Indice

INTRODUZIONE

Da sempre esiste in informatica una disputa espressiva tra programmi scritti secondo il paradigma imperativo e programmi scritti con linguaggi funzionali. In questa tesi parlerò di due processi di trasformazione che costituiscono un'importante trade-off tra questi due stili differenti di programmazione.

Il lambda-lifting di un programma è , infatti , una trasformazione di un programma strutturato in un set di equazioni ricorsive.

Il lambda dropping invece è la trasformazione simmetrica , che avendo in input un set di equazioni ricorsive restituisce un programma con struttura a blocchi .

Varie sono le applicazioni di questi processi di trasformazione , il più importante è la *partial evaluation* , intesa come tecnica di ottimizzazione di un programma attraverso la specializzazione.

L' obbiettivo principale di questa tesi , è quello di creare delle implementazioni algoritmiche delle fasi di trasformazione che portano alla realizzazione di programmi lambda-liftati e lambda-droppati e inoltre di fornirne esempi esplicativi su programmi scritti in linguaggi funzionali e su programmi residui ottenuti tramite specializzazione.

La tesi è quindi organizzata nel seguente modo. Il Capitolo 1 fornisce una prima definizione generale dei due processi oggetto della tesi , dando uno sguardo alle caratteristiche generali dei paradigmi di programmazione imperativa e funzionale.

Nel Capitolo 2 avremo una definizione formale dei programmi del prim'ordine e una realizzazione algoritmica delle trasformazioni caratterizzanti lambda-lifting e lambda-dropping applicate a questo tipo di programmi.

Nel Capitolo 3 descriverò come gli algoritmi implementati nel capitolo 2 saranno applicati a programmi di ordine superiore.

Il Capitolo 4 descrive più nel dettaglio la motivazione principale dell'utilizzo di lambda lifting e lambda dropping con un esempio di applicazione su un programma specializzato tramite un partial evaluator.

Il Capitolo 5 chiude la tesi con una discussione sulle diverse direttive per i futuri lavori, con uno sguardo a quelle che possono essere le altre applicazioni come per esempio la forma SSA di un programma , l'ottimizzazione dei compilatori e l'analisi di dipendenza di Peyton Jones.

CAPITOLO 1

λ-lifting e λ-dropping

1.1 Programmazione Imperativa

La **programmazione imperativa** è un paradigma di programmazione secondo cui un programma viene inteso come un insieme di **istruzioni** (dette anche **direttive** o **comandi**), ciascuna delle quali può essere pensata come un "ordine" che viene impartito alla macchina.

La programmazione imperativa consiste nel :

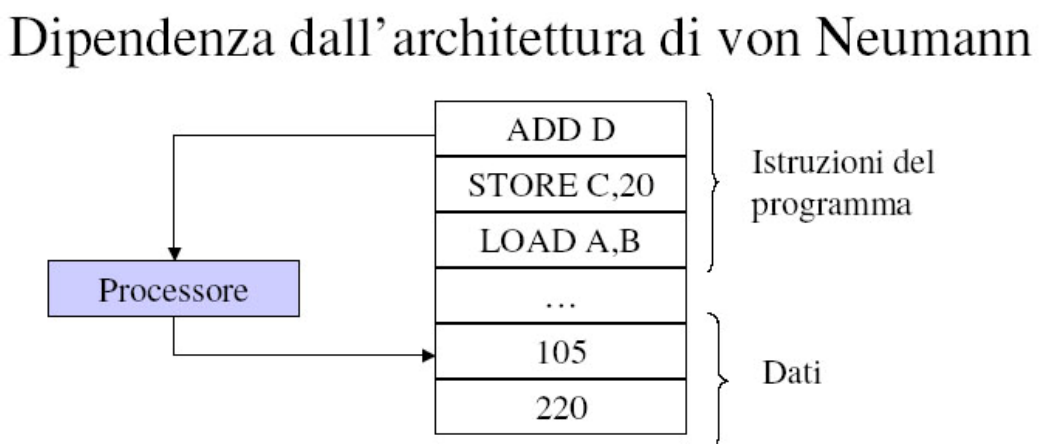
- Determinare quali valori saranno richiesti per la computazione
- Rappresentare questi valori associandoli a locazioni di memoria.
- Determinare una sequenza per passi di trasformazioni di quanto si trova in memoria.

La base del paradigma imperativo è la manipolazione , totalmente specificata e controllata ,dei dati in passi successivi.

I dati sono rappresentati da variabili , il cui nome (identificatore) è legato ad una posizione di memoria e ad un valore.

Le caratteristiche essenziali della programmazione imperativa sono strettamente legate all'architettura di Von Neumann che brevemente possiamo defeinire come una architettura costituita da due componenti fondamentali:

- memoria (componente passiva)
- processore (componente attiva)



La principale attività del processore e' eseguire calcoli e assegnare valori (svolge quindi un ruolo attivo) a celle di memoria (che è quindi passiva) (si denoti a questo proposito anche il concetto di variabile come astrazione logica di una cella di memoria).

L'approccio imperativo è l'approccio dominante in programmazione. La programmazione imperativa viene generalmente contrapposta a

quella dichiarativa, in cui un programma consiste in un insieme di "affermazioni" (non "ordini") che la macchina virtuale del linguaggio è (implicitamente) tenuta a considerare vere e/o rendere vere. Un esempio di paradigma dichiarativo è la programmazione logica.

1.2 PROGRAMMAZIONE FUNZIONALE

La **programmazione funzionale** è un paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche. Solitamente questo approccio viene usato maggiormente in ambiti accademici piuttosto che industriali. La programmazione funzionale pura è caratterizzata dal seguente principio :*il valore di un'espressione dipende solo dalle sue (eventuali) sotto-espressioni.*

Il valore dell'espressione $a+b$ è semplicemente la somma dei valori di a e b . Poichè questo principio bandisce ogni tioo di *side-effecting* (<< effetto collaterale >>) delle espressioni , la programmazione funzionale pura può essere anche caratterizzata come programmazione senza assegnamenti. In assenza di side-effect ogni

valutazione di una stessa espressione produce sempre lo stesso risultato. Un'altra caratteristica dei linguaggi funzionali è che gli utenti non devono occuparsi della memorizzazione dei dati : le operazioni predefinite sui dati allocano memoria secondo necessità. La memoria non più accessibile viene deallocata automaticamente . L'assenza di codice esplicito per la deallocazione rende i programmi più semplici e brevi. Una conseguenza di questo approccio è che l'implementazione deve realizzare la cosiddetta *garbage collection* per recuperare la memoria non più accessibile .le funzioni vengono trattate come dati di prima classe, cioè una funzione è come un qualunque altro valore . Una funzione può rappresentare il valore di un'espressione , può essere passata come argomento e può essere posta in una struttura dati. Il trattamento delle funzioni come dati di prima classe consente la creazione di operazioni molto potenti su collezioni di dati , come mapcar in Lisp e map in Haskell, che prendono una funzione e una lista come input ed applicano la funzione in input ad ogni elemento della lista, di restituire una lista dei risultati.

I linguaggi funzionali permettono inoltre una tecnica chiamata *currying*, che permette di trasformare una funzione con parametri multipli in una funzione con un solo parametro che mappa ad un'altra funzione con un solo parametro e così via, fino all'esaurimento dei

parametri. La funzione trasformata può essere applicata ad un sottoinsieme dei suoi parametri iniziali e dare come risultato una nuova funzione dove i parametri di quel sottoinsieme sono costanti e il resto dei parametri hanno valori non ancora specificati. Infine questa nuova funzione può essere applicata ai parametri rimanenti per ottenere il risultato finale. Per esempio, una funzione somma(x,y) = x + y può essere trasformata in modo tale che il valore di ritorno di somma(2) (nota che non c'è il parametro y) sia una funzione anonima equivalente alla funzione somma2(y) = 2 + y. Questa nuova funzione ha un solo parametro a cui somma 2. Questa tecnica non sarebbe possibile se le funzioni non fossero entità *di prima classe*.

L'esempio più vecchio di linguaggio funzionale è il *Lisp*, anche se né il LISP originale né i Lisp moderni, come il Common Lisp, sono puramente funzionali. Le varianti del Lisp includono il *Logo*, lo *Scheme* e *Dylan*. Esempi di linguaggi funzionali moderni sono l'*Haskell* e i linguaggi della famiglia ML.

1.2.1 Un primo linguaggio funzionale : Il linguaggio del λ-calcolo

Definizione del Calcolo

I Principi. L'attività matematica di calcolo si può ridurre in gran parte alla trasformazione di espressioni (le formule matematiche) per mezzo di regole convenzionalmente stabilite. Combinando trasformazioni sostanzialmente elementari si possono elaborare computazioni anche di grande complessità. Per esempio pensiamo a come viene definita nella pratica (non nella teoria matematica!) l'operazione di somma tra numeri naturali che supponiamo scritti nell'usuale base decimale.

- Per prima cosa si stabilisce per mezzo di una tabella qual `è il risultato della somma di numeri di una sola cifra.
- Quindi estenderemo le regole per poter calcolare anche la somma di numeri composti di più cifre.
- Per finire ci doteremo di regole che ci consentano di manipolare espressioni che contengano somme anche simboliche.

Ognuna di queste fasi può essere codificata in modo piuttosto semplice per mezzo di regole di calcolo meccanico. La tabella può esser descritta da una collezione finita di regole (esattamente 100) del tipo:

$$0 + 0 = 0,\ 0 + 1 = 1,\ 0 + 2 = 2,\ \ldots\ 1 + 0 = 1,$$

E` chiaro che possiamo ridurre il numero di regole se ammettiamo di avere una nuova regola che ci permetta di scambiare il ruolo degli addendi (la proprietà commutativa!) che potremmo scrivere così:

$$x + y = y + x.$$

In questa regola compaiono, però, oggetti sintattici che chiamiamo variabili delle quali è necessario chiarire ruolo e significato.

Le teorie del λ-calcolo e dei combinatori consentono di chiarire il ruolo delle regole e delle variabili nella definizione di tipo funzionale dei processi di computazione e inoltre forniscono un ambiente per i concetti di globalità e località presenti in tutti i linguaggi di programmazione ad alto livello.

Il linguaggio del λ-calcolo. L'alfabeto del λ-calcolo è costituito da :

- Un'insieme finito V di simboli per variabili;
- Le parentesi tonde "(" e ")" ;
- Due simboli speciali : il *punto* "." e *l'operatore di astrazione* "λ".

L'insieme Λ delle espressioni del λ-calcolo - dette λ-termini o semplicemente termini - è definito dalla seguente grammatica:

$$\Lambda \quad :== V \mid () \mid (V\ )$$

Seguendo la tradizione consolidata del linguaggio matematico le variabili saranno normalmente indicate con le ultime lettere dell'alfabeto latino arricchite eventualmente da pedici:

u, x, y, z, . . . , x1, x2, . . .

I termini generici saranno indicati in generale con t, t , t , t1, t2, . . . o più liberamente con altre lettere dell'alfabeto latino se la scrittura dovesse diventare di difficile lettura.

Il linguaggio del λ-calcolo può essere descritto anche in forma diversa, cioè per mezzo di un numero finito di regole di deduzione.

Questo modo di presentare un insieme di espressioni è utile quando si tratta di integrare un particolare linguaggio all'interno di un sistema logico-deduttivo.

L'insieme dei λ-termini è definito dalle seguenti regole :

(atomica)

$$\frac{x \in V}{x \in \Lambda}$$

(astrazione) $$\frac{x \in V \quad t \in \Lambda}{(\lambda x t) \in \Lambda}$$

(applicazione) $$\frac{t, t' \in \Lambda}{(tt') \in \Lambda}$$

Alcune espressioni possono essere pressoché incomprensibili per la grande quantità di parentesi; si adottano allora delle convenzioni per migliorarne la leggibilità, eccone un elenco completo :

$\lambda x.t$ sta per $(\lambda x t)$

(astrazione) $e_1 e_2 \ldots e_{n-1} e_n$ sta per $((\ldots(e_1 e_2)\ldots e_{n-1})e_n)$

(associativa)

$\lambda x_1 x_2 \ldots x_n.e$ sta per $(\lambda x_1(\lambda x_2 \ldots(\lambda x_n e)\ldots))$

(associativa)

Si noti che l'applicazione "associa a sinistra", mentre l'astrazione "associa a destra".

Variabili libere e variabili legate. Bisogna ora spiegare il significato intuitivo dell'astrazione nel λ-calcolo partendo da alcuni esempi. Si consideri per cominciare la definizione di una funzione f(x) di variabile reale data nel modo usuale:

$$f(x) = x^2$$

Il significato della definizione è abbastanza ovvio: la funzione f calcola il quadrato di un numero reale. L'espressione x non è un numero reale, ma una variabile che rappresenta l'argomento della funzione; il valore è rappresentato dall'espressione x^2; il lato sinistro dell'uguaglianza non ha un ruolo matematico essenziale, infatti se scrivessimo :

$$g(x) = x^2$$

staremmo semplicemente definendo la stessa funzione dandole un nome diverso. Dunque quello che è davvero importante è l'espressione x^2 che descrive in modo matematicamente chiaro di che funzione si tratta.

Supponiamo ora di voler generalizzare la funzione, ammettendo che l'esponente sia un qualunque numero naturale n fissato:

$$f(x)=x^n$$

Questa funzione potrebbe generare delle ambiguità, in quanto essa può essere interpretata come una funzione a due argomenti oppure come una funzione sempre ad un solo argomento, ma nella quale l'esponente va inteso come fissato ma non specificato in un primo momento.

Normalmente l'interpretazione più comune è quest'ultima, ma in questo caso la semplice espressione x^n non è abbastanza esplicativa. La situazione è ancora peggiore se volessimo scrivere:

$$f_n(x) = x^n$$

Si potrebbe infatti pensare che stiamo davvero scrivendo una funzione a due argomenti oppure che f_n è semplicemente un nome che comprende una lettera che compare anche nel definiens.

Poiché nel λ-calcolo non sono ammesse questo tipo di ambiguità, si introduce l'operatore di astrazione che indica esplicitamente quali entità linguistiche debbano essere intese come argomenti della funzione e quali no.

Rifacendosi agli esempi appena illustrati, la prima funzione viene descritta bene dall'espressione

$$\lambda x.x^2$$

indicando così esplicitamente che la lettera x è l'argomento; si dice che x è legata dall'operatore di astrazione. Nel secondo caso la funzione è adeguatamente descritta da

$$\lambda x.x^n$$

e dal momento che solo x è legata dall'operatore di astrazione si deduce che x è argomento dell'espressione funzionale, mentre n è da intendersi come variabile libera. Se invece si vuole considerare tale espressione come funzione di due argomenti si dovrà scrivere nx.xn oppure xn.xn che non è la stessa cosa.
A questo punto diamo le seguenti definizioni :

Definizione 3. Si dice occorrenza di una variabile $x \in V$ nel termine t ogni simbolo linguistico x che compare nell'espressione linguistica t. Ad esempio, la variabile x occorre due volte nell'espressione (xx), una volta in (x(yy)) e mai in (yy).

Definizione 4. Definiamo con mutua induzione sulla costruzione dei λ-termini,le nozioni di occorrenza libera e legata.

· Una variabile x che viene introdotta dalla regola atomica occorre libera nel termine x.

· Le occorrenze libere (legate) delle variabili in un termine del tipo (tt) sono esattamente le occorrenze libere (legate) delle variabili in t e in t .

· Le occorrenze libere (legate) delle variabili in un termine del tipo (x) sono esattamente le occorrenze libere (legate) delle variabili in t con la sola eccezione delle occorrenze libere di x in t che nel termine (xt) sono legate.

Riduzioni

Nel λ-calcolo le computazioni vengono sostanzialmente eseguite per mezzo delle sostituzioni delle occorrenze libere delle variabili con termini.

Definizione 5 (sostituzione). Siano t,t e x V ; definiamo t[t /x], la sostituzione di t per ogni occorrenza libera di x in t, induttivamente sulla costruzione del termine.

· x[t /x] = t;

· z[t /x] = z se z = x;

· se t = (t1t2) allora t[t /x] = (t1t2) = ((t1[t /x])(t2[t /x]));

· se t = (xt1) allora t[t /x] = (xt1)[t /x] = t (t rimane inalterato perchè x non `e libera in t).

