DOCUMENTAZIONE DEL PROGETTO DI LABORATORIO DI ANALISI E VERIFICA DI PROGRAMMI AA 2005/2006

Possamai Lino*

Indice

1 Introduzione **2**

2 Copy Propagation **3**
2.1 Linguaggio WHILE 4
2.2 Definizione dell'analisi 5
2.3 Analisi Lazy 6
2.4 Analisi Eager 6
2.5 Esempio di applicazione 11
2.6 Correttezza e Terminazione 14

3 CP con Program Analyzer Generator **17**
3.1 Implementazione analisi con Pag 18
3.2 Sezione Type e Problem 18
3.3 Sezione Tranfer 19
3.4 Sezione Support 20

4 Testing **21**
4.1 Analisi Lazy 21
4.2 Analisi Eager 23

5 Future Work **26**

6 Conclusioni **26**

A Appendice **27**

*lino@possamai.it, matricola 800509

1 Introduzione

L'analisi data flow corrisponde all'analisi statica di codice scritto in un qualche linguaggio di programmazione con l'obiettivo di analizzare, data una proprietà, i punti del programma in cui questa è valida.

Prima di poter applicare una qualsiasi analisi data flow, è necessario trasformare il codice sorgente in un grafo di flusso (detto appunto control flow graph, CFG) in cui ogni nodo del grafo corrisponde ad un comando (ad una procedura oppure ad un insieme di comandi) e gli archi saranno direzionati in base al flusso di esecuzione del programma. La differenza tra un'analisi *intra-procedurale* e *inter-procedurale* riguarda il tipo di comandi che possono esserci all'interno dei nodi del control flow graph. Nella prima, si analizza una procedura alla volta (solitamente si assegna un comando ad un nodo) mentre nella seconda, ci possono essere dei nodi che contengono dei comandi `call`.

Informalmente, costruire un'analisi data flow consiste nel definire delle equazioni (solitamente ricorsive o mutuamente ricorsive) che modellano una situazione valida prima e dopo l'esecuzione di un comando/procedura presente in un certo nodo. Per poter risolvere queste equazioni è necessario inserire un caso iniziale ed applicare un algoritmo di punto fisso.

A seconda della proprietà che si deve analizzare, si distinguono analisi forward e backward. Le prime assumono di partire dal nodo iniziale e di seguire il normale flusso di esecuzione del programma, mentre le seconde prevedono invece di partire dal nodo terminale ed andare quindi a ritroso. La scelta della proprietà determina non solo la direzione dell'analisi ma anche la precisione della stessa. Infatti, ci sono alcune analisi, definite *possible*, che calcolano il massimo insieme di oggetti (variabili per esempio) per cui la proprietà è valida. Viceversa, le analisi cosiddette *definite* calcolano il minimo insieme di oggetti per cui la proprietà in questione è soddisfatta per un certo nodo.

In letteratura sono state proposte (ed effettivamente utilizzate) molte tecniche di analisi data flow. Tra le più importanti ricordiamo l'analisi di *liveness*, la *reaching definitions*, l'*available expressions* e la *very busy expressions*. Ognuna di queste affronta tipologie diverse di ottimizzazioni motivo per il quale tutte quante fanno parte integrante di ogni compilatore.

Studiando in dettaglio queste analisi, si possono trovare diverse parti in comune, motivo per il quale in [4] si è proposto un pattern che le unifica in un unico framework generale. Questo è rappresentato dalle seguenti espressioni ricorsive:

$$GA_\circ(p) = \begin{cases} \iota & \text{se p} \in E \\ \oplus\{GA_\bullet(q) | q \in F\} & \text{altrimenti} \end{cases}$$

e $GA_\bullet(p) = f(GA_\circ(p))$ in cui:

- **E** rappresenta l'insieme dei nodi iniziali o terminali del grafo (nel primo caso l'analisi sarà forward, backward altrimenti);
- ι specifica l'informazione assegnata ai nodi iniziali (forward) o finali (backward);
- **F** è l'insieme di nodi del grafo successivi (backward) o precedenti (forward);
- $\oplus$ è l'operazione insiemistica di intersezione (definite) o unione (possible);
- f è la funzione di transfert che calcola come l'informazione su una proprietà deve essere modificata dopo l'esecuzione di un comando.

Ogni istanziazione dei parametri precedenti da luogo ad una tipologia diversa di analisi effettuabile con questo framework.

Dopo questa breve introduzione, descriviamo come si intende procedere in questa relazione: nel capitolo successivo viene presentata in dettaglio l'analisi di copy propagation, obiettivo di questo lavoro, cercando di formalizzare il più possibile i concetti introdotti. Nel capitolo 3 viene introdotto lo strumento che è stato utilizzato per effettuare l'analisi e come è stato programmato per effettuare la copy propagation. Infine, nel capitolo 4, vengono presentati tutti i test effettuati con alcuni programmi di prova. La relazione si conclude con alcuni spunti per future modifiche dell'analisi.

