*Questa è la* Dedica*:*
*ognuno può scrivere quello che vuole,*
*anche nulla . . .*

# Abstract

Lo studio presente in questo documento è il frutto di una ricerca più ampia condotta da Accattoli, Condoluci e Sacerdoti Coen ed esposto nel documento Sharing Equality is Linear [1].
Il lavoro svolto consiste in un'implementazione efficiente delle regole di riduzione di un $\lambda$-termine, codificando le stesse in un algoritmo, seguite dall'applicazione di tali regole per ridurre e comparare due $\lambda$-espressioni apparentemente differenti. L'output di tale algoritmo è un feedback che riporta se tali espressioni sono effettivamente differenti o se rappresentano la stessa espressione ridotta. Due $\lambda$-espressioni vengono considerate equivalenti a meno di un $\alpha-rinominazione$ ovvero di comparare i $\lambda$-termini a meno dei nomi delle variabili ad essi legate. Quindi due $\lambda$-espressioni equivalenti verranno considerate bi-simili.
Lo studio presente nel documento Sharing Equality is Linear [1] si basa su precedenti lavori condotti da Accattoli e Dal Lago ed esposti nel documento NOME_DOC [2], nel quale viene dimostrato che la riduzione forte di un $\lambda$-termine è consentita ovunque ed è ragionevole implementare un algoritmo che la esegua.
L'algoritmo ideato da Accattoli et alt. è ispirato all'algoritmo di Paterson e Wegman [3] per l'unificazione del primo ordine dove i $\lambda$-termini delle $\lambda$-espressioni vengono rappresentati sotto forma di DAG (Directed Acyclic Graph - Grafo Aciclico Diretto).
Nel seguito del documento verranno illustrate le regole di riduzione, i costrutti e le tecniche utilizzate nell'implementazione dell'algoritmo.

# Indice

# Capitolo 1

# Introduzione

Il progetto di tesi svolto consiste nella realizzazione di un algoritmo per la visita e la riduzione di $\lambda$-espressioni, per permetterne la comparazione e quindi determinare se due $\lambda$-espressioni sono bi-simili.
Le $\lambda$-espressioni prese in considerazione sono un formalismo per descrivere un programma software, nonché il fine ultimo dell'algoritmo è determinare se 2 programmi, a parità di input, producono lo stesso output senza eseguire tali programmi.
La modalità di rappresentazione delle $\lambda$-espressioni è la stessa suggerita da Paterson e Wegman ossia tramite un grafo aciclico diretto. Tale rappresentazione è risultata essere ottimale allo scopo prefissato poiché permette di creare una gerarchia fra i $\lambda$-termini dell'espressione e quindi determinare padri e figli di un termine in modo naturale.

L'algoritmo implementato consiste in una procedura che prende in input due $\lambda$-espressioni e ne compara i $\lambda$-termini sottostanti tramite una relazione di vicinato ($\sim$neighbour).
Ogni $\lambda$-espressione è codificata tramite un DAG e ogni nodo di tale grafo è inserito in un array globale utilizzato per visitare esattamente una volta ogni nodo. L'algoritmo può iniziare la valutazione da un qualsiasi nodo del DAG e tramite procedure ad-hoc risale sino al nodo root per poi iniziare la

comparazione con il secondo DAG.

La particolarità legata al progetto in esame è di eseguire tale comparazione in tempo lineare al numero di nodi presenti nell'array globale. Tale vincolo non è banale poiché l'algoritmo deve riuscire a visitare esattamente una volta ogni nodo e determinare se il ∼neighbour corrispondente rappresenta in realtà lo stesso termine ridotto a meno di $\alpha - rinominazione$. Nel *Capitolo 4 Algoritmo - AlphaLinearConversion* vedremo come tale relazione viene semplificata attraverso la definizione di un nodo canonico per ogni nodo. Tale canonico sarà il nodo stesso o il ∼neighbour corrispondente nell'altra $\lambda$-espressione, sempre che non accada prima che l'algoritmo rifiuti la comparazione perché ha rilevato che i 2 DAG presi in input non sono bi-simili.

Un'ulteriore particolarità è che la comparazione e la valutazione avvengano in parallelo, ovvero non vi è una vera è propria parallelizzazione dell'esecuzione, ma mentre analizza un termine per compararlo con i suoi ∼neighbour, se questo non è stato ancora valutato, verrà richiamata la procedura di valutazione. Un approccio di questo tipo permette di ottenere in minor tempo una risposta dall'algoritmo nel caso in cui non siano bi-simili poiché non vengono valutati tutti i termini.

I costrutti implementati nell'algoritmo comprendono tutti i costrutti tipici di un linguaggio funzionale e sono stati opportunamente codificati tramite nodi del grafo. Suddetti costrutti sono:

- variabili libere da contesto: $x$;
- variabili legate ad un ambiente chiamate binders: $b$;
- applicazioni per rappresentare un operazione binaria: $t@t$
- $\lambda$-astrazioni: $\lambda\, x : t.t$;
- sorte: $s$;

- tipi induttivi: $i$;
- costruttori j-esimo con n argomenti: $k_j[s_1, ..., s_n]$;
- costanti non definite in libreria: $a$;
- costanti definite in libreria: $c$;
- match: $match\ i\ t_0\ t\ \vec{t_i}$;
- piai (tipo delle funzioni): $\Pi\, x : t.t$;
- let per la dichiarazione di variabili locali o per abbreviazioni: $Let\ x : t := t\ in\ t$;
- funzioni ricorsive: $f^{(n)}\ [t_1, ..., t_n]$;
- funzioni co-ricorsive: $g^{(n)}\ [t_1, ..., t_n]$.

Suddetti costrutti verranno descritti formalmente nel *Capitolo 4 Algoritmo - AlphaLinearConversion* ma alcuni di questi verranno nominati nel *Capitolo 2 Regole di riduzione* per descrivere le regole di riduzione coinvolte nella loro valutazione.

Il linguaggio scelto per l'implementazione è stato C. Tale scelta è motivata dalle feature offerte dal linguaggio, ossia non aggiunge alcun overhead computazionale garantendo di preservare la complessità programmata e permette la gestione esplicita della memoria. Quest'ultima feature è stata utilizzata per identificare univocamente un nodo tramite il suo indirizzo di memoria evitando un overhead procedurale, come ad esempio il calcolo dell'hash del nodo, inoltre l'utilizzo dei puntatori facilita notevolmente lo sharing dei nodi.

# Capitolo 2

# Regole di riduzione

Il $\lambda$-calcolo è un pratico formalismo per la traduzione di un programma software in un'espressione. Ciò permette di applicare procedure deterministiche per la valutazione dei termini della $\lambda$-espressione. Tale formalismo consente inoltre di sostituire parti dell'espressione con forme ridotte della stessa. Ridurre un $\lambda$-termine non significa ridurre il numero di componenti ma trasformare tale termine in un nuovo termine per semplificare il calcolo. Le riduzioni che permettono tali sostituzioni sono state opportunamente formalizzate dal professore Sacerdoti Coen Claudio, relatore di questa tesi, e opportunamente codificate nell'algoritmo oggetto di codesta tesi. Di seguito verranno presentate tali riduzioni.

$\beta$-riduzione

La $\beta$-riduzione è applicabile quando si valuta un $\lambda$-termine applicazione che ha come $\lambda$-termine figlio sinistro una $\lambda$-astrazione. Dopo aver applicato la $\beta$ il nodo applicazione verrà sostituito da un nuovo nodo formato dal $\lambda$-termine figlio destro della $\lambda$-astrazione, il quale sarà collegato alla variabile legata all'astrazione (il binder). In seguito verrà settato come figlio del binder il nodo destro del nodo applicazione.
Formalmente: $(\lambda x : T.t)M \rightarrow_\beta t[M/x]$

Quando si valuta un $\lambda$-termine match è possibile applicare due tipi di riduzione a seconda del $\lambda$-termine contenuto nel body del match. Le riduzioni applicabili sono la J-riduzione o la J$\delta$c-riduzione.