· se t = (zt1) e x = z allora t[t /x] = (zt1)[t /x] = (z(t1[t /x])).

Conversioni.

Possiamo ora definire le regole di cui abbiamo necessità per eseguire i calcoli :

Nella scrittura di algoritmi con linguaggi di programmazione ad altro livello si scopre presto che il nome delle variabili, dei parametri formali di una funzione o di una procedura è irrilevante, purchè una volta attribuito un nome ad una certa entità tale nome venga usato con coerenza.

Questi fatti sono tutti aspetti di uno stesso fatto: le variabili sono da un punto di vista linguistico semplici nomi e il nome delle variabili legate può esser liberamente cambiato pur rispettando alcune prescrizioni.

Questa posizione è espressa chiaramente nel λ-calcolo con la regola chiamata α-conversione e definita qui di seguito.

Definizione 6. Sia t = x.t e sia y una variabile che non occorre libera in t; in tal caso il termine t pu`o esser modificato secondo la regola detta di λ–conversione così definita:

(α) $\lambda x.t' \ \alpha \ \lambda y.(t' [y/x])$

La seconda regola è quella che davvero esegue il calcolo, infatti essa è sufficiente per calcolare tutte le funzioni calcolabili.

Partiamo dall'esempio della funzione $f(x) = x^2$; si è detto che tale funzione si può rappresentare in λ-calcolo con $\lambda x.x^2$ l'applicazione della funzione ad uno specifico argomento (ad esempio il numero 5) viene comunemente indicata con f(5); e nel -calcolo?

Nel λ-calcolo abbiamo la regola di applicazione (di una funzione al suo argomento) che permette di formare

$$(x.x^2)5$$

ma non abbiamo ancora alcun metodo per ottenere quella che viene considerata

usualmente la scrittura esplicita di f(5), cio`e 5^2.

Sarà l'operatore di sostituzione ha metter in pratica questo passaggio mediante la regola di conversione descritta nella definizione che segue.

Definizione 7. Siano $\lambda x.t, t'$; allora il termine risultante dall'applicazione del primo termine al secondo può essere modificato secondo la regola detta di β-conversione così definita:

$$(\beta) \qquad (\lambda x.t)t' \alpha \ \lambda y.(t[t'/x])$$

purchè nessuna occorrenza libera di variabili di t' diventi legata (cattura) per effetto

della sostituzione.

Consideriamo le espressioni matematiche $f(x) = x^2 + 2x + 1$ e $g(x) = (x + 1)^2$; siamo tutti d'accordo nel dire che definiscono la stessa funzione matematiche anche se suggeriscono modi diversi per il calcolo.

Questo aspetto è molto importante nella trattazione degli algoritmi in informatica: due algoritmi anche se diversi per la loro strategia di risoluzione, possono calcolare la stessa funzione. Nel λ-calcolo così per come lo abbiamo definito non è in grado non possibile considerare

diversi algoritmi (λ-termini) che calcolano la stessa funzione insiemistica.

Per rendere possibile ciò, quando ciò è utile ci si avvale di un'apposita regola di calcolo:

Definizione 8. Siano λx.(tx); dove x non compare libera in t; allora il termine in oggetto può essere modificato secondo la regola detta di *η-conversione* così definita:

(η) (λx.(tx) α t

1.3 PROGRAMMAZIONE FUNZIONALE VS. PROGRAMMAZIONE IMPERATIVA

Se paragonata alla programmazione imperativa, può sembrare che la programmazione funzionale manchi di molti costrutti spesso (ma incorrettamente) ritenuti essenziali per un linguaggio imperativo, come il C o il Pascal. Per esempio, nella programmazione funzionale rigorosa, non c'è alcuna esplicita allocazione di memoria o assegnazione di variabile, ma queste operazioni avvengono automaticamente quando una funzione è invocata: l'allocazione di memoria avviene per creare lo spazio necessario per i parametri e il

valore di ritorno e l'assegnazione avviene per copiare i parametri nel nuovo spazio allocato e per copiare il valore di ritorno alla funzione chiamante. Entrambe le operazioni possono avvenire solo alla chiamata di una funzione e al ritorno da essa e quindi gli effetti collaterali sono eliminati. Eliminando gli effetti collaterali dalle funzioni, si ottiene ciò che viene chiamata *trasparenza referenziale*, che assicura che il risultato di una funzione sia lo stesso per uno stesso insieme di parametri, indifferentemente da quando e dove questa funzione venga valutata. La trasparenza referenziale rende molto più facile sia la dimostrazione della correttezza del programma sia l'identificazione automatica delle computazioni indipendenti per l'esecuzione parallela.

Le iterazioni, un altro costrutto della programmazione imperativa, sono ottenute attraverso il costrutto più generale delle chiamate ricorsive a funzioni. Le funzioni ricorsive invocano se stesse permettendo di eseguire più e più volte una stessa operazione. La ricorsione nella programmazione funzionale può assumere molte forme e, in generale, è una tecnica più potente dell'iterazione. Per questa ragione, quasi tutti i linguaggi imperativi la supportano (con la notevole eccezione del Fortran 77 e del Cobol, prima del 2002).

1.4 Lambda-lifting e Lambda-dropping

I processi di lambda-lifting e lambda dropping risolvono la disputa tra lo stile dei programmi strutturati (imperativi) e quello piuttosto delle funzioni ricorsive. Infatti , questi ci permettono di trasformare programmi scritti in uno qualsiasi dei due stili di programmazione in quello opposto. Più in generale , programmi scritti in uno stile ibrido (in parte liftato e in parte droppato) possono essere interamente lambda-liftati o lambda-droppati.

Questo portò, alla metà degli anni ottanta, Hughes Johnsson e Peyton Jones a progettare lambda-lifting come una trasformazione simmetrica di programmi strutturati in funzioni ricorsive.

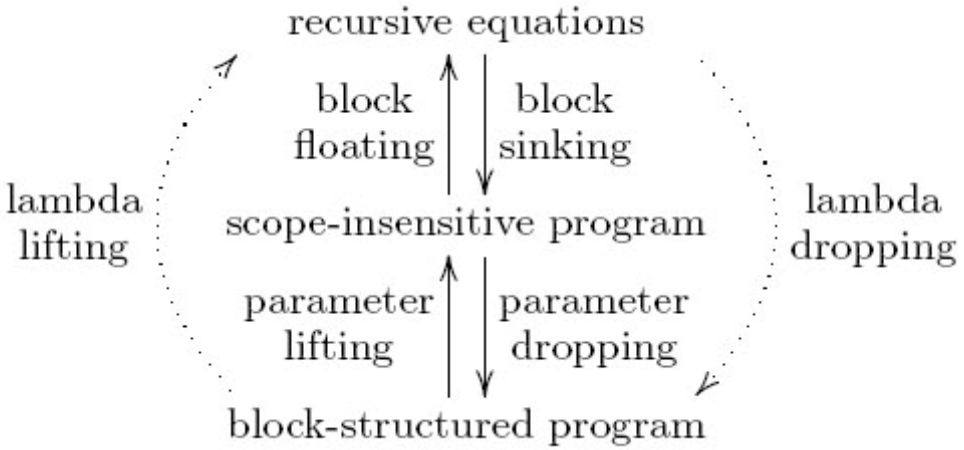
Il lambda lifting è un processo che trasforma programma strutturato in un set di equazioni ricorsive. Il lambda-dropping invece è il processo inverso e ripristina la struttura di un programma a partire da un set di equazioni ricorsive.

- Nel lambda-lifting ogni variabile che occorre libera nel corpo di una funzione sarà passata come parametro della funzione stessa.

- Nel lambda dropping i parametri che sono usati nello stesso ambito di visibilità della loro definizione , non necessitano di essere passati come parametri.

Un programma i cui blocchi non usano variabili libere è detto *scope-insensitive,* pochè il legame variabile-valore non dipende dallo scope lessicale della variabile ma è legato al valore del parametro formale corrispondente alla variabile. In un programma scope-insensitive i blocchi locali ad ogni funzione sono liberi di emergere (per il lambda lifting) o di affondare (per il lambda dropping).

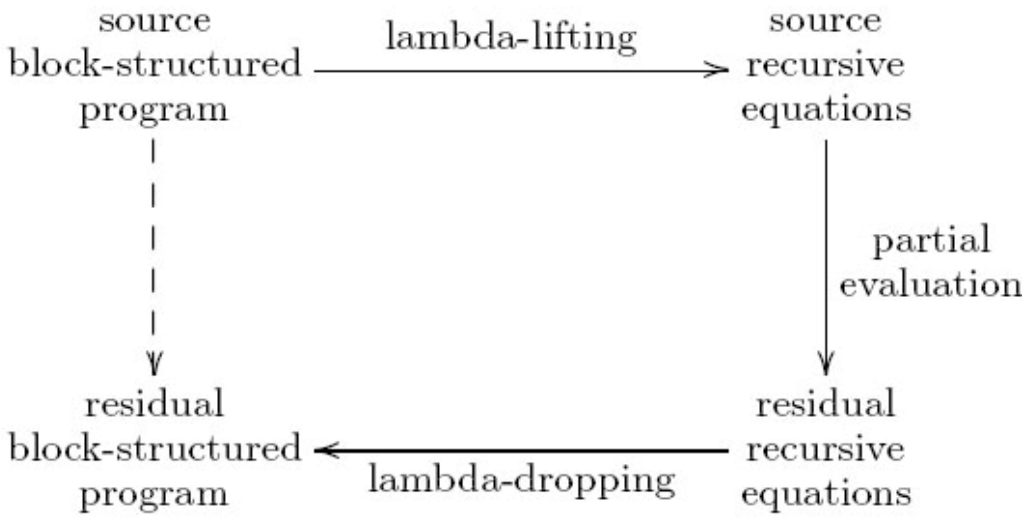
Per riassumere :



La prima applicazione di questi due processi è la *partial evaluation* , intesa come tecnica di ottimizzazione di un programma attraverso la specializzazione.

1.4.1 Partial Evaluation e lambda lifting / lambda dropping

Molti valutatori parziali per programmi procedurali operano su equazioni ricorsive. Per questo essi lambda liftano il programma sorgente in fase di pre-elaborazione. L'inconveniente è che spesso , i valutatori parziali producono automaticamente equazioni ricorsive residue , con dozzine di parametri che molti compilatori non trattano efficientemente. Questo problema critico si risolve lambda droppando i programmi residui in fase di post elaborazione migliorando così i loro tempi di compilazione e di esecuzione.



Il lambda lifting nei compilatori per programmi funzionali è stato presentato come una rappresentazione intermedia. Anche lambda dropping è usato come una trasformazione intermedia di programmi

sorgente : si è notato infatti che lambda droppare un programma corrisponde a trasformarlo nella rappresentazione funzionale della sua forma *SSA* ottimale. ma vedremo questi concetti più avanti , per adesso il mio obbiettivo sarà quello di entrare nello specifico dei due processi , analizzando il loro comportamenti prima su programmi del prim'ordine e poi su programmi di ordine superiore.

CAPITOLO 2

Programmi del prim'ordine

2.1 Introduzione

La sintassi dei linguaggi del prim'ordine può essere descritta attraverso le seguenti regole :

$$
\begin{array}{rcll}
p \in \mathsf{Program} & ::= & \{d_1, \ldots, d_m\} & \\
d \in \mathsf{Def} & ::= & f \equiv \lambda(v_1, \ldots, v_n).e & \\
e, \cancel{e} \in \mathsf{Exp} & ::= & \ell & \\
& | & v & \\
& | & f\,(e_1, \ldots, e_n) & \\
& | & \mathrm{LetRec}\,\{d_1, \ldots, d_k\}\,e_0 & \text{where } k > 0 \\
\ell \in \mathsf{Literal} & & & \\
v, \cancel{v} \in \mathsf{Variable} & & & \\
f \in \mathsf{FunctionName} \cup \mathsf{PredefinedFunction} & & &
\end{array}
$$

Figure 1: Simplified syntax of first-order programs.

Queste regole sono scritte in BNF (Backus Naur Form) , una notazione comunemente usata per descrivere la sintassi dei linguaggi di programmazione.

Nella tavola si può notare che un programma è definito come un insieme di definizioni.

Ciascuna delle occorrenze di una definizione è espansa in una funzione descritta attraverso l'espressione $\lambda(v_1,\ldots v_n).e$, che specifica esplicitamente , attraverso l'operatore di astrazione λ , che $(v_1,\ldots v_n)$ sono gli argomenti della funzione.

A sua volta ogni espressione può essere : un letterale (costante), o una variabile, o una funzione che ha come argomenti delle espressioni, o la definizione locale di altre definizioni.

Si noti , inoltre, come alcune espressioni sono segnate con una linea orizzontale , per le trasformazioni successive del programma.

La particolarità di un programma del prim'ordine è che le funzioni hanno come argomenti solo variabili o costanti , nella lista dei loro parametri formali non ci possono essere altre funzioni.

Un programma è scritto come un insieme di funzioni , ognuna delle quali può contenere altre funzioni annidate nei suoi blocchi . Gli identificatori delle funzioni si trovano solamente nella posizione in cui la funzione stessa è definita. In una applicazione si possono trovare sia funzioni definite dai programmatori (FunctionName) e sia gli operatori messi gia a disposizione dal linguaggio (PredefinedFunction).

Useremo inoltre la notazione FF per indicare set di variabili libere dichiarate come funzioni *named* (definite dall'utente) e FV per indicare un set di variabili libere dichiarate come parametri formali.

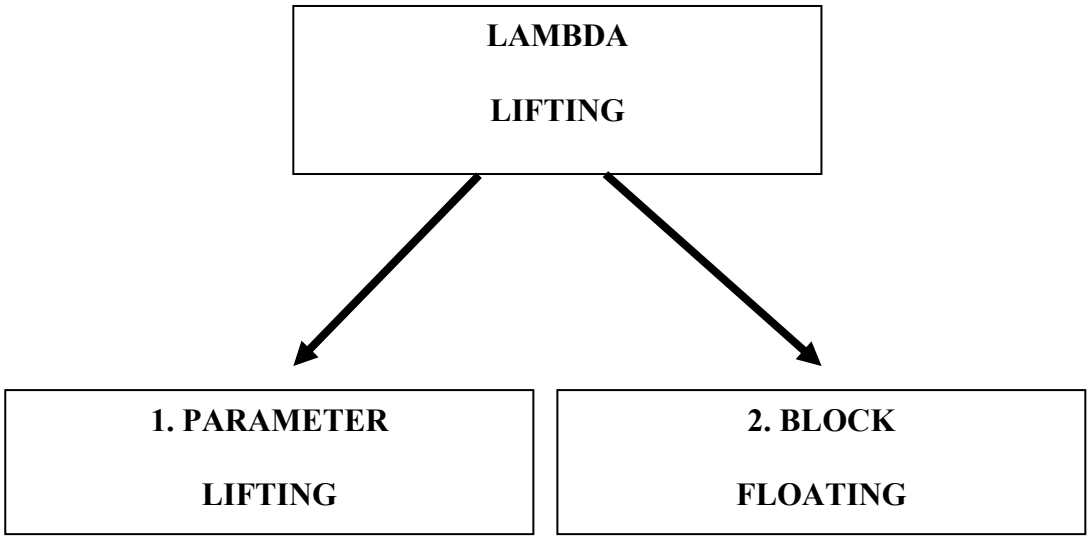
2.2 Basi di Lambda lifting e lambda dropping

Presenteremo lambda-lifting e lambda dropping in modo simmetrico attraverso quattro trasformazioni :

- Lifting dei parametri
- Emersione dei blocchi
- Affondamento dei blocchi
- Dropping dei parametri

2.2.1 Le basi del lambda lifting

Il lambda lifting si ottiene attraverso due trasformazioni :



Il lifting dei parametri :

Il lifting dei parametri rende un programma *scope insensitive,* aggiungendo ad ogni funzione , dei parametri formali extra , per ogni variabile che occorre libera nel corpo della stessa funzione.