2 Copy Propagation

L'analisi di copy propagation consiste nella rilevazione, dato un CFG, di tutti i cammini che partono da un comando di assegnazione del tipo `x:=y` (lo chiameremo comando copia) e terminano in un comando in cui si assegna un qualsiasi altro valore ad una delle due variabili in questione. In tutti questi cammini la proprietà che due variabili sono uguali rimane valida per cui tutti i riferimenti alle due variabili nei nodi del cammino possono essere sostituiti da un'unica variabile, risparmiando l'uso di un registro. Inoltre, in alcuni casi, il nodo che contiene il comando copia può essere eliminato, mantenendo inalterata la semantica del programma.

Supponiamo ora di avere il seguente frammento di codice:

$$[\texttt{y:=1}]^1; [\texttt{x:=y}]^2; [\texttt{z=1+x}]^3; [\texttt{k=x+2}]^4;$$

L'analisi di copy propagation identifica che nel cammino $2 \rightarrow 3 \rightarrow 4$ le variabili x e y mantengono sempre lo stesso valore per cui è possibile effettuare le sostituzioni $[\texttt{z:=1+x}]^3[\texttt{x/y}]$ e $[\texttt{k:=x+2}]^4[\texttt{x/y}]$ e cancellare il comando 2 trasformando quindi il programma precedente in un

altro che è semanticamente equivalente (nella semantica che considera i valori delle variabili calcolati dal programma) al seguente:

$$[\texttt{y:=1}]^1; [\texttt{z=1+y}]^3; [\texttt{k=y+2}]^4.$$

Prima di entrare in dettaglio nei formalismi dell'analisi, è d'obbligo definire brevemente sintassi e semantica del linguaggio imperativo su cui si basa l'analisi, il linguaggio WHILE.

2.1 Linguaggio WHILE

Il linguaggio WHILE è un linguaggio imperativo molto semplice e allo stesso tempo molto potente (Turing Complete). La sintassi del linguaggio è definita nella tabella 1. Da come si vede nella tabella, un'espressione aritmetica a può essere un numerale, una variabile oppure un'espressione a_1 op a_2 qualsiasi con $op \in \{+, *, -, /\}$. Un'espressione booleana può valere `true`, `false`, un'espressione a_1 op a_2 qualsiasi con $op \in \{=, <, >, \leq, \geq, <>\}$ oppure un'espressione con la negazione. I comandi si possono costruire utilizzando uno dei seguenti costrutti: l'assegnamento, lo skip (utile per l'eliminazione del ramo else nel comando if-then-else), il ';', l'if-then-else ed il costrutto while.

Una fase indispensabile, dopo quella di definizione della sintassi, è rappresentata dalla specifica della semantica di un linguaggio di programmazione. Sebbene siano diversi gli approcci alla definizione della semantica (denotazionale, operazionale e assiomatica), noi ne sceglieremo uno presentando le sue regole.

Tra tutte le semantiche disponibili, abbiamo scelto quella operazionale perché ci tornerà utile (vedi sezione 2.6) nel momento in cui dimostreremo la correttezza dell'analisi proposta. Informalmente, la semantica operazionale consiste di un modello ambiente-memoria (A-M). Su questo modello si descrive, appunto operazionalmente, a partire da una certa sintassi, l'effetto di ogni comando sul modello A-M.

a	::=	$n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \mid a_1/a_2 \mid -a_1$
b	::=	$\text{true} \mid \text{false} \mid a_1 < a_2 \mid a_1 \leq a_2 \mid a_1 > a_2 \mid a_1 \geq a_2 \mid a_1 = a_2 \mid a_1 <> a_2 \mid \neg b_1$
S	::=	$x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if b then } S_1 \text{ else } S_2 \mid \text{while b do S}$

Tabella 1: Sintassi del Linguaggio WHILE

Le regole della semantica operazionale sono elencate nella tabella 2. In questa tabella si fa riferimento alla funzione $\mathcal{A}$ definita in questo modo: $\mathcal{A} : Aexp \rightarrow (State \rightarrow \mathbb{Z})$ con $State : Var \rightarrow \mathbb{Z}$ che rappresenta la funzione di valutazione della semantica delle espressioni aritmetiche.

La funzione $\mathcal{B} : Bexp \rightarrow (State \rightarrow \{tt, ff\})$ rappresenta invece la funzione di valutazione delle espressioni booleane. Per una spiegazione delle regole, si rimanda a [4].

$[ass]$	$\langle [x := a]^l, \sigma \rangle \rightarrow \sigma[x \rightarrow \mathcal{A}[\![a]\!]\sigma]$	
$[skip]$	$\langle [skip]^l, \sigma \rangle \rightarrow \sigma$	
$[seq_1]$	$\dfrac{\langle S_1, \sigma \rangle \rightarrow \langle S_1', \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S_1'; S_2, \sigma' \rangle}$	
$[seq_2]$	$\dfrac{\langle S_1, \sigma \rangle \rightarrow \langle \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S_2, \sigma' \rangle}$	
$[if_1]$	$\langle \text{if [b]}^l \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$	se $\mathcal{B}[\![b]\!]\sigma$=true
$[if_2]$	$\langle \text{if [b]}^l \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle$	se $\mathcal{B}[\![b]\!]\sigma$=false
$[wh_1]$	$\langle \text{while [b]}^l \text{ do } S, \sigma \rangle \rightarrow \langle (S; \text{while [b]}^l \text{ do } S), \sigma \rangle$	se $\mathcal{B}[\![b]\!]\sigma$=true
$[wh_2]$	$\langle \text{while [b]}^l \text{ do } S, \sigma \rangle \rightarrow \sigma$	se $\mathcal{B}[\![b]\!]\sigma$=false

