Indice

Capitolo 1

Introduzione

Il lavoro da me svolto consiste nell'implementazione dell'algoritmo di type inference di Haskell in ELPI.

1.1 Stato dell'arte

1.1.1 Premesse

Iniziamo questa trattazione ponendo alcune basi. Esse faciliteranno la comprensione degli argomenti successivi.

Haskell Haskell è un linguaggio di programmazione che adotta il paradigma di programmazione funzionale. Al suo interno è presente il lambda calcolo e il meccanismo delle type class; è presente inoltre una parte più ampia che corrisponde alle librerie.

Type inference La type inference è il rilevamento automatico del tipo di dato di un'espressione in un linguaggio di programmazione. La capacità di dedurre i tipi automaticamente semplifica molte attività di programmazione, lasciando il programmatore libero di omettere le annotazioni sui tipi pur consentendo il type check.

ELPI ELPI è un linguaggio di programmazione logico. Esso è un'estensione con vincoli del linguaggio λProlog, il quale a sua volta è un'estensione di Prolog a una logica di ordine superiore.

Prolog Prolog è un linguaggio di programmazione che adotta il paradigma di programmazione logica. Si basa sul calcolo dei predicati (logica del prim'ordine); la sintassi è composta da formule dette clausole. L'esecuzione di un programma Prolog è comparabile alla dimostrazione di un teorema mediante la regola di inferenza detta risoluzione (essa permette di passare da un numero finito di proposizioni assunte come premesse a una proposizione che funge da conclusione). I concetti fondamentali di questo linguaggio sono l'unificazione, la ricorsione in coda e il backtracking.

Clausola La clausola è una disgiunzione di letterali del prim'ordine. Essa è della forma:

$$H :- A_1, \ldots, A_n. \tag{1.1}$$

La semantica è quella di una implicazione rovesciata: $H \Leftarrow A_1 \wedge \ldots \wedge A_n$. Se n = 0 il simbolo $:-$ è omesso, ma non il punto finale. Un programma logico è un insieme di clausole. Una query è una sequenza di atomi $A_1, \ldots, A_n$.

λProlog λProlog è, come già detto, un'estensione di Prolog. Le caratteristiche principali in aggiunta, rispetto a Prolog, sono il polimorfismo, la programmazione di ordine superiore e il lambda calcolo tipato.

1.1.2 Limiti di λProlog

Non essendo λProlog un linguaggio di programmazione con vincoli risulta impossibile implementare la type inference mediante esso. I limiti dell'utilizzo di tale linguaggio si riscontrano in particolare nel tentativo di codificare il tipaggio per i costrutti del let-in e della type class.

L'unica strategia percorribile sarebbe quella di codificare interamente il sistema punto per punto. In tal caso però si creerebbe un sistema estremamente rigido, il che rende tale strategia impraticabile. Ad esempio, infatti, si potrebbe codificare tutto nelle stringhe e svolgere ogni operazione attraverso di esse; a quel punto ogni cosa sarebbe implementabile, il sistema in questione sarebbe infatti Turing completo. Il problema però è il fatto che in tal caso si dovrebbero abbandonare tutte le features del linguaggio λProlog; infatti la rigidezza del sistema creato non consentirebbe l'utilizzo delle stesse, poiché si risulterebbe vincolati alla nuova struttura.
Si è reso dunque necessario l'utilizzo di ELPI, la cui maggiore espressività (non nel senso della Turing completezza) permette di svolgere operazioni impossibili da codificare in λProlog. Prendiamo come esempio i due casi indicati precedentemente:

- Per codificare il tipaggio del let-in è necessario l'utilizzo del meccanismo mode di ELPI. Infatti esso offre un maggior controllo sugli elementi del codice poiché permette di accorgersi se essi sono delle variabili non istanziate, così da poterle gestire in modo appropriato.

- Per codificare il tipaggio della type class è necessario l'utilizzo dei vincoli. Infatti questi possono sussistere anche non totalmente istanziati e quindi permettono, ad esempio, di fissare l'obbligo di appartenenza di una variabile di tipo non istanziata ad un'istanza di type class.

Entrambi i requisiti sono caratteristiche presenti in ELPI ma non in λProlog. Risulta dunque evidente la necessità di utilizzare ELPI come linguaggio di programmazione per poter raggiungere lo scopo prefissato.

