Sistema F1, (lambda calcolo tipato semplice)

- quasi un frammento minimo di Haskell,
  ma tipi espliciti, nessun polimorfismo
- Tipi `... A, B`
  - un insieme di tipi base `K in Basic`
  - tipi funzioni `A -> B`
- Termini, codice `... M, N`
  - variabili `x`
  - costanti `c : K`
  - funzioni `\ x:A . M`
  - applicazioni `M N`
- si associa un tipo al parametro formale di ogni funzione
  - diversamente da Haskell
  - similmente a quasi tutti gli altri linguaggi

Regole

- ovvie per i giudizi di 'buona formazione' e 'buon tipo',
  - definibili mediante grammatiche libere
- (Var)

```
G, x:A, G' |- x:A
```

- (Fun)

```
     G, x:A |- M:B
------------------------
G |- (\ x:A . M) : A->B
```

- (App)

```
G |- M : A->B      G |- N:A
--------------------------
    G |-  MN :  B
```


Booleani

Costanti - (true) (false)

```
G |- true : Bool      G |- false : Bool
```

Funzioni collegate: - (ite)

```
 G |- (if_A _ then _ else _) : Bool -> A -> A -> A
```

o alternativamente

```
G |- M : Bool   G |- N1 : A  G |- N2 : A
----------------------------------------
   G |- if_A M then N1 else N2 : A
```

Marcare il costrutto `if_A` con il tipo `A` semplifica l'Inferenza altrimenti più complessa

Naturali

linguaggio minimale

```
G |- 0 : Nat        G |- succ : Nat -> Nat

G |- pred : Nat -> Nat        G |- isZero : Nat -> Bool
```

estensioni

- insieme infinito di regole per infinite constanti

```
G |- 1 : Nat      G |- 2 : Nat      G |- 3 : Nat  ...
```

- operazioni aritmetiche

```
G |- N1 : Nat  G |- N2 : Nat
---------------------------
   G |- N1 + N2 : Nat
```

Naturali

O in alternativa:

```
G |- (+) : Nat -> Nat -> Nat
```


Esempio

```
\ y:Nat . \ z:Nat . (y + 2) + z
```


Tipi prodotto, coppia

- nuovo costruttore di tipi `A * B`
  in Haskell `(A,B)`

Regole

- (Pair)

```
G |- M : A     G |- N : B
--------------------------
    G |- (M,N) : A*B
```

- (First) (Second)

```
G |- first : A*B -> A     G |- second : A*B -> B
```


Tipi unione

- nuovo costruttore di tipi `A + B`

Regole

- (InLeft), (InRight)

```
G |- inLeft : A -> A+B     G |- inRight : B -> A+B
```

- (IsLeft), (IsRight)

```
G |- isLeft : A+B -> Bool     G |- isRight : A+B -> Bool
```

- (AsLeft), (AsRight)

```
G |- asLeft : A+B -> A      G |- asRight : A+B -> B
```

possono causare errore di tipo a tempo di esecuzione
forzano un type checking dinamico (dynamic checking)

segue

È possibile evitare il type checking dinamico con il costrutto (case)

```
G |- M : A1 + A2     G, x1:A1 |- N1:B     G, x2:A2 |- N2:B
-------------------------------------------------------------
G |- case M of(inLeft(x1:A1) -> N1)(inRight(x2:A2) -> N2)   : B
```

- sintassi alternativa [Cardelli]
  - `case M of (inLeft(x1:A1) -> N1) (inRight(x2:A2) -> N2) : B`
    scritto come:
    `case M of x1:A1 then N1 | x2:A2 then N2`

Record (Struct)

- Estensione del tipo prodotto
  - un numero arbitrario di componenti
  - sistema di etichette `l`
- nuovo costruttore di tipo `{ l1 : A1, ... , ln : An }`

regole

- (Record)

```
G |- M1 : A1      G |- M2:A2     ...     G |- Mn:An
-------------------------------------------------------
G |- { l1:M1, ..., ln:Mn } : { l1:A1, ..., ln:An }
```

- (Record Select)

```
G |- M : { l1 : A1, ... , ln : An }
-----------------------------------
        G |-  M.li : Ai
```


segue

Un alternativa, poco usata, a (Record Select)