Tabella 2: Semantica Operazionale del Linguaggio WHILE

2.2 Definizione dell'analisi

Dopo aver introdotto la sintassi e la semantica del linguaggio gestito dall'analisi, è ora possibile entrare in dettaglio e presentare in termini più formali, la copy propagation. Rispetto al general framework presentato nell'introduzione, vediamo come poterlo istanziare creando un'analisi che fa comodo a noi. Per questioni di uniformità, da questo punto in poi, sarà utilizzata la stessa notazione dell'analisi data flow presente in [4].

Di seguito sono presentate due tipologie di analisi. Le due analisi si distinguono per l'approccio adottato nel momento in cui si combinano le informazioni provenienti da diversi antenati di un nodo.

La prima è un'analisi lazy, nel senso che se in due rami diversi di computazione vengono definite due variabili copia [`x:=y`], in ingresso al nodo di confluenza non viene propagata nessuna informazione. Secondo questa analisi e rispetto al CFG di figura 1(c), in ingresso al nodo 70, le variabili x e y non sono uguali.

Per far fronte a questo problema nasce la seconda analisi. Quest'ultima cerca di aumentare la precisione nel caso di informazioni in ingresso a nodi di confluenza. Infatti, sempre in riferimento

alla stessa figura, per qualsiasi cammino di computazione venga intrapreso e per qualsiasi valore assegnato alla variabile y, al nodo 70 è possibile sostituire i riferimenti ad x con riferimenti ad y, senza alterare la semantica del programma.

2.3 Analisi Lazy

Il primo elemento che bisogna specificare è il dominio astratto usato per rappresentare la proprietà. Per quanto riguarda l'analisi di copy propagation, possiamo considerare il seguente insieme L:

$$\mathrm{L} = \mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathbf{Lab}_*)$$

con l'ordine parziale $\subseteq$ e l'operatore di greatest lower bound $\cap$. (L,$\subseteq$) è un reticolo completo. Per poter aumentare la precisione, potrebbe essere interessante considerare un nuovo ordine parziale $\sqsubseteq$, sempre rispetto a L, definito nel seguente modo:

$$\begin{aligned} \forall c \in L \quad &: \quad \emptyset \sqsubseteq c \\ \forall a, b \in L \quad &: \quad a \sqsubseteq b \text{ sse } \forall (a_1, a_2, a_3) \in a,\ \exists (b_1, b_2, b_3) \in b \mid \\ &\qquad (a_1 = b_1) \wedge (a_2 = b_2) \wedge (a_3 \geq b_3) \end{aligned}$$

e rispettivamente l'operazione di greatest lower bound sarà definita così:

$$\begin{aligned} a \sqcap b \quad = \quad & \{(x, y, l) \mid \exists (a_1, a_2, a_3) \in a, (b_1, b_2, b_3) \in b\ , \\ & (x = a_1 = b_1) \wedge (y = a_2 = b_2) \wedge (l = max(a_3, b_3))\}. \end{aligned}$$

Il reticolo (L,$\sqsubseteq$) appena definito, va bene per rappresentare l'informazione sulla copia di due variabili e sulla label in cui questa copia è definita, ma anche in questo caso non è sufficiente per gestire completamente situazioni come quelle rappresentate in figura 1(c). Se utilizziamo il reticolo $(L, \sqsubseteq)$, l'informazione in ingresso al nodo 70 sarà (x,y,50), senza però considerare che esiste un altro cammino in cui le variabili x e y sono uno copia dell'altra: $10 \rightarrow 70$.

La definizione degli altri elementi dell'analisi non è stata inserita qui perché identica (con le dovute semplificazioni) rispetto ad una estensione di questa analisi, detta eager, definita nella sezione successiva.

2.4 Analisi Eager

Nell'analisi precedente e in questa che presenteremo, il dominio concreto è rappresentato dall'insieme di tutti i possibili stati delle variabili. Per quanto riguarda il nuovo dominio astratto, che dovrebbe risolvere tutti i problemi che sono stati sollevati precedente, si considera l'insieme:

$$\mathrm{L} = \mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathcal{P}(\mathbf{Lab}_*))$$