Quindi si ha l'ampliamento della lista dei parametri formali di ogni lambda astrazione , con nuove variabili corrispondenti alle variabili libere che occorrono nella funzione. Allo stesso modo si procede con la lista dei parametri attuali della funzione, ad ogni punto di attivazione di quest'ultima.

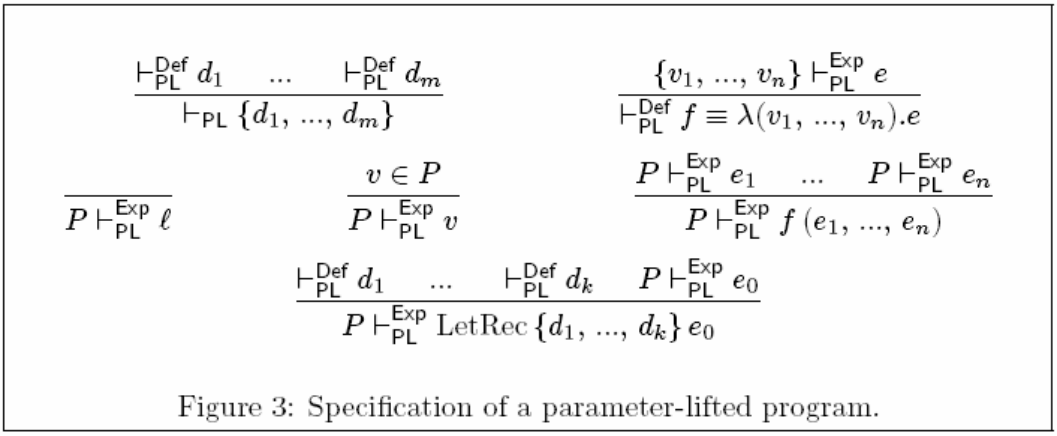


Figure 3: Specification of a parameter-lifted program.

La tavola sopra ci mostra la sintassi di un programma *parameter-lifted,* attraverso una serie di regole di inferenza che ci indicano il modo in cui i costrutti propri dei programmi del prim'ordine vengono riscritti all'interno di un programma con lifting dei parametri.

Definiamo più precisamente la notazione $V \vdash_G^D \varphi$.

Essa sta ad indicare che la formula φ,appartenente all'insieme D e avente valore in un'insieme di variabili V (ambiente), è derivabile nella grammatica G e " $\vdash$ " è il simbolo di derivazione.

Quindi, secondo le regole in figura 3 , un programma parameter-lifted :

- è costituito da un insieme di definizioni parameter-lifted
- ogni espressione avente variabili libere $\{v_1, \ldots\ldots\ldots, v_n\}$ all'interno di un programma PL (Parameter-Lifted) , è sostituita da una lambda-astrazione $\lambda(v_1, \ldots\ldots\ldots\ldots, v_n)$.e , nella quale le variabili $\{v_1, \ldots\ldots\ldots, v_n\}$ sono legate alla lambda astrazione.

- In ogni punto di un programma PL ci può essere un letterale *l* definito sull'insieme di variabili P.
- Ad ogni variabile *v* appartenente a P si può sostituire nel programma PL una variabile *v* nell'ambiente P.
- Se le espressioni $e_1,\ldots\ldots\ldots e_n$ sono espressioni parameter-lifted con variabili in P , allora esse sono argomenti di una forma $f(e_1,\ldots\ldots\ldots e_n)$.
- Se abbiamo le definizioni $d_1,\ldots\ldots\ldots,d_k$ e l'espressione e_0 , nel programma avremo una definizione di una funzione locale basata sulle definizioni $d_1,\ldots\ldots\ldots,d_k$ e il cui corpo sarà costituito dall'espressione e_0 (LetRec$\{ d_1,\ldots\ldots\ldots,d_k \}e_0$).

Pertanto la caratteristica che distingue un programma Parameter-Lifted da un normale programma del prim'ordine , è la totale assenza di variabili libere.

Le variabili presenti nel programma sono tutte legate alla definizione di una qualche lambda-astrazione .

In generale , quindi , un programma strutturato è interamente parameter-lifted se nessuna delle sue funzioni ha variabili libere .Infatti tutte le variabili che occorrono nel corpo della funzione sono dichiarate come parametri formali della stessa.

Un programma *p* , una definizione *d* e un'espressione *e* i cui parametri sono denotati da P , sono completamente parameter-lifted , se sono soddisfatte rispettivamente le seguenti condizioni :

$$\vdash_{\mathsf{PL}} p \qquad \vdash^{\mathsf{Def}}_{\mathsf{PL}} d \qquad P \vdash^{\mathsf{Exp}}_{\mathsf{PL}} e$$

Emersione dei blocchi

Il processo di emersione dei blocchi elimina la strutturazione del programma , globalizzando ogni blocco e rendendo ogni funzione definita localmente ad esso , come una funzione ricorsiva globale .

Le funzioni locali presenti in un blocco vengono spostate a livello dello scope di chiusura , ogni qualvolta esse non usano funzioni definite localmente o si riferiscano a variabili locali libere.

La sintassi di un programma BF (Block Floated) è espressa dalle seguenti regole di inferenza :

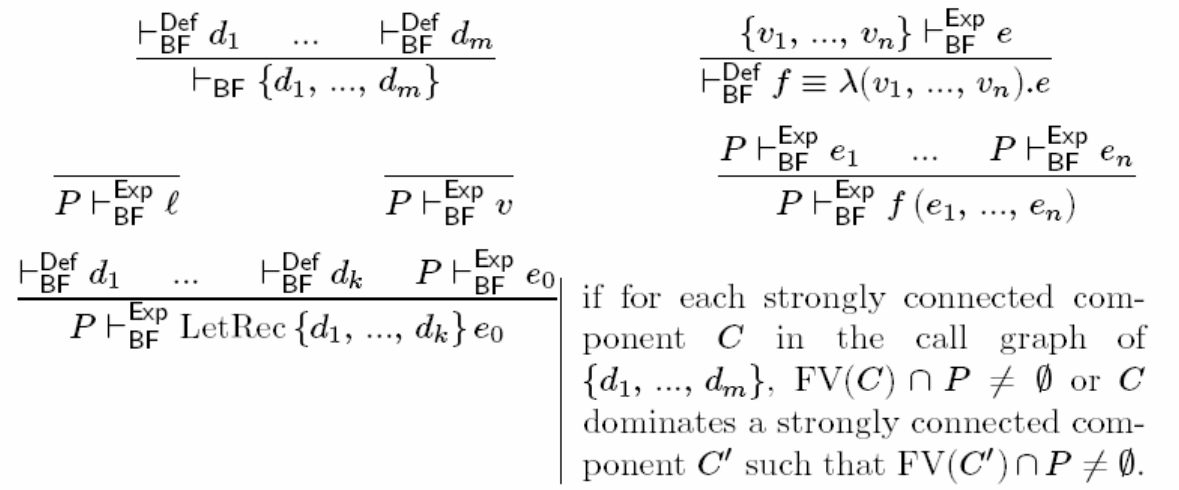


Figure 4: Specification of a block-floated program.

Secondo tali regole:

- Un programma BF è costituito da un'insieme di definizioni block-floated
- Un'espressione *e* block-floated , definita sull'insieme di variabili $\{v_1, \ldots\ldots, v_n\}$, appare nel programma come corpo di una lambda-astrazione avente parametri formali $v_1, \ldots\ldots, v_n$.
- In un programma con emersione dei blocchi ci possono essere sia letterali che variabili aventi come ambiente l'insieme di variabili P.
- Le espressioni block-floated $e_1, \ldots\ldots e_n$ definite sulle variabili di P , sono argomenti della funzione $f(e_1, \ldots\ldots e_n)$.
- Se abbiamo le definizioni $d_1, \ldots\ldots, d_k$ e l'espressione e_0 , nel programma avremo una definizione di una funzione locale basata

sulle definizioni $d_1,\ldots\ldots,d_k$ e il cui corpo sarà costituito dall'espressione e_0 (LetRec{ $d_1,\ldots\ldots,d_k$ }e_0).

In generale , quindi , un programma è completamente block-floated se tutte le sue funzioni sono globali.

Un programma p , una definizione d e un'espressione e i cui parametri sono denotati da P , sono completamente block-floated , se sono soddisfatte rispettivamente le seguenti condizioni :

$$\vdash_{BF} p \qquad \vdash_{BF}^{Def} d \qquad P \vdash_{BF}^{Exp} e$$

2.2.2 Algoritmi di lambda lifting

In questo paragrafo si realizzano delle soluzioni algoritmiche che implementano le due trasformazioni caratterizzanti il lambda lifting : lifting dei parametri e emersione dei blocchi .

Lifting dei parametri .

Per fare il lifting dei parametri di un programma , tutte le variabili libere di una funzione devono essere esplicitamente passate come parametri formali della medesima funzione; in tal modo tutte le

attivazioni della funzione dovranno passare , al momento della chiamata , dei parametri attuali che facciano riferimento ai nuovi parametri formali.

L'algoritmo , attraversa tutto il programma , analizzando prima le definizioni globali e poi scendendo ricorsivamente ad analizzare le definizioni locali e le espressioni , mentre allo stesso tempo costruisce un set di soluzioni. Una soluzione è una coppia (FunName,Set(Variable)) , ed associa ogni funzione all'insieme minimo di variabili che bisognerà passarle come argomenti.

parameterLiftProgram p=map(ParameterLiftDefØ) p

ParameterLiftDef S ($f \equiv \lambda(v_1,\ldots\ldots v_n).$(parameterLiftExp S e))=applySolutionToDef S

La funzione "parameterLiftExp" analizza le espressioni e agisce in questo modo : se l'espressione che sta analizzando è un letterale o una variabile allora restituisce l'espressione stessa , se l'espressione è del tipo $f(e_1,\ldots\ldots e_n)$ allora la funzione "parameterLiftExp" viene applicata ricorsivamente a $e_1,\ldots\ldots e_n$. Se invece l'espressione è una definizione locale delle definizioni $d_1,\ldots\ldots,d_k$ (LetRec$\{ d_1,\ldots\ldots,d_k \}e_0$) allora , l'insieme di variabili libere (V_{fi}=FV($f_i \equiv l_i$)) che occorrono nel corpo della funzione sarà ampliato con le variabili libere che occorrono in ogni blocco della stessa ; la stessa cosa si fa con

l'insieme delle funzioni che occorrono libere nel corpo della funzione (F_{fi}=FF($f_i \equiv l_i$)).

```
parameterLiftExp :: Set(FunName,Set(Variable)) → Exp→ Exp
parameterLiftExp S (ℓ) = ℓ
parameterLiftExp S (v) = v
parameterLiftExp S (f(e1, ..., en)) =
        applySolutionToExp S (f (map parameterLiftExp (e1, ...,
en)))
parameterLiftExp S (LetRec f(d1, ..., dk)e0) =
    foreach (fi ≡ li)Є f(d1, ..., dk) do
      Vfi := FV(fi ≡ li);
      Ffi := FF(fi ≡ li)
   foreach Ffi Є {Ff1,…., Ffk}  do
       foreach (g, Vg)Є S such that gЄ Ffi do
          Vfi := Vfi U Vg;
           Ffi := Ffi \ {g}
```

Quindi ogni blocco da luce ad una serie di equazioni che descrivono quali variabili saranno candidate ad essere passate come argomenti alle funzioni definite dal blocco.

Abbiamo equazioni ricorsive per funzioni mutuamente ricorsive , pertanto , per il *teorema del punto fisso di Kleene* , saranno risolte da *iterazioni a punto fisso*. Il set di soluzioni quindi sarà esteso con la

soluzione del set di equazioni e usato per analizzare il corpo del blocco di ogni funzione locale.

fixpoint over $\{V_{f1},\ldots\ldots,V_{fk}\}$ by
  foreach Ffi Є $\{F_{f1},\ldots\ldots, F_{fk}\}$ do
    foreach g Є F_{fi} do
      V_{fi} := V_{f}i U V_{g}
let $S^{'}$ = S U {(f1, Vf1),……, (fk, Vfk)}
  f_0 = map (parameterLiftDef $S^{'}$) {d1,…..,dk}
  $e_0^{'}$= parameterLiftExp S_0 e_0
in (LetRec f_0 $e_0^{'}$)

Il risultato sarà :

```
parameterLiftProgram :: Program → Program
parameterLiftProgram p = map (parameterLiftDef ∅) p

parameterLiftDef :: Set(FunName,Set(Variable)) → Def → Def
parameterLiftDef S (f ≡ λ(v1, ..., vn).e) =
    applySolutionToDef S (f ≡ λ(v1, ..., vn).(parameterLiftExp S e))

parameterLiftExp :: Set(FunName,Set(Variable)) → Exp → Exp
parameterLiftExp S (ℓ) = ℓ
parameterLiftExp S (v) = v
parameterLiftExp S (f (e1, ..., en)) =
    applySolutionToExp S (f (map parameterLiftExp (e1, ..., en)))
parameterLiftExp S (LetRec {d1, ..., dk} e0) =
    foreach (fi ≡ li) ∈ {d1, ..., dk} do
        Vfi := FV(fi ≡ li);
        Ffi := FF(fi ≡ li)
    foreach Ffi ∈ {Ff1,...,Ffk} do
        foreach (g, Vg) ∈ S such that g ∈ Ffi do
            Vfi := Vfi ∪ Vg;
            Ffi := Ffi \ {g}
    fixpoint over {Vf1,...,Vfk} by
        foreach Ffi ∈ {Ff1,...,Ffk} do
            foreach g ∈ Ffi do
                Vfi := Vfi ∪ Vg
    let S' = S ∪ {(f1,Vf1),...,(fk,Vfk)}
        fe = map (parameterLiftDef S') {d1,...,dk}
        e'0 = parameterLiftExp S' e0
    in (LetRec fe e'0)

applySolutionToDef :: Set(FunName,Set(Variable)) → Def → Def
applySolutionToDef {...,(f,{v1, ..., vn}),...} (f ≡ λ(v'1, ..., v'n).e) =
    (f ≡ λ(v1, ..., vn, v'1, ..., v'n).e)
applySolutionToDef S d = d

applySolutionToExp :: Set(FunName,Set(Variable)) → Exp → Exp
applySolutionToExp {...,(f,{v1, ..., vn}),...} (f (e1, ..., en)) =
    (f (v1, ..., vn, e1, ..., en))
applySolutionToExp S e = e
```

Figure 5: Parameter lifting – free variables are made parameters.

Esempio

Adesso prendiamo in esame un programma scritto in linguaggio funzionale multiparadigma *Scheme*. Il mio obbiettivo è quello di mostrare come si comporta l'algoritmo descritto in precedenza e come trasformerà il programma .

```
(define main
  (lambda (i1 j1 k1)
    (letrec ([f2 (lambda (j2 k2)
                   (if (< k2 100)
                       (letrec ([f7 (lambda (j4 k4)
                                      (f2 j4 k4))])
                         (if (< j2 20)
                             (f7 i1 (+ k2 1))
                             (f7 k2 (+ k2 1))))
                       j2))])
      (f2 j1 k1))))
;(main 1 1 0)
```

Figure 7: A textbook example.

Nel programma le due funzioni f1 e f2 sono mutuamente ricorsive , poiché si riferiscono ricorsivamente l'una con l'altra e sono localizzate nel corpo della funzione principale *main*.

Il parametro formale i1 occorre libero nei corpi di f1 e f2.