- (Record With)

```
G |- M:{l1:A1, ..., ln:An}      G, x1:A1,... , xn:An |- N : B
-------------------------------------------------------------
     G |- case M of {x1:A1,... , xn:An} -> N : B
```


Costrutti e regole

- (Ref)

```
  G |- M : A
------------------
G |- ref M : Ref A
```

`Ref M` definisce una nuova locazione, inizializzata con valore `M`

- (Deref) accedo al contenuto

```
G |- deref : (Ref A) -> A
```

`deref` corrisponde a `!` in ML `!`, o $*$ in C

- (Assign)

```
G |- M : Ref A     G |- N : A
------------------------------
   G |- M = N  :  Unit
```

Linguaggio imperativo dentro uno funzionale

- posso tradurre

```
var x = M;
N
```

- con

```
let x = ref M in N
```

- oppure con

```
(\ x . N) (ref M)
```

- posso tradurre la composizione di comandi `C1; C2` con

```
`(\ y : Unit . C2) C1`
```

in un linguaggio call-by-value

Esempio:

- i costruttori di liste si possono definire come:

```
Nil  =  fold(inLeft unit)
Cons x xs =  fold(inRight (x,xs))
```

- i distruttori di liste

```
head xs =  first (asRight (unfold xs))
tail xs =  second (asRight (unfold xs))
```

- i costrutti (`mu, fold, unfold`) sono:
  concettualmente semplici, ma poco pratici, i costrutti standard modellabili attraverso di essi
- Tutti i dati ricorsivi, non mutuamente ricorsivi e non parametrici, di Haskell, trattabili in questo modo.
- codifica non troppo complessa

Esempio: la regola per Composition derivabile

- (Composition)

```
G |- C1 : Unit     G |- C2 : Unit
-----------------------------
   G |- C1; C2  :  Unit
```

```
(C1;C2)  ::=  (\ y : Unit . C2) C1
```


Linguaggio imperativo, simil C.

Considero tre categorie sintattiche

- Espressioni

```
E ::= const  |  id  |  E binop E  |  unop E
```

- Comandi

```
C ::= id = E  |  C; C  |  while E {C}  |
          if E then C else C  |  I(E, ... E)  |  {D ; C}
```

- Dichiarazione

```
D  ::=  A id = E  |  id(A1 id1, ... , An idn) { C }  |
          epsilon  |  D; D
```

Regole, espressioni

Per le espressione, valgono le corrispondenti regole del linguaggio funzionale

- (Id, (Var))

```
G, id:A, G' |- id : A
```

```
G |- true : Bool     G |- false : Bool
```

- (ite)

```
 G |- (if _ then _ else _) : Bool -> A -> A -> A
```

```
G |- 1 : Nat      G |- 2 : Nat     G |- 3 : Nat  ...
```

- operazioni aritmetiche

```
G |-  E1 : Nat     G |- E2 : Nat
------------------------------
    G |-  E1 + E2 : Nat
```


segue

- (If Then Else)

```
G |- E : Bool     G |- C1 : Unit  G |- C2 : Unit
-----------------------------------------------
   G |- if E then C1 else C2  :  Unit
```

- (Procedure)

```
G |- id : (A1 * ... * An) -> Unit    G |- E1 : A1 ...  G |- En
-------------------------------------------------------------
   G |- id (E1, ..., En) : Unit
```

- (Blocco)

```
G |- D :: G1     G, G1 |- C : Unit
---------------------------------
   G |- {D;C} : Unit
```


Comandi

- (Assign)

```
G |- id : A     G |- E : A
--------------------------
   G |- id = E  :  Unit
```

- (Sequence)

```
G |- C1 : Unit  G |- C2 : Unit
------------------------------
  G |- C1; C2  :  Unit
```

- (While)

```
G |- E : Bool     G |- C : Unit
-------------------------------
   G |- while E {C}  :  Unit
```


Dichiarazioni

Le dichiarazione necessitano di un nuovo tipo di giudizio

- Una dichiarazione crea un ambiente, che viene utilizzato nel blocco della dichiarazione
- Giudizi nella forma

```
G |- D :: G1
```