J-riduzione

La J-riduzione è applicabile quando il body è un $\lambda$-termine costruttore j-esimo. In questo caso verrà sostituito l'intero nodo match con una serie di nodi applicazione costruiti in modo ricorsivo come segue. L'applicazione foglia Aj avrà come figlio destro il termine s1 del costruttore e come figlio sinistro il ramo tj del nodo match. L'applicazione al livello immediatamente dopo avrà come figlio sinistro l'applicazione Aj e come figlio destro il nodo s2 del costruttore. La costruzione continua in modo iterativo fino all'applicazione che avrà come figlio destro il nodo sk del costruttore. Quest'ultimo nodo applicazione sarà il nodo che sostituirà il nodo match.
Formalmente: $Match\, i\, t_0\, (k_j[s_1, ..., s_k])\, \vec{t_i} \rightarrow_J t_j\, s_1, ..., s_k$

J$\delta$c-riduzione

La J$\delta$c-riduzione è applicabile quando il body del nodo match è un $\lambda$-termine funzione co-ricorsiva. In questo caso il nodo match continuerà ad esistere ma verrà sostituito il suo nodo body da una serie di nodi applicazione costruiti come segue. L'applicazione foglia Ab avrà come figlio destro il termine b ossia il corpo della funzione e come figlio sinistro il nodo s1 primo termine in input alla funzione. L'applicazione al livello successivo avrà come figlio sinistro l'applicazione Ab e come figlio destro il nodo $s_2$, secondo termine in input alla funzione. La costruzione continua in modo iterativo fino all'applicazione che avrà come figlio destro il nodo sn, n-esimo termine in input alla funzione. Quest'ultimo nodo applicazione sarà il nodo che sostituirà il body del match.

Formalmente: $\texttt{Match}\, i\, t_0\, (g^{(n)}[s_1, ..., s_n])\, \vec{t_i} \rightarrow_{J\delta c} Match\, i\, t_0\, (b\, s_1, ..., s_n)\, \vec{t_i}$

J$\delta$i-riduzione

La J$\delta$i-riduzione è applicabile quando si valuta un $\lambda$-termine rappresentante una funzione ricorsiva che ha come n-esimo termine in input un costruttore j-esimo. Dopo aver applicato la J$\delta$i il nodo rappresentante la funzione verrà sostituito da una serie di applicazioni costruite come segue. L'applicazione foglia Ab avrà come figlio destro il termine b ossia il corpo della funzione e come figlio sinistro il nodo t1 primo termine in input alla funzione. L'applicazione al livello immediatamente dopo avrà come figlio sinistro l'applicazione Ab e come figlio destro il nodo t2 secondo termine in input alla funzione. La costruzione continua in modo iterativo fino all'applicazione che avrà come figlio destro il nodo tn n-esimo termine in input alla funzione ovvero il nodo costruttore j-esimo. Quest'ultimo nodo applicazione sarà il nodo che sostituirà il nodo funzione.
Formalmente: $f^{(n)}\ [t_1, ..., t_{n-1}, k_j[s_1, ..., s_m]]) \rightarrow_{J\delta i}\ b\, t_1, ..., t_{n-1}, k_j[s_1, ..., s_m]$

$\zeta$-riduzione

La $\zeta$-riduzione è applicabile quando si valuta un $\lambda$-termine let. Dopo aver applicato la $\zeta$ il nodo let verrà sostituito da un nuovo nodo formato dal $\lambda$-termine t3 ossia il corpo del nodo let, il quale sarà collegato alla variabile legata al let (il binder). In seguito verrà settato come figlio del binder il nodo t2 ovvero il valore della variabile dichiarata nel let.
Formalmente: $Let\ x : t_1 := t_2\ in\ t_3 \rightarrow_\zeta t_3\ [t_2/x]$

$\delta$-riduzione

La $\delta$-riduzione è applicabile quando si valuta un $\lambda$-termine rappresentante una costante definita in libreria. Dopo aver applicato la $\delta$ il nodo costante verrà sostituito da un nuovo nodo variabile valorizzato dal contenuto della costante.
Formalmente: $c \rightarrow_\delta\ b$

L'applicazione di una delle riduzioni appena presentate richiede ulteriori operazioni sulla struttura del grafo. Ovvero quando inseriamo un nuovo nodo, oltre a rifattorizzare il nodo in esame, dobbiamo aggiornare i padri dei nuovi nodi con quelli della precedente struttura e rivalutare gli stessi. La procedura di rivalutazione è necessaria poiché potrebbe accadere che l'applicazione di una riduzione crei l'ambiente adatto all'applicazione di un'altra. Ciò crea una forma di loop che viene risolta nel momento in cui una nuova valutazione non apporta modifiche alla struttura.

# Capitolo 3

# Grafo Aciclico Diretto

Un grafo aciclico diretto (Directed Acyclic Graph - DAG) è un grafo diretto o digrafo che non presenta cicli diretti, ovvero eseguendo una visita in profonditá nel grafo non riscontriamo mai archi all'indietro.

In questo lavoro si è deciso di utilizzare tale struttura per rappresntare le $\lambda$-espressioni, proprio come suggerito da Paterson e Wegman nell'algoritmo per l'unificazione del primo ordine [3]. Tale algoritmo non si poneva il problema dei cicli indotti dai binders legati ad un $\lambda$-termine perché non computa l'equivalenza-$\alpha$ sino alle variabili legate, mentre in codesta implementazione la valutazione dei $\lambda$-termini viene estesa anche alle variabili legate ossia i binders.
L'affermazione precedente induce la struttura ad abbandonare la forma di un DAG e ad avvicinarsi a quella di un automa a stati finiti deterministico, ma in realtá il ciclo indotto dai binders non è veramente tale perché il termine puntato dal binder non è un sottotermine ma semplicemente un ambiente a cui è appunto legato il binder. Tecnicamente tale arco è una rappresentazione grafica dell'ambiente e quindi puó essere visto come il dominio a cui appartiene il binder. Inoltre durante la visita profonda del DAG viene valutata prima la variabile legata e poi l'ambiente ovviando il problema della struttura ciclica.

Per rappresentare una $\lambda$-espressione sotto forma di DAG abbiamo bisogno di codificare sotto forma di nodi i vari costrutti presentati nel *Capitolo 1 Introduzione*.

Le variabili libere da contesto sono rappresentate con un singolo nodo variabile e non hanno figli. Tale rappresentazione vale anche per le sorte, i tipi induttivi, le costanti definite in libreria e le costanti non definite in libreria. Rappresentato in Figura 3.1.

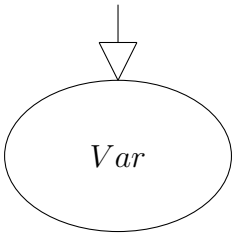


Figura 3.1: Esempio nodo variabile

Le variabili legate ad un ambiente sono rappresentate con un singolo nodo e hanno un arco all'indietro che punta all'ambiente a cui appartengono. Codesta implementazione vale anche per le piai. Rappresentato in Figura 3.2.

Le applicazioni rappresentano un'operazione binaria e sono rappresentate da un nodo applicazione contenente due figli uno destro e uno sinistro. I figli rappresentano gli operandi dell'operazione. Rappresentato in Figura 3.3.

Le $\lambda$-astrazioni sono rappresentate da un nodo $\lambda$ contenente due figli ossia il corpo della $\lambda$-astrazione e la variabile legata all'ambiente. Tale variabile è una variabile locale visibile solo nell'ambiente della $\lambda$-astrazione mentre il corpo dell'astrazione potrá utilizzare anche le variabili globali. Rappresenta-

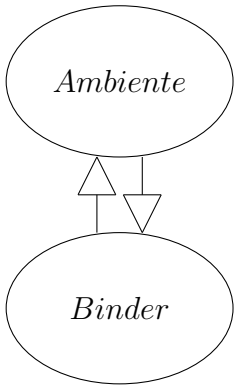


Figura 3.2: Esempio nodo Binder

to in Figura 3.4.

I costruttori j-esimo con n argomenti vengono rappresentati da un nodo costruttore con $j$+1 figli dove il primo è un intero $j$ che indica la lunghezza del costruttore e i restatnti figli sono i $j$ argomenti del costruttore. Rappresentato in Figura 3.5.