L'algoritmo di lifting dei parametri agisce sul programma applicando la funzione "parameterLiftProgram" a *main* :

parameterLiftDProgram {define main…} : Applichiamo "parameterLiftDef" a *main* con un set vuoto di soluzioni (S= Ø).

parameteLiftDefØ (define main…) : scendiamo ricorsivamente nel corpo della funzione.

parameterLiftExpØ (letrec[f2…] …) : creiamo due insiemi V_{f2} ={i1} F_{f2}=Ø, che rimarranno invariati durante la prima iterazione a punto fisso. Estendiamo l'insieme vuoto di soluzioni con (f2,{i1}) e poi continuiamo ricorsivamente su f2 e sul corpo della letrec.

parameterLiftDef S [f2(lambda (j2 k2)…)] : l'insieme di soluzioni S={(f2,{i1})} fa in modo che la lista dei parametri formali di f2 viene ampliata con (i1 j2 k2).

parameterLiftExp S (letrec ([f7…]) …) : creiamo due insiemi V_{f7}=Ø e F_{f7}={f2} e poiché f2 è descritta dall'insieme di soluzioni S=(f2,{i1}) estendiamo V_{f7} a V_{f7}={i1}.Il set rimane invariato durante l'intera iterazione a punto fisso. Estendiamo quindi il set di soluzioni con (f7,{i1} e continuiamo ricorsivamente su f7 e sul corpo della letrec.

parameterLiftDef S [f7 (lambda (j4 k4) …)]: Il set di soluzioni S={(f7,{i1}), …} ci induce ad espandere la lista dei parametri formali di f7 a (i1 j4 k4).

parameterLiftExp S (f2 j4 k4) : il set di soluzioni S={(f2,{i1}), …} ci porta ad inserire i1 come primo argomento di f2.

parameterLiftExp S (f7 i1 (…)) : il set di soluzioni S={(f7,{i1}), …} ci porta ad inserire i1 come primo argomento di f7.

parameterLiftExp S (f7 k2 (…)) : il set di soluzioni S={(f7,{i1}), …} ci porta ad inserire i1 come primo argomento di f7.

parameterLiftExp S (f2 j1 k1) : il set di soluzioni S={(f2,{i1}), …} ci porta ad inserire i1 come primo argomento di f2.

Il risultato dell'applicazione di tale algoritmo è :

```
(define main
  (lambda (i1 j1 k1)
    (letrec ([f2 (lambda (x1 j2 k2)
                   (if (< k2 100)
                       (letrec ([f7 (lambda (y1 j4 k4)
                                      (f2 y1 j4 k4))])
                         (if (< j2 20)
                             (f7 x1 x1 (+ k2 1))
                             (f7 x1 k2 (+ k2 1))))
                       j2))])
      (f2 i1 j1 k1))))
;(main 1 1 0)
```

Figure 8: The program of Figure 7, after parameter lifting.

Algoritmo di emersione dei blocchi

Adesso descriverò la parte di emersione dei blocchi dell'algoritmo di lambda lifting.

Solitamente l'emersione delle funzioni al di fuori dei blocchi , è strettamente correlata all'ambito di visibilità dei parametri formali e dei nomi delle funzioni dagli utenti.

Tuttavia , un programma che ha già subito lifting dei parametri , è scope-insensitive , ossia nessuna funzione è definita nello scope dei parametri formali di qualche altra funzione. Inoltre , una funzione globale è visibile a tutte le altre funzioni globali del programma , così i riferimenti ai nomi delle funzioni sono facilmente risolti.

In questo algoritmo non si fa altro che analizzare tutto il programma , partendo dalle definizioni per arrivare in modo ricorsivo ai letterali e alle variabili ; tutte le funzioni locali verranno riunite in un insieme (F_{new}) attraverso la funzione "blockFloatExp" e tutti i blocchi saranno sostituiti dai loro corpi .

Le funzioni dell'insieme F_{new} saranno, poi aggiunte al programma come funzioni globali.

Il risultato sarà :

```
blockFloatProgram :: Program → Program
blockFloatProgram p = foldr makeUnion ∅ (map blockFloatDef p)

blockFloatDef :: Def → (Set(Def),Def)
blockFloatDef (f ≡ λ(v1, ..., vn).e) = let (Fnew,e') = blockFloatExp e
                                       in (Fnew, f ≡ λ(v1, ..., vn).e')

blockFloatExp :: Exp → (Set(Def),Exp)
blockFloatExp (ℓ) = (∅,ℓ)
blockFloatExp (v) = (∅,v)
blockFloatExp (f (e1, ..., en)) = let x = map blockFloatExp (e1, ..., en)
                                      Fnew= foldr (∪) ∅ (map fst x)
                                      (e'1, ..., e'n) = map snd x
                                  in (Fnew,f (e'1, ..., e'n))
blockFloatExp (LetRec {d1, ..., dk} e0) =
                 let x = map blockFloatDef {d1, ..., dk}
                     b = blockFloatExp e0
                     Fnew = foldr makeUnion ∅ x
                 in (Fnew ∪ (fst b),snd b)

makeUnion :: (Set(Def),Def) → Set(Def) → Set(Def)
makeUnion (Fnew, d) S = Fnew ∪ {d} ∪ S
```

Figure 6: Block floating – flattening of block structure.

Esempio

Torniamo alla versione con lifting dei parametri del nostro programma con funzioni mutuamente ricorsive.

```
(define main
  (lambda (i1 j1 k1)
    (letrec ([f2 (lambda (x1 j2 k2)
                   (if (< k2 100)
                       (letrec ([f7 (lambda (y1 j4 k4)
                                      (f2 y1 j4 k4))])
                         (if (< j2 20)
                             (f7 x1 x1 (+ k2 1))
                             (f7 x1 k2 (+ k2 1))))
                       j2))])
      (f2 i1 j1 k1))))
;(main 1 1 0)
```

Figure 8: The program of Figure 7, after parameter lifting.

Vediamo come si comporta l'algoritmo su questo programma :

blockFloatProgram {main…} : applichiamo la funzione "blockFloatDef" alla definizione di main.

blockFloatDef(main) : crea la coppia (F_{new}, e') dove f_{new} sarà l'insieme costituito dalle definizioni delle funzioni locali e l'espressione e' sarà ottenuta attraverso l'applicazione della funzione "blockFloatedExp". Inizialmente F_{new}=Ø.

blockFloatExp(letrec[f2…]): l'insieme delle funzioni sarà ampliato con la funzione f2 (cioè F_{new}={f2}) , mentre e' sarà rappresentata da " ….letrec[f7] ". Quindi scendiamo nel corpo della prossima letrec.

blockFloatExp(letrec[f7]) : F_{new} viene ampliato con la definizione di f7 , quindi F_{new}={f2,f7}

A questo punto andiamo a sostituire le soluzioni della funzione "blockFloatDef" in "blockFloatProgram". Si otterrà che :

blockFloatProgram ({main})=foldr makeUnion Ø (({f1}, ...),({f1,f2}, ...))=

= makeUnion(({f1}, ...) , makeUnion(({f1,f2}), ...))=makeUnion(({f1}, ...) , ({f1,f2}), ...))= { main , f1 , f2}

Quindi il programma risultante sarà caratterizzato dalle funzioni globali main , f1 , f2 e non avrà blocchi locali :

```
(define main
  (lambda (i1 j1 k1)
    (f2 i1 j1 k1)))

(define f2
  (lambda (x1 j2 k2)
    (if (< k2 100)
        (if (< j2 20)
            (f7 x1 x1 (+ k2 1))
            (f7 x1 k2 (+ k2 1)))
        j2)))

(define f7
  (lambda (y1 j4 k4)
    (f2 y1 j4 k4)))

;(main 1 1 0)
```

Figure 9: The program of Figure 8, after block floating.

Inversione del lambda lifting

Come abbiamo visto il lambda-lifting prima rende le funzioni scope-insensitive espandendone la lista dei parametri formali, poi procede a rendere tutte le funzioni globali attraverso il processo di emersione dei blocchi.

Un modo per ottenere il processo inverso del lambda-lifting è quello di rendere locali le appropriate funzioni locali , poi renderle scope-sensitive riducendo la loro lista di parametri formali.

Affondamento dei blocchi

Per invertire l'effetto del lambda-lifting esaminiamo il programma precedente.

La funzione principale del programma è *main*, le altre due funzioni sono solo usate dalla funzione principale e quindi sono localizzabili a *main*. Quindi sostituiremo il corpo di *main* con un blocco che contiene queste funzioni e che ha il corpo originale di *main* come suo corpo.

```
define main=letrec f2 = …..
                   f7=…..
               in ….
```

Ma come possiamo vedere il corpo di f2 riferisce la funzione f7.La funzione *main* non usa f7 e questo sta a significare che f7 può essere localizzata in f2.

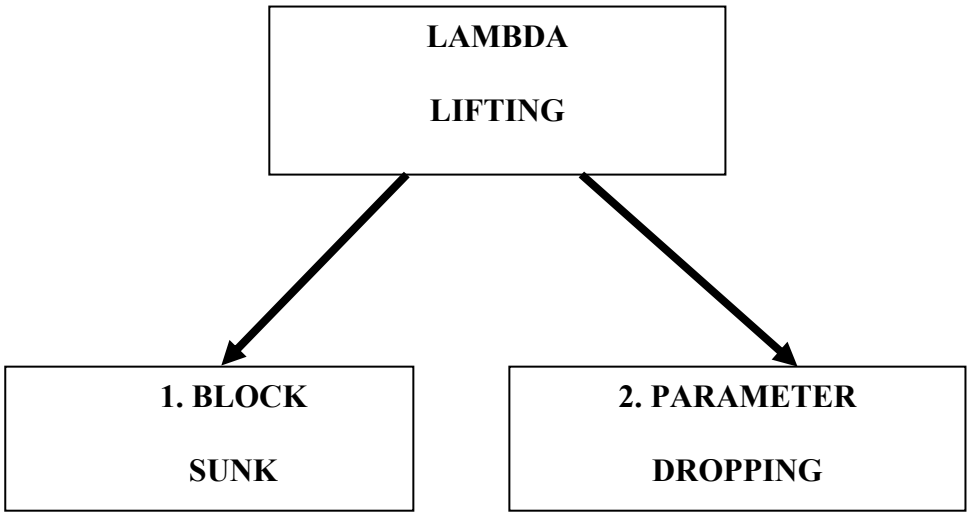
```
define r=letrec f2=letrec f7=….
                          in….
                in….
```

2.2.3 Le basi del Lambda Dropping

Adesso specificheremo il lambda-dropping in maniera più formale. Fare lambda-dropping su un programma significa ridurre al minimo il passaggio dei parametri ; lambda-dropping è spesso usato come processo inverso del lambda lifting.

Le definizioni delle funzioni sono localizzate in una struttura a blocchi con scope lessicale. I parametri che diventano ridondandi a causa della creazione di questo nuovo scope vengono eliminati.

Il processo di lambda dropping è costituito da due trasformazioni :



In questa parte si fa largo uso dei grafi di cui adesso diamo una definizione.

Cenni sui grafi

Un *grafo di chiamata* (anche conosciuto con il nome di multigrafo) è un grafo diretto che rappresenta le relazioni di chiamata tra i sottoprogrammi di un programma. A seconda dell'algoritmo e del linguaggio di programmazione usati , il grafo di chiamata può supportare o non supportare la ricorsione. Un grafo di chiamata , di un

programma che non usa la ricorsione , è definito grafo diretto aciclico

.

I grafi si possono distinguere in dipendenti da contesto e liberi da contesto. Nei grafi liberi da contesto ogni funzione è rappresentata da un nodo e ogni arco orientato rappresenta tutte le possibili chiamate tra i nodi. Nel caso dei grafi dipendenti da contesto si prendono in considerazione anche i parametri di ogni chiamata della funzione .

Un *grafo di flusso di controllo* è una rappresentazione , attraverso la notazione dei grafi , di tutti i percorsi che un programma può attraversare durante la sua esecuzione. Ogni nodo nel grafo rappresenta un blocco base , ossia un porzione di programma senza alcuna istruzione di salto. Gli archi diretti sono usati per rappresentare i salti nel flusso di controllo. Un grafo di flusso di controllo ha due nodi principali : un *blocco di entrata* attraverso il quale il controllo entra nel grafo di flusso e un *blocco di uscita* attraverso il quale il controllo termina.

Definizione : si definisce *albero dei dominatori* di un grafo , un albero in cui un nodo *a* precede un nodo *b* se *a* domina *b* nel grafo.

Definizione : in un grafo un nodo *a domina* un nodo *b,* se *a* è presente in qualsiasi percorso che dalla radice porta a *b*.

La descrizione degli algoritmi di lambda dropping fa largo uso delle funzioni standard per la manipolazione dei grafi. Per creare il grafo di

flusso di un programma del prim'ordine si può usare un algoritmo di questo tipo :

```
Graph.addNode :: Graph(α) → α → α
Graph.addNode (G as (V, E)) a = V := V ∪ {a}; a

Graph.addEdge :: Graph(α) → (α, α) → (α, α)
Graph.addEdge (G as (V, E)) (a, b) =
    if a ∉ V then V := V ∪ {a};
    if b ∉ V then V := V ∪ {b};
    if (a, b) ∉ E then E := E ∪ {(a, b)};
    (a, b)

Graph.findDominators :: (Graph(α), α) → (Graph(α), α)
Graph.findDominators (G, r) : Returns the dominator tree of G
                              [3]. In the dominator tree, node b
                              is a successor of node a if and only
                              if all paths from r to b go through
                              a.

Graph.flowGraph :: Program → (Graph(DefNode), DefNode)
Graph.flowGraph P : Returns the flowgraph of P [3]. The flow-
                    graph has an edge from node f_a to node g_b
                    iff g is invoked from within f, binding the
                    formal argument b of g to a, where a is a
                    formal parameter of f.

Graph.isPath :: Graph(α) → α → α → Bool
Graph.ispath G a b : true if there is a path from a to b in G.
```

Dove le funzioni "Graph.addNode" e "Graph.addEdge" servono rispettivamente ad aggiungere un nodo e ad inserire un arco orientato nel grafo .

Affondamento nei blocchi

Un programma strutturato si dice *completamente affondato* quando nessuna sua funzione può essere definita localmente ad una definizione nella quale è usata. Nessuna funzione che è usata nel corpo di un blocco può essere usata da una funzione definita nello stesso blocco. La sintassi dei programmi BS (Block Sunk) è la seguente :

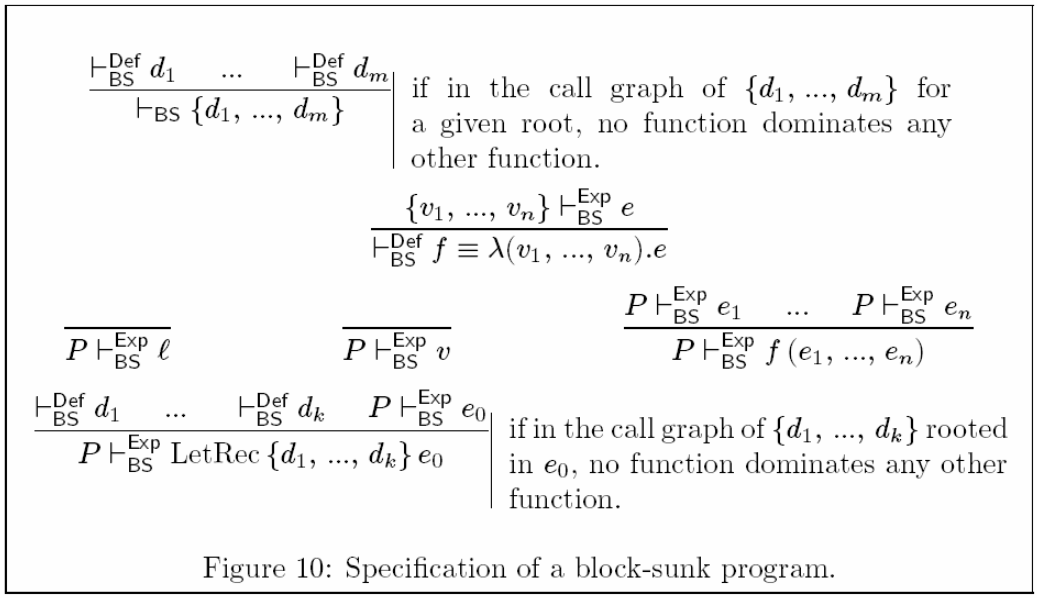


Figure 10: Specification of a block-sunk program.

Ossia :

- Un programma BS , è un insieme ($\{d_1, \ldots\ldots, d_k\}$) , ottenuto a partire da una serie di definizioni , a loro volta Block Sunk , se e solo se nel grafo di chiamata di $\{d_1, \ldots\ldots, d_k\}$, nessuna funzione domina un'altra funzione.
- Se abbiamo un'espressione *e* affondata in un blocco e quest'espressione è definita sull'insieme di variabili $\{v_1, \ldots\ldots, v_n\}$, allora nel programma BS tale espressione

comparirà in una lambda astrazione avente come parametri proprio $v_1, \ldots\ldots\ldots, v_n$.