1.2 ELPI

1.2.1 Introduzione al linguaggio

ELPI, così come λProlog, è un linguaggio logico di ordine superiore (HOLP language - Higher Order Logic Programming language). La loro

differenza consiste nel fatto che ELPI possiede in aggiunta il sistema dei vincoli: infatti esso è un Higher Order constraint Logic Programming language. Spieghiamo alcuni concetti:

HOLP La programmazione di ordine superiore è uno stile di programmazione che usa elementi del linguaggio (ad esempio funzioni, oggetti, ...) come valori. Ad esempio le funzioni possono essere passate come argomenti di altre funzioni oppure possono essere il valore di ritorno di altre funzioni. È solitamente istanziato con il modello di computazione del lambda calcolo, il quale utilizza funzioni di ordine superiore.

CLP La programmazione logica con vincoli (Constraint Logic Programming), estensione della programmazione logica, è un paradigma di programmazione dove le relazioni fra variabili possono essere dichiarate in forma di vincoli. Un vincolo è una formula della logica del prim'ordine (solitamente una congiunzione di formule atomiche) che usa solo predicati di significato predefinito.

1.2.2 Sistema di propagazione dei vincoli

Spieghiamo ora il sistema dei vincoli, nel caso specifico di ELPI.
Generalmente l'approccio con ELPI è il seguente:

1. Si dichiara il `mode`;
2. Lo si trasforma in un vincolo;
3. Se si ha dei teoremi li si aggiunge, essi sono utili per evitare un accumulo non necessario di vincoli.

Primo punto Si raggiunge definendo il `mode` di un costrutto. Saranno presenti un numero di `i` e di `o` pari, rispettivamente, alla quantità di elementi in input e in output al costrutto.
Struttura:

`mode (`*costrutto* `i o).`

Secondo punto Si risolve dichiarando che un predicato contenente quel costrutto, il quale presenta in input una o più variabili non istanziate, può essere soddisfatto aggiungendo al programma un vincolo contentente quel costrutto al cui interno saranno presenti le variabili non istanziate; viene inoltre fissata una lista delle variabili non istanziate la quale, una volta soddisfatta (cioè una volta istanziate le variabili in essa contenute), dà il via alla processazione del vincolo. Infatti un vincolo viene ricordato senza essere processato: sarà processato soltanto nel momento in cui le variabili ad esso associate saranno istanziate.
Struttura:

costrutto `(uvar _ as` *variabile non istanziata*`) :-`
`!, declare_constraint (`*vincolo*`) [`*variabili non istanziate*`].`

Terzo punto Si ottiene definendo uno o più blocchi `constraint`. Al suo interno saranno presenti una o più `rule`, regole di riscrittura dei vincoli. Esse sono della forma:

`rule` *tengo* `\` *tolgo* `<=>` *condizione* `|` *aggiungo*`.`

Oppure possono trovarsi in una delle forme ristrette:

`rule \` *tolgo* `<=>` *condizione* `|` *aggiungo*`.`
`rule` *tengo* `\` *tolgo* `<=>` *aggiungo*`.`
`rule \` *tolgo* `<=>` *aggiungo*`.`

Struttura:

`constraint` *costrutto* `{`*regole*`}`

Il sistema di propagazione dei vincoli consiste nella riscrittura di insiemi di vincoli in altri insiemi di vincoli.

Esempi

Per comprendere meglio il concetto presento ora due frammenti di codice, estrapolati dal mio lavoro.

- Implementazione di `of`, il costrutto utilizzato per codificare il tipaggio di un termine.

```
type of term -> tipo -> prop.
mode (of i o).
of (uvar _ as X) T :- !, declare_constraint (of X T) [X].
```

- Implementazione di `eqp_tipo`, il costrutto utilizzato per la codifica dell'uguaglianza di due tipi.

```
type eqp_tipo tipo -> tipo -> prop.
constraint eqp_tipo {
 rule \ (eqp_tipo A A) <=> true.
 rule \ (eqp_tipo A B) <=> false.
}
```

1.3 Haskell

Il mio lavoro comprende l'implementazione di una parte di Haskell, non di tutto il linguaggio. Presento ora gli argomenti da me trattati.