Il costrutto Match è rappresentato tramite un nodo match con $i$+1 figli. Il primo rappresenta il corpo su cui fare match mentre i restanti $i$ figli rappresentano i vari rami del match ovvero i nodi da eseguire dopo aver messo a confronto i vari casi. Rappresentato in Figura 3.6.

Il costrutto Let è utilizzato per dichiarare variabili locali o abbreviaviazioni ed è rappresentato tramite un nodo let con tre figli. Il primo è una variabile $t$ legata all'ambiente, il secondo è il corpo del let e il terzo è il valore della variabile dichiarata. Rappresentato in Figura 3.7.

Le funzioni ricorsive sono rappresntate da un nodo funzione con $i$+1 figli. Il primo è un intero pari ad $i$ che indica il numero di argomenti in input alla

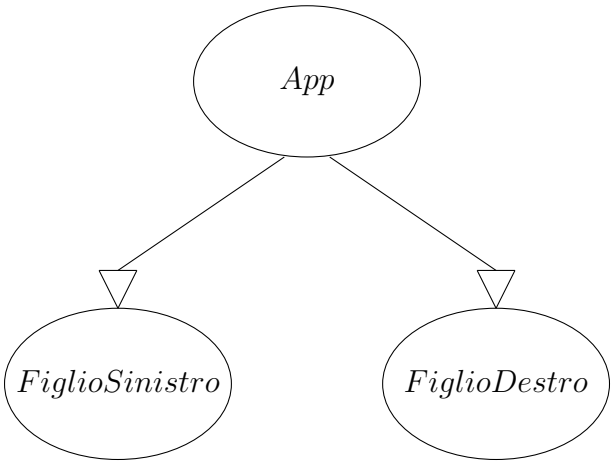


Figura 3.3: Esempio nodo applicazione

funzione, mentre i restanti $i$ figli sono esattamente gli $i$ input. La medesima rappresentazione la ritroviamo anche nelle funzioni co-ricorsive. Rappresentato in Figura 3.8.

