Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica (Classe L-31)

Compilazione di un linguaggio funzionale in Java

Laureando
Massimo Pavoni

Matricola 124377

Relatore
Prof. Luca Padovani

A.A. 2023/2024

Indice

Elenco dei codici

Elenco delle figure

Elenco delle tabelle

1. Introduzione

Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur. Duis aute irure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Nel Capitolo 1 illustreremo prima le motivazioni che ci hanno spinto a perseguire l'obiettivo descritto e quindi la struttura della tesi.

1.1 Motivazione

1.2 Obiettivi

1.3 Struttura della Tesi

2. Funx

Questo capitolo descrive brevemente i linguaggi funzionali e le scelte effettuate durante l'ideazione del linguaggio usato per il progetto: **Funx**.
Il nome nasce dall'unione dei due termini anglosassoni *functional* e *expression*; viene quindi pronunciato ['fʌnɪk's] in inglese, [fan·èks] o [fan·ìks] in italiano.

2.1 Linguaggi funzionali

Nonostante molti linguaggi non si possano confinare all'interno di un solo paradigma, parlando di linguaggi di programmazione si fa spesso riferimento a due grandi categorie: linguaggi imperativi e linguaggi dichiarativi.

I primi hanno caratteristiche direttamente legate al modello di calcolo di *John Von Neumann*, a sua volta non dissimile dalla macchina di *Alan Turing*. Questi linguaggi sono usati per scrivere codice che segue una precisa sequenza di istruzioni, la quale descrive più o meno esplicitamente i passi necessari per risolvere il problema affrontato.
Appartengono alla famiglia dei linguaggi di programmazione imperativi sia linguaggi procedurali come `Fortran`, `Cobol` e `Zig`, sia i linguaggi orientati agli oggetti, tra cui `Kotlin`, `C#` e `Ruby`.

I linguaggi dichiarativi, invece, sono fondamentali per lo scopo del progetto: tali linguaggi sono generalmente di altissimo livello e permettono allo sviluppatore di concentrarsi sull'obiettivo da raggiungere piuttosto che sui dettagli implementativi.
Fanno parte di questa categoria linguaggi di interrogazione come `SQL`, linguaggi logici come `Prolog` e soprattutto i linguaggi funzionali: `Lisp`, `Clojure`, `Elixir`, `OCaml` e `Haskell` sono alcuni esempi.

Alla base di ogni linguaggio funzionale vi è il **lambda calcolo**[1]: un sistema formale definito dal matematico *Alonzo Church* (supervisore di *Alan Turing* durante il dottorato), equivalente alla macchina di Turing, ma fondato sulle funzioni pure.

[1] «A Set of Postulates for the Fundation of Logic», [Chu32] e «A Set of Postulates for the Foundation of Logic» (Second paper), [Chu33]

La grammatica del lambda calcolo verrà presentata poco più avanti (sezione 2.2), ma le regole che ne governano il funzionamento e il modo in cui queste vengano utilizzate per ridurre le espressioni ad una forma normale esulano dai fini di questo documento.
Rimane comunque rilevante elencare le principali qualità che un linguaggio funzionale usualmente matura grazie al lambda calcolo:

- **funzioni come entità di prima classe**: le funzioni possono essere passate come argomenti e restituite come risultato di altre funzioni;
- **immutabilità**: le variabili utilizzate sono immutabili;
- **purezza**: le funzioni sono libere da effetti collaterali (non modificano lo stato del programma) e restituiscono sempre lo stesso output per input identici;
- **ricorsione**: la ricorsione è il meccanismo più idiomatico per esprimere l'iterazione su una struttura dati.

2.1.1 ML, Haskell e Funx

Nonostante le funzioni pure tipiche di un linguaggio funzionale siano un concetto molto attraente dal punto di vista della correttezza della computazione, i vincoli così imposti possono risultare stringenti a tal punto da rendere difficile, se non impossibile, la scrittura di programmi che interagiscano con il mondo reale.

Per questo motivo, molti linguaggi funzionali permettono invece di utilizzare particolari funzioni impure o di effettuare almeno operazioni di input/output. Inoltre, molti linguaggi prevalentemente imperativi adottano ormai da tempo alcune caratteristiche tipiche dei linguaggi funzionali (e.g. `Rust`, il linguaggio più amato[2] dagli sviluppatori secondo i sondaggi di *Stack Overflow*, eredita molto dal linguaggio con cui era scritto il suo primo compilatore, `OCaml`, ed è dotato quindi di funzioni di prima classe, immutabilità di default, strutture dati algebriche, ecc.).

`ML` è un linguaggio funzionale sviluppato negli anni '70 presso l'Università di Edimburgo, costituente la base per moltissimi dei linguaggi sviluppati in seguito. `ML` permette effettivamente l'uso di funzioni impure, ma fra i suoi discendenti vi è `Haskell`, uno dei pochi linguaggi invece completamente puri.
`Haskell` si avvale di un pattern di programmazione chiamato *monadi*[3] per gestire le operazioni di input/output e altre operazioni impure, mantenendo le funzioni pure.

Nell'ideare **Funx** l'ispirazione viene proprio da `Haskell`, ma è presente la possibilità di dichiarare un'unica funzione impura (il cosiddetto *main*) per permettere di visualizzare a schermo un risultato. Il linguaggio non è quindi allo stesso livello di purezza di `Haskell`, e naturalmente non supporta molte delle funzionalità più avanzate di quest'ultimo (come le *classi di tipi* e il *pattern matching*), ma ne mutua altre comunque interessanti, tra cui l'uso di alcuni operatori infissi e il *polimorfismo parametrico*.

[2] Stack Overflow Developer Survey 2023 (`https://survey.stackoverflow.co/2023`), *Rust is the most admired language*
[3] «Notions of Computation and Monads», [Mog91]

2.2 Sintassi

La sintassi di **Funx** risulta molto simile a quella di `Haskell`, con poche differenze dovute a tre principali motivi:

- libera scelta di nomi e simboli per le parole chiave;
- necessità di successiva traduzione in `Java`;
- difficoltà e scarso valore all'interno del progetto dell'implementazione di un parser dipendente dall'indentazione.