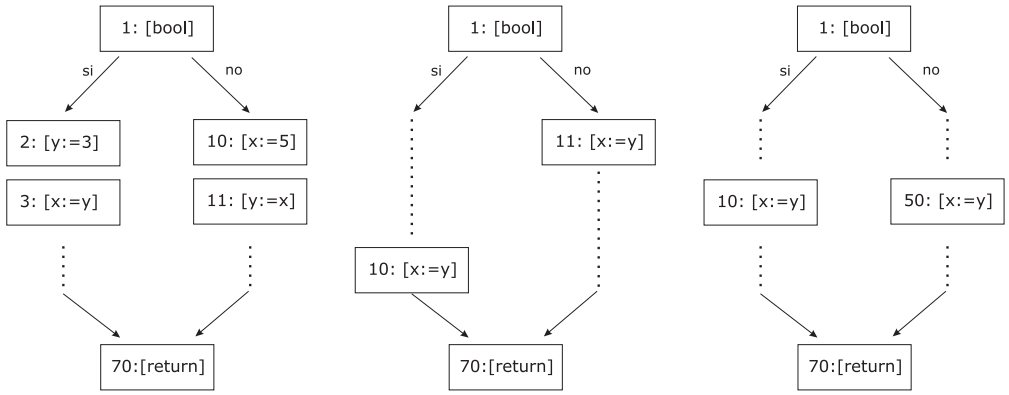


Figura 1: Tre situazioni che si possono verificare con il costrutto if-then-else. In figura (a) all'ingresso del nodo 70, non è possibile affermare che x e y sono uguali. Questo vuol dire che l'ordine di assegnamento è importante. In (b) e (c) altri due esempi che testimoniano la necessità di specificare, in ingresso al nodo 70, che in più di un cammino, x e y hanno lo stesso valore.

e si definisce la relazione $\trianglelefteq$ nel seguente modo:

$$\begin{aligned} \forall c \in L \quad &: \quad \emptyset \trianglelefteq c \\ \forall a, b \in L \quad &: \quad a \trianglelefteq b \text{ sse } \forall (a_1, a_2, \{a_3\}) \in a,\ \exists (b_1, b_2, \{b_3\}) \in b \mid \\ &\qquad (a_1 = b_1) \wedge (a_2 = b_2) \wedge (\{a_3\} \subseteq \{b_3\}) \end{aligned} \tag{1}$$

La relazione $\trianglelefteq$ è un ordine parziale perché valgono le seguenti proprietà:

- riflessiva: $\forall a \in L$, $a \trianglelefteq a$, infatti se a=$(a_1, a_2, \{a_3\})$ vale ovviamente che $a_1 = a_1$, $a_2 = a_2$ e $\{a_3\} \subseteq \{a_3\}$.
- transitiva: $\forall a = (a_1, a_2, \{a_3\}), b = (b_1, b_2, \{b_3\}), c = (c_1, c_2, \{c_3\}) \in L$, tale che $(a \trianglelefteq b) \wedge (b \trianglelefteq c) \Rightarrow a \trianglelefteq c$. Questa proprietà è verificata perché $a_1 = b_1 = c_1$, $a_2 = b_2 = c_2$ e $\{a_3\} \subseteq \{b_3\} \subseteq \{c_3\}$ implica che $a_1 = c_1$, $a_2 = c_2$ e $\{a_3\} \subseteq \{c_3\}$.
- antisimmetrica: $\forall a = (a_1, a_2, \{a_3\}), b = (b_1, b_2, \{b_3\})$ con $(a \trianglelefteq b) \wedge (b \trianglelefteq a) \Rightarrow a = b$. Vera perché $a_1 = b_1$, $a_2 = b_2$ e $\{a_3\} \subseteq \{b_3\}$, $\{b_3\} \subseteq \{a_3\}$ implica che $a_3 = b_3$.

L'operatore di greatest lower bound $\sqcap$ (identificato da qui in poi con il simbolo $\bigtriangledown$) è così definito:

$$a \bigtriangledown \bot \;=\; \bot \bigtriangledown a \;= \bot,\ \forall a \in L$$

$$
\begin{aligned}
a \bigtriangledown \top &= \top \bigtriangledown a = a,\ \forall a \in L \\
a \bigtriangledown b &= \{(x, y, \{l\}) \mid \exists (a_1, a_2, \{a_3\}) \in a, (b_1, b_2, \{b_3\}) \in b\ , \\
&\quad (x = a_1 = b_1) \wedge (y = a_2 = b_2) \wedge (l = \{a_3\} \cup \{b_3\})\}.
\end{aligned} \tag{2}
$$

che informalmente corrisponde ad un operatore di intersezione leggermente modificato. Rispettivamente, $a \bigtriangleup b$ è definito come:

$$
\begin{aligned}
a \bigtriangleup b &= \{(a_1, a_2, a_3), (b_1, b_2, b_3) \in L \mid \forall (a_1, a_2, a_3) \in a, (b_1, b_2, b_3) \in b, \\
&\quad a_1 \neq b_1 \wedge a_2 \neq b_2\} \cup (a \bigtriangledown b)
\end{aligned}
$$

(unione per le variabili copia diverse, mentre se a e b hanno in comune degli elementi, vale $a \bigtriangledown b$). Il reticolo (L,$\trianglelefteq$) è un reticolo completo perché ogni sottoinsieme di L ha least upper bound e greatest lower bound.