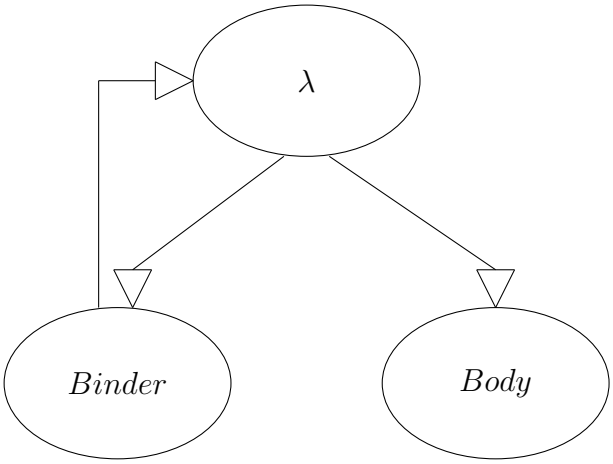


Figura 3.4: Esempio nodo $\lambda$

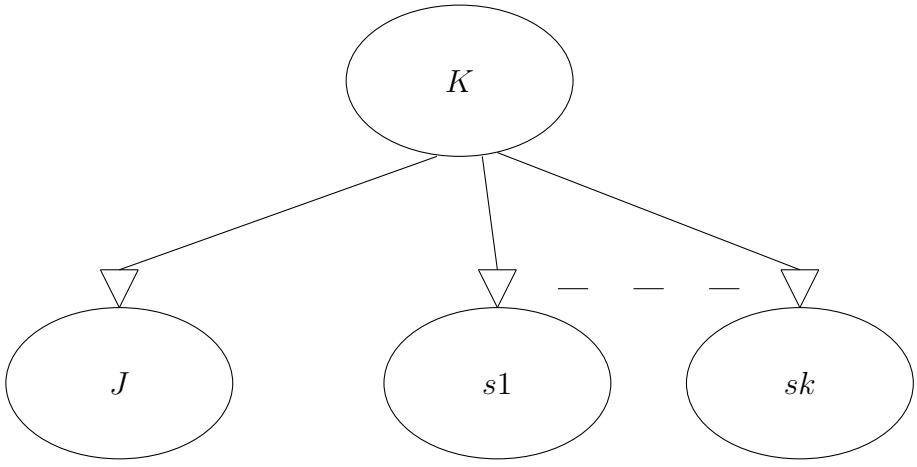


Figura 3.5: Esempio nodo costruttore j-esimo

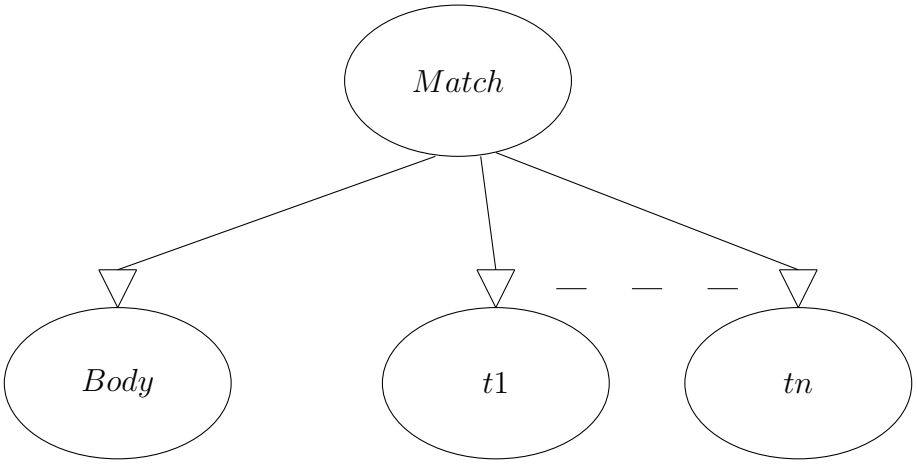


Figura 3.6: Esempio nodo match

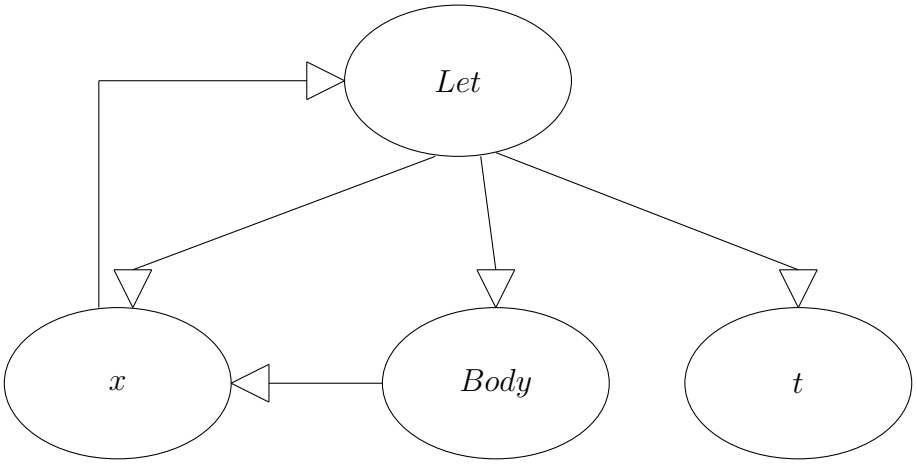


Figura 3.7: Esempio nodo let

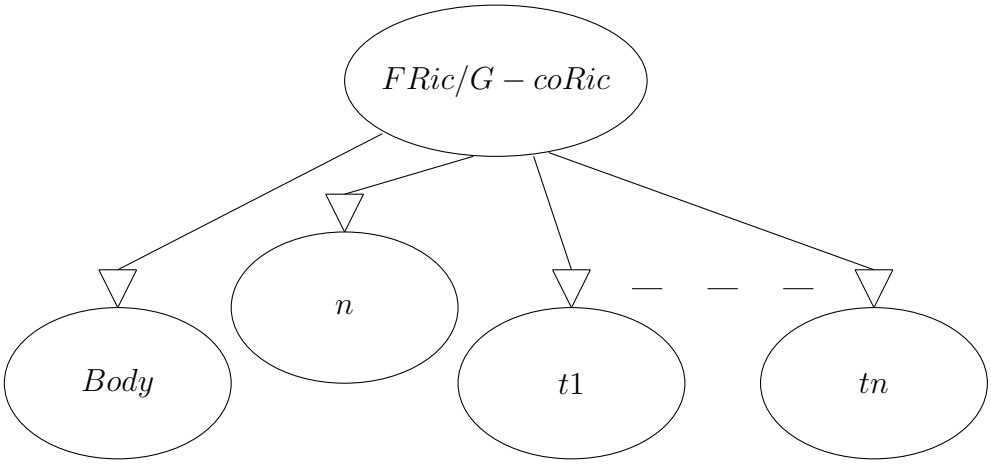


Figura 3.8: Esempio nodo funzione ricorsiva/co-ricorsiva

# Capitolo 4

# Algoritmo - AlphaLinearConversion - SISTEMARE

L'algoritmo implementato consiste in una serie di procedure che consento la visita esaustiva di un grafo aciclico diretto permettendo la valutazione e la comparazione dei termini presenti nei nodi del grafo.
L'input dell'algoritmo è un array contenente tutti i nodi del grafo. Questi possono anche essere ordinati in modo casuale poiché sarà l'algoritmo stesso a risalire dai nodi figli verso i padri e viceversa durante la valutazione. L'output dell'algoritmo è banale. Se il programma termina vuol dire che le 2 $\lambda$-espressioni sono bi-simili ossia hanno strutture diverse ma possono essere collassate nello stesso $\lambda$-termine. Mentre se avviene un uscita anomala corrisponde a dimostrare che le 2 $\lambda$-espressioni rappresentano due espressioni realmente diverse e quindi non sono bi-simili.

## 4.1 Strutture Dati

L'algoritmo lavora su nodi appartenenti ad un grafo ma, non essendo necessaria, non esiste una vera e propria struttura dati che rappresenti il grafo. Per strutturare le $\lambda$-espressioni sotto forma di DAG vengono creati una serie di nodi collegati fra loro, generando dietro le quinte la struttura del grafo.

I nodi vengono rappresentati tramite un ADT in C ovvero tramite una variabile union, che contiene tutte le forme possibili del nodo, legata ad una label che distingue le varie tipologie di nodo possibili. La struct che definisce il nodo viene arricchita da ulteriori informazioni necessarie alla valutazione del nodo stesso e alla visita del grafo. Di seguito è riportata la struct in questione.

```
struct Node {
    enum TypeNode label;
    union {
        struct NodeFVar fvar;
        struct NodeBVar bvar;
        struct NodePiai piai;
        struct NodeShared shared;
        struct NodeApp app;
        struct NodeLam lam;
        struct NodeMatch match;
        struct NodeFRic fRic;
        struct NodeGCoRic gCoRic;
        struct NodeLet let;
        struct NodeJthConstr jCostr;
        struct NodeConst constant;
    } content;
    Node *canonic;
    Node *copy;
```

```
    Bool building;
    List *parentNodes;
    List *neighbour;
    List *queue;
    Bool root;
    Bool reachable;
    Bool visited;
    int rc;
};
enum TypeNode {
    FVar, BVar, Piai, Shared, App, Lam, Match,
    Let, FRic, GCoRic, Constructor, Constant
};
```

Il content del Nodo è rappresentato da un costrutto chiamato union che permette di raggruppare in un unico tipo un insieme di tipi. La rappresentazione in memoria di un tipo union sarà sempre la stessa in tutti i casi ossia occuperà l'area di memoria massima fra tutti i tipi che rappresenta. Una volta popolato il content non sarà più possibile risalire al tipo del dato contenuto, se non tramite una label opportunamente valorizzata, proprio perché non possono essere confrontati i puntatori essendo uniformi.

Come possiamo notare la label che differenzia il content del Nodo è un tipo enumerato. Ciò permette di essere più efficienti durante i vari switch...case utilizzati nelle procedure perché la label viene vista come un intero e la comparazione fra due interi è più veloce della comparazione fra aree di memoria o fra stringhe. Inoltre utilizzando i tipi enumerati anziché dei semplici interi preserviamo la legibilità del codice facilitando future operazioni di manutenzione o espanzione del progetto.

La struct ListHT *parentNodes è una puntatore alla testa della lista dei genitori di un nodo e serve appunto per creare la stuttura del DAG e per risalire al nodo root del grafo durante la valutazione. Il nodo root avrà tale lista vuota mentre i restanti nodi dovranno avere almeno un genitore

prima che inizi la valutazione. Il contenuto della lista in questione potrà solo crescere e in alcuni casi subire refactoring durante le procedure di riduzione.

La struct ListHT *neighbour è anch'essa un puntatore alla testa di una lista. Tale lista è utilizzata per creare i collegamenti fra i nodi dei due DAG e tali relazioni vengono definite ∼neighbour. Nel momento in cui viene scandita la lista si avviano procedure di comparazione che determinano se due nodi sono bi-simili o meno.
Inizialmente solo il nodo root di ogni DAG sarà collegato da una relazione di ∼neighbour, mentre i restanti nodi popoleranno tale lista durante la valutazione.

I rimanenti campi della struct Nodo sono utilizzati esclusivamente per valutare e confrontare i nodi nell'algoritmo e possono essere considerate delle variabili di appoggio per semplificare il codice. Inialmente sono settate su un valore di default come NULL o False.

La struct Node *canonic serve per settare il canonico di un nodo durante la valutazione per poi confrontare il canonico di due nodi in alcune casistiche della fase di comparazione.

La enum Bool building è un tipo enumerato che può assumere il valore True o False. È sempre settata su False tranne quando il nodo è in fase di analisi. La variabile in questione è utilizzata come un assert poiché se stiamo analizzando i parent di un nodo che hanno giá il nodo canonico definito se tale nodo è in fase di analisi, ovvero ha building settato a true, l'algoritmo effettua un exit perché vuol dire che abbiamo delle forme cicliche nel grafo, cosa che non dovrebbe accadere.

La struct Node *copy è utilizzata per salvare una copia del nodo prima di richiamare delle procedure di riduzione.

La struct ListHT *queue è una lista dove vengono salvati i nodi da comparare con il nodo attualmente in analisi.

Le forme che può assumere il content di un Nodo sono molteplici e corrispondono ai costrutti elencati nel *Capitolo 1 Introduzione*. Alcuni costrutti

vengono collassati nella stessa struttura poiché seguono le medesime regole e quindi sarebbe stato inutile differenziare le casistiche.

La struct NodeFVar contiene un singolo campo e rappresenta le variabili libere da contesto, le sorte, i tipi induttivi e le costanti non definite in libreria. Di seguito è riportata la relativa struttura.

```
struct NodeFVar {
    int var;//dummy
};
```

La struct NodeBVar contiene il campo binder e rappresenta le variabili legate e le piai (tipo delle funzioni). Di seguito è riportata la relativa struttura.

```
struct NodeBVar {
    Node *binder;
};
```

La struct NodePiai CONTINUARE.....

```
struct NodePiai {
    Node *var;
    Node *body;
};
```

La struct NodeApp contiene i due figli di un'applicazione e appunto è utilizzata per rappresentare le operazioni binarie. Di seguito è riportata la relativa struttura.

```
struct NodeApp {
    Node *left;
    Node *right;
};
```

La struct NodeLam è utilizzata per rappresentare le $\lambda$-astrazioni e contiene due variabili, ovvero il corpo della $\lambda$-astrazione e la variabile legata all'ambiente la quale dovra essere necessariamente di tipo NodeBVar. Di seguito è riportata la relativa struttura.

```
struct NodeLam {
    Node *var;//NodeBVar
    Node *body;
};
```

La struct NodeJthConstr rappresenta i costruttori j-esimo con n argomenti e contiene due campi un intero j e una lista di argomenti. Di seguito è riportata la relativa struttura.

```
struct NodeJthConstr {
    int j;
    List *arg;
    int n;
};
```