- Possiamo avere nel programma una funzione del tipo $f(e_1, \ldots\ldots\ldots e_n)$, se $e_1, \ldots\ldots\ldots e_n$ sono block sunk e aventi parametri formali nell'insieme P.
- Date le definizioni BS $d_1, \ldots\ldots\ldots, d_k$ e l'espressione e_0 anch'essa BS possiamo avere nel nostro programma una definizione locale (LetRec$\{ d_1, \ldots\ldots\ldots, d_k \} e_0$) ; purchè nel grafo di chiamata , di radice e_0 , del programma , nessuna funzione domini un'altra.

Quindi giungiamo alla conclusione che un programma p , una definizione d e un'espressione e, i cui parametri sono denotati da P , sono affondate , se soddisfatte , rispettivamente , le seguenti condizioni :

$$\vdash_{\mathsf{BS}} p \qquad \vdash^{\mathsf{Def}}_{\mathsf{BS}} d \qquad P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} e$$

Dropping dei parametri

In un programma ogni parametro formale di una funzione , che ad ogni chiamata della stessa è sempre legato ad una variabile usata nel

corpo della funzione stessa , è considerato ridondante e quindi può essere rimosso.

Su questo principio si basa il processo di *dropping dei parametri.*

Un programma è completamente *parameter-dropped ,* quando nessuna variabile visibile nel corpo di una funzione è passata come parametro della stessa funzione.

La figura seguente definisce formalmente la sintassi dei programmi *parameter-dropped* :

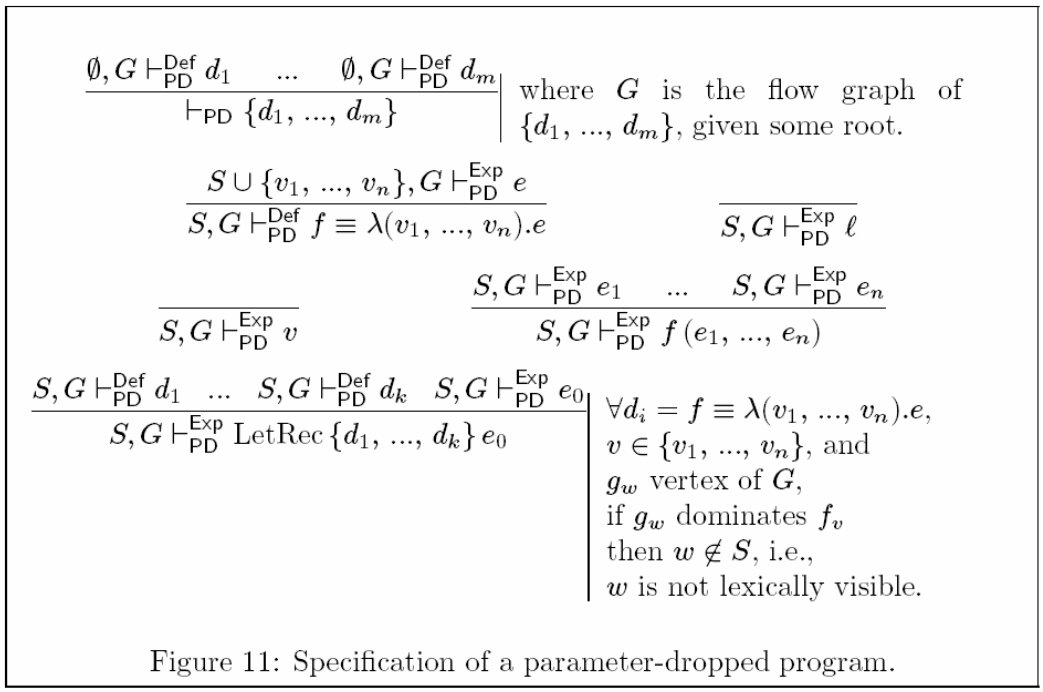


Figure 11: Specification of a parameter-dropped program.

Vale a dire che :

- Un programma PD (parameter-dropped) è definito come un'insieme di definizioni basate su un set vuoto di variabili e

derivate dal grafo di chiamata dello stesso insieme di definizioni.

- Un'espressione *e* che occorre nello scope delle variabili $v_1, \ldots\ldots\ldots, v_n$ e delle variabili contenute nel set S , allora *e* può essere scritta come una lambda astrazione avente come parametri $v_1, \ldots\ldots\ldots, v_n$.
- La funzione $f(e_1, \ldots\ldots\ldots e_n)$ è PD se lo sono anche le espressioni $e_1, \ldots\ldots\ldots e_n$ appartenenti allo scope delle variabili di S e al grafo di chiamata del programma.
- Se abbiamo k definizioni PD in S e G e un'espressione e_0 PD in S e G allora possiamo inserire nel programma la definizione locale (LetRec$\{ d_1, \ldots\ldots\ldots, d_k \} e_0$).

In generale quindi un programma *p* , una definizione *d* e una espressione *e* , il cui scope si estende allo scope delle variabili contenute nel set S , sono completamente parameter dropped se e solo se sono verificate , rispettivamente ,le seguenti condizioni :

$$\vdash_{\mathrm{PD}} p \qquad S, G \vdash^{\mathrm{Def}}_{\mathrm{PD}} d \qquad S, G \vdash^{\mathrm{Exp}}_{\mathrm{PD}} e$$

2.2.4 Gli algoritmi del lambda dropping

L´algoritmo del lambda-dropping funziona su qualsiasi tipo di programma ma si capisce forse meglio se operante su programmi con lambda-lifting. L'algoritmo è composto di due stadi: la caduta dei *blocchi* e il *dropping dei parametri*.

La parte dell'algoritmo basata sull'affondamento dei blocchi consiste dei seguenti passi:

Si forniscono due punti di entrata. Il primo mantiene la funzione "*main*" e ogni funzione che non è usata come funzione globale, e l´altro permette al chiamante di specificare un set di funzioni da mantenere come globali oltre alla funzione "*main*" di default.

La funzione "blockSinkProgram2" applicata al programma e ad un set di funzioni globali fa sì che si costruisca il grafo di chiamata del programma (attraverso la funzione "buildCallGraph") , in questo modo : crea un grafo vuoto con radice "*root*" ; per ogni funzione *f* del programma e per ogni funzione *f'* che occorre libera nel corpo di *f* , si aggiungono nel grafo due nodi *d* e *d'* corrispondenti rispettivamente a *f* e *f'* , legati da una corrispondenza diretta ; ogni funzione globale è rappresentata , nel grafo , da un nodo collegato direttamente alla radice.

Si costruisce l'albero dei dominatori del grafo di chiamata e al suo interno ogni funzione *g* dominata da un'altra funzione *f* sarà il suo successore.

L'albero dei dominatori induce così una nuova struttura a blocchi nel programma . Avremo così un nuovo programma , costruito dalla funzione "buildProgram" , nel quale ogni funzione sarà dichiarata localmente al suo predecessore nell'albero dei dominatori.

Il risultato è :

```
blockSinkProgram :: Program → Program
blockSinkProgram p = blockSinkProgram2 p {main}

blockSinkProgram2 :: Program → Set(FunName) → Program
blockSinkProgram2 p globalFns =
   let buildCallGraph :: () → (Graph(Def),Def)
       buildCallGraph () =
          let (G as (V, E)) = ref (∅, ∅)
              root = Graph.addNode G "root"
          in foreach ((d as (f ≡ l)) ∈ p) do
                foreach f' ∈ (FF(d)\{f}) do
                   let (d' as (f' ≡ l')) ∈ p in Graph.addEdge G (d, d')
             foreach f ∈ globalFns do
                let (d as (f ≡ l)) ∈ p in Graph.addEdge G (root, d)
             foreach d ∈ V do
                if (∀d' ∈ V : (d', d) ∉ E) then Graph.addEdge G (root, d)
             (G, root)
       buildProgram :: (Graph(Def),Def) → Program
       buildProgram (G as (V, E), root) =
          let succ :: Def → Set(Def)
              succ d = {d' ∈ V|(d, d') ∈ E}
              build :: Def → Def
              build (d as f ≡ λ(v1, ..., vn).e) =
                    let S = map build (succ d)
                    in if S = ∅
                       then d
                       else (f ≡ λ(v1, ..., vn).(LetRec S e))
          in map build (succ root)
   in buildProgram (Graph.findDominators (buildCallGraph ()))
```

Figure 12: Block sinking – re-creation of block structure.

Per fare invece il dropping dei parametri del programma ci avvaliamo del grafo del flusso di controllo del programma.

Se nel grafo di flusso di un programma , un parametro formale *w* di una funzione *g* è dominato da qualche parametro formale *v* di una funzione *f* , allora *w* denoterà sempre il valore *v* . Se *v* , inoltre , è visibile nella definizione di *g* , allora *w* potrà essere rimosso dalla lista

dei parametri formali di *g* e sarà sostituito da *v* nel corpo della funzione.

La funzione "processGlobalDef" è invocata su una funzione globale, e procede in tre stadi, prima marcando i parametri formali per la rimozione, poi gli argomenti corrispondenti e infine rimuovendoli.

La funzione "markFormalsDef" è usata per visitare il programma e marcare quei parametri formali che possono subire dropping.

L´invocazione di "markFormalsDef" su una funzione i cui parametri formali sono dominati solo dal nodo radicale restituisce sempre gli stessi parametri (funzione identità), poiché queste variabili non possono essere state rese ridondanti da altre variabili. Per questo motivo descriviamo solo le invocazioni di questa funzione su quelle funzioni i cui parametri formali hanno un predecessore che non è il nodo radice.

La funzione "markArgumentsDef" è usata per visitare il programma e marcare quegli argomenti di funzioni che corrispondono a parametri formali che sono stati marcati come rimuovibili.

Infine la funzione "removeMarkedPartsDef" è usata per rimuovere tutte le parti marcate del programma, sia come parametri formali che come parametri effettivi.

```
parameterDropProgram :: Program → Program
parameterDropProgram p =
  let (G as (V,E),r) = Graph.findDominators (Graph.flowGraph p)
      processGlobalDef :: Def → Def
      processGlobalDef =   removeMarkedPartsDef
                         ∘ (markArgumentsDef ∅)
                         ∘ (markFormalsDef ∅)
      markFormalsDef :: Set(Variable) → Def → Def
      markFormalsDef S (f ≡ λ(v1, ..., vn).e) =
        let markFormalsExp :: Set(Variable) → Exp → Exp
            markFormalsExp S e =
              ... descend recursively, calling markFormalsDef on definitions...
            mark :: Variable → (Exp, List(Variable)) → (Exp, List(Variable))
            mark v (e,s) =
              if ∃g_w ∈ V :  (Graph.isPath G g_w f_v) ∧ w ∈ S
                           ∧((r,g_w) ∈ E ∨ ∃h_x ∈ V : (h_x,g_w) ∈ E ∧ x ∉ S)
              then (e[v/w], ↯ :: s)
              else (e, v :: s)
            (e',s') = foldr mark (e,[]) (v1, ..., vn)
        in (f ≡ λs'.(markFormalsExp (S ∪ {v1, ..., vn}) e'))
      markArgumentsDef :: Set(Def) → Def → Def
      markArgumentsDef S (f ≡ λ(v1, ..., vn).e) =
        let markArgumentsExp :: Set(Def) → Exp → Exp
            markArgumentsExp S (ℓ) = ℓ
            markArgumentsExp S (v) = v
            markArgumentsExp S (f (e1, ..., en)) =
              let (f ≡ λ(v1, ..., vn).e) ∈ S
                  mark :: (Variable, Exp) → Exp
                  mark (↯, e) = ↯
                  mark (v, e) = markArgumentsExp S e
              in (f (map mark (zip (v1, ..., vn) (e1, ..., en))))
            markArgumentsExp S (LetRec B e) =
              LetRec {map (markArgumentsDef (S ∪ B)) B}
                     (markArgumentsExp (S ∪ B) e)
            body = markArgumentsExp (S ∪ {f ≡ λ(v1, ..., vn).e}) e
        in (f ≡ λ(v1, ..., vn).body)
      removeMarkedPartsDef :: Def→Def
      removeMarkedPartsDef d =
        ...descend recursively, removing all marked parts of the definition ...
  in map processGlobalDef p
```

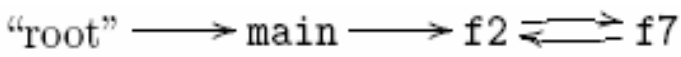
Figure 13: Parameter dropping – removing parameters.

Esempio

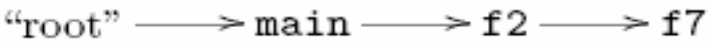
Operiamo adesso il lambda-dropping alla versione liftata del programma che implementa due funzioni mutuamente ricorsive .

Affondamento nei Blocchi

L´algoritmo di caduta dei blocchi è usato sul programma attraverso l'applicazione della funzione "blockSinkProgram" al set delle funzioni globali. L´applicazione di "buildCallGraph" costruisce il diagramma di chiamata del programma:



È facile computare l´albero dominatore di questo diagramma:



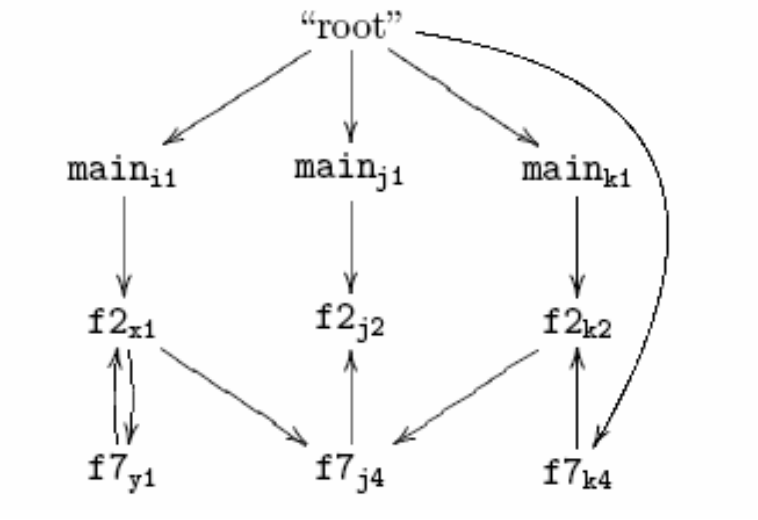
La funzione "buildProgram" costruisce un nuovo programma invocando "build" su main. Questo crea la definizione di main con il suo successore f2 come funzione locale. Similmente la funzione f2 è creata con f7 come funzione locale. Il risultato è :

```
(define main
  (lambda (i1 j1 k1)
    (letrec ([f2 (lambda (x1 j2 k2)
                   (if (< k2 100)
                       (letrec ([f7 (lambda (y1 j4 k4)
                                      (f2 y1 j4 k4))])
                         (if (< j2 20)
                             (f7 x1 x1 (+ k2 1))
                             (f7 x1 k2 (+ k2 1))))
                       j2))])
      (f2 i1 j1 k1))))
;(main 1 1 0)
```

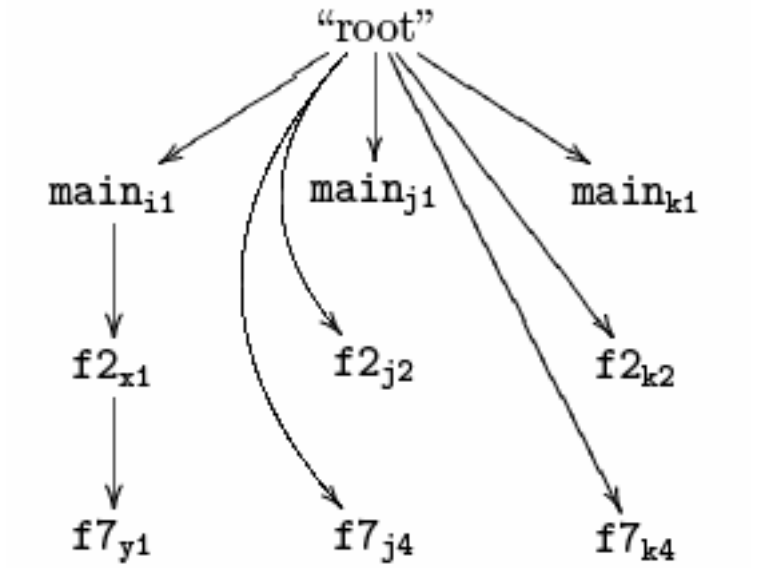
Figure 8: The program of Figure 7, after parameter lifting.