Regole

- (Id)

```
G |-  E : A     (A tipo memorizzabile)
-------------------------------------
    G |-  A id = E  ::  (id : A)
```

- (Proc)

```
G, id1 : A1, ..., idn : An |-  C : Unit
-----------------------------------------------------------
G |- id(A1 id1, ... idn){ C } :: id : (A1 x...x  An) -> Unit
```

- (Recursive Proc)

```
G, id1 : A1, ... An, id : (A1 *...* An) -> Unit  |-  C : Unit
-------------------------------------------------------------
G |- id(A1 id1, ... idn){ C } :: id : (A1 x...x  An) -> Unit
```


segue

- (Sequenza)

```
G  |-  D1 :: G1        G, G1 |-  D2 :: G2
-----------------------------------------
        G |-  D1; D2 :: G1, G2
```


Array

- Devo formalizzare le due differente interpretazioni delle espressioni
  - a sinistra della assegnazione '='
  - a destra dell'assegnazione
- finora a sinistra solo identificatori 'id'

Tipi, aggiungo un costruttore di tipo

- `A[B]` con le opportune restrizioni
  - `A` a memorizzabile
  - `B` tipo enumerazione

Espressioni, distinguo tra espressioni sinistre e destre

- `LE  ::= id  |  LE[RE]`
- `RE ::= LE  | const  |  RE binop RE  |  unop RE`

Una nuova versione dell'assegnamento

- `C ::=  LE = RE  |  ...`

Regole

Per le espressioni distinguo due giudizi,

- `G  |-l  E : A` (`E` denota una locazione di tipo `A`)
- `G  |-r  E : A` (`E` denota un valore di tipo `A`)
- (Assign)

```
G |-l  E1 : A     G |-r  E2 : A
-------------------------------
G |-l  E1 = E2 : Unit
```

- (Left-Right)

```
G |-l  E : A
--------------
G |-r  E : A
```

- (Var)

```
G, x:A, G' |-l x : A
```

segue

- (Array)

```
G |-l  E : A[B]     G |-r  E1 : B
---------------------------------
G |-l  E[E1] : A
```

- Dichiarazione

```
--------------------------------
G |-  id : A[B]  ::  id : A[B]
```

- Le restanti regole per le espressioni restano inalterate, diventando regole per giudizi right `|-r`.

Polimorfismo di sottotipo.

Maggiore flessibilità permetto ad un espressione di avere più tipi

- Varie forme di polimorfismo
  - ad hoc
  - di sottotipo
  - parametrico
  - combinazioni dei precedenti
- Sottotipo (e ad hoc)
  - introduco una relazione di sottotipo, con relativi giudizi e regole

    `G |- A <: B`
- (Subsumtion)

```
G |-r  E : A     G |-  A <: B
-----------------------------
       G |-r  E : B
```


Regole per i giudizi di sottotipo

Posso trattare il polimorfismo ad-hoc, con assiomi del tipo.

```
G |-  Int <: Float
```

Polimorfismo dei linguaggi ad oggetti:

- un tipo oggetto con più campi, sottoggetto di uno con meno.
- formalizziamolo con i tipi Record

```
G |- A1 <: B1 ...     G |- Am <:Bm     m < n
-----------------------------------------------
G |- { l1:A1, ..., ln:An } <: { l1:B1, ..., lm:Bm}
```


segue

Definisco regole di sottotipo per ogni costruttore di tipi.

- (Prod <:)

```
G |- A1 <: B1      G |- A2 <:B2
-------------------------------
G |- A1 x A2  <:  B1 x B2
```

Regole sempre covariante con eccezione

- (Arrow <:)

```
G |- A1 <: B1      G |- A2 <:B2
-------------------------------
G |- B1 -> A2  <:  A1 -> B2
```

`controvariante sul primo argomento.`

Regola non ovvia, tipi ricorsivi.

- (Mu <:)

```
G, X <: Y  |-  A <: B
-------------------------
G |- mu X. A  <:  mu Y . B
```

la regola

- (Mu <: Wrong)

```
G, X  |-  A <: B
-------------------------
G |- mu X. A  <:  mu X . B
```

mi permette di derivare

```
G |- mu X. X -> Int  <:  mu X . X ->  Float
```

non corretto.