Per avere un'idea sulla rappresentazione grafica del reticolo completo (L,$\trianglelefteq$) si veda la figura 2. Prima di procedere, è necessaria una spiegazione del motivo per cui si è scelto di definire l'ordine parziale $\trianglelefteq$ e l'operatore di greatest lower bound $\bigtriangledown$ in tale modo. Per poter motivare la definizione dei primi due vincoli in (1) è necessario considerare la figura 1(a). Prima dell'ultimo comando, l'analisi di CP calcolerebbe $(x, y, \{3\})$ e $(y, x, \{11\})$, informazioni provenienti dai due rami dell'if. A prima vista si potrebbe pensare di propagare l'informazione che le due variabili, x e y, sono una copia dell'altra, indifferentemente dall'ordine di definizione, invece, questo non può essere fatto perché potrebbe portare ad un risultato errato se in precedenza, come si vede dalla figura, le variabili sono inizializzate con valori differenti. L'ultimo vincolo della definizione di ordine parziale, $\{a_3\} \subseteq \{b_3\}$, è giustificato dal fatto che un'informazione sulla definizione di copia che considera un insieme più esteso, risulta più informativa.

Per quanto riguarda la definizione (2), l'ultimo vincolo $l = \{a_3\} \cup \{b_3\}$ è giustificato (vedi figura 1(b) e 1(c)) dal fatto che la definizione delle variabili copia può provenire da definizioni all'interno di due (o più) rami.

L'obiettivo dell'analisi è quello di cercare i cammini $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_m$ che partono da un assegnamento del tipo $[x := y]$ con $x, y \in \mathbf{Var}_*$ in cui solo alla fine una delle due variabili x o y viene ridefinita (cioè vale $s_m \equiv [x := e_1]$ oppure $s_m \equiv [y := e_1]$, per una qualche espressione e_1) allora se ne deduce che la direzione dell'analisi è *forward*.

Quest'ultima scelta implica che in $\mathbf{E}$ ci sarà il nodo iniziale del CFG (in tutti i casi è rappresentato dal nodo 1), in ι si specificherà l'informazione iniziale, che definiamo come $\emptyset$, con $\mathbf{F}$ si indica l'insieme degli antenati di un nodo.

Per decidere se l'analisi è *definite* o *possible*, consideriamo il seguente frammento di codice S:

$$
[\texttt{g:=3}]^1; [\texttt{x:=2}]^2; (\texttt{if}\ \ [\texttt{k} \leq \texttt{3}]^3\ \texttt{then}\ [\texttt{y:=x}]^4\ \texttt{else}\ [\texttt{y:=g}]^5); [\texttt{m:=y+3}]^6
$$

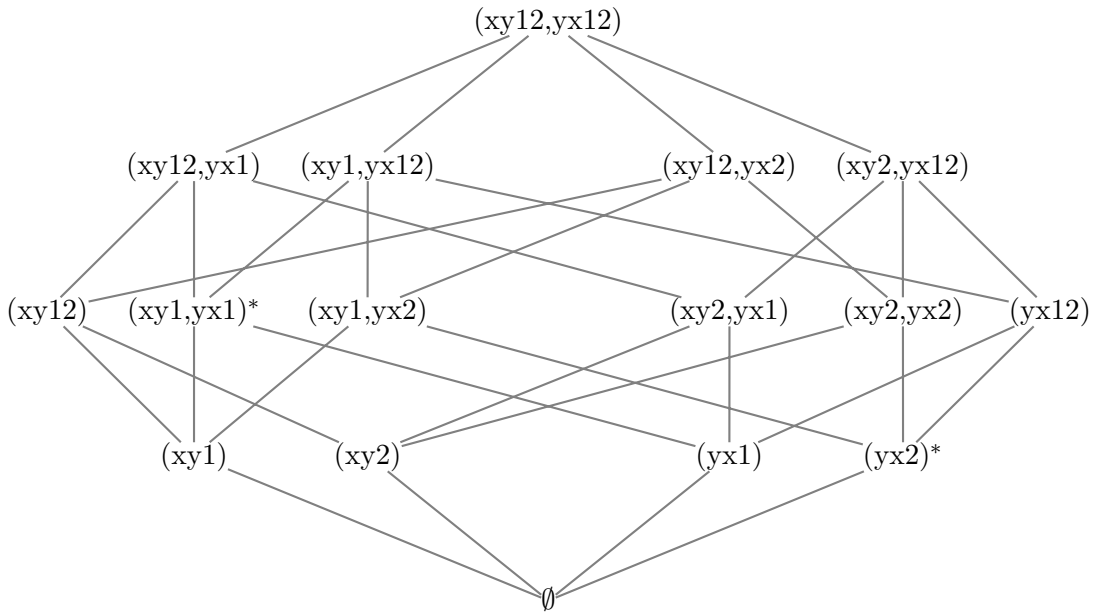


Figura 2: Il grafo indica un esempio di rappresentazione grafica del reticolo completo (L,$\trianglelefteq$) quando $\mathbf{Var}_*=\{x,y\}$ e $\mathbf{Lab}_*=\{1,2\}$. Ogni nodo è un insieme di elementi della forma $xyn=(x,y,n)$. Ogni sottoinsieme di L ha least upper bound e greatest lower bound. Infatti se per esempio consideriamo l'insieme Y=$\{(xy1,yx1),(yx2)\}$ (gli elementi sono contrassegnati nel grafo con un asterisco) allora $\sqcap$Y=$\emptyset$ mentre $\sqcup$Y=$\{(xy1,yx12),(xy12,yx2),(xy12,yx12)\}$.

che è rappresentato dal CFG di figura 3.