La struct NodeConst rappresenta le costanti definite in libreria e contiene un singolo campo rappresentante il valore della costante. Di seguito è riportata la relativa struttura.

```
struct NodeConst {
    Node *var;
};
```

La struct NodeMatch rappresenta il costrutto match e contiene un body e la lista dei vari casi del match. Di seguito è riportata la relativa struttura.

```
struct NodeMatch {
    Node *body;
    List *branches;
    int n;
};
```

La struct NodeLet rappresenta il costrutto let e contiene tre campi ossia il binder legato all'ambiente che sarà di tipo NodeBVar, il valore t2 della variabile dichiarata e il corpo t3 dove utilizzo la variabile dichiarata. Di seguito è riportata la relativa struttura.

```
struct NodeLet {
    Node *var;//NodeBVar
    Node *t2;
    Node *t3;
};
```

La struct NodeFRic rappresenta le funzioni ricorsive mentre la struct NodeGCoRic rappresenta le funzioni co-ricorsive. Entrambe contengono un binder legato all'ambiente di tipo NodeBVar, il corpo della funzione, un intero che indica il numero di argomenti in input e la lista di tali argomenti. Di seguito sono riportate le relative strutture.

```
struct NodeFRic {
    Node *var;//NodeBVar
    Node *t;
    int n;
    List *arg;
};
struct NodeGCoRic {
    Node *var;//NodeBVar
    Node *t;
    int n;
    List *arg;
};
```

La struct ListHT presente nelle strutture sovracitate rappresenta una lista di elementi di tipo ListElement. Al suo interno contiene un puntatore alla testa della lista e un contatore utili per iterarare sulla lista e un puntatore

alla coda necessario per appendere nuovi elementi. Ogni elemento conterrà un puntatore ad un Nodo e un puntatore al successivo elemento della lista. Si è deciso di implemetare tale struttura, anziché utilizzare strutture di libreria, per preservare la complessità desiderata e quindi essere più efficienti nelle operazioni in cui è coinvoilta. Di seguito sono riportate le relative strutture.

```
struct ListItem {
    Node *node;
    ListItem *next;
};
struct List {
    ListItem *head;
    ListItem *tail;
};
```

## 4.2 Propagazione ∼neighbour

La diffusione dei ∼neighbour avviene tramite la funzione Propagate che prende in input due nodi m e c e testa se questi rispettano gli invarianti di propagazione. Se ciò avviene l'algoritmo continua a girare e in alcuni casi aggiunge sulla lista dei neighbour dei figli di m i corrispettivi figli del nodo c e viceversa. Mentre se non vengono rispettati gli invarianti avviene un exit e l'algoritmo termina segnalando che le espressioni non sono bi-simili.

Gli invarianti di propagazione variano a seconda del tipo di nodo analizzato. Esistono anche tipologie di nodi che non si propagano mai ovvero quelli di tipo Shared, Let o Constant. Le restanti tipologie di nodo propagano e/o testano la bisimilarità, esse hanno un'invariante comune soddisfatto se m e c sono dello stesso tipo.
Un nodo di tipo FVar o BVar non propaga mai i suoi figli ma verifica un'ulteriore invariante ovvero che il canonico della variabile contenuta in m deve essere uguale al canonico della variabile contenuta in c.

Un nodo di tipo App propaga i rispettivi figli destro e sinistro di m con quelli di c.
Un nodo di tipo Lam o Match propaga unicamente i rispettivi body di m e c.
Un nodo di tipo FRic o GCoRic verifica un invariante aggiuntivo ossia che il numero di argomenti in input deve essere uguale. Se ciò avviene propaga il corpo della funzione e l'intera lista di argomenti.
Un nodo di tipo Constructor verifica che i due nodi abbiano lo stesso numero di argomenti e se ciò avviene propaga l'intera lista di argomenti fra i due nodi.

Le procedure di propagazione descritte vengono utilizzate per popolare la lista dei ∼neighbour dei nodi analizzati, la quale verrà visitata nella procedura BuildClass descritta nel paragrafo successivo 4.3 Valutazione e riduzione.

## 4.3 Valutazione e Riduzione

Il processo di valutazione e riduzione avviene in 2 fasi. Inanzitutto viene invocata la funzione WeakCbVEval sul nodo root di entrambi i DAG e successivamente viene invocata la funzione DAGCheckAndEval fornendo come input l'array contenente tutti i nodi dei due DAG.
La procedura DAGCheckAndEval visita l'array in input e solo se un nodo non è stato ancora valutato, ossia se non ha il nodo canonico definito, richiama la procedura BuildClass che si occupa della sua valutazione.

### 4.3.1 BuildClass

La funzione BuildClass si occupa della valutazione di un nodo c ricevuto in input. Inizialmente setta lo stato di alcune variabili di processo del nodo c e successivamente avvia un ciclo while nel quale scorre la queue di tale nodo. La queue inizialmente conterrà unicamente un puntatore al nodo in analisi ma durante tale ciclo verrà popolata da ulteriori nodi. La funzione terminerà

quando la queue sarà esaurita eseguendo un'operazione di coda che setta lo stato building di c su False.

Durante ogni passo del ciclo while viene estratto un nodo n dalla queue e vengono eseguiti ulteriori cicli sui genitori e sui ∼neighbour di n.
Durante la visita dei genitori, salvati nella lista parentNodes, se il genitore analizzato non è stato ancora valutato viene richiamata ricorsivamente la funzione BuildClass sul genitore, mentre se è già stato valutato viene testato un'invariante che potrebbe provocare un exit dell'algoritmo. Tale invariante verifica che la variabile building del canonico del genitore sia settata su False, se ciò non si verifica l'algoritmo termina.
Terminata la visita dei genitori se n non ha ∼neighbour la funzione termina normalmente settando lo stato building su False ed effettuando un return al chiamante. Prima di terminare vengono testati due invarianti ossia che la queue sia stata esaurita e che l'ultimo nodo estratto dalla queue sia proprio c. Quindi se uno dei due invarianti non viene rispettato, quando n non ha ∼neighbour, avviene un exit segnalando la non bi-similarità.
Se n ha ∼neighbour avviene la visita della relativa lista. Durante tale ciclo si estrae un ∼neighbour dalla lista e si verifica se il suo canonico è definito. Se non è definito si aggiunge il ∼neighbour analizzato nella queue di c e si setta il suo canonico su c, mentre se è definito ed è diverso da c avviene un exit segnalando la non bi-similarità dei due DAG.
Di seguito se n rappresenta un nodo $\lambda$-astrazione viene effettuato un refactoring del suo body con l'output della funzione WeakCbVEval richiamata sul body di n.
Infine, prima di passare all'iterata successiva, viene invocata la funzione Propagate con input n e c.

La funzione BuildClass non produce alcun output ma, come descritto precedentemente, può provocare la terminazione dell'algoritmo e può modificare direttamente i figli dei nodi che analizza.

### 4.3.2 WeakCbVEval

La funzione WeakCbVEval si occupa della riduzione dei $\lambda$-termini rappresentati nei nodi dei DAG. Essa riceve in input un nodo n e ritorna al chiamante o n stesso o un nodo ridotto, nel caso in cui è possibile applicare delle regole di riduzione sull'input.
I casi in cui ritorna n stesso sono quelli in cui il tipo del nodo in input è FVar, BVar, Lam, GCoRic o Constructor, ma anche in alcune casistiche per nodi di tipo Match e FRic che vedremo più avanti in codesta sezione.

Se n è di tipo Shared o Constant semplicemente viene eliminato il doppio puntatore tornando il corpo di n, nel primo caso il nodo sherato mentre nel secondo la variabile rappresentante il valore della costante.