A prescindere da ciò, il cuore del linguaggio è lo stesso di ogni altro linguaggio derivato dal lambda calcolo: la sua definizione si può agilmente comprendere visualizzando la grammatica del lambda calcolo e confrontandola con quella (leggermente semplificata) di **Funx**, facendo attenzione alle regole aggiuntive.

Espressione	E	$::=$	x	variabile
		$\mid$	$E_l\ E_r$	applicazione
		$\mid$	$\lambda x\ .\ E$	astrazione

Figura 2.1: Grammatica del lambda calcolo

Le tre regole in Figura 2.1 indicano le tre componenti indispensabili del lambda calcolo:

- **variabile**: simbolo rappresentante un parametro;
- **applicazione**: applicazione di funzione ad un argomento (entrambi espressioni);
- **astrazione**: definizione di una funzione anonima, con un solo input x (variabile vincolata) e un solo output E (espressione); per definire funzioni con più parametri si debbono usare molteplici astrazioni annidate (tecnica detta *currying*).

Modulo	M	::=	$nome \cdot L$	
Dichiarazione	D	::=	$?(schema\ di\ tipo) \cdot id = E$	funzione
Espressione	E	::=	c	costante
		|	x	variabile
		|	$E_l\ E_r$	applicazione
		|	$\lambda x\ .\ E$	astrazione
		|	L	let
		|	**if** E_c **then** E_t **else** E_e	if
Let	L	::=	**let** $\cdot$ $D\ (\cdot\ D)^* \cdot$ **in** E	

Figura 2.2: Grammatica di Funx

È facile constatare la presenza delle ulteriori produzioni per la definizione del modulo corrente (informazione inclusa a prescindere dal fatto che il linguaggio ad ora non supporti l'importazione di moduli esterni che non siano la libreria standard) e di funzioni con nome: lo *schema di tipo* è un'informazione opzionale relativa al tipo della funzione e di cui si parlerà più approfonditamente nella sezione 3.1.2.
Per quanto riguarda invece le espressioni, vengono introdotte tre nuove regole:

- **costante**: rappresenta un valore letterale, come un numero o una stringa;
- **let**: permette di avere dichiarazioni locali utilizzabili all'interno di un'espressione;
- **if**: la più classica istruzione condizionale controllata da un'espressione booleana.

2.2.1 Zucchero sintattico

Con lo scopo di rendere il codice più leggibile, conciso e semplice, **Funx** introduce dello zucchero sintattico (del tutto simile a quello di `Haskell`). In Tabella 2.1 sono riportati l'indispensabile per evitare il parsing dell'indentazione, le semplificazioni comuni utili all'arrichimento del lambda calcolo, e infine tutti gli operatori simbolici supportati al momento (assieme alla notazione per indicarne associatività e precedenza).

Zucchero	**Sostituzione**
`\x -> e`	`λx . e`
`\x y -> e`	`λx . λy . e`
`f x y = e`	`f = λx . λy . e`

`let`
`f1 = e1`
`f2 = e2`
`in e3`
	`let f1 = e1 · f2 = e2 in e3`

`f3 = e3`
`with`
`f1 = e1`
`f2 = e2`
`out`
	`f3 = let f1 = e1 · f2 = e2 in e3`

`main = e3`
`f1 = e1`
`f2 = e2`
	`main = let f1 = e1 · f2 = e2 in e3`

`if b then e1 else e2 fi`	`if b then e1 else e2`
`e1 . e2` `infixr 9`	`compose e1 e2`
`e1 / e2` `infixl 7`	`divide e1 e2`
`e1 % e2` `infixl 7`	`modulo e1 e2`
`e1 * e2` `infixl 7`	`multiply e1 e2`
`e1 + e2` `infixl 6`	`add e1 e2`
`e1 - e2` `infixl 6`	`subtract e1 e2`
`e1 > e2` `infix 4`	`greaterThan e1 e2`
`e1 >= e2` `infix 4`	`greaterThanEquals e1 e2`
`e1 < e2` `infix 4`	`lessThan e1 e2`
`e1 <= e2` `infix 4`	`lessThanEquals e1 e2`
`e1 == e2` `infix 4`	`equalsEquals e1 e2`
`e1 != e2` `infix 4`	`notEquals e1 e2`
`!!e` `prefix 4`	`not e`
`e1 && e2` `infixr 3`	`if e1 then e2 else False`
`e1 || e2` `infixr 2`	`if e1 then True else e2`
`e1 $ e2` `infixr 0`	`apply e1 e2`

Tabella 2.1: Zucchero sintattico

Come già accennato, il Capitolo 5 illustrerà come l'albero sintattico astratto (**AST**) di un programma viene ottenuto, annotato e tradotto in `Java`; la sezione 4.2 esporrà invece il motivo della traduzione degli operatori booleani binari in if.

Alcuni esempi di funzioni sono presentati nel Codice 2.1; seppur superflua, l'indentazione è inclusa per maggiore chiarezza.

```
1  main = factorial 20
2
3  factorial : Int -> Int
4  factorial n = if n == 0 then 1 else n * factorial (n - 1) fi
5
6  even : Int -> Bool
7  even = let
8         even1 : Int -> Bool
9         even1 n = if n == 0 then True else odd (n - 1) fi
10
11        odd : Int -> Bool
12        odd n = if n == 0 then False else even1 (n - 1) fi
13     in even1
14
15 gcd : Int -> Int -> Int
16 gcd a b = if b == 0 then a else gcd b (a % b) fi
17
18 xor : Bool -> Bool -> Bool
19 xor a b = (a || b) && !!(a && b)
```

Codice 2.1: Esempio di programma

3. Inferenza di tipo

Dopo aver discusso la sintassi di **Funx**, è importante far notare come i programmi non abbiano bisogno di annotazioni di tipo, nonostante siano stati adottati tipi statici.
In questo capitolo affronteremo l'argomento dei sistemi di tipo e dell'inferenza, meccanismo proprio di molti linguaggi, funzionali e non, che rende possibile la deduzione automatica del tipo di un termine basandosi sull'utilizzo delle variabili e delle funzioni.

