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 (e.g. 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.

Punto 1. Si raggiunge definendo il `mode` di un predicato. Saranno presenti un numero di `i` e di `o` pari, rispettivamente, alla quantità di elementi in input e in output al predicato.
Struttura:

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

Punto 2. Si risolve dichiarando che una proposizione contenente un predicato, il quale presenta in input una o più variabili non istanziate, può essere soddisfatto aggiungendo al programma un vincolo contenente tale predicato al cui interno saranno presenti le variabili non istanziate; viene inoltre fissata una lista delle variabili non istanziate la quale, una volta soddisfatta (i.e. 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:

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

Punto 3. 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` *predicato* `{`*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 predicato 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 predicato 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 Case

Il case è un costrutto che prende in input un termine e restituisce un determintato termine di output in base al genere di termine ricevuto. Esso è costituito da una serie di alternative ($a_1 \ldots a_n$), che sono coppie che associano ad un genere di termini di input un termine di output.
Per capire il genere dell'input viene utilizzata la tecnica del pattern matching sul termine stesso confrontandolo con gli input delle alternative del case.

1.3.3 Let-in

Il costrutto del let-in è costituito da:

- Una lista di dichiarazioni ($d_1 \ldots d_n$) che associano, ciascuna, un nome ad un termine;
- Un corpo che è un'espressione all'interno della quale si possono utilizzare i termini definiti nelle dichiarazioni.

1.3.4 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

Viene presentato 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, il linguaggio utilizzato è ELPI.
Per la consultazione del codice completo si rimanda al repository di GitHub (vedere il riferimento [1] della bibliografia).

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

Codice

```
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.
```

Spiegazione

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 di ELPI: il `pi` e l'implicazione (`=>`).

Quantificazione universale Il costrutto `pi` corrisponde al quantificatore universale ($\forall$). 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 ($\Rightarrow$). Esso, assumendo l'ipotesi, tenta di soddisfare la tesi.
Struttura:

$$\textit{ipotesi} \texttt{ => } \textit{tesi} \ .$$

2.2 Case e funzioni ricorsive

Implementare il costrutto case ha richiesto l'introduzione di una codifica per gli algebraic data types, una sorta di tipi composti formati dalla combinazione di altri tipi. Essi infatti permettono di classificare il termine in input ricollegandolo ad una classe di appartenenza (e.g. numeri naturali, liste, alberi, ...).
Per la consultazione dell'implementazione del costrutto case si rimanda all'appendice A.

Nell'implementazione di Haskell sono comprese anche le funzioni ricorsive. Dopo aver dichiarato una funzione, infatti, essa può essere richiamata anche all'interno del corpo della medesima.

2.3 Type class e schema

Per codificare il tipaggio nel meccanismo delle type class è stato inserito all'interno dell'implementazione il concetto di schema. In particolare esso viene utilizzato nelle dichiarazioni di funzione, per definire il tipo della stessa. Lo schema, infatti, è un tipo con, al suo interno, delle variabili quantificate universalmente ($\forall$, in codice `for_all`). Esso è costituito da due parti: la lista di tali variabili e il tipo.
Mostriamone la sintassi con un esempio.
Per il tipo

$$\forall x, y : x \to y \tag{2.1}$$

la struttura del costrutto è la seguente

```
for_all x \ for_all y \ base (arr x y)
```

2.3.1 Implementazione

Codice

```
kind schema type.
type for_all (tipo -> schema) -> schema.
type base tipo -> schema.

kind nome_fun type.
kind declaration type.
type fun_decl nome_fun -> schema -> declaration.
kind implementation type.
type fun_impl nome_fun -> term -> implementation.

type fun nome_fun -> term.
of (fun N) T :-
 typeclass N1 T1 L,
 is_fun_in_decls L N S,
 instantiate S T,
 istanza N1 T1 _.

kind nome_classe type.
type typeclass
 nome_classe -> tipo -> list declaration -> prop.
type istanza
 nome_classe -> tipo -> list implementation -> prop.
mode (istanza i i o).
istanza N (uvar _ as X) L :-
 !, declare_constraint (istanza N X L) [X].
```

```
type is_fun_in_decls
 list declaration -> nome_fun -> schema -> prop.

kind holds type.
type is_instance nome_classe -> tipo -> holds.
type implies holds -> schema -> schema.
type instantiate schema -> tipo -> prop.
instantiate (base T) T.
instantiate (for_all S) O :- instantiate (S T_) O.
instantiate (implies (is_instance N T) S) O :-
 istanza N T _, instantiate S O.
```