Dropping dei parametri

L´algoritmo di dropping di parametri è usato sul programma scope-insensitive precedente applicando la funzione "parameterDropProgram" alla funzione main. La funzione costruisce prima il diagramma di flusso del programma:



Dove la notazione f_x indica il parametro formale *x* della funzione *f*. L´albero dei dominatori di questo grafo è:



L´albero dei dominatori ci mostra che il solo parametro che è passato ad altre funzioni su ogni invocazione è i1, che può essere legato a x1 e y1. Tutti gli altri parametri sono successori diretti del nodo radice, il che significa che sono legati a differenti variabili nei loro punti di applicazione.

La funzione "markArgumentsDef" è usata per visitare il programma e marcare quegli argomenti di funzioni che corrispondono a parametri formali che sono stati marcati come rimuovibili. Così il primo argomento di f2 e f7 è rimosso in tutte le invocazioni. Infine la funzione "removeMarkedPartsDef" è usata per rimuovere tutte le parti marcate del programma, sia come parametri formali che come parametri effettivi. Il risultato è il programma :

markFormalsDef [f2 (lambda (x1 j2 k2) ...)]: ogni parametro formale è elaborato in sequenza, usando la funzione "mark". Il parametro formale x1 è dominato dal parametro formale i1, che è nel set *S* di parametri formali visibili. Poiché (i1, x1) è un ~~arc~~o nell´albero dei dominatori , x1 è rimpiazzato da x1 e i1 è sostituito a x1 in tutto il corpo di f2. Le altre due variabili sono dominate solo dal nodo radicale e non rivestono quindi alcun interesse.

markFormalsDef [f7 (lambda (y1 j2 k2) ...)]: ancora, ogni parametro formale è elaborato in successione, usando la funzione "mark". Il parametro formale y1 è dominato dai parametri formali i1 e x1, entrambi i quali sono nel set *S* di parametri formali visibili . Comunque, la variabile i1 è un successore diretto della radice, così sarà usata come rimpiazzo. La variabile y1 è rimpiazzata da ~~y1~~ e i1 è

sostituito a y1 in tutto il corpo di f7. Ancora, le altre due variabili sono dominate solo dal nodo radice, e quindi non rivestono alcun interesse .

```
(define main
  (lambda (i1 j1 k1)
    (letrec ([f2 (lambda (j2 k2)
                   (if (< k2 100)
                       (letrec ([f7 (lambda (j4 k4)
                                      (f2 j4 k4))])
                         (if (< j2 20)
                             (f7 i1 (+ k2 1))
                             (f7 k2 (+ k2 1))))
                       j2))])
      (f2 j1 k1))))
;(main 1 1 0)
```

Figure 7: A textbook example.

CAPITOLO 3

Programmi di ordine superiore

3.1 Introduzione

Un potente meccanismo spesso utilizzato nella programmazione funzionale è quello delle *funzioni di ordine superiore* (o *funzioni higher-order*). Una funzione si dice di ordine superiore quando può prendere altre funzioni come parametri e/o restituire funzioni come risultato. L'operatore differenziale in matematica è un esempio di funzione che mappa una funzione ad un'altra funzione.

Le funzioni di ordine superiore erano studiate dalla teoria del lambda calcolo ben prima che la nozione di programmazione funzionale esistesse e sono presenti nel design di molti linguaggi di programmazione funzionali, quali lo Scheme e l'Haskell.

Più in generale si può dire che le funzioni di ordine superiore sono parte del linguaggio naturale. Per esempio, gli avverbi possono modificare i verbi (azioni) creando verbi derivati. Con lo stesso ragionamento si può dire che i verbi imperativi sono simili alle funzioni "ordinarie" dei linguaggi di programmazione.

Le funzioni di ordine superiore sono cruciali nel paradigma della programmazione a livello funzionale .

I programmi di ordine superiore possono subire lambda-lifting senza considerare le applicazioni di funzioni di ordine più elevato che sono state trasferite come parametri, allo stesso modo possono subire lambda-dropping considerando solo le applicazioni di funzioni di prim´ordine.

L´introduzione di funzioni di ordine più elevato nel nostro linguaggio quindi implica solo mutamenti leggeri nelle specifiche e negli algoritmi del lambda-lifting e del lambda-dropping.

Per programmi d´ordine più elevato generalizziamo la sintassi dei programmi di prim'ordine permettendo alle funzioni di apparire come argomenti e variabili da applicare come funzioni.

3.2 Lambda lifting

Un programma di ordine più elevato ha lifting di parametri totale quando nessuna funzione ha variabili libere. Un programma di ordine superiore ha emersione completa dei blocchi quando le sue funzioni non possono essere spostate ulteriormente all´esterno della struttura a blocchi.

Per operare lifting di parametri su un programma di ordine più alto può essere usato un algoritmo quasi identico a quello per programmi di prim´ordine. Nei programmi di prim´ordine e di ordine superiore le

funzioni possono avere variabili libere. Comunque in un programma d´ordine più elevato una funzione può essere passata come argomento e applicata altrove con un nome diverso,ciò ci permette di creare funzioni di *Curry* ogni volta che ci troviamo di fronte a funzioni di ordine *superiore*. L´approccio è così quello trasformare nella sua versione *curried* ogni funzione che ha variabili libere, facendo delle variabili libere i parametri formali della nuova lambda astrazione. Così riscriviamo una funzione *f* con variabili libere $\{v'_1,..., v'_n\}$ come segue:

$$f \equiv \lambda(v_1, ..., v_n).e \Rightarrow f \equiv \lambda(v'_1, ..., v'_n).\lambda(v_1, ..., v_n).e$$

Similmente ogni occorrenza del nome della funzione è rimpiazzata da un´applicazione alle sue variabili libere. Per la funzione *f*:

$$f \Rightarrow (f\,(v'_1, ..., v'_n))$$

Usando questa tecnica il lifting di parametri può essere realizzato con una versione leggermente modificata dell´algoritmo esistente. L´unica differenza sta nel modo in cui la definizione della funzione è estesa con le sue variabili libere e in cui le occorrenze dell´applicazione della funzione da espandere ,con le sue variabili libere, possono essere in qualsiasi espressione.

Per far emergere i blocchi di un programma d´ordine superiore scope-insensitive si può usare esattamente lo stesso algoritmo dei programmi di prim´ordine. Una funzione curried è considerata come un insieme ed è aggiunta all'insieme delle funzioni globali di cui sarà composto il programma block-floated.

Esempio

Prendiamo adesso in considerazione un programma che mappa[1] una funzione di ordine superiore in una lista. La funzione filter, che filtra i valori che non soddisfano alcuni predicati, è mappata su una lista di numeri.

Per operare il lifting di parametri del programma della figura 14 le variabili libere pred ed i devono essere trasferite a filter. Similmente la variabile libera f deve essere trasferita alla funzione loop. In entrambi i casi leghiamo queste variabili estendendo la dichiarazione di funzione con una lista corrente di parametri. Tutte le occorrenze di queste funzioni sono rimpiazzate da applicazioni delle funzioni sulle variabili con le quali la sua dichiarazione è stata estesa.

[1] "map" è una funzione che ha come argomenti una funzione f e una lista xs e restituisce una lista i cui elementi sono ottenuti dall'applicazione della funzione f a tutti gli elementi della lista xs

Questo rende il programma scope-insensitive e ci permette il consueto passaggio di emersione dei blocchi.

```
(define main
  (lambda (pred i ls)
    (letrec ([filter (lambda (j)
                       (if (pred j) j i))]
             [map (lambda (f xs)
                    (letrec ([loop (lambda (s)
                                     (if (null? s)
                                         '()
                                         (cons (f (car s))
                                               (loop (cdr s))))))])
                      (loop xs)))])
      (map filter ls))))
```

Figure 14: Example involving a higher-order function.

```
(define main
  (lambda (pred i ls)
    (letrec ([filter (lambda (pred i)
                       (lambda (j)
                         (if (pred j) j i)))]
             [map (lambda (f xs)
                    (letrec ([loop (lambda (f)
                                     (lambda (s)
                                       (if (null? s)
                                           '()
                                           (cons (f (car s))
                                                 ((loop f)
                                                  (cdr s))))))])
                      ((loop f) xs)))])
      (map (filter pred i) ls))))
```

Figure 15: The program of Figure 14, after parameter-lifting.

```
(define main
  (lambda (pred i ls)
    (map (filter pred i) ls)))

(define filter
  (lambda (pred i)
    (lambda (j)
      (if (pred j) j i))))

(define map
  (lambda (f xs)
    ((loop f) xs)))

(define loop
  (lambda (f)
    (lambda (s)
      (if (null? s)
          '()
          (cons (f (car s))
                ((loop f) (cdr s)))))))
```

Figure 16: The program of Figure 15, after block-floating.

3.2.1 Inversione del lambda lifting

Come nel caso dei programmi di prim´ordine, l´inversione del lambda-lifting deve essere fatta prima ricreando la struttura a blocchi e poi rimuovendo i parametri formali ridondanti.

Le variabili libere in una funzione nel programma originale sono passate direttamente come argomenti della funzione. Comunque il lambda-lifting ha introdotto un livello extra di currying. Così una funzione curried senza argomenti, un "thunk", che è sempre applicata direttamente per ottenere il suo valore curried, dovrebbe essere "thawed" rimuovendo questo livello extra di currying.

3.3 Lambda dropping

Un programma d´ordine superiore ha affondamento totale dei blocchi quando i riferimenti ai nomi di funzione tra funzioni impediscono ulteriore localizzazione. Un programma di ordine superiore ha dropping completo di parametri quando nessun parametro che è passato come argomento diretto di una funzione è ridondante (cioè non deve appartenere allo scope di una funzione che domina , nel grafo , di chiamata la funzione nella quale la variabile stessa è usata).

Per operare caduta dei blocchi di un programma di ordine superiore , si può usare l´algoritmo di caduta dei blocchi per i programmi del prim'ordine. L´unica differenza è concettuale : le funzioni libere ora non rappresentano necessariamente chiamate , ma piuttosto dipendenze tra funzioni, riguardanti l'ambito di visibilità. Così , piuttosto che iniziare costruendo il grafico di chiamata del programma, procediamo esattamente nello stesso modo e costruiamo il grafico di dipendenza del programma.

Per operare il dropping di parametri di un programma di ordine superiore si può usare un algoritmo quasi identico a quello per i programmi di prim´ordine. Nei programmi di prim´ordine e di ordine superiore le sole variabili che consideriamo come ridondanti sono quelle passate come argomenti diretti di una funzione. Tuttavia in un programma di ordine

superiore , si potrebbero avere funzioni con nessun argomento , dopo aver droppato i parametri ridondanti.

Una funzione curried di nessun argomento che occorra solo come applicazione dovrebbe essere *uncurried*. Per una funzione *f* , che è sempre applicata, l´uncurrying è come segue:

$$f \equiv \lambda().\lambda(v_1, \ldots, v_n).e \Rightarrow f \equiv \lambda(v_1, \ldots, v_n).e$$

Similmente ogni occorrenza della funzione è un´applicazione, che è rimpiazzata dalla funzione stessa :

$$(f\,()) \Rightarrow f$$

Questa trasformazione di "**thawing**" dovrebbe essere l´ultimo stadio del dropping di parametri.

Le funzioni che sono passate come argomenti sono semplicemente ignorate, poiché non possono avere parametri ridondanti che siano passati ad esse direttamente.

Esempio rivisitato.

```
(define main
  (lambda (pred i ls)
    (map (filter pred i) ls)))

(define filter
  (lambda (pred i)
    (lambda (j)
      (if (pred j) j i))))

(define map
  (lambda (f xs)
    ((loop f) xs)))

(define loop
  (lambda (f)
    (lambda (s)
      (if (null? s)
          '()
          (cons (f (car s))
                ((loop f) (cdr s)))))))
```

Figure 16: The program of Figure 15, after block-floating.

Ricostruire il programma porta la struttura a blocchi del programma originale dopo il lifting di parametri che abbiamo effettuato nel precedente esempio.

```
(define main
  (lambda (pred i ls)
    (letrec ([filter (lambda (pred i)
                       (lambda (j)
                         (if (pred j) j i)))]
             [map (lambda (f xs)
                    (letrec ([loop (lambda (f)
                                     (lambda (s)
                                       (if (null? s)
                                           '()
                                           (cons (f (car s))
                                                 ((loop f)
                                                  (cdr s))))))])
                      ((loop f) xs)))])
      (map (filter pred i) ls))))
```

Figure 15: The program of Figure 14, after parameter-lifting.

Il dropping di parametri procede come nel caso di prim´ordine, ignorando ogni argomento curried di una funzione. Questo elimina i parametri formali pred ed i di filter, e f di loop.

Entrambe le funzioni hanno subito la rimozione dei programmi, così nessuna di loro è trasferita come argomento ad altre parti del programma. Così siamo sicuri che in ogni occorrenza di queste funzioni esse sono applicate a zero argomenti. Possiamo quindi rimuovere il currying vuoto nella definizione e nelle applicazioni di ciascuna di queste funzioni, producendo il programma :

```
(define main
  (lambda (pred i ls)
    (letrec ([filter (lambda (j)
                       (if (pred j) j i))]
             [map (lambda (f xs)
                    (letrec ([loop (lambda (s)
                                     (if (null? s)
                                         '()
                                         (cons (f (car s))
                                               (loop (cdr s)))))])
                      (loop xs)))])
      (map filter ls))))
```

Figure 14: Example involving a higher-order function.

CAPITOLO 4

Applicazioni

La nostra motivazione principale per vagliare il lambda lifting ed il lambda dropping è la *partial evaluation*.

4.1 Partial Evaluation

Una funzione ad un argomento può essere ottenuta a partire da una funzione a due argomenti specializzandola, cioè fissando un input per un valore particolare. In analisi questo è chiamato "*restrizione*" o "*proiezione*" di una funzione. La partial evaluation, tuttavia, prende in considerazione i programmi piuttosto che le funzioni.

Un ***partial evaluator*** (*valutatore parziale*) è un algoritmo che, dato un programma e alcuni dei suoi dati in input, produce un cosiddetto programma *residuo* o *specializzato*. Eseguendo il programma residuo con i restanti dati in input, si ottiene lo stesso risultato che si otterrebbe eseguendo il programma originale con tutti i dati in input.

La partial evaluation è una tecnica di trasformazione di programma, denominata anche ***program specialization***, il cui

obiettivo è quello di generare programmi efficienti da programmi generali in modo completamente automatico.

4.1.1 Principi della Partial Evalution

Si può produrre un nuovo programma a partire da uno gia dato e da un insieme di input,fissando alcuni di essi a dei particolari valori. Un *partial evaluator*, o *program specializator*, è uno strumento, completamente automatico, che trasforma un programma specializzandolo rispetto ad alcuni dei suoi dati in input, considerati come costanti, per i quali vengono forniti dei valori concreti, solitamente dall'utente.

Generalmente, il programma residuo generato dal partial evaluator è considerato più efficiente rispetto al programma iniziale, sia in termini di memoria che in termini di prestazioni. Infatti, utilizzando le informazioni relative alle *invarianti*, lo specializzatore è in grado di eseguire l'elaborazione dei dati conosciuti al momento della specializzazione, prima della esecuzione del programma, riducendo così la quantità di lavoro che deve essere effettuata a run-time dal programma residuo. In questo modo si ottiene una applicazione più veloce e più efficiente.

4.1.2 Partial Evaluation = Program Specialization

L'idea di specializzare i programmi è tutt'altro che nuova, infatti essa è stata formulata e provata circa quarant'anni fa' nel teorema s-m-n di Kleene ed è un fondamento della teoria delle funzioni ricorsive. Comunque la costruzione di Kleene dava dei programmi specializzati che erano più lenti degli originali. Quando un programma ha bisogno di più di un input, ed uno degli input varia più lentamente rispetto agli altri, allora la specializzazione del programma genera, rispetto a quell'input, un programma specializzato più veloce.