3.1 Sistemi di tipo

Durante la genesi di ogni linguaggio di programmazione, una delle scelte più significative riguarda l'introduzione di un sistema per gestire i tipi di variabili ed espressioni.
Tali sistemi di tipo sono di fatto insiemi di regole logiche che permettono di assegnare una proprietà *"tipo"* a ciascuno dei termini del linguaggio che ne necessitano.
Sono principalmente suddivisi in due categorie:

- **tipizzazione statica**: i tipi sono definiti a tempo di compilazione e non possono cambiare mentre il programma è in esecuzione;
- **tipizzazione dinamica**: i tipi vengono stabiliti durante l'esecuzione e possono cambiare in qualsiasi momento.

Oltre a questa distinzione esistono varie sfumature e approcci differenti, informalmente classificati in base alla rigidità delle regole di tipizzazione. Si parla di *tipizzazione debole* quando ad esempio sono consentite conversioni implicite tra tipi diversi, *tipizzazione forte* se sono impedite, oppure qualora sia o meno disponibile l'aritmetica dei puntatori.

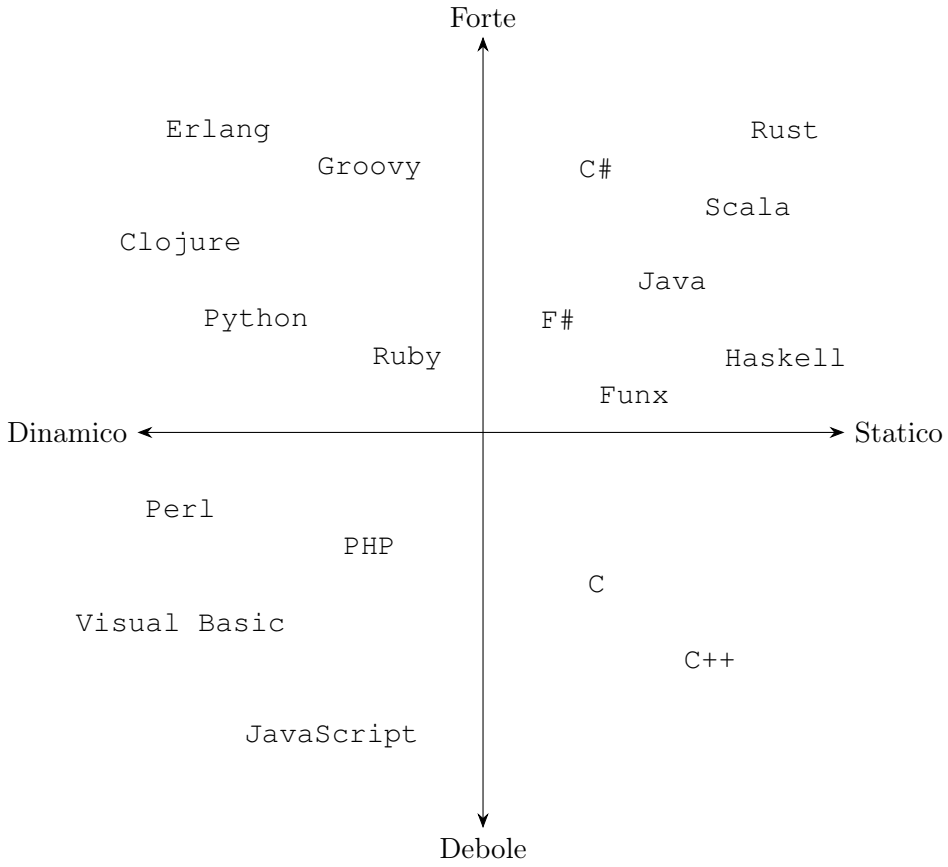

Figura 3.1: Alcuni linguaggi e loro sistemi di tipo

Grazie ai tipi dinamici, linguaggi quali `Python` e `JavaScript` permettono veloce prototipazione, flessibilità e codice più conciso, a discapito però di una più alta probabilità di incontrare errori importanti a runtime, piuttosto che in fase di compilazione.

Al contrario, i tipi statici spesso migliorano naturalmente la mantenibilità di un progetto: viene limitata la possibilità di scorciatoie nello sviluppo, ma si hanno maggiori garanzie di correttezza, in quanto il compilatore può implementare ulteriori controlli e segnalare errori semantici più precisi già prima dell'esecuzione del programma.

D'altro canto, l'obbligo di specificare i tipi di ogni variabile, oggetto, funzione e parametro può risultare tedioso e talvolta ridondante; molti linguaggi moderni, tra cui `Haskell` e `Rust`, ovviano magistralmente a quest'inconvenienza tramite l'uso dell'inferenza di tipo.

Gli algoritmi di inferenza introducono numerosi benefici, in particolare:

- la scrittura del codice è meno onerosa per lo sviluppatore a prescindere dal sistema di tipi utilizzato, e diviene quindi estremamente vantaggioso utilizzare tipi statici;
- le annotazioni ora opzionali possono essere aggiunte dal programmatore quando vi sono casi difficili da disambiguare automaticamente, oppure per migliorare la leggibilità del codice;
- gli strumenti di sviluppo per il linguaggio possono sfruttare informazioni fornite dal motore di inferenza per suggerire il tipo delle espressioni e arricchire i messaggi di errore e di warning.