1.3.1 BNF

T ::= t | T $\rightarrow$ T

E ::= e | E E | λe.E | case E of $a_1 \ldots a_n$ | let $d_1 \ldots d_n$ in E | if E then E else E

Indichiamo con T i tipi e con E i termini (o espressioni).

1.3.2 Let-in

Il costrutto del let-in è costituito da:

- Una lista di dichiarazioni che associano, ciascuna, un nome ad un termine;
- Un corpo che è un'espressione che contiene i termini definiti nelle dichiarazioni.

1.3.3 Type class

Un altro elemento da cui Haskell è costituito è il meccanismo delle type class. Una type class è un costrutto di un sistema di tipi che supporta il polimorfismo parametrico. Esso è ottenuto aggiungendo vincoli alle variabili di tipo. Questi tipicamente coinvolgono una type class e una variabile di tipo e indicano che la variabile può essere istanziata solamente con un tipo che soddisfa le operazioni associate alla type class. Una type class può quindi essere considerata come una sorta di insieme vincolato di tipi.
Paragonando questo sistema con il paradigma di programmazione orientato agli oggetti potremmo dire che, rispettivamente, le type class stanno ai tipi come le classi stanno agli oggetti. Il meccanismo delle type class, infatti, nasce come risposta ad alcuni problemi dovuti all'utilizzo del meccanismo dell'ereditarietà nella programmazione orientata agli oggetti.

Capitolo 2

Implementazione

Presento di seguito il mio lavoro, con alcuni estratti del codice. Esso consiste nella codifica della sintassi di Haskell e nella implementazione della type inference con le type class di Haskell.

2.1 STLC

Con Simply Typed Lambda Calculus si fa riferimento, per l'appunto, al lambda calcolo tipato semplice. Esso è una forma di teoria dei tipi; è un'interpretazione tipata del lambda calcolo con un solo tipo costruttore ($\rightarrow$) il quale costruisce i tipi delle funzioni. I termini del lambda calcolo possono essere: un nome di variabile, l'applicazione di un termine come argomento di un altro termine oppure l'astrazione di un termine rispetto ad una variabile. Diamo la sua sintassi mediante la seguente grammatica:

$$\mathrm{T} ::= \mathrm{t} \mid \mathrm{T} \rightarrow \mathrm{T}$$

$$\mathrm{E} ::= \mathrm{e} \mid \mathrm{E}\ \mathrm{E} \mid \lambda \mathrm{e}.\mathrm{E}$$

Indichiamo con T i tipi e con E i termini (o espressioni).

2.1.1 Implementazione

```
kind tipo type.
```

```
type arr tipo -> tipo -> tipo.

kind term type.
type app term -> term -> term.
type lam (term -> term) -> term.

type of term -> tipo -> prop.
mode (of i o).
of (uvar _ as X) T :- !, declare_constraint (of X T) [X].
of (app X Y) B :- of X (arr A B), of Y A.
of (lam F) (arr A B) :- pi x \ of x A => of (F x) B.
```

Per comprendere l'ultima linea di codice è necessaria una spiegazione aggiuntiva rispetto a quanto detto fino ad ora.
Essa permette di tipare la lambda astrazione. Per ottenere questo risultato è stato necessario l'utilizzo di due costrutti particolari: il `pi` e l'implicazione (`=>`).

Quantificazione universale Il costrutto `pi` corrisponde al quantificatore universale (∀). Esso restituisce una costante, in questo caso la `x`, quantificata universalmente da utilizzare all'interno dell'espressione.
Struttura:

`pi` *costante* `\` *espressione* `.`

Implicazione Il costrutto `=>` corrisponde all'implicazione (⇒). Esso, assumendo l'ipotesi, tenta di soddisfare la tesi.
Struttura:

ipotesi `=>` *tesi* `.`

2.2 Funzioni ricorsive, match

2.3 Type class, istanza, schema

2.4 Let-in

Capitolo 3

Conclusioni

3.1 Riassumendo

3.2 Finalità

3.3 La mia esperienza con ELPI

3.4 Il mio lavoro

3.5 Sviluppi futuri

3.5.1 Parser

3.5.2 Testing

3.5.3 Estensioni

Ringraziamenti