In figura 1 è rappresentato un partial evaluator nella sua forma classica, che identificheremo con "***mix***", che ha l'obiettivo di velocizzare l'esecuzione di un programma **p** con input **I = {in1,in2}**, dove **in1** è il sottoinsieme degli input conosciuti al momento della specializzazione (*input statico*), mentre **in2** è il sottoinsieme degli input che non sono disponibili fino al momento della esecuzione (*input dinamico*).

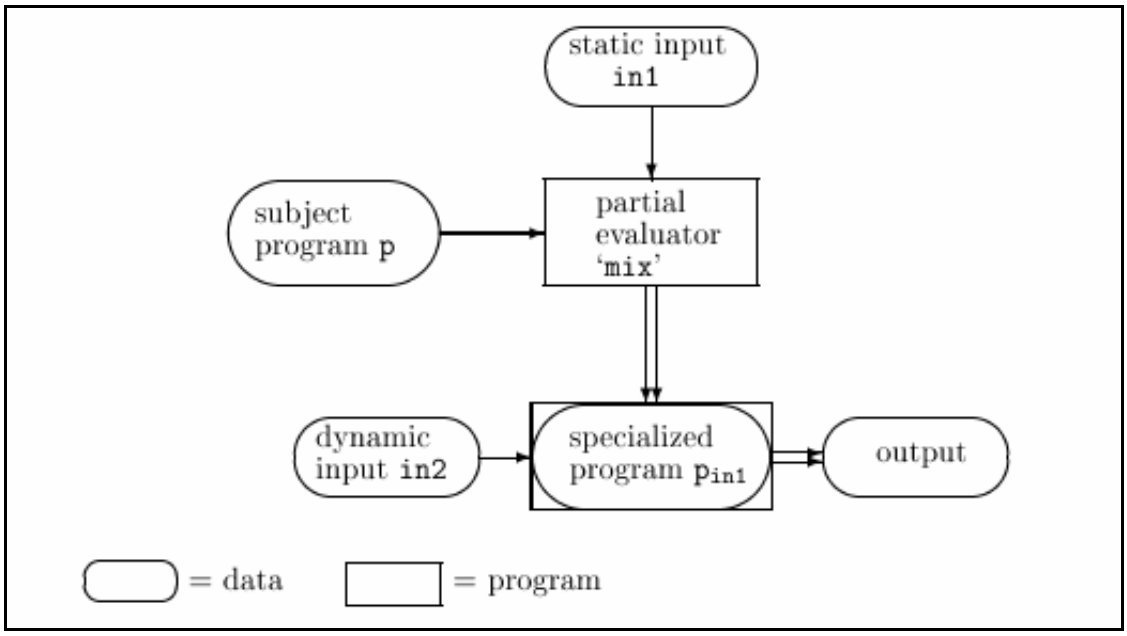


FIGURA 1: Il Partial Evaluator "*mix*"

Il partial evaluator ha in input il programma soggetto **p** con una parte dei dati in input **in1.** Il suo effetto è di costruire un nuovo programma $\mathbf{p_{in1}}$ detto ***residuo***. Quando a p_{in1} viene fornito il resto degli input **in2**, genera gli stessi risultati che avrebbe generato il programma p per l'input I = {in1,in2}, ma presumibilmente più velocemente.

Nel complesso il programma generale sarà più semplice ma meno efficiente delle versioni specializzate prodotte da un partial evaluator. In altre parole, quindi, un partial evaluator è proprio uno specializzatore di programma.

Possiamo notare nella figura che i cerchi contengono i dati, mentre i rettangoli contengono i programmi. Il programma specializzato p_{in1} inizialmente è considerato un dato, successivamente come codice, e

quindi è contenuto in entrambi. Inoltre le frecce singole indicano i dati in input al programma, e le frecce doppie gli output. Il partial evaluator ha due input, mentre p_{in1}, che rappresenta l'output di mix, ne ha soltanto uno.

ESEMPIO

Consideriamo una funzione a due variabili definita come segue

f(x, y) = if x = 0 then g(x) else h(y)

Supponiamo di specializzare f per il valore x = 2

f2(y) = f(2, y) = h(y)

f2 si ottiene "valutando" il corpo di f al tempo della specializzazione. Poiché il test del condizionale è sempre falso, si può rimpiazzare il condizionale con il suo ramo else. Si ottiene quindi che f2 è più efficiente di f per qualunque valore di y.

4.1.3 Perchè Partial Evaluation?

Speed-up della Partial Evaluation

La motivazione principale per adottare la partial evaluation e la velocità: il programma p_{in1} è spesso più veloce di p.

Cerchiamo di descrivere questo concetto più precisamente.

Per ogni p, d_1, . . . , d_n appartenente a D, sia t_p $(d_1, \ldots, d_n)$ il tempo impiegato per computare □p□[d_1 . . . d_n]. Ad esempio, t_p $(d_1, \ldots, d_n)$ può essere il numero di cicli-macchina per eseguire il codice di p in linguaggio macchina su un qualsiasi calcolatore.

In alcuni casi, la partial evaluation può dare buoni risultati in una singola unità di esecuzione, poiché l'esecuzione del programma specializzato più la sua generazione risulta essere più veloce della esecuzione del programma generale.

$$t_{mix}(p, in1) + t_{pin1}(in2) \leq t_p(in1, in2)$$

In generale la specializzazione è chiaramente più efficiente quando alcuni parametri del programma cambiano meno frequentemente. In questo caso sarà generato un nuovo programma generalizzato solo quando gli input selezionati per la specializzazione vengono modificati. Se ad esempio in2 cambia più frequentemente rispetto ad in1, ogni volta che in1 cambia si può costruire un nuovo p_{in1} specializzato, più veloce di p, e quindi farlo girare con vari in2 fino a che in1 cambi ancora.

4.2 Applicazione di lambda lifting e lambda dropping su un source program

Come gia detto , le equazioni ricorsive offrono un format conveniente di valutatore parziale. *Similix* e *Schism*, per esempio, sono due esempi di valutatori parziali che operano lambda-lifting su programmi sorgente prima della specializzazione e producono programmi residui in forma di equazioni ricorsive. Molto spesso, comunque, queste equazioni ricorsive sono afflitte da un numero massiccio di parametri, il che aumenta enormemente il loro tempo di compilazione, talvolta al punto di rendere tutto il processo di valutazione parziale impraticabile. Operiamo quindi il lambda-dropping dei programmi residui per snellire le loro funzioni residue. . Come beneficio collaterale, il lambda-dropping ricrea anche una struttura a blocchi che è spesso simile all´annidamento del programma sorgente corrispondente, incrementando così la leggibilità.

Il nostro lambda dropper opera sul linguaggio output di Schism , cioè su un sistema di partial evaluation per linguaggi applicativi di ordine superiore puri .

Esempio : la funzione fold

```
(define-type binary-tree
  (leaf alpha)
  (node left right))

(define binary-tree-fold
  (lambda (process-leaf process-node init)
    (letrec ([traverse (lambda (t)
                         (case-type t
                           [(leaf n)
                            (process-leaf n)]
                           [(node left right)
                            (process-node (traverse left)
                                          (traverse right))]))])
      (lambda (t) (init (traverse t))))))

(define main
  (lambda (t x y)
    ((binary-tree-fold (lambda (n) (leaf (* (+ x n) y)))
                       (lambda (r1 r2) (node r1 r2))
                       (lambda (x) x)) t)))
```

Figura 18 : programma sorgente

Il programma applica la funzione standard **`fold`** di Scheme su un albero binario.

L'albero binario è definito come una struttura costituita dagli elementi foglia (*leaf*) e nodo che può essere sinistro o destro (*left right*).

La funzione **`traverse`** è definita localmente alla funzione **`bynary-tree-fold`** e realizza la visita dell'albero , utilizzando come parametri le funzioni di ordine superiore : *process-leaf* , *process-node* e *init*, definite poi all'interno di **`main`**.

Dopo la specializzazione: la funzione **`traverse`** è adesso globale ; inoltre , senza alcun input statico , le astrazioni statiche sono propagate da Schism dalla funzione **`main`** alla funzione **`fold.`** Il programma residuo è composto da due equazioni ricorsive e i parametri dinamici *x* e *y* sono mantenuti e occorrono adesso come parametri residui. Essi compongono la funzione **`traversal`** e si candidano ovviamente (essendo parametri ridondanti) per il lambda dropping.

```
(define (main-1 t x y)
  (traverse:1-1 t y x))

(define (traverse:1-1 t y x)
  (casetype t
    [(leaf n)
     (leaf (* (+ x n) y))]
    [(node left right)
     (node (traverse:1-1 left y x)
           (traverse:1-1 right y x))]))
```

Figure 19: Specialized (lambda-lifted) version of Figure 18.

La corrispondente versione droppata di questo programma , che è ottenuta automaticamente , sarà :

```
(define (main-1 t x y)
  (letrec ([traverse:1-1 (lambda (t)
                           (case-type t
                             [(leaf n)
                              (leaf (* (+ x n) y))]
                             [(node left right)
                              (node (traverse:1-1 left)
                                    (traverse:1-1 right))]))])
    (traverse:1-1 t)))
```

Figure 20: Lambda-dropped version of Figure 19.

Le figure 19 e 20 ci mostrano i parametri dinamici *x* e *y* siano gli unici superstiti delle due astrazioni statiche nel programma sorgente. Si noti anche come risulta più chiaro confrontare i programmi in figura 18 e in figura 20 , anziché quelli in figura 18 e 19. la figura 20 ci mostra più chiaramente, infatti , come il blocco della letrec del programma sorgente sia stato localizzato e specializzato nella funzione **`main.`**

Esempio 2 : la proiezione di Futamura

Un ‘esempio particolarmente interessante di partial evaluator è la **proiezione di Futamura** : tecnica di specializzazione di un programma sorgente rappresentatati da un inteprete per un linguaggio di programmazione.

Se l’input statico è il programma sorgente che deve essere interpretato allora la partial evaluation produrrà una nuova versione del

programma oggetto , la quale costituirà una nuova versione dell'interprete specializzata ad eseguire solo quel programma.

Consideriamo adesso un linguaggio imperativo ,basato su cicli "WHILE" , com'è tradizione nella valutazione parziale e nei compilatori basati sulla semantica.

```
{
  int res=1; int n=4; int cnt=1;
  while (cnt > 0) { res = 1;
                    n = 4;
                    while (n > 0) { res = n * res;
                                    n = n - 1;
                    }
                    cnt = cnt - 1;
  }
}
```

Figura 21 : Esempio di programma imperativo

La figura mostra un programma C con molti cicli "WHILE".

La specializzazione del corrispondente interprete definizionale (non mostrato qui) usando lo Schism , rispetto a questo programma sorgente da il seguente programma residuo :

```
(define (evprogram-1 s)
  (evwhile-1
    (intupdate 2 1 (intupdate 1 4 (intupdate 0 1 s)))))

(define (evwhile-1 s)
  (if (gtint (fetchint 2 s) 0)
      (evwhile-2 (intupdate 1 4 (intupdate 0 1 s)))
      s))

(define (evwhile-2 s)
  (if (gtint (fetchint 1 s) 0)
      (let ([s-1 (intupdate 0
                            (mulint (fetchint 1 s) (fetchint 0 s))
                            s)])
        (evwhile-2 (intupdate 1 (subint (fetchint 1 s-1) 1) s-1)))
      (evwhile-1 (intupdate 2 (subint (fetchint 2 s) 1) s))))
```

Figure 22: Specialized (lambda-lifted) version of the definitional interpreter with respect to Figure 21.

Ogni ciclo *while* del programma sorgente genera , nel programma specializzato , un'equazione ricorsiva.

Mostriamo adesso la versione droppata del programma , che è ottenuta automaticamente :

```
(define (evprogram-1 s)
  (letrec ([evwhile-1
            (lambda (s)
              (letrec ([evwhile-2
                        (lambda (s)
                          (if (gtint (fetchint 1 s) 0)
                              (let ([s-1 (intupdate 0
                                                    (mulint
                                                      (fetchint 1 s)
                                                      (fetchint 0 s))
                                                    s)])
                                (evwhile-2
                                  (intupdate 1
                                             (subint (fetchint 1 s-1)
                                                     1)
                                             s-1)))
                              (evwhile-1
                                (intupdate 2
                                           (subint (fetchint 2 s) 1)
                                           s)))))])
                (if (gtint (fetchint 2 s) 0)
                    (evwhile-2 (intupdate 1 4 (intupdate 0 1 s)))
                    s)))])
    (evwhile-1 (intupdate 2 1 (intupdate 1 4 (intupdate 0 1 s))))))
```

Figure 23: Lambda-dropped version of Figure 22.

CAPITOLO 5

Conclusioni :

prospettive del lambda lifting e del lambda dropping

Il lambda-dropping è stato indirettamente citato in altri contesti come : SSA (Static Single Assignment), l'ottimizzazione dei compilatori e in relazione ad una simile trasformazione di struttura a blocchi.

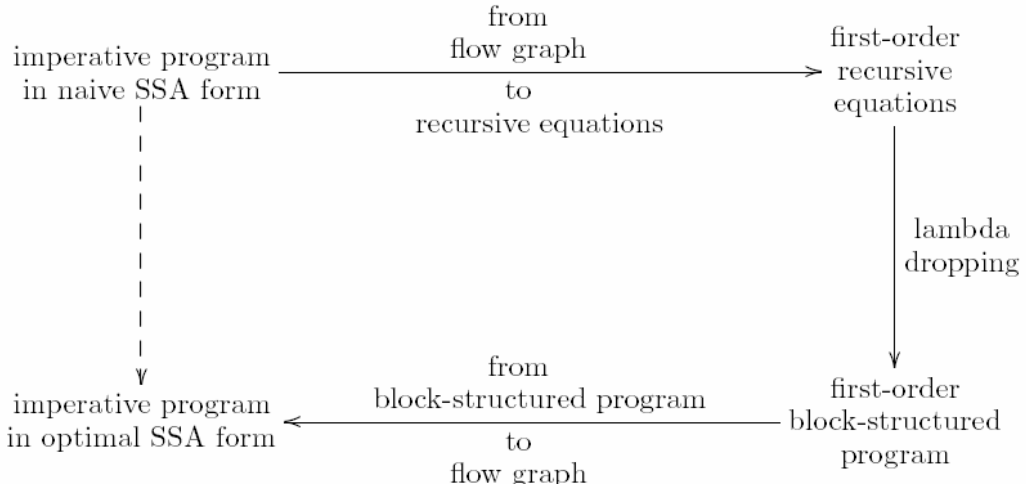
5.1 Forma SSA

Nella progettazione dei compilatori lo static single assignment,spesso abbreviato con SSA, è una rappresentazione intermedia(IR), in cui ogni variabile è assegnata solo a un singolo punto del programma. In presenza di espressioni che possono generare ambiguità nell'assegnazione di una variabile , ci si avvale delle *Φ-unzioni*che assegnano alla variabile un valore differente a seconda del ramo del grafo di flusso del programma da cui arriva il controllo.

Quindi , valori da rami differenti del grafo di flusso del programma sono uniti usando "Φ-nodi". Un algoritmo elementare per trasformare un programma imperativo in forma SSA crea Φ -nodi per tutte le variabili in tutti i punti di confluenza del programma. La minimizzazione del numero di Φ -nodi dà un programma in forma SSA ottimale.

Esistono diversi algoritmi per trasformare un programma in una forma SSA ottimale. Operano creando un albero dominatore per il grafo di flusso del programma. La conversione di un programma imperativo in forma SSA (rappresentata dal suo grafo di flusso) in forma funzionale crea un programma funzionale strutturato a blocchi, dove ciascuna funzione corrisponde a un blocco fondamentale. La conversione da una forma SSA elementare a una forma SSA ottimale corrisponde al dropping di parametri.