Spiegazione

Per comprendere la sintassi di un tipo che deve sottostare ad una type class presentiamo un esempio.
Per il tipo (2.1), con x che debba soddisfare un'ipotetica type class *comparable* e y una type class *printable*, la struttura del costrutto è la seguente:

```
for_all x \ for_all y \
 (implies (is_instance comparable x)
  (implies (is_instance printable y)
   (base (arr x y))
```

La proposizione `of (fun N) T` tipa una generica funzione contenuta in una type class. Tale risultato è ottenuto attraverso la collaborazione di più predicati:

1. Il predicato `typeclass` cerca nel programma una definizione di type class.

2. Il predicato `is_fun_in_decls`, presi in input il nome della funzione e la lista delle dichiarazioni di funzione della type class, controlla che sia presente nella lista una funzione con tale nome (i.e. che la type

class contenga al suo interno quella specifica funzione) e restituisce lo schema ad essa associato.
L'implementazione di tale predicato è omessa in questa trattazione.

3. Il predicato `instantiate` istanzia lo schema ad essa associato in un tipo che sarà, per l'appunto, il tipo della funzione all'interno della proposizione.

4. Il predicato `istanza` aggiunge un vincolo di appartenenza di tipo all'istanza della type class.

2.4 Let-in

L'implementazione del costrutto del let-in non verrà presentata nella sua interezza. Sarà mostrata, con le dovute spiegazioni, la parte più ardua alla comprensione e i punti di snodo tra i diversi passaggi dell'implementazione del tipaggio del costrutto.

Per facilitare la comprensione del codice, e della sua spiegazione, si introduce ora il concetto della generalizzazione.
Esso consiste nel generalizzare un tipo in uno schema, una sorta di operazione inversa a quella del predicato `instantiate` (mostrato nella sezione 2.3.1). Ogni variabile non istanziata presente nel tipo soggetto a tale operazione verrà generalizzata, attraverso il predicato `for_all`, nel rispettivo schema; lo schema risultante conterrà, quindi, il tipo con le variabili generalizzate.
Ad esempio, il tipo

$$x \to int \tag{2.2}$$

(in codice)

```
arr X int
```

sarà generalizzato in

$$\forall x : x \to int \tag{2.3}$$

(in codice)

```
for_all x \ base (arr x int)
```

2.4.1 Implementazione

Codice

```
kind decl type.
kind nome_decl type.
type let_in list decl -> term -> term.
type coppia_decl nome_decl -> term -> decl.

of (let_in L EXPR) T :-
 associa_list_let L L OUT, (OUT => of EXPR T).

type associa_list_let list decl -> list decl -> prop -> prop.
associa_list_let [] L OUT :- of_list_let L L OUT.
associa_list_let [coppia_decl N _C | TL] L OUT :-
 associa_schema N (base T_) => associa_list_let TL L OUT.

type of_list_let list decl -> list decl -> prop -> prop.
of_list_let [] L OUT :- generalize_list_let L OUT.
of_list_let [coppia_decl N C | TL] L OUT :-
 associa_schema N (base T),
 of C T,
 of_list_let TL L OUT.

type generalize_list_let list decl -> prop -> prop.
generalize_list_let [] true.
generalize_list_let
 [coppia_decl N C | TL] (associa_schema N S, OUT) :-
  associa_schema N (base T),
```

```
  free_vars_let C L,
  vars_in_type L L',
  generalize L' T S,
  generalize_list_let TL OUT.

type associa_schema nome_decl -> schema -> prop.
type decl_let nome_decl -> term.
of (decl_let N) T :-
 associa_schema N S, instantiate S T.

type generalize list tipo -> tipo -> schema -> prop.
generalize Lnot T S :-
 free_vars T Lnot L1,
 generalize_forall L1 L1 T S.

type generalize_forall
 list tipo -> list tipo -> tipo -> schema -> prop.
generalize_forall [] L T S :-
 generalize_instance_aux L T S.
generalize_forall [X|TL] L T (for_all S) :-
 pi x \ assoc X x => generalize_forall TL L T (S x).

type generalize_instance_aux
 list tipo -> tipo -> schema -> prop.
generalize_instance_aux [] T S :-
 generalize_instance [] [] T S.
generalize_instance_aux [X|TL] T S :-
 retrieve_constraints X LI,
 generalize_instance LI TL T S.

type generalize_instance
```

```
 list prop -> list tipo -> tipo -> schema -> prop.
generalize_instance [] [] T S :- copy T S.
generalize_instance [] TL T S :-
 generalize_instance_aux TL T S.
generalize_instance
 [istanza N X _ | LI] TL T (implies (is_instance N Y) S) :-
  assoc X' Y, test_eqp_tipo X X',
  generalize_instance LI TL T S.

type assoc tipo -> tipo -> prop.
mode (assoc i o).
```

Spiegazione

La proposizione `of (let_in L EXPR) T` tipa il costrutto del let-in.
Per raggiungere tale scopo essa si avvale del lavoro sinergico tra diversi diversi predicati.