3.1.1 λ-cubo

Al fine di comprendere quale sistema il linguaggio **Funx** implementi, prima di discutere l'inferenza si vuol descrivere brevemente il *λ-cubo*, lambda cubo[1], un modello introdotto per classificare i sistemi di tipo applicabili al lambda calcolo.
In Figura 3.2 è possibile osservare come la struttura del cubo abbia all'origine il *lambda calcolo semplicemente tipato* ($\lambda\!\rightarrow$) e come le tre dimensioni in cui si sviluppa rappresentino ciascuna un'estensione del sistema:

- **tipi dipendenti** ($\rightarrow$): la definizione dei tipi può dipendere dai valori delle variabili (implementati da linguaggi funzionali come `Agda`, `Coq` e `Idris`);
- **polimorfismo parametrico** ($\uparrow$): i tipi possono essere polimorfi, generalizzati tramite variabili di tipo (presenti nei sistemi adottati da `ML`, `OCaml` e `Haskell`);
- **costruttori di tipo** ($\nearrow$): capacità di costruire nuovi tipi a partire da tipi esistenti (`Haskell` ne fa grande uso poiché ogni nuovo tipo, dichiarato con la keyword `data`, è un nuovo costruttore di tipo).

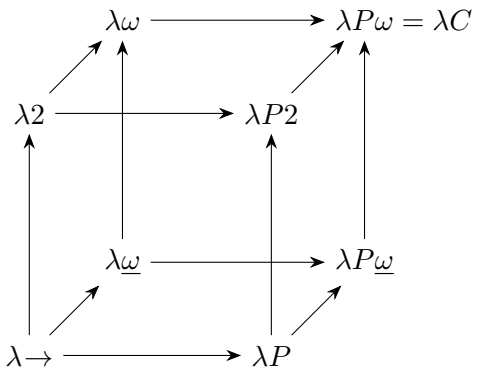

Figura 3.2: λ-cubo

Senza entrare troppo nei dettagli, in ordine crescente di potenza espressiva:

- $\lambda\!\rightarrow$ (*lambda calcolo semplicemente tipato*): tipi monomorfi;
- $\lambda\underline{\omega}$ (*lambda weak omega*): costruttori di tipo;
- $\lambda 2$ (*lambda due, lambda F, lambda calcolo polimorfico*): polimorfismo parametrico;
- λP (*lambda P*): tipi dipendenti;
- $\lambda P\underline{\omega}$ (*lambda P weak omega*): costruttori di tipo e tipi dipendenti;
- $\lambda\omega$ (*lambda omega*): costruttori di tipo e polimorfismo parametrico;
- $\lambda P2$ (*lambda P due*): polimorfismo parametrico e tipi dipendenti;
- $\lambda P\omega = \lambda C$ (*lambda P omega, lambda C, calcolo delle costruzioni*): cstronglyombinazione di tutte le tre estensioni.

[1] «Introduction to generalized type systems», [Bar91]

3.1.2 Sistema FC

Tra i vari sistemi di tipo per il lambda calcolo, uno dei più interessanti è *lambda F* (vertice $\lambda 2$ in Figura 3.2) poiché molto utile per la generalizzazione delle funzioni: un problema molto ricorrente nella programmazione con qualsiasi linguaggio è infatti la duplicazione di codice per funzioni che svolgono operazioni simili su tipi diversi.

Il *sistema F* risolve tale problema introducendo il **polimorfismo parametrico** e di conseguenza la distinzione tra tipi monomorfi (monotipi) e tipi polimorfi (politipi).

I tipi delle funzioni possono essere caratterizzati tramite quantificatori universali e variabili di tipo ove sia necessario un tipo generico (spesso vengono usate singole lettere dell'alfabeto greco o latino).

Tuttavia, $\lambda 2$ nella sua forma più pura, oltre a non essere un sistema Turing-completo (è possibile definire solamente la ricorsione primitiva), rende l'inferenza di tipo trattata nella sezione 3.2 un problema non decidibile[2].

Pertanto, il linguaggio `Haskell` non implementa semplicemente il *sistema F*, ma piuttosto una versione ristretta di $\lambda\omega$ chiamata *sistema FC*[3].

Quest'ultima include anche i costruttori di tipo (*funzioni di tipo* in **Funx**), frenando però il polimorfismo ai cosiddetti *tipi polimorfici di rango 1* (*polimorfismo predicativo*): tale limitazione si manifesta nella scrittura di tutti i quantificatori universali all'inizio di un tipo polimorfo (che prende il nome di *schema di tipo*).

Le versioni invece più espressive e più vicine a $\lambda\omega$ sono:

- *polimorfismo di rango superiore*: supporta quantificatori universali in qualsiasi punto nelle definizioni delle funzioni (e.g. `Bool -> (forall b . b -> b)`); `Haskell` lo realizza con l'estensione `RankNTypes` del compilatore `GHC`, mentre offre anche l'estensione `Rank2Types`, per la quale l'inferenza rimane decidibile;

- *polimorfismo impredicativo*: permette di quantificare le variabili di tipo in modo arbitrario, anche e soprattutto all'interno dei costruttori di tipo (e.g. `Maybe (forall a . a -> a) -> Bool`, possibile in `Haskell` abilitando l'estensione `ImpredicativeTypes`).

Il linguaggio **Funx** ovviamente non è correntemente in grado di supportare queste estensioni del sistema di tipo, così come non è possibile definire nuovi tipi o fare uso di *classi di tipo* simili a quelle proprie di `Haskell`. Affermare che **Funx** adotti il *sistema FC* potrebbe lasciare intendere un linguaggio più espressivo di quanto non sia in realtà: è dunque più opportuno realizzare il *sistema HM*, di cui il *sistema FC* è un ampliamento, e che comunque ben si presta allo scopo principe di traduzione in `Java`.

In Tabella 3.1 si possono osservare i tipi di alcune funzioni polimorfe di `Haskell`: la sintassi di **Funx** è molto simile (identica in ognuno dei casi presentati), con l'eccezione che la parola chiave `forall` è completamente assente dal linguaggio, in quanto ogni identificatore che inizia con una lettera minuscola è considerato una variabile di tipo da quantificare universalmente (vedi sezione **??**).