Un programma in forma SSA elementare che è tradotto nella sua controparte funzionale, attraverso il dropping di parametri e poi ritradotto in forma SSA, è una forma SSA ottimale. Allo stesso modo la traduzione dalla forma SSA a un programma con struttura a blocchi si fa usando l´albero dominatore del diagramma di flusso, il che corrisponde esattamente alla caduta dei blocchi.



L'uso di grafici dominatori ha semplificato la nostra presentazione complessiva. Dà anche un miglioramento sostanziale del lambda-dropping nella complessità temporale.

5.2 Ottimizzazione dei Compilatori

Peyton Jones, Partain e Santos ottimizzano i programmi usando l´emersione dei blocchi, la caduta dei blocchi e il dropping dei parametri nel compilatore Glasgow Haskell[2]. In particolare i blocchi non sono necessariamente fatti emergere sino al livello massimo. I programmi sorgente non sono più sistematicamente sottoposti a lambda-lifting.

La preoccupazione primaria è ottimizzare funzioni ricorsive singole (opposte a quelle mutuamente ricorsive). Piuttosto che operare dropping di parametri si da una definizione che fornisce lo scope che abilita il compilatore a effettuare dropping di parametri. Nel compilatore Glasgow

[2] Compilatore di istruzioni macchina per linguaggio funzionale Haskell.

Haskell l´ottimizzazione è stata poi giudicata di scarso effetto, mentre nell' ML standard l´ottimizzazione ha dato miglioramenti consistenti nel tempo di compilazione, nel tempo di esecuzione e consumo di spazio.

5.3 L'analisi di dipendenza di Peyton Jones

Nel suo manuale sull´implementazione dei linguaggi funzionali Peyton Jones usa un´analisi per generare struttura a blocchi da equazioni ricorsive il cui algoritmo è abbastanza simile a quello usato per la caduta dei blocchi,anche se il suo scopo è diverso.

Supponiamo di estenderela versione liftata del programma che implementa un textbox con un'espressione generale che chiama la funzione *main* con gli opportuni parametri .

L'analisi di dipendenza di Peyton-Jones darebbe allora il seguente programma :

```
(letrec ([f2 (lambda (x1 j2 k2) ...)]
         [f7 (lambda (y1 j4 k4) ...)])
  (letrec ([main (lambda (i1 j1 k1) ...)])
    (main 1 1 0)))
```

In questo scheletro ogni funzione è visibile ad alter funzioni che si riferiscono ad essa, i loro parametri formali non sono visibili a queste altre funzioni e quindi non possono subire dropping. L´analisi di Peyton-Jones non pone alcuna funzione nello spazio di alcun parametro formale e così inibisce il dropping di parametri.

5.4 Questioni di correttezza

Abbiamo progettato il lambda-dropping come l'inverso del lambda lifting , per programmi massimalmente annidati. Poiché i programmi sorgente sono spesso parzialmente lifted e parzialmente dropped, non possiamo sperare di fornire un inverso unico per il lambda-lifting.

Date diverse versioni dello stesso programma con annidamento vario, il lambda-lifting le mappa nello stesso set di equazioni mutuamente ricorsive. Così introduciamo le seguenti due proprietà:

Proprietà 1: il lambda-dropping è l´inverso del lambda-lifting su tutti i programmi che hanno subito lambda-dropping.

Proprietà 2: il lambda-lifting è l´inverso del lambda-dropping su tutti i programmi che hanno subito lambda-lifting.

La proprietà 1 è presumibilmente la più complessa delle due. Il lambda-dropping di un programma richiede la ricostruzione della struttura a blocchi che è stata eliminata dal lambda-lifting e l´omissione dei parametri formali che hanno subito lifting dall´operatore del lambda-lifting.

Limitandoci a fornire un inverso di sinistra del lambda-lifting eliminiamo il bisogno di un´analisi. Se una funzione non ha variabili libere prima del lambda-lifting nessun parametro addizionale è aggiunto e non abbiamo bisogno di far cadere parametri per fornire un inverso proprio. Se la funzione aveva variabili libere queste variabili sono applicate come argomenti alla funzione nel punto in cui è trasferita come argomento. Così i parametri extra sono fatti cadere facilmente, poiché sono associati senza ambiguità alla funzione.

L´algoritmo del lambda-lifting di Johnsson nomina esplicitamente le forme lambda anonime con espressioni let, ed elimina le espressioni ler convertendole in applicazioni.

5.5 Complessità temporale

L´algoritmo del lambda-lifting ha una complessità temporale di O ($n^3 + m$ log m), dove n è il numero massimo di funzioni dichiarate in un blocco e m è la dimensione del programma. La componente n^3 è derivata dalla

soluzione delle equazioni di set durante lo stadio di lifting dei parametri. L´algoritmo del lambda-dropping ha una complessità temporale di O (n log n), dove n è la dimensione del programma. Questa complessità assume l´uso di un algoritmo di tempo lineare per trovare i grafici dominatori.

5.6 Studio empirico

Il lambda-dropping di un programma rimuove i parametri formali dalle funzioni, liberando le variabili localmente legate dalla funzione. In linguaggi di ordine superiore è usualmente necessario memorizzare i legami di queste variabili libere quando si trasferisce una funzione come un valore.

5.6.1 Risultati

Esiste un legame naturale tra il trasferimento di argomenti come parametri formali e l'accesso ad essi tramite chiusura.

Generalmente in ogni chiusura i valori devono essere copiati occupando spazio sia nello stack che nella heap ,così la funzione può essere chiamata molte volte con la stessa chiusura. Questo è rilevante nel caso di funzioni ricorsive, poiché il blocco che le circonda è già stato penetrato quando la

funzione chiama se stessa e così la stessa chiusura è usata in ogni invocazione ricorsiva della funzione.

Mentre il trasferimento di parametri formali nello stack, sebbene sia più semplice, deve essere fatto su ogni invocazione della funzione.

Una funzione ricorsiva in un programma scritto senza struttura a blocchi deve esplicitamente manipolare ogni cosa che le serve dall´ambiente a ogni chiamata ricorsiva. Per contrasto, se la funzione usa invece variabili libere, ad esempio, dopo il lambda-dropping, la performance del programma può essere migliorata:

- meno valori hanno bisogno di essere inseriti nello stack ad ogni chiamata ricorsiva; questo riduce il numero di istruzioni di macchina spese a ogni invocazione della funzione.
- Se una variabile libera è usata solo in casi speciali, potrebbe non essere manipolata durante l´esecuzione del corpo della funzione. Questo riduce la quantità di manipolazione dei dati, potenzialmente riducendo lo spilling del registro e migliorando la performance della cache.

5.6.2 Esperimenti

Gli esperimenti iniziali suggeriscono che il lambda-dropping può migliorare la performance delle funzioni ricorsive.

Il miglioramento dipende dall´implementazione, il numero di parametri rimossi, il numero risultante di parametri, e la profondità della ricorsione. Secondo l'esperienza il lambda-dropping incrementa la performance per le seguenti implementazioni dei linguaggi funzionali con struttura a blocchi:

- Scheme: Chez Scheme e SCM;
- ML: Standard ML del New Jersey e ML di Mosca;
- Haskell: il Compilatore Haskell di Glasgow;
- Miranda.

e anche per le implementazioni dei linguaggi imperativi con struttura a blocchi come Delphi, Pascal, Gnu C e Java 1.1.

Ad eccezione di Miranda, sono stati tutti realizzati su una macchina Pentium. Gli esperimenti con Java e Pascal sono stati fatti con Windows e tutti gli altri con Linux.

L´accelerazione riferita $S_{L \rightarrow D}$ è il tempo impiegato dalla versione con lambda-lifting del programma diviso per il tempo impiegato dalla versione con lambda-dropping.

Language	$S_{L\to D}$
ML (Moscow ML)	1.77
Miranda*	1.50
Pascal (Delphi)	1.09
Haskell (GHC 2.01)	1.02
"C" (GCC)	1.00
ML (SML/NJ 110)	1.00
Scheme (SCM)	1.00
Scheme (Chez v5)	0.98
Java (Sun JDK 1.1.1)	0.40

Append
(many invocations on small lists)

Language	$S_{L\to D}$
ML (Moscow ML)	1.96
Miranda*	1.52
Pascal (Delphi)	1.10
Haskell (GHC 2.01)	1.01
"C" (GCC)	1.00
ML (SML/NJ 110)	1.00
Scheme (SCM)	1.00
Scheme (Chez v5)	0.95
Java (Sun JDK 1.1.1)	0.90

Append
(few invocations on long lists)

Questi primi due esperimenti valutano l´efficienza della funzione *append* in diverse situazioni. Il terzo esperimento, valuta invece l´efficienza delle funzioni mutuamente ricorsive che effettuano semplici computi aritmetici, dove cinque su sette parametri sono ridondanti. Rappresenta un caso ideale per l´ottimizzazione del lambda-dropping.

Language	$S_{L\to D}$
ML (SML/NJ 110)	2.74
Miranda*	2.02
ML (Moscow ML)	1.42
Haskell (GHC 2.01)	1.23
Scheme (SCM)	1.16
Java (Sun JDK 1.1.1)	1.13
Pascal (Delphi)	1.12
Scheme (Chez v5)	1.02
"C" (GCC)	1.01

Mutually recursive functions
(many parameters)

I risultati indicano che l´applicazione del lambda-dropping localmente può ottimizzare le funzioni ricorsive. Questi esperimenti sono stati condotti su

una scala molto limitata usando due compilatori maturi: il Compilatore Glasgow Haskell e l' ML Standard del New Jersey. L' ML Standard del New Jersey in particolare trae beneficio dal lambda-dropping di funzioni mutuamente ricorsive.

Bibliografia

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University
Press, New York, 1992.

[2] Andrew W. Appel. *Loop headers in lambda-calculus or CPS. Lisp and Symbolic Computation,* 1994.

[3] Andrew W. Appel. *Modern Compiler Implementation in {C, Java, ML}*.
Cambridge University Press, New York, 1998.

[4] Lennart Augustsson. *Compiling Lazy Functional Languages, part II*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, 1988.

[5] Anindya Banerjee and David A. Schmidt. *A categorical interpretation of Landin's correspondence principl*e. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*,* number 802 in Lecture Notes in Computer Science.
New Orleans, Louisiana, April 1993. Springer-Verlag.

[6] Henk Barendregt. *The Lambda Calculus | Its Syntax and Semantics.*
North-Holland, 1984.

[7] Lars Birkedal and Morten Welinder. *Partial evaluation of Standard ML*. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1993. DIKU Rapport 93/22.

[8] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland*, Oregon, January 1994. ACM Press.

[9] Anders Bondorf and Jesper Jorgensen. *Efficient analyses for realistic online partial evaluation. Journal of Functional Programming*.

[10] William Clinger and Lars Thomas Hansen. *Lambda, the ultimate label, or a simple optimizing compiler for Scheme*. In Talcott [45].

[11] Charles Consel. *A tour of Schism: A partial evaluation system for higherorder applicative languages*. In David A. Schmidt, editor, Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 145-154, Copenhagen, Denmark, June 1993. ACM Press.

[12] Charles Consel and Olivier Danvy. *Static and dynamic semantics processing.*
In Robert (Corky) Cartwright, editor, Proceedings of the Eighteenth
Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, January 1991. ACM Press.

[13] Charles Consel and Olivier Danvy. *Tutorial notes on partial evaluation.*
In Susan L. Graham, editor, Proceedings of the Twentieth Annual
ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993. ACM Press.

[14] Olivier Danvy. *Type-directed partial evaluation*. In Guy L. Steele Jr., editor.
Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida , January 1996. ACM Press.

[15] Olivier Danvy. *An extensional characterization of lambda-lifting and lambda-dropping*. Technical Report BRICS RS-98-2, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 1998.

[16] Olivier Danvy, Nevin C. Heintze, and Karoline Malmkj_r. Resourcebounded partial evaluation. *ACM Computing Surveys,* June 1996.

[17] Olivier Danvy and Ulrik Pagh Schultz. *Lambda-dropping: transforming recursive equations into programs with block structure*. In Charles Consel editor, Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997. ACM Press.

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237-247, Albuquerque, New Mexico, June 1993. ACM Press.

[19] Pascal Fradet. *Syntactic detection of single-threading using continuations.*
In John Hughes, editor*,* Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture, number 523 in Lecture Notes in Computer Science, Cambridge, Massachusetts,
August 1991. Springer-Verlag.

[20] Carsten K. Gomard and Neil D. Jones. *A partial evaluator for the untyped lambda-calculus*. Journal of Functional Programming, 1(1):21{69, 1991.

[21] John Hughes. *Super combinators: A new implementation method for applicative languages*. In Daniel P. Friedman and David S. Wise, editors, 54 Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, pages 1-10, Pittsburgh, Pennsylvania, August 1982. ACM Press.

[22] John Hughes. *Type specialisation for the lambda calculus; or, a new paradigm for partial evaluation based on type inference*. In Olivier Danvy, Robert Gluck, and Peter Thiemann, editors, Partial Evaluation, number 1110 in Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996. Springer-Verlag.

[23] Thomas Johnsson. *Lambda lifting: Transforming programs to recursive equations*. In Jean-Pierre Jouannaud, editor, Functional Programming Languages and Computer Architecture, number 201 in Lecture Notes in Computer Science, Nancy, France, September 1985. Springer-Verlag.

[24] Thomas Johnsson. *Compiling Lazy Functional Languages*. PhD thesis , Department of Computer Sciences, Chalmers University of Technology,
Goteborg, Sweden, 1987.

[25] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[26] Neil D. Jones, Peter Sestoft, and Harald S_ndergaard. *MIX: A self applicable partial evaluator for experiments in compiler generation. Lisp and Symbolic Computation*, 1989.

[27] Peter J. Landin. *The mechanical evaluation of expressions*. Computer Journal.

[28] Peter J. Landin. *The next 700 programming languages*. Communications of the ACM 1966.

[29] Karoline Malmkj_r, Nevin Heintze, and Olivier Danvy. *ML partial evaluation using set-based analysis*

[30] Karoline Malmkjær and Peter Orbæk. *Polyvariant specialization for higherorder, block-structured languages*. In William L. Scherlis, editor June 1995. ACM Press.

[31] Austin Melton, David A. Schmidt, and George Strecker. *Galois connections and computer science applications*. In David H. Pitt et al., editors.

[32] Simon Peyton Jones, Will Partain, and Andr_e Santos. *Let-floating: moving bindings to give faster programs*.
Philadelphia, Pennsylvania, May 1996. ACM Press.

[33] Simon L. Peyton Jones. *An introduction to fully-lazy supercombinators.*
Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors.

[34] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International Series in Computer Science.
Prentice-Hall International, 1987.

[35] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages.*
Prentice Hall International Series in Computer Science. Prentice-Hall, 1992.

[36] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University.
Stanford, California, February 1993. Technical report CSL-TR-93-563.

[37] Andr_e Santos. *Compilation by transformation in non-strict functional languages.*

PhD thesis, Department of Computing, University of Glasgow, Glasgow, Scotland, 1996.

[38] David A. Schmidt. *Detecting global variables in denotational definitions.*
ACM Transactions on Programming Languages and Systems, April 1985.

[39] Ulrik P. Schultz. *Implicit and explicit aspects of scope and block structure.*
Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1997.

[40] Peter Sestoft. Replacing function parameters by global variables. Master's
thesis, DIKU, Computer Science Department, University of Copenhagen.
Copenhagen, Denmark, October 1988.

[41] Peter Sestoft. *Replacing function parameters by global variables*. In Joseph E. Stoy, editor, Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture, London, England, September 1989. ACM Press.

[42] Zhong Shao and Andrew W. Appel. *Space efficient closure representations.*
In Talcott [45].

[43] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMUCS-91-145.

[44] Paul A. Steckler and Mitchell Wand. *Lightweight closure conversion.* ACM Transactions on Programming Languages and Systems, January 1997.

[45] Carolyn L. Talcott, editor. *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming, LISP Pointers,* Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[46] Mads Tofte and Jean-Pierre Talpin. *Implementation of the typed callby-value lambda-calculus using a stack of regions*. In Boehm [8].

[47] Philip Wadler. *Deforestation: Transforming programs to eliminate trees.*
Theoretical Computer Science, 1989. Special issue on ESOP'88, the Second European Symposium on Programming, Nancy,
France, March , 1988.