as follows:

$$\Omega_\iota \equiv \mathtt{pred}\,\tilde{0}\,, \qquad\qquad \Omega_{\mu\to\tau} \equiv \lambda \mathtt{x}^{\mu}.\Omega_\tau\,.$$

By using Ω_σ, it is possible to define terms $\mathtt{Y}^k_\sigma$ $(k \in \mathbb{N})$ in the following way:

$$\mathtt{Y}^0_\sigma \equiv \Omega_{(\sigma\to\sigma)\to\sigma}\,, \qquad\qquad \mathtt{Y}^{k+1}_\sigma \equiv \lambda \mathtt{x}^{\sigma\to\sigma}.\mathtt{x}(\mathtt{Y}^k_\sigma\,\mathtt{x})\,.$$

Theorem 16

Let $\mathtt{M}_0, ..., \mathtt{M}_m$ be a sequence of terms $(m \geq 0)$.

1. If $\Omega_\sigma \mathtt{M}_0 ... \mathtt{M}_m$ is a program then $\Omega_\sigma \mathtt{M}_0 ... \mathtt{M}_m \Uparrow_e$.
2. Let $\mathtt{Y}_\sigma \mathtt{M}_0 ... \mathtt{M}_m$ be a program.
   $\mathtt{Y}_\sigma \mathtt{M}_0 ... \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}$ if and only if $\mathtt{Y}^k_\sigma \mathtt{M}_0 ... \mathtt{M}_m \Downarrow_N \tilde{\mathtt{n}}$, for some $k \in \mathbb{N}$.

Proof. 1. In case $m = 0$ then $\sigma = \iota$ and $\Omega_\iota \equiv \mathtt{pred}\,\tilde{0}$, so the proof is trivial. By induction on $m \geq 1$, it is easy to see that the $\Omega_\sigma \mathtt{M}_0 ... \mathtt{M}_m \Downarrow_N$ would imply $\mathtt{pred}\,\tilde{0} \Downarrow_{\mathtt{N}}$, absurdly.

2. We just consider programs, i.e. $\mathtt{Y}_\sigma \mathtt{M}_0 ... \mathtt{M}_m$ is a closed ground term. Let $\mathtt{C}[\iota]$ be a closing context context.
We prove that $\mathtt{C}[\mathtt{Y}_\sigma \mathtt{M}_0 ... \mathtt{M}_\mathtt{m}] \Downarrow_{\mathtt{N}} \tilde{\mathtt{n}}$ if and only if, exist $k \in \mathbb{N}$ such that $\forall h \geq k$, $\mathtt{C}[\mathtt{Y}^\mathtt{h}_\sigma \mathtt{M}_0 ... \mathtt{M}_\mathtt{m}] \Downarrow_{\mathtt{N}} \tilde{\mathtt{n}}$.
Both implications can be proved by induction on derivations proving the hypothesis.
$\Rightarrow$ The proof is done by induction on the derivation $\Downarrow_N$. If the last rule is different from $\mathtt{Y}$ then the proof is trivial, so we just consider it. There are two cases. If $\mathtt{C}[\iota]$ has shape $\mathtt{Y}_\tau \mathtt{C}_1[\tau_1] \ldots \mathtt{C}_1[\tau_n]$ for some $n \in \mathbb{N}$ then the proof follows easily by induction. In the remaining case $\mathtt{C}[\iota]$ has the shape $\mathtt{C}[\iota]\mathtt{C}_1[\tau_1] \ldots \mathtt{C}_1[\tau_\mathtt{n}]$. Since the type of $\mathtt{Y}_\sigma$, $m \geq 1$. Thus the premise of the last rule applied on the derivation $\mathtt{C}[\mathtt{Y}_\sigma \mathtt{M}_0 ... \mathtt{M}_\mathtt{m}] \Downarrow_{\mathtt{N}} \tilde{\mathtt{n}}$ must to have premise

$$\mathtt{M}_0(\mathtt{Y}_\sigma \mathtt{M}_0 ... \mathtt{M}_\mathtt{m})\mathtt{C}_1[\mathtt{Y}_\sigma \mathtt{M}_0 ... \mathtt{M}_\mathtt{m}] \ldots \mathtt{C}_\mathtt{n}[\mathtt{Y}_\sigma \mathtt{M}_0 ... \mathtt{M}_\mathtt{m}] \Downarrow_{\mathtt{N}} \tilde{\mathtt{n}}$$

but, it is the conclusion of a shorter derivation. Therefore, the proof follows straightforwardly by induction. $\Leftarrow$ Similar, but easier.