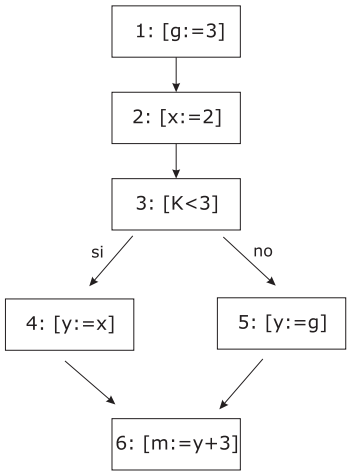


Figura 3: Control Flow Graph del codice S

Come si può notare, nel nodo 6 non è possibile propagare l'informazione sulla proprietà calcolata nei nodi 4 e 5 (fare l'unione in altre parole) perché se così fosse, in 6 potremo scegliere fra due sostituzioni possibili (g o x), ma queste due variabili potrebbero contenere valori diversi (com'è il caso dell'esempio). Questo implica l'impossibilità di poter applicare una tecnica *possible* a questa analisi, a differenza di un'analisi *definite* che produce un risultato corretto (e sicuro).

Per quanto riguarda la funzione di transfert, cioè la funzione che rappresenta la semantica astratta di un certo comando, può essere definita nel seguente modo:

$$f(GA_\circ(p)) = \text{gen(p)} \cup (GA_\circ(p) - \text{kill(p)})$$

La funzione è definita in modo simile alle altre analisi presentate in [2]. Il punto in cui si differenziano riguarda la definizione delle funzioni di generazione e uccisione delle variabili copia.

La funzione di generazione dell'informazione sarà pertanto definita in questo modo:

$$gen : \mathbf{Lab}_* \rightarrow \mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathcal{P}(\mathbf{Lab}_*)).$$

e

$$gen(l) = \{(x, y, \{l\}) \mid [x := y]^l\}.$$

con $x, y \in \mathbf{Var}_*$ e $l \in \mathbf{Lab}_*$. Questa funzione, dato in input un nodo del CFG, altro non fa che controllare il tipo di comando corrispondente e se questo è un assegnamento variabile-variabile allora restituisce un insieme formato da una tripla che contiene le due variabili copia e un insieme inizializzato con la label di quel comando. Per gli altri comandi, la funzione gen restituisce l'insieme vuoto.

La funzione di uccisione, è definita in questo modo:

$$kill : \mathbf{Lab}_* \rightarrow \mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathcal{P}(\mathbf{Lab}_*)),$$

$$kill(l) = \{k \mid [x := e_1]^l \ \wedge (k = (x, y, \{i\}) \in gen(j) \vee k = (y, x, \{i\}) \in gen(j), 1 \leq j < l)\}.$$

La funzione accetta in input un nodo del CFG e se il comando corrispondente è un comando di assegnamento in cui si assegna un valore ad una delle variabili presente in qualche insieme gen dei nodi precedenti, allora restituirà questi insiemi. Negli altri casi restituisce l'insieme vuoto.

Riassumendo, l'analisi di copy propagation è definita attraverso i seguenti vincoli:

$$CP_{entry}(p) = \begin{cases} \emptyset & \text{se p è il nodo iniziale} \\ \bigtriangledown\{CP_{exit}(q) \mid q = prec[p]\ \} & \text{altrimenti} \end{cases}$$

e $CP_{exit}(p) = \text{gen}(p) \cup (CP_{entry}(p) - \text{kill}(p))$. L'elemento meno informativo, come si vede nella tabella 3, è $\mathbf{Var}_* \times \mathbf{Var}_* \times \mathcal{P}(\mathbf{Lab}_*)$.

	A. Eager	A. Lazy
L	$\mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathcal{P}(\mathbf{Lab}_*))$	$\mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathbf{Lab}_*)$
$\sqsubseteq$	$\trianglelefteq$	$\subseteq$
$\sqcup$	$\bigtriangledown$	$\cap$
$\bot$	$(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathcal{P}(\mathbf{Lab}_*))$	$(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathcal{P}(\mathbf{Lab}_*))$
ι	$\emptyset$	$\emptyset$
E	$\{\text{init}(S_*)\}$	$\{\text{init}(S_*)\}$
F	$\text{flow}(S_*)$	$\text{flow}(S_*)$
$\mathcal{F}$	$\{f : L \rightarrow L \mid \exists l_k, l_g \in L : f(l) = (l - l_k) \cup l_g\}$	
f_l	$f_l(m) = (m - kill([B]^l)) \cup gen([B]^l)$ dove $[\text{B}]^l \in blocks(S_*)$	