- Il predicato `associa_list_let` esegue una prima passata delle dichiarazioni del let-in (una lista di proposizioni contenti il predicato `coppia_decl`) per associare uno schema (come variabile fresca, i.e. non istanziata) a ciascuna dichiarazione e assume tali associazioni attraverso l'implicazione (`=>`).
  Resituisce in output una serie di and ($\wedge$, in codice '`,`') fra queste associazioni, con gli schemi istanziati correttamente
  (e.g. `associa_schema N1 S1, ..., associa_schema Nn Sn`).

- Il predicato `of_list_let` esegue una seconda passata sulle stesse dichiarazioni per istanziare lo schema associato a ciascuna dichiarazione con il tipo del corpo della dichiarazione.

- Il predicato `generalize_list_let` esegue una terza, ed ultima, passata sulle medesime dichiarazioni per generalizzare il tipo associato a ciascuna dichiarazione nel rispettivo schema.

- Il predicato `free_vars_let` trova tutte le variabili libere (i.e. non legate dalla lambda astrazione (`lam`) o dal let-in) presenti nel corpo di ciascuna dichiarazione.
  L'implementazione di tale predicato è omessa in questa trattazione.

- Il predicato `vars_in_type` restituisce le variabili (di tipo) non istanziate contenute nei tipi delle variabili (di termine) libere del corpo delle dichiarazioni.
  L'implementazione di tale predicato è omessa in questa trattazione.

- Il predicato `associa_schema` associa il nome di una dichiarazione al rispettivo schema.
  Per tipare una dichiarazione del let-in (con la proposizione `of (decl_let N) T`) si cerca lo schema associato al nome di quella dichiarazione e viene restituito il tipo risultante dall'istanziazione di tale schema.

- Il predicato `generalize` implementa il meccanismo di generalizzazione dei tipi (escludendo alcune variabili di tipo, prese in input, dalla lista di quelle da generalizzare).

- Il predicato `free_vars` colleziona la lista, priva di duplicati, delle variabili nel tipo in input (escludendo quelle presenti nella lista presa in input).
  L'implementazione di tale predicato è omessa in questa trattazione.

- Il predicato `generalize_forall` mette, al posto delle variabili da generalizzare, delle costanti quantificate con un `for_all`.

- Il predicato `generalize_instance_aux` recupera l'eventuale vincolo di istanza di type class sulle variabili da generalizzare.

- Il predicato `generalize_instance` trasforma quei vincoli di istanza di type class in uno schema tramite i predicati `implies` e `is_instance`.

- Il predicato `copy` fa la copia ricorsiva di un tipo nel rispettivo schema (soltanto la parte del predicato `base`) sostituendo le variabili non istanziate con la rispettiva costante ad esse associata tramite il predicato `assoc`.
  L'implementazione di tale predicato è omessa in questa trattazione.

- La proposizione `assoc X x` associa la variabile `X` con la costante `x`.

- Il predicato `test_eqp_tipo` testa se due metavariabili (di tipo) sono uguali, senza unificarle.
  L'implementazione di tale predicato è omessa in questa trattazione.

Infine, dopo aver assunto la serie di proposizioni contenenti il predicato `associa_schema` (di ritorno del predicato `associa_list_let`) attraverso l'implicazione (`=>`), viene tipato il corpo del let-in.

Capitolo 3

Conclusioni

3.1 Riassumendo

Il problema da risolvere era il seguente: implementare in un linguaggio logico con vincoli di ordine superiore la type inference del linguaggio di programmazione Haskell.
Per risolvere tale problema si è utilizzato il linguaggio di programmazione ELPI, senza alcuna modifica aggiuntiva. L'impiego delle caratteristiche peculiari di ELPI è risultato necessario ed ha quindi vincolato all'impossibilità di utilizzare un linguaggio meno espressivo di ELPI (e.g. λProlog).

L'idea che si potesse utilizzare il linguaggio di programmazione ELPI e le sue features per risolvere tale problema annoso è stata suggerita dal dr. Enrico Tassi.

3.2 Finalità

Lo scopo del mio lavoro di tesi è duplice:

- Il primo, pienamente raggiunto, è mostrare che le estensioni a λProlog presenti in ELPI permettono di risolvere problemi non risolvibili in λProlog.