Se n rappresenta un nodo applicazione prima richiamo ricorsivamente la funzione WeakCbVEval sul figlio destro e sinistro di n calcolando due nuovi nodi n2 e n1.
Di seguito se n1 è un nodo $\lambda$-astrazione posso applicare la $\beta$-riduzione e quindi chiamare la funzione Inst fornendo come input il body di n1, il nodo n1 e un nuovo nodo Shared con body n2. In questo caso l'output della funzione al top level sarà l'output della chiamata ricorsiva di WeakCbVEval con input l'output della chiamata alla funzione Inst.
Mentre se n1 non è un nodo $\lambda$-astrazione l'output sarà un nuovo nodo applicazione con figlio sinistro n1 e destro n2.

Quando n rappresenta un nodo Match si presentano tre casistiche differenti condizionate dal tipo del body del match.
Se il body è un costruttore j-esimo prima si prova a ridurre il ramo j-esimo effettuando una chiamata ricorsiva a WeakCbVEval e salvando l'output in n1. Di seguito si calcola la sequenza di applicazioni, come descritto nel *Capitolo 2 Regole di riduzione*, per applicare la J-riduzione salvando il nodo root della sequenza nel nodo n2. Infine l'output sarà il nodo restituito dalla

chiamata ricorsiva di WeakCbVEval sul nodo n2.
Mentre se il body rappresenta una funzione co-ricorsiva verrà applicata la J$\delta$c-riduzione mantenendo il nodo match ma creando la sequenza di applicazioni sul body del match. L'output sarà la chiamata ricorsiva di WeakCbVEval su n stesso ma con body modificato.
Quando il body del match non ricade in uno dei due casi precedenti l'output sarà il nodo n fornito in input.

Se n è di tipo Let verrà applicata la $\zeta$-riduzione ritornando come output la chiamata ricorsiva della funzione WeakCbVEval con input t3. Prima di effettuare tale chiamata modificheremo il binder puntato dalla bvar del nodo Let con il figlio t2 dello stesso.

Se n rappresenta una funzione ricorsiva per prima cosa si proverà a ridurre l'argomento j-esimo tramite WeakCbVEval. Di seguito se tale argomento è un costruttore j-esimo potrà essere applicata la J$\delta$i-riduzione calcolando la sequenza di applicazioni, come descritto nel *Capitolo 2 Regole di riduzione*. In questo caso l'output sarà la chiamata ricorsiva di WeakCbVEval sul nodo root della sequenza di applicazioni, altrimenti l'output sarà n stesso dopo aver richiamato sul body t di n la funzione WeakCbVEval.

.

.
Trovare conclusione capitolo
.
.

# Capitolo 5

# Test e Risultati

Esporre fase di debug con il tool graphviz???

Test e Risultati TESTO

Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO

Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO
Test e Risultati TESTO

# Capitolo 6

# Conclusioni

Conclusioni TESTO

# Appendice A

# Listato dei programmi

## A.1 DAGCheckAndEval

```
typedef struct Node Node;
typedef struct List List;
typedef struct ListItem ListItem;
typedef enum Bool Bool;
List *nodesHT;//Node List

void DAGCheckAndEval(List *nodesHT, Node *root1, Node *root2) {
    root1->root = True;
    root2->root = True;
    PushNeighbour(root1, root2);
    ListItem *nodes = nodesHT->head;
    while (nodes->node != NULL) {
        if (nodes->node->canonic == NULL)
            BuildClass(nodes->node);
        nodes = nodes->next;
    }
}
```

## A.2 BuildClass

```
void BuildClass(Node *c) {
    c->building = True;
    c->queue = InitListHT();
    ListItem *iterQueue = c->queue->head;
    Enqueue(c, c);
    while (iterQueue->node != NULL) {
        Node *n = iterQueue->node;
        SearchIfReachable(n);
        if (n->reachable == False) {
            assert(n->neighbour->head->node == NULL);
            c->building = False;
            return;
        }
        // visit parents
        assert(n->parentNodes != NULL);
        ListItem *iterParents = n->parentNodes->head;
        while (iterParents->node != NULL) {
            if (iterParents->node->canonic == NULL)
                BuildClass(iterParents->node);
            else if (iterParents->node->canonic->building == True)
                PrintExit(2);
            iterParents = iterParents->next;
        }
        // visit neighbours
        assert(n->neighbour != NULL);
        ListItem *iterNeighbour = n->neighbour->head;
        while (iterNeighbour->node != NULL) {
            Enqueue(iterNeighbour->node, c);
            iterNeighbour = iterNeighbour->next;
        }
```

```
        if (n != c)
            Propagate(n, c);
        iterQueue = iterQueue->next;
    }
    c->building = False;
}

void Enqueue(Node *n, Node *c) {
    if (n->canonic == NULL) {
        PushToListHT(c->queue, n);
        n->canonic = c;
    } else if (n->canonic != c)
        PrintExit(2);
}

void SearchIfReachable(Node *n) {
    n->reachable = n->root;
    // visit parents
    assert(n->parentNodes != NULL);
    ListItem *m = n->parentNodes->head;
    while (m->node != NULL && n->reachable != True) {
        if (m->node->visited == False)
            SearchIfReachable(m->node);
        n->reachable = m->node->reachable;
        m = m->next;
    }
    n->visited = True;
}
```

## A.3 Propagate

```
void Propagate(Node *m, Node *c) {
    switch (m->label) {
        case FVar:
            if (m != c)
                PrintExit(3);
            break;
        case BVar:
            if (c->label == BVar) {
                assert(m->content.bvar.binder != NULL);
                assert(c->content.bvar.binder != NULL);
                if (m->content.bvar.binder->canonic != c->content.bvar.binder->can
                    PrintExit(3);
            } else
                PrintExit(3);
            break;
        case Piai:
            if (c->label == Piai)
                PushNeighbour(m->content.piai.body, c->content.piai.body);
            else
                PrintExit(3);
            break;
        case Shared:
            assert(0);
        case App:
            if (c->label == App) {
                PushNeighbour(m->content.app.left, c->content.app.left);
                PushNeighbour(m->content.app.right, c->content.app.right);
            } else
                PrintExit(3);
            break;
```

```
        case Lam:
            if (c->label == Lam)
                PushNeighbour(m->content.lam.body, c->content.lam.body);
            else
                PrintExit(3);
            break;
        case Match:
            if (c->label == Match && m->content.match.n == c->content.match.n) {
                PushNeighbour(m->content.match.body, c->content.match.body);
                ListItem *iter = m->content.match.branches->head;
                ListItem *iter2 = c->content.match.branches->head;
                for (int i = 0; i < m->content.match.n; ++i) {
                    PushNeighbour(iter->node, iter2->node);
                    iter = iter->next;
                    iter2 = iter2->next;
                }
            } else
                PrintExit(3);
            break;
        case Let:
            assert(0);
        case FRic:
            if (c->label == FRic && m->content.fRic.n == c->content.fRic.n) {
                PushNeighbour(m->content.fRic.t, c->content.fRic.t);
                ListItem *iter = m->content.fRic.arg->head;
                ListItem *iter2 = c->content.fRic.arg->head;
                for (int i = 0; i < m->content.fRic.n; ++i) {
                    PushNeighbour(iter->node, iter2->node);
                    iter = iter->next;
                    iter2 = iter2->next;
                }
```

```
        } else
            PrintExit(3);
        break;
    case GCoRic:
        if (c->label == GCoRic && m->content.gCoRic.n == c->content.gCoRic.n)
            PushNeighbour(m->content.gCoRic.t, c->content.gCoRic.t);
            ListItem *iter = m->content.gCoRic.arg->head;
            ListItem *iter2 = c->content.gCoRic.arg->head;
            for (int i = 0; i < m->content.gCoRic.n; ++i) {
                PushNeighbour(iter->node, iter2->node);
                iter = iter->next;
                iter2 = iter2->next;
            }
        } else
            PrintExit(3);
        break;
    case Constructor:
        if (c->label == Constructor && m->content.jCostr.j == c->content.jCost
            m->content.jCostr.n == c->content.jCostr.n) {
            ListItem *iter = m->content.jCostr.arg->head;
            ListItem *iter2 = c->content.jCostr.arg->head;
            for (int i = 0; i < m->content.jCostr.n; ++i) {
                PushNeighbour(iter->node, iter2->node);
                iter = iter->next;
                iter2 = iter2->next;
            }
        } else
            PrintExit(3);
        break;
    case Constant:
        assert(0);
```

```
    }
}

void PushNeighbour(Node *m, Node *c) {
    Node *m1 = WeakCbVEval(m);
    Node *c1 = WeakCbVEval(c);

    PushToListHT(m1->neighbour, c1);
    PushToListHT(c1->neighbour, m1);
}
```