Tabella 3: Istanza del General Framework: Analisi di Copy Propagation versione lazy ed eager

2.5 Esempio di applicazione

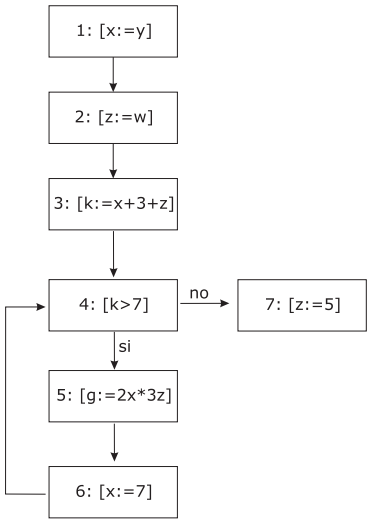
Prima di vedere come si costruisce un'analisi con PAG/WWW [1], analizziamo un esempio di analisi copy propagation rispetto a come è stata definita nella sezione precedente.

Consideriamo quindi il seguente programma S:

$$[\texttt{x:=y}]^1; [\texttt{z:=w}]^2; [\texttt{k:=x+3+z}]^3; (\texttt{while } [\texttt{k} > 7]^4 \texttt{ do } [\texttt{g:=2x*3z}]^5\texttt{;}[\texttt{x:=7}]^6); [\texttt{z:=5}]^7$$

il cui CFG è rappresentato in figura 4(a).

(a)



(b)

nodo	**gen**	**kill**
1	(x,y,{1})	$\emptyset$
2	(z,w,{2})	$\emptyset$
3	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$
5	$\emptyset$	$\emptyset$
6	$\emptyset$	(x,y,{1})
7	$\emptyset$	(z,w,{2})

Figura 4: In figura (a) è schematizzato il CFG a partire dal codice sorgente dell'esempio, mentre in (b) sono elencati i rispettivi insiemi kill e gen che sono stati calcolati.

Prima di tutto è necessario definire gli insiemi gen e kill, necessari alla risoluzione delle equazioni. Come si può vedere dalla tabella 4(b), i nodi 1 e 2 *generano* variabili copia mentre i nodi 6 e 7 uccidono variabili copia perché fanno terminare il cammino in cui due variabili, in questo caso x,y e z,w hanno valori identici. Tutti gli altri nodi non generano né uccidono variabili copia perché non alterano il valore delle variabili in questione.

Nella tabella 4 sono schematizzati i tre step di esecuzione dell'algoritmo che risolve le equazioni dell'analisi (a partire dagli insiemi gen e kill calcolati dal programma S). Guardando l'ultimo run, cioè la soluzione minima, possiamo quindi concludere di aver rilevato due cammini copia:

$$\text{(x,y): } 1 \rightarrow 2 \rightarrow 3 \text{ e}$$

$$\text{(z,w): } 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$$

per cui è lecito, rispetto alla copy propagation, effettuare la seguente trasformazione

$$CP \vdash S \rhd S'$$

con S$'$ uguale a:

$$[\texttt{x:=y}]^1; [\texttt{k:=y+3+w}]^3; (\texttt{while } [\texttt{k} \leq \texttt{7}]^4 \texttt{ do } [\texttt{g:=2x*3w}]^5\texttt{;}[\texttt{x:=7}]^6); [\texttt{z:=5}]^7$$

in cui sono state effettuate le sostituzioni: $[\texttt{k:=x+3+z}]^3$[x/y][z/w], $[\texttt{g:=2x*3z}]^5$[z/w] ed è stato cancellato il comando che fa riferimento alla label 2 perché superfluo. Il comando in 1 non può

(a)

nodo	CP_{entry}	CP_{exit}	CP_{entry}	CP_{exit}
1	$\emptyset$	(x,y,{1})	$\emptyset$	(x,y,{1})
2	(x,y,{1})	(x,y),(z,w,{2})	(x,y,{1})	(x,y,{1}),(z,w,{2})
3	(x,y,{1}),(z,w,{2})	(x,y,{1}),(z,w,{2})	(x,y,{1}),(z,w,{2})	(x,y,{1}),(z,w,{2})
4	(x,y,{1}),(z,w,{2})	(x,y,{1}),(z,w,{2})	(z,w,{2})	(z,w,{2})
5	(x,y,{1}),(z,w,{2})	(x,y,{1}),(z,w,{2})	(z,w,{2})	(z,w,{2})
6	(x,y,{1}),(z,w,{2})	(z,w,{2})	(z,w,{2})	(z,w,{2})
7	(x,y,{1}),(z,w,{2})	(x,y,{1})	(z,w,{2})	$\emptyset$

(b)

nodo	CP_{entry}	CP_{exit}
1	$\emptyset$	(x,y,{1})
2	(x,y,{1})	(x,y,{1}),(z,w,{2})
3	(x,y,{1}),(z,w,{2})	(x,y,{1}),(z,w,{2})
4	(z,w,{2})	(z,w,{2})
5	(z,w,{2})	(z,w,{2})
6	(z,w,{2})	(z,w,{2})
7	(z,w,{2})	$\emptyset$

Tabella 4: Insiemi CP_{entry} e CP_{exit} nei tre run di esecuzione dell'algoritmo

essere eliminato perché la variabile x serve in 5. Il nuovo programma S' sarà semanticamente equivalente a S.