Si è dimostrato quindi che ELPI è più espressivo di λProlog, ma non nel senso della Turing completezza. Infatti, come spiegato nella sezione 1.1.2, è possibile ottenere la Turing completezza già con il linguaggio λProlog ma i limiti che necessariamente, per il raggiungimento di tale obiettivo, verrebbero imposti al linguaggio non rende tale strategia percorribile.

- Il secondo, lasciato agli sviluppi futuri, è fare del mio lavoro uno strumento di prova per testare, implementare e studiare nuove estensioni al meccanismo delle type class di Haskell.
  Il problema delle type class infatti è un problema complesso. Aggiungere estensioni alle type class nel codice di Haskell è molto complicato, dal momento che Haskell è un'implementazione complessa ed è implementato in un linguaggio più di basso livello. In ELPI invece dovrebbe risultare molto banale, essendo la mia implementazione più di alto livello e più semplice (grazie alle peculiari features del linguaggio).

3.3 La mia esperienza con ELPI

La mia esperienza con ELPI ha avuto inizio durante lo svolgimento del tirocinio curriculare interno, in preparazione alla tesi, presso l'Alma Mater Studiorum - Università di Bologna ed è proseguita poi nel lavoro di tesi. Ho lavorato sotto la supervisione del professore Claudio Sacerdoti Coen.
Il mio approccio al linguaggio è risultato faticoso essendo scarsa la mia conoscenza personale del paradigma di programmazione logico. Nonostante tale considerazione sono rimasti tuttora vivi in me, come precedentemente a questa esperienza, il desiderio e la curiosità di approfondire e lavorare ulteriormente con tale paradigma di programmazione (assieme a quello funzionale).

3.4 Il mio lavoro

Una stima del monte ore complessivo del mio lavoro di tesi è di circa 180 ore, per un totale di circa 600 linee di codice scritte nel linguaggio ELPI.

3.5 Sviluppi futuri

Si indicano ora i possibili lavori futuri, come sviluppo del mio lavoro di tesi.

Parser Un primo lavoro consiste nella realizzazione di un parser che prenda in input veri programmi Haskell (i.e. scritti in sintassi Haskell) e li trasformi nella presentazione ELPI (i.e. programmi Haskell scritti nella sintassi ELPI), così da renderlo comprensibile al mio programma. Infatti, al momento, la mia implementazione riesce ad elaborare programmi Haskell soltanto se scritti nella sintassi ELPI, quindi la presentazione ELPI dei programmi.

Testing Successivamente sarebbe doveroso effettuare un'operazione di test sul lavoro svolto. In tal modo si potrebbe capire la capacità computazionale effettiva della mia implementazione. Infatti, per mancanza di tempo, il codice è stato testato solamente in ridotti test di esempio e non in frammenti di codice o in librerie ampie.
Tale lavoro è impossibile da svolgere senza aver antecedentemente realizzato il parser di cui sopra. Infatti sarrebbe impossibile eseguire dei test esaustivi senza un valido sistema di input e quindi non si potrebbe conoscere l'efficienza dell'implementazione: non si saprebbe infatti quanto tempo impieghi nell'esecuzione di programmi complessi né si saprebbe la complessità effettiva dell'algoritmo. Senza lo svolgimento di test esaustivi si pone la domanda non tanto sulla correttezza quanto sull'efficienza della mia implementazione.

Estensioni Infine, come ultimo spunto per uno sviluppo futuro, si potrebbe implementare le estensioni al meccanismo base delle type class in Haskell. Esse sono meccanismi più complicati che, per l'appunto, estendono le funzionalità base delle type class.

Quindi, in conclusione, i possibili lavori futuri sono:

1. realizzare un parser per leggere vero codice Haskell;
2. fare test usando quel codice vero e capire la capacità computazionale effettiva della mia implementazione;
3. implementare le estensioni al meccanismo delle type class.

Appendice A

Implementazione del costrutto case

```
kind nome_costruttore_adt type.
type constructor nome_costruttore_adt -> list term -> term.

type case_of term -> list pair -> term.
kind pair type.
type dato (term -> pair) -> pair.
type branch term -> term -> pair.

of (case_of X L) T1 :- of X T2, check_list_case L T1 T2.

type check_list_case list pair -> tipo -> tipo -> prop.
check_list_case [] _ _.
check_list_case [branch A B | TL] T1 T2 :-
 of A T2, of B T1, check_list_case TL T1 T2.
check_list_case [dato F | TL] T1 T2 :-
 pi x \ of x TX_ => check_list_case [(F x) | TL] T1 T2.
```

Bibliografia

[1] https://github.com/danielepolidori/tesi_git - repository di GitHub contenente il mio lavoro di tesi.