## A.4 WeakCbVEval

```
Node *WeakCbVEval(Node *n) {
    ListItem *listElement;
    Node *n1;
    Node *n2;
    Node *nodeReturn;
    switch (n->label) {
        case FVar:
            return n;
        case BVar:
            return n;
        case Piai:
            return n;
        case Shared:
            nodeReturn = n->content.shared.body;
            RefactoringNode(n, nodeReturn);
```

```
    return nodeReturn;
case App:
    n2 = WeakCbVEval(n->content.app.right);
    n1 = WeakCbVEval(n->content.app.left);
    if (n1->label == Lam) {
        nodeReturn = WeakCbVEval(Inst(n1->content.lam.body, n1, n2));
        RefactoringNode(n, nodeReturn);
        return nodeReturn;
    } else
        return n;
case Lam:
    return n;
case Match:
    switch (n->content.match.body->label) {
        case Constructor://j
            listElement = n->content.match.branches->head;
            for (int i = 0; i < n->content.match.body->content.jCostr.n -
                listElement = listElement->next;
            }
            n2 = AppReplacement(listElement->node, n->content.match.body->
            nodeReturn = WeakCbVEval(n2);
            RefactoringNode(n, nodeReturn);
            return nodeReturn;
        case GCoRic://jDelta-c
            n2 = InitGCoRic(n->content.match.body->content.gCoRic.var, n->
                            0, InitListHT());
            n1 = Inst(n->content.match.body->content.gCoRic.t, n->content.
            if (!(n1->label == Shared && n1->content.shared.body == n2))
                FreeRC(n2);
            n2 = AppReplacement(n1, n->content.match.body->content.gCoRic.
            RefactoringNode(n->content.match.body, n2);
```

```
                n->content.match.body = n2;
                nodeReturn = WeakCbVEval(n);
                return nodeReturn;
            default:
                return n;
        }

    case Let:
        n->content.let.var->content.bvar.binder = WeakCbVEval(
                n->content.let.t2);
        n->content.let.var->content.bvar.binder->rc++;
        nodeReturn = WeakCbVEval(n->content.let.t3);
        RefactoringNode(n, nodeReturn);
        return nodeReturn;
    case FRic:
        listElement = n->content.fRic.arg->head;
        for (int i = 0; i < n->content.fRic.n - 1; ++i) {
            listElement = listElement->next;
        }
        listElement->node = WeakCbVEval(listElement->node);
        if (listElement->node->label == Constructor) {
            n1 = Inst(n->content.fRic.t, n->content.fRic.var, n);//b
            nodeReturn = WeakCbVEval(AppReplacement(n1, n->content.fRic.arg)
            RefactoringNode(n, nodeReturn);
            return nodeReturn;
        } else {
            n->content.fRic.t = WeakCbVEval(n->content.fRic.t);
            return n;
        }
    case GCoRic:
        return n;
```

```
        case Constructor:
            return n;
        case Constant:
            nodeReturn = n->content.constant.var;
            RefactoringNode(n, nodeReturn);
            return nodeReturn;
    }
}
```

## A.5 RefactoringNode

```
void RefactoringNode(Node *oldNode, Node *newNode) {
    newNode->root = oldNode->root;
    newNode->reachable = oldNode->reachable;

    ListItem *parent = oldNode->parentNodes->head;
    while (parent->node != NULL) {
        UpdateSon(oldNode, newNode, parent->node);
        newNode->rc++;
        PushToListHT(newNode->parentNodes, parent->node);
        parent = parent->next;
    }
    FreeRC(oldNode);
}

void UpdateSon(Node *oldSon, Node *newSon, Node *parent) {
    int find = 0;
    switch (parent->label) {
        case FVar:
```

```
        assert(0);
    case BVar:
        if (parent->content.bvar.binder == oldSon)
            parent->content.bvar.binder = newSon;
        else
            assert(0);
        break;
    case Piai:
        if (parent->content.piai.body == oldSon)
            parent->content.piai.body = newSon;
        else if (parent->content.piai.var == oldSon)
            parent->content.piai.var = newSon;
        else
            assert(0);
        break;
    case Shared:
        if (parent->content.shared.body == oldSon)
            parent->content.shared.body = newSon;
        else
            assert(0);
        break;
    case App:
        if (parent->content.app.left == oldSon)
            parent->content.app.left = newSon;
        else if (parent->content.app.right == oldSon)
            parent->content.app.right = newSon;
        else
            assert(0);
        break;
    case Lam:
        if (parent->content.lam.body == oldSon)
```

```
            parent->content.lam.body = newSon;
        else if (parent->content.lam.var == oldSon)
            parent->content.lam.var = newSon;
        else
            assert(0);
        break;
    case Match:
        if (parent->content.match.body == oldSon)
            parent->content.match.body = newSon;
        else {
            assert(parent->content.match.branches != NULL);
            ListItem *branches = parent->content.match.branches->head;
            while (find == 0 && branches->node != NULL) {
                if (branches->node == oldSon) {
                    branches->node = newSon;
                    find = 1;
                }
                branches = branches->next;
            }
            if (find == 0)
                assert(0);
        }
        break;
    case Let:
        if (parent->content.let.t3 == oldSon)
            parent->content.let.t3 = newSon;
        else if (parent->content.let.t2 == oldSon)
            parent->content.let.t2 = newSon;
        else if (parent->content.let.var == oldSon)
            parent->content.let.var = newSon;
        else
```

```
            assert(0);
        break;
    case FRic:
        if (parent->content.fRic.t == oldSon)
            parent->content.fRic.t = newSon;
        else if (parent->content.fRic.var == oldSon)
            parent->content.fRic.var = newSon;
        else {
            assert(parent->content.fRic.arg != NULL);
            ListItem *arg = parent->content.fRic.arg->head;
            int i = 0;
            while (find == 0 && arg->node != NULL) {
                if (arg->node == oldSon) {
                    arg->node = newSon;
                    find = 1;
                }
                ++i;
                arg = arg->next;
            }
            if (find == 0)
                assert(0);
        }
        break;
    case GCoRic:
        if (parent->content.gCoRic.t == oldSon)
            parent->content.gCoRic.t = newSon;
        else if (parent->content.gCoRic.var == oldSon)
            parent->content.gCoRic.var = newSon;
        else {
            assert(parent->content.gCoRic.arg != NULL);
            ListItem *arg = parent->content.gCoRic.arg->head;
```

```
            while (arg->node != NULL) {
                if (arg->node == oldSon) {
                    arg->node = newSon;
                    find = 1;
                }
                arg = arg->next;
            }
            if (find == 0)
                assert(0);
        }
        break;
    case Constructor:
        assert(parent->content.jCostr.arg != NULL);
        ListItem *arg = parent->content.jCostr.arg->head;
        while (find == 0 && arg->node != NULL) {
            if (arg->node == oldSon) {
                arg->node = newSon;
                find = 1;
            }
            arg = arg->next;
        }
        if (find == 0)
            assert(0);
        break;
    case Constant:
        if (parent->content.constant.var == oldSon)
            parent->content.constant.var = newSon;
        else
            assert(0);
        break;
    default:
```

```
            assert(0);
    }
}

void FreeRC(Node *node) {
    RemoveHT(nodesHT, node);
    ListItem *listElement;
    switch (node->label) {
        case FVar:
            break;
        case BVar:
            break;
        case Piai:
            RefactoringSon(node->content.piai.var, node);
            RefactoringSon(node->content.piai.body, node);
            break;
        case Shared:
            RefactoringSon(node->content.shared.body, node);
            break;
        case App:
            RefactoringSon(node->content.app.left, node);
            RefactoringSon(node->content.app.right, node);
            break;
        case Lam:
            RefactoringSon(node->content.lam.body, node);
            RefactoringSon(node->content.lam.var, node);
            break;
        case Match:
            RefactoringSon(node->content.match.body, node);
            listElement = node->content.match.branches->head;
            while (listElement->node != NULL) {
```

```
            RefactoringSon(listElement->node, node);
            listElement = listElement->next;
        }
        break;
    case Let:
        RefactoringSon(node->content.let.t3, node);
        RefactoringSon(node->content.let.t2, node);
        RefactoringSon(node->content.let.var, node);
        break;
    case FRic:
        RefactoringSon(node->content.fRic.t, node);
        RefactoringSon(node->content.fRic.var, node);
        listElement = node->content.fRic.arg->head;
        while (listElement->node != NULL) {
            RefactoringSon(listElement->node, node);
            listElement = listElement->next;
        }
        break;
    case GCoRic:
        RefactoringSon(node->content.gCoRic.t, node);
        RefactoringSon(node->content.gCoRic.var, node);
        listElement = node->content.gCoRic.arg->head;
        while (listElement->node != NULL) {
            RefactoringSon(listElement->node, node);
            listElement = listElement->next;
        }
        break;
    case Constructor:
        listElement = node->content.jCostr.arg->head;
        while (listElement->node != NULL) {
            RefactoringSon(listElement->node, node);
```

```
                listElement = listElement->next;
            }
            break;
        case Constant:
            RefactoringSon(node->content.constant.var, node);
            break;
    }

    free(node);
}

void RefactoringSon(Node *node, Node *parent) {
    node->rc--;
    RemoveHT(node->parentNodes, parent);
    if (node->root == False && node->rc < 1)
        FreeRC(node);
}
```