[2] «Typability and type checking in System F are equivalent and undecidable», [Wel99]
[3] «System FC, as implemented in GHC», [Eis15]

Funzione	**Schema**
`id`	`forall a . a -> a`
`const`	`forall a b . a -> b -> a`
`(.)`	`forall b c a . (b -> c) -> (a -> b) -> a -> c`
`flip`	`forall a b c . (a -> b -> c) -> b -> a -> c`
`($)`	`forall a b . (a -> b) -> a -> b`
`(&)`	`forall a b . a -> (a -> b) -> b`

Tabella 3.1: Esempi di funzioni polimorfe

In Figura 3.3 è mostrata la grammatica per la definizione dei tipi implementati nel linguaggio **Funx**. Si noti come i tipi monomorfi siano solo variabili di tipo o applicazioni di funzioni ad altri tipi; al momento il linguaggio mette a disposizione le funzioni di tipo più elementari, la cui arietà è indicata in pedice.

Schema di tipo	σ	$::=$	τ	monotipo
		$\mid$	$\forall\ \alpha\ .\ \sigma$	politipo
Tipo	τ	$::=$	α	variabile di tipo
		$\mid$	$F\ \tau \ldots \tau$	applicazione di funzione di tipo
Funzione di tipo	F	$::=$	$\rightarrow_2$	funzione
		$\mid$	$Bool_0$	booleano
		$\mid$	Int_0	intero

Figura 3.3: Grammatica del sistema di tipo di **Funx**

3.2 Inferenza secondo Hindley–Milner

Come già accennato, il sistema *Hindley–Milner (HM)*[4], o *Damas-Hindley-Milner*[5], è un sistema di tipo per il lambda calcolo con polimorfismo parametrico largamente utilizzato in molti moderni linguaggi di programmazione ad alto livello. Il maggiore punto di forza del sistema è il relativo metodo di inferenza, in grado di dedurre automaticamente il tipo di un termine senza annotazioni esplicite fornite dagli sviluppatori.

Prima di presentare l'algoritmo di inferenza è necessario complementare le nozioni di lambda calcolo esteso (Figure 2.1 e 2.2) e del sistema di **Funx** (Figura 3.3) con due concetti fondamentali (Figura 3.4):

- **contesto**: una insieme di associazioni tra variabili e schemi di tipo, che rappresenta lo stato corrente dell'ambiente in cui un termine viene tipato; l'unione tra la grammatica del linguaggio e il sistema di tipi è data da un giudizio di tipo effettuato nel contesto su un termine (espressione);
- **variabili libere**: l'insieme delle variabili libere di un tipo è semplicemente il complemento delle variabili vincolate, quantificate universalmente in un politipo.

Contesto	Γ	$::=$	ϵ	contesto vuoto
		$\mid$	$\Gamma + x : \sigma$	aggiunta di associazione
Giudizio di tipo		$::=$	$\Gamma \vdash E : \sigma$	

$free(\forall\ \alpha\ .\ \sigma)$	$=$	$free(\sigma) - \{\alpha\}$	tipo polimorfo
$free(\alpha)$	$=$	$\{\alpha\}$	variabile di tipo
$free(F\ \tau_1 \ldots \tau_n)$	$=$	$\bigcup_{i=1}^{n} free(\tau_i)$	applicazione di funzione di tipo
$free(\Gamma)$	$=$	$\bigcup_{x:\ \sigma \in \Gamma} free(\sigma)$	contesto
$free(\Gamma \vdash E\ :\ \sigma)$	$=$	$free(\sigma) - free(\Gamma)$	giudizio di tipo

Figura 3.4: Definizioni di contesto e variabili libere

Le regole di inferenza[6] del *sistema HM*, riportate in Figura 3.5, informano il comportamento dell'*algoritmo* $\mathcal{W}$, implementazione dell'inferenza di tipo.

[4] «The Principal Type-Scheme of an Object in Combinatory Logic», [Hin69] e «A Theory of Type Polymorphism in Programming», [Mil78]

[5] «Principal type-schemes for functional programs», [DM82]

[6] «A Simple Applicative Language: Mini-ML», [Clé+86]

In aggiunta a principi già noti, le peculiarità non immediatamente chiare sono:

- *constantType*: funzione che restituisce il tipo di una costante;
- $\sigma \sqsubseteq \tau$: indica intuitivamente che σ è più generale di τ (τ è un'*istanza* di σ);
- $Clos_\Gamma$: ottiene la *chiusura*, ossia tipo più generale, di una variabile quantificando universalmente le variabili libere del tipo iniziale.

$$\frac{\tau = constantType(c)}{\Gamma \vdash c : \tau} \quad \text{[costante]}$$

$$\frac{x : \sigma \in \Gamma \qquad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau} \quad \text{[variabile]}$$

$$\frac{\Gamma \vdash E_l : \tau \to \beta \qquad \Gamma \vdash E_r : \tau}{\Gamma \vdash E_l\ E_r : \beta} \quad \text{[applicazione]}$$

$$\frac{\Gamma + x : \beta \vdash E : \tau}{\Gamma \vdash \lambda x\ .\ E : \beta \to \tau} \quad \text{[astrazione]}$$

$$\frac{\overline{\Gamma} = \Gamma + \vec{x} : \vec{\beta}}{\dfrac{\overline{\Gamma} \vdash \vec{E} : \vec{\kappa} \qquad \overline{\Gamma}\{Clos_\Gamma(\vec{\kappa})/\vec{\beta}\} \vdash E_i : \tau}{\overline{\Gamma} \vdash \textbf{let}\ \vec{x} = \vec{E}\ \textbf{in}\ E_i : \tau}} \quad \text{[let]}$$

$$Clos_\Gamma(\tau) = \forall\ \hat{\alpha}\ .\ \tau \qquad \hat{\alpha} = free(\tau) - free(\Gamma)$$

$$\frac{\Gamma \vdash E_c : Bool \qquad \Gamma \vdash E_t : \tau \qquad \Gamma \vdash E_e : \tau}{\Gamma \vdash \textbf{if}\ E_c\ \textbf{then}\ E_t\ \textbf{else}\ E_e : \tau} \quad \text{[if]}$$