Syntactic Sugar

Exercise 17

1. Write explicitly the term $\mathtt{Y}^3_{\iota\to\iota}$.
2. Prove that $(\mathtt{Y}^3_{\iota\to\iota}\mathtt{F}^{(\iota\to\iota)\to\iota\to\iota})\,\tilde{\mathtt{n}} \Downarrow_{\mathtt{N}} \tilde{\mathtt{m}}$
   whenever $m \in \mathbb{N}$ is the factorial of $n \leq 3$.
   ($\mathtt{F}^{(\iota\to\iota)\to\iota\to\iota}$ is defined before the Exercise 9).
3. Compare the previous point with the Exercise 9.

Turing Completeness

A function $f : \mathbb{N}^m \to \mathbb{N}$ is definable, if there is a closed term $\mathtt{F}$ such that $\mathtt{F}\tilde{n}_1 \dots \tilde{n}_m \Downarrow \underline{f(n_1, \dots, n_m)}$ for all $n_1, \dots, n_m \in \mathbb{N}$.

Clearly Z, S and U_i^m are respectively defined by:

$$\begin{array}{l}\lambda x^\iota.\mathtt{if}\ \mathtt{x}\ \tilde{0}\ \tilde{0}\\ \mathtt{succ}\\ \lambda x_1^\iota \ldots x_m^\iota.\mathtt{if}\ (\mathtt{x_1 and} \ldots \mathtt{and}\, \mathtt{x_m})\ \mathtt{x_i}\ \mathtt{x_i}\end{array}$$

where $\mathtt{M\ and\, N}$ abbreviates $\mathtt{if\ M\ (if\ N\ 0\ 1)\ (if\ N\ 1\ 1)}$. The composition between $\mathtt{G}, \mathtt{H}_1, \ldots \mathtt{H_n}$ is defined by

$$\lambda x_1^\iota \dots x_m^\iota.\mathtt{G}(\mathtt{H}_1 x_1^\iota \dots x_m^\iota) \dots (\mathtt{H_n} x_1^\iota \dots x_m^\iota)$$

The primitive recursion on linear functions defined by $\mathtt{G}, \mathtt{H}$ can be represented as follows,

$$\begin{array}{l}\mu F.\lambda z^\iota x_1^\iota \dots x_m^\iota.\ \mathtt{if}\ z^\iota\\ \qquad (\mathtt{G} x_1^\iota \dots x_m^\iota)\\ \qquad (\mathtt{H}\ (F\ (\mathtt{pred}\ z^\iota) x_1^\iota \dots x_m^\iota)\ (\mathtt{pred}\ z^\iota) x_1^\iota \dots x_m^\iota).\end{array}$$

Let $\mathtt{G}$ to represent a partial recursive function $g : \mathbb{N}^{m+1} \to \mathbb{N}$ and let $\mathtt{F}$ be

$$(\mu F.\lambda z^\iota x_1^\iota \dots x_m^\iota.\mathtt{if}\ (\mathtt{G} z^\iota x_1^\iota \dots x_m^\iota)\ \ z^\iota\ \ (F\ (\mathtt{succ}\ z^\iota) x_1^\iota \dots x_m^\iota)\ )\tilde{0}.$$

Assume $\tilde{n}_1, \dots, \tilde{n}_m \in \mathcal{N}$. Thus, $\mathtt{F}\tilde{n}_1 \dots \tilde{n}_m \Downarrow \underline{k}$ if and only if $\mathtt{G}\underline{k}\ \tilde{n}_1 \dots \tilde{n}_m \Downarrow \tilde{0}$ and for each $h \leq k \in \mathbb{N}$ exists $\tilde{n} \in \mathcal{N}$ such that $\mathtt{G}\underline{h}\ \tilde{n}_1 \dots \tilde{n}_m \Downarrow \mathtt{succ}\,(\tilde{n})$.

Hence Turing completeness is achieved.

Higher-type

- A higher-order function (also functional) is a function that takes one or more functions as an input.
- Formally, we define the order by induction on types

$$\text{ORDER}(\sigma) = \begin{cases} 0 & \text{if } \sigma = \iota \\ \max\{\text{ORDER}(\sigma_0) + 1, \text{ORDER}(\sigma_1)\} & \text{if } \sigma = \sigma_0 \rightarrowtail \sigma_1 \end{cases}$$

  - Order 0: Ground/atomic data (i.e. non function data)
  - Order 1: Functions with domain of order 0
  - Order 2: Functions with domain of order 1
  - Order k: Functions with domain of order k-1
- Programs of order i ($i >= 2$) are called higher-type programs.
- A higher-order (higher-type) language supports higher-order functions and allows functions to be constituents of data structures.