## A.6 Inst

```
Node *Inst(Node *n, Node *l, Node *sub) {
    Node *n1;
    Node *temp;
    ListItem *iterList;
    struct List *arg = InitListHT();
    switch (n->label) {
        case FVar:
            return n;
        case BVar:
            if (n->content.bvar.binder == l)
                return InitShared(sub);
```

```
        else {
            if (n->content.bvar.binder->copy == NULL)
                return n;
            else
                return InitBVar(n->content.bvar.binder->copy);
        }
    case Piai:
        n1 = InitPiai(InitBVar(NULL), n->content.piai.body);
        n->copy = n1;
        temp = n1->content.piai.body;
        n1->content.piai.body = Inst(temp, l, sub);
        if (temp != n1->content.piai.body) {
            PushToListHT(n1->content.piai.body->parentNodes, n1);
            n1->content.piai.body->rc++;
            RemoveHT(temp->parentNodes, n1);
            temp->rc--;
        }
        n->copy = NULL;
        return n1;
    case Shared:
        return n;
    case App:
        return InitApp(Inst(n->content.app.left, l, sub), Inst(n->content.app.
    case Lam:
        n1 = InitLam(InitBVar(NULL), n->content.lam.body);
        n->copy = n1;

        temp = n1->content.lam.body;
        n1->content.lam.body = Inst(temp, l, sub);
        if (temp != n1->content.lam.body) {
            PushToListHT(n1->content.lam.body->parentNodes, n1);
```

```
            n1->content.lam.body->rc++;
            RemoveHT(temp->parentNodes, n1);
            temp->rc--;
        }
        n->copy = NULL;
        return n1;
    case Match:
        return n;
    case Let:
        n1 = InitLet(InitBVar(NULL), n->content.let.t2, n->content.let.t3);
        n->copy = n1;
        temp = n1->content.let.t2;
        n1->content.let.t2 = Inst(temp, l, sub);
        if (temp != n1->content.let.t2) {
            PushToListHT(n1->content.let.t2->parentNodes, n1);
            n1->content.let.t2->rc++;
            RemoveHT(temp->parentNodes, n1);
            temp->rc--;
        }
        temp = n1->content.let.t3;
        n1->content.let.t3 = Inst(temp, l, sub);
        if (temp != n1->content.let.t3) {
            PushToListHT(n1->content.let.t3->parentNodes, n1);
            n1->content.let.t3->rc++;
            RemoveHT(temp->parentNodes, n1);
            temp->rc--;
        }
        n->copy = NULL;
        return n1;
    case FRic:
        iterList = n->content.fRic.arg->head;
```

```
        while (iterList->node != NULL) {
            PushToListHT(arg, iterList->node);
            iterList = iterList->next;
        }

        n1 = InitFRic(InitBVar(NULL), n->content.fRic.t, n->content.fRic.n, ar
        n->copy = n1;
        temp = n1->content.fRic.t;
        n1->content.fRic.t = Inst(temp, l, sub);
        if (temp != n1->content.fRic.t) {
            PushToListHT(n1->content.fRic.t->parentNodes, n1);
            n1->content.fRic.t->rc++;
            RemoveHT(temp->parentNodes, n1);
            temp->rc--;
        }
        iterList = n1->content.fRic.arg->head;
        while (iterList->node != NULL) {
            temp = iterList->node;
            iterList->node = Inst(temp, l, sub);
            if (temp != iterList->node) {
                PushToListHT(iterList->node->parentNodes, n1);
                iterList->node->rc++;
                RemoveHT(temp->parentNodes, n1);
                temp->rc--;
            }
            iterList = iterList->next;
        }
        n->copy = NULL;
        return n1;
    case GCoRic:
        iterList = n->content.gCoRic.arg->head;
```

```
        while (iterList->node != NULL) {
            PushToListHT(arg, iterList->node);
            iterList = iterList->next;
        }
        n1 = InitGCoRic(InitBVar(NULL), n->content.gCoRic.t, n->content.gCoR
        n->copy = n1;
        temp = n1->content.gCoRic.t;
        n1->content.gCoRic.t = Inst(temp, l, sub);
        if (temp != n1->content.gCoRic.t) {
            PushToListHT(n1->content.gCoRic.t->parentNodes, n1);
            n1->content.gCoRic.t->rc++;
            RemoveHT(temp->parentNodes, n1);
            temp->rc--;
        }
        iterList = n1->content.gCoRic.arg->head;
        while (iterList->node != NULL) {
            temp = iterList->node;
            iterList->node = Inst(temp, l, sub);
            if (temp != iterList->node) {
                PushToListHT(iterList->node->parentNodes, n1);
                iterList->node->rc++;
                RemoveHT(temp->parentNodes, n1);
                temp->rc--;
            }
            iterList = iterList->next;
        }
        n->copy = NULL;
        return n1;
    case Constructor:
        iterList = n->content.jCostr.arg->head;
        while (iterList->node != NULL) {
```

```
                PushToListHT(arg, iterList->node);
                iterList = iterList->next;
            }
            n1 = InitConstructor(n->content.jCostr.j, arg, n->content.jCostr.n);
            iterList = n1->content.jCostr.arg->head;
            while (iterList->node != NULL) {
                temp = iterList->node;
                iterList->node = Inst(temp, l, sub);
                if (temp != iterList->node) {
                    PushToListHT(iterList->node->parentNodes, n1);
                    iterList->node->rc++;
                    RemoveHT(temp->parentNodes, n1);
                    temp->rc--;
                }
                iterList = iterList->next;
            }
            return n1;
        case Constant:
            return n;
    }
}
```

## A.7 AppReplacement

```
Node *AppReplacement(Node *tj, List *args) {
    Node *result = tj;
    ListItem *arg = args->head;
    while (arg->node != NULL) {
        result = InitApp(result, arg->node);
        arg = arg->next;
```

```
    }
    return result;
}
```

# Bibliografia

[1] AGGIUNGERE-RIFERIMENTO - Accattoli, Condoluci e Sacerdoti Coen, Sharing Equality is Linear.

[2] AGGIUNGERE-RIFERIMENTO - Accattoli e Dal Lago.

[3] M.S. Paterson and M.N. Wegman. 1978. Linear unification. J. Comput. System Sci. 16, 2 (1978), 158 - 167. https://doi.org/10.1016/0022-0000(78)90043-0.

# Ringraziamenti

Qui possiamo ringraziare il mondo intero!!!!!!!!!!
Ovviamente solo se uno vuole, non è obbligatorio.