Figura 3.5: Regole di inferenza del *sistema HM* in **Funx**

L'*algoritmo* $\mathcal{W}$[7] riportato in Figura 3.6 fornisce un'implementazione del metodo finora descritto, servendosi delle regole di inferenza e dettagliando gli step per ogni espressione disponibile in **Funx**.
L'algoritmo appare nel codice del compilatore in una forma più complessa, soprattutto nel caso dell'espressione `let` (vedi sezione **??**), volendo supportare la ricorsione senza far uso di alcun *fixed-point combinator*.

[7] «Proofs about a Folklore Let-Polymorphic Type Inference Algorithm», [LY98]

$$\mathcal{W} : Context \times Expression \to Substitution \times Type$$

$$\begin{array}{lcll}
\mathcal{W}(\Gamma, c) & = & (id, constantType(c)) \\
\mathcal{W}(\Gamma, x) & = & \text{let} & \forall\ \vec{\alpha}\ .\ \tau = \Gamma(x),\ new\ \vec{\beta} \\
 & & \text{in} & (id, \{\vec{\beta}/\vec{\alpha}\}\tau) \\
\mathcal{W}(\Gamma, E_l\ E_r) & = & \text{let} & (S_l, \tau_l) = \mathcal{W}(\Gamma, E_l) \\
 & & & (S_r, \tau_r) = \mathcal{W}(S_l\Gamma, E_r) \\
 & & & S_a = \mathcal{U}(S_r\tau_l, \tau_r \to \beta),\ new\ \beta \\
 & & \text{in} & (S_aS_rS_l, S_a\beta) \\
\mathcal{W}(\Gamma, \lambda x\ .\ E) & = & \text{let} & (S, \tau) = \mathcal{W}(\Gamma + x : \beta, E),\ new\ \beta \\
 & & \text{in} & (S, S\beta \to \tau) \\
\mathcal{W}(\Gamma, \textbf{let}\ \vec{x} = \vec{E}\ \textbf{in}\ E_i) & = & \text{let} & \overline{\Gamma} = \Gamma + \vec{x} : \vec{\beta},\ new\ \vec{\beta} \\
 & & & (\vec{S}, \vec{\kappa}) = \mathcal{W}(\overline{\Gamma}, \vec{E}) \\
 & & & (S_i, \tau) = \mathcal{W}(\vec{S}\,\overline{\Gamma}\{Clos_\Gamma(\vec{\kappa})/\vec{\beta}\}, E_i) \\
 & & \text{in} & (S_i\vec{S}, \tau) \\
\mathcal{W}(\Gamma, \textbf{if}\ E_c\ \textbf{then}\ E_t\ \textbf{else}\ E_e) & = & \text{let} & (S_c, \tau_c) = \mathcal{W}(\Gamma, E_c) \\
 & & & (S_t, \tau_t) = \mathcal{W}(S_c\Gamma, E_t) \\
 & & & (S_e, \tau_e) = \mathcal{W}(S_tS_c\Gamma, E_e) \\
 & & & S_{cb} = \mathcal{U}(\tau_c, Bool) \\
 & & & S_{te} = \mathcal{U}(S_e\tau_t, \tau_e) \\
 & & \text{in} & (S_{te}S_{cb}S_eS_tS_c, S_{te}\tau_t)
\end{array}$$

Figura 3.6: Algoritmo $\mathcal{W}$

Dato un contesto in cui analizzare un'espressione, $\mathcal{W}$ restituisce un tipo per l'espressione e una sostituzione. Quest'ultima può essere vuota (id), generata manualmente ($\{\tau_2/\tau_1\}$) o dall'*unificazione* di altri tipi, e rappresenta una mappatura tra variabili di tipo e altri tipi; può essere applicata a tipi e contesti.

La funzione *new* esprime la creazione di nuove variabili di tipo, con il vincolo che non siano state ancora usate dall'algoritmo.

Infine, l'*unificazione* (funzione $\mathcal{U}$) è un'operazione altrettanto importante per l'inferenza di tipo, con la quale si cerca di trovare una sostituzione che renda due tipi uguali.

$$\mathcal{U} : Type \times Type \to Substitution$$

$$\begin{array}{lcl}
\mathcal{U}(\alpha, \alpha) & = & id \\
\mathcal{U}(\alpha, \tau) & = & \{\tau/\alpha\} \\
\mathcal{U}(\tau, \alpha) & = & \{\tau/\alpha\} \\
\mathcal{U}(F\ \vec{\tau}, F\ \vec{\kappa}) & = & \mathcal{U}(\vec{\tau}, \vec{\kappa})
\end{array}$$

Figura 3.7: Funzione $\mathcal{U}$

4. Java

Parallelamente alle prime fasi di sviluppo è stata svolta un'analisi di `Java`[1] per valutare quali fossero le caratteristiche del linguaggio utili alla traduzione di codice **Funx**.
Questo capitolo riporta pertanto una breve panoramica delle principali funzionalità impiegate, accompagnate da esempi di traduzione che illustrano alcuni dei risultati ottenibili con il compilatore (si ricorda che altri esempi sono esibiti nel Capitolo 6).

Le funzioni non dichiarate all'interno degli esempi stessi provengono dalla piccola libreria standard (suddivisa in `FunxPrelude` e `JavaPrelude`), parzialmente presentata nella Tabella 2.1 con gli operatori simbolici.

4.1 Interfacce funzionali

Nel tradurre un linguaggio funzionale viene naturale pensare immediatamente alle *interfacce funzionali* e *lambda espressioni* introdotte in `Java` 8[2] per rappresentare funzioni anonime: l'interfaccia generica `Function` è la più adatta a riprodurre il comportamento dell'astrazione, come mostrato nei Codici 4.1 e 4.2.

```
1 constant : a -> b -> a
2 constant x = \y -> x
```

Codice 4.1: Semplice funzione in **Funx**

```
1 public static <a, b> Function<a, Function<b, a>> constant() {
2     return (x -> (y -> x));
3 }
```

Codice 4.2: Corrispondente metodo in `Java`

Dato il naturale *currying* di oggetti `Function`, questo tipo di traduzione ha il vantaggio di permettere l'applicazione parziale di funzioni (tramite una sequenza di `apply()` in numero minore rispetto al totale degli input), ma il grande svantaggio della creazione di una nuova istanza della funzione per ogni chiamata.

Poiché `constant` ha tipo polimorfo (vedi sezione 4.3), il metodo utilizza parametri di tipo e deve quindi necessariamente restituire un nuovo oggetto con ogni chiamata: nonostante la performance delle traduzioni non sia un obiettivo primario del progetto, la versione attuale del compilatore fa uso di alcune piccole ottimizzazioni nella trasposizione delle funzioni monomorfe, approfondite nella sezione **??**.

[1] OpenJDK (`https://openjdk.org`), Open source JDK implementation
[2] OpenJDK 8 (`https://openjdk.org/projects/jdk8`)

In aggiunta alla classe `Function`, nella parte nativa (codice `Java`) della libreria standard di **Funx** è definita un'ulteriore interfaccia funzionale per creare espressioni `let`: in questo caso vi è un grande utilizzo di classi anonime, potenzialmente annidate, rappresentanti le dichiarazioni locali e l'espressione principale.

```java
@FunctionalInterface
public interface Let<T> {
   T _eval();
}
```

Codice 4.3: Interfaccia funzionale per espressioni `let`

```
hundredsSum : Int -> Int -> Int
hundredsSum = let
      on : (a -> a -> b) -> (c -> a) -> c -> c -> b
      on op f x y = op (f x) (f y)
   in on add (multiply 100)
```

Codice 4.4: Espressione `let` in **Funx**

```java
public static Function<Long, Function<Long, Long>> hundredsSum;

static {
   hundredsSum = (new Let<>() {
      private <a, b, c>
         Function<
               Function<a, Function<a, b>>,
               Function<Function<c, a>, Function<c, Function<c, b>>>>
            on() {
         return (op -> (f -> (x -> (y -> op.apply(f.apply(x)).apply(f.apply(y)))))));
      }

      @Override
      public Function<Long, Function<Long, Long>> _eval() {
         return this.<Long, Long, Long>on().apply(add).apply(multiply.apply(100L));
      }
   })._eval();
}
```

Codice 4.5: Corrispondente classe anonima in `Java`

Nei Codici 4.4 e 4.5 si può vedere che la funzione `hundredsSum` è implementata attraverso la chiamata al metodo principale dell'interfaccia funzionale `Let`, realizzato internamente alla classe anonima con il supporto del metodo polimorfo `on`.

Inoltre, è immediatamente evidente come le traduzioni in `Java` siano progressivamente più complesse e meno leggibili con l'introduzione di nuove funzionalità: l'esempio più eclatante è dato proprio dalla *signature* del metodo locale `on`, divenuta di difficile comprensione rispetto alla sintassi molto concisa del linguaggio funzionale.

4.2 Operatore ternario

Una delle peculiarità della traduzione da **Funx** a `Java` è l'uso dell'operatore ternario (`condition ? thenBranch : elseBranch`) ogni qualvolta siano presenti espressioni condizionali `if-then-else`.

I linguaggi funzionali sfruttano spesso una caratteristica (non menzionata nella sezione 2.1) che prende il nome di *lazy evaluation* (valutazione pigra): fino a quando il risultato di un'espressione non è richiesto per un successivo calcolo, questa non verrà completamente valutata.
Oltre ad offrire molteplici possibilità di ottimizzazione dal punto di vista del tempo di esecuzione, tale comportamento è molto comodo nella scrittura di funzioni che per esempio potrebbero terminare prima del previso o magari effettuare computazioni su strutture dati infinite. I linguaggi che sono *lazy evaluated* di default impegano nella maggior parte dei casi un *garbage collector* per liberare la memoria occupata dalle espressioni non valutate e non più rilevanti.

Il linguaggio `Java` non adotta la *lazy evaluation* di default se non in casi particolari, tra cui gli operatori booleani binari, il costrutto `if-then-else` (e corrispondente operatore ternario) e altre funzionalità più avanzate tra cui gli *stream* e le *lambda espressioni* già viste. Utilizzando quest'ultime si potrebbero ottenere risultati simili, in termini di valutazione pigra, a quelli di un linguaggio funzionale; tuttavia, rendere **Funx** un linguaggio completamente pigro avrebbe comportato una traduzione indubbiamente ancora più complessa, molteplici rischi di peggiorare le prestazioni dei programmi e un'implementazione del compilatore che va oltre lo scopo di questo progetto.

Nonostante ciò, la scelta di ridurre gli operatori booleani binari (*and* e *or*) ad espressioni con operatore ternario è stata considerata quasi obbligatoria per conservarne la natura pigra: gli operatori ternari utilizzati a questo scopo derivano direttamente dalla costruzione dell'**AST** (sezione **??**), motivo per cui non vengono riconvertiti in operatori nativi di `Java` in fase di traduzione.

```
power : Int -> Int -> Int
power b e = if e == 0 then 1 else b * power b (e - 1) fi

xor : Bool -> Bool -> Bool
xor a b = (a || b) && !!(a && b)
```

Codice 4.6: If e operatori booleani in **Funx**

```
public static Function<Long, Function<Long, Long>> power;

public static Function<Boolean, Function<Boolean, Boolean>> xor;

static {
   power = (b -> (e -> ((JavaPrelude.<Long>equalsEquals().apply(e).apply(0L))
      ? (1L)
      : (multiply.apply(b)
         .apply(power.apply(b).apply(subtract.apply(e).apply(1L)))))));

   xor = (a -> (b ->
      ((((a) ? (true) : (b))) ? (not.apply(((a) ? (b) : (false)))) : (false))));
}
```

Codice 4.7: Corrispondenti operatori ternari in `Java`

4.3 Tipi generici

Il sistema di tipo di **Funx** necessita la traduzione di funzioni polimorfe, e la soluzione più semplice e idiomatica in `Java` è l'utilizzo dei *generics*: tramite i parametri di tipo generici è possibile definire classi e metodi che agiscono su molteplici tipi di dati, implementando comportamenti che possono essere condivisi dai diversi elementi del dominio di tipi delle funzioni rappresentate.

Nel contesto del *sistema HM* di **Funx**, le variabili quantificate universalmente nei politipi hanno una diretta corrispondenza con i parametri di tipo che possono essere dichiarati tra i modificatori di visibilità e il tipo di ritorno di un metodo, il quale a sua volta combacia con la parte interna dello schema di tipo.

Nei Codici 4.8 e 4.9 si può notare come `Java` non sia in grado di inferire i tipi desiderati per le funzioni polimorfe: queste devono infatti essere istanziate esplicitamente usando la classe di appartenenza e le parentesi angolari (questa limitazione richiederà alcuni espedienti in casi limite illustrati nella sezione **??**).

```
1  sumToN : Int -> Int
2  sumToN = let
3        ap : (a -> b -> c) -> (a -> b) -> a -> c
4        ap op f x = op x (f x)
5    in (flip divide 2) . ap multiply (add 1)
```

Codice 4.8: Scrittura e utilizzo di funzioni polimorfe in **Funx**

```
1  public static Function<Long, Long> sumToN;
2
3  static {
4     sumToN = (new Let<>() {
5        private <a, b, c>
6           Function<
7              Function<a, Function<b, c>>,
8              Function<Function<a, b>, Function<a, c>>>
9                 ap() {
10          return (op -> (f -> (x -> op.apply(x).apply(f.apply(x)))));
11       }
12
13       @Override
14       public Function<Long, Long> _eval() {
15          return FunxPrelude.<Long, Long, Long>compose()
16             .apply(FunxPrelude.<Long, Long, Long>flip().apply(divide).apply(2L))
17             .apply(this.<Long, Long, Long>ap().apply(multiply).apply(add.apply(1L)));
18       }
19    })._eval();
20 }
```

Codice 4.9: Corrispondente traduzione in `Java`

5. Compilatore

Il software sviluppato per il progetto è stato sviluppato mantenendo il codice sul repository `GitHub` **Funx-jt**[1], nel cui nome *"jt"* è un acronimo per *"Java Transpiler"*.
Una versione stabile è disponibile sul repository come `Funx-jt-0.1.0`; la versione di `Java` utilizzata è la più recente *Long Term Support (LTS)*, versione 21 di OpenJDK[2].

Nel corso di questo capitolo si discuteranno le fasi di compilazione e la struttura del software, analizzando nel dettaglio le parti più importanti.

[1] massimopavoni/Funx-jt (`https://github.com/massimopavoni/Funx-jt`)
[2] OpenJDK 21 (`https://openjdk.org/projects/jdk/21`)

6. Esempi di traduzione

7. Conclusioni

Bibliografia

[Bar91] Henk Barendregt. «Introduction to generalized type systems». In: *Journal of Functional Programming* 1.2 (apr. 1991), pp. 125–154. DOI: 10.1017/s0956796800020025.

[Chu32] Alonzo Church. «A Set of Postulates for the Fundation of Logic». In: *Annals of Mathematics* 33.2 (apr. 1932), pp. 346–366. DOI: 10.2307/1968337.

[Chu33] Alonzo Church. «A Set of Postulates for the Foundation of Logic». In: *Annals of Mathematics* 34.4 (ott. 1933). Second paper, pp. 839–864. DOI: 10.2307/1968702.

[Clé+86] Dominique Clément et al. «A Simple Applicative Language: Mini-ML». In: *Proceedings of the 1986 ACM conference on LISP and functional programming - LFP '86* (8 ago. 1986), pp. 13–27. DOI: 10.1145/319838.319847.

[DM82] Luis Damas e Robin Milner. «Principal type-schemes for functional programs». In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82* (25 gen. 1982), pp. 207–212. DOI: 10.1145/582153.582176.

[Eis15] Richard A. Eisenberg. «System FC, as implemented in GHC». In: (2015). Last revised in 2020. URL: https://raw.githubusercontent.com/ghc/ghc/master/docs/core-spec/core-spec.pdf.

[Hin69] J. Roger Hindley. «The Principal Type-Scheme of an Object in Combinatory Logic». In: *Transactions of the American Mathematical Society* 146 (dic. 1969), pp. 29–60. DOI: 10.2307/1995158.

[LY98] Oukseh Lee e Kwangkeun Yi. «Proofs about a Folklore Let-Polymorphic Type Inference Algorithm». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.4 (1 lug. 1998), pp. 707–723. DOI: 10.1145/291891.291892.

[Mil78] Robin Milner. «A Theory of Type Polymorphism in Programming». In: *Journal of Computer and System Sciences* 17.3 (dic. 1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[Mog91] Eugenio Moggi. «Notions of Computation and Monads». In: *Information and Computation* 93.1 (lug. 1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

[Wel99] Joe B. Wells. «Typability and type checking in System F are equivalent and undecidable». In: *Annals of Pure and Applied Logic* 98.1-3 (30 giu. 1999), pp. 111–156. DOI: 10.1016/s0168-0072(98)00047-5.

Ringraziamenti

Ringrazio...