2.6 Correttezza e Terminazione

In letteratura, le dimostrazioni di correttezza dell'analisi hanno uno schema comune. Si parte dal presupposto di avere una semantica di un linguaggio, che specifica come un programma p trasforma un valore $v_1 \in V$ del dominio concreto (valore in questo caso può essere anche una configurazione) in un altro v_2, cioè:

$$p \vdash v_1 \rightsquigarrow v_2$$

Allo stesso tempo, un'analisi di un programma identifica un insieme di proprietà e specifica come, dato un programma, trasforma una proprietà in un'altra:

$$p \vdash l_1 \rhd l_2$$

Ogni analisi deve essere corretta rispetto alla semantica. Questa cosa è stabilita grazie ad una *relazione di correttezza* definita in questo modo:

$$\sim : V \times L \rightarrow \{true, false\}$$

L'idea è proprio quella di formalizzare con $v_1 \sim l_1$ il fatto che il valore v_1 è rappresentato dalla proprietà l_1. Per dimostrare quindi che l'analisi è corretta è necessario dimostrare che la relazione di correttezza definita è preservata rispetto alla computazione, che formalmente può essere espressa come:

$$(v_1 \sim l_1) \wedge (p \vdash v_1 \rightsquigarrow v_2) \wedge (p \vdash l_1 \rhd l_2) \Rightarrow v_2 \sim l_2$$

Questo approccio è valido solamente per quelle analisi dove le proprietà descrivono direttamente i valori (per le analisi definite di primo ordine). Per quanto riguarda le altre analisi, quale per esempio la liveness o la copy propagation, bisogna invece dimostrare qualcosa di leggermente diverso. Questa dimostrazione fa uso di una serie di concetti che sono ampiamente descritti in [4] e che elencheremo solamente:

- Analisi dataflow utilizzando l'approccio Constraint Based considerando quindi $\mathrm{CP}^{\subseteq}$(S) con S un qualsiasi programma.
- Programma S label consistent, cioè se l è una label e corrisponde a due comandi (o blocchi di comandi) necessariamente questi sono uguali.
- $CP_{entry}(l) = N(l)$ e $CP_{exit}(l) = X(l)$.

Se consideriamo un insieme di funzioni cp: $cp_{entry}, cp_{exit} : \mathbf{Lab}_* \to \mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathcal{P}(\mathbf{Lab}_*))$, con $cp \models CP^{\subseteq}(S)$ si indica che cp risolve $CP^{\subseteq}(S)$.

Il teorema centrale della dimostrazione di correttezza è il seguente:

Teorema 1. Se $cp \models CP^{\subseteq}(S)$ con S label consistent, allora:

1. se $\langle S, \sigma_1\rangle \to \langle S', \sigma_1'\rangle$ e $\sigma_1 \sim_{N(init(S))} \sigma_2$ allora esiste σ_2' tale che $\langle S, \sigma_2\rangle \to \langle S', \sigma_2'\rangle$ e $\sigma_1' \sim_{N(init(S'))} \sigma_2'$.
2. se $\langle S, \sigma_1\rangle \to \sigma_1'$ e $\sigma_1 \sim_{N(init(S))} \sigma_2$ allora esiste σ_2' tale che $\langle S, \sigma_2\rangle \to \sigma_2'$ e $\sigma_1' \sim_{X(init(S))} \sigma_2'$.

Per poter dimostrare il teorema, è necessario vedere a cosa corrisponde la relazione di correttezza per la copy propagation. Possiamo per esempio definirla in questo modo:

$$\sigma_1 \sim_V \sigma_2 \text{ sse } \forall (x, y, \{l\}) \in V, \sigma_1(x) = \sigma_1(y) = \sigma_2(x) = \sigma_2(y)$$

che per appesantire meno la notazione, potremmo riscriverla come

$$\sigma_1 \sim_V \sigma_2 \text{ sse } \forall (x, y, \{l\}) \in V,\ \eta_{\sigma_1\sigma_2}(x, y)$$

con $\sigma_1(x) = \sigma_1(y) = \sigma_2(x) = \sigma_2(y)$ sse $\eta_{\sigma_1\sigma_2}(x, y)$. Questa relazione indica banalmente che le variabili copia negli stati σ_1 e σ_2 hanno lo stesso valore.

La dimostrazione deve essere fatta per induzione sulla struttura della semantica operazionale definita nella tabella 2 (pagina 5). Dato che l'analisi genera o uccide variabili copia solamente nel caso di comandi di assegnamento, allora la parte più impegnativa della dimostrazione consiste nel verificare il punto 2 del teorema quando la regola scelta della semantica operazionale è $[ass]$.

Dimostrazione. Supponiamo quindi valida la prima parte del teorema, cioè $\langle S, \sigma_1\rangle \to \sigma_1'$ e $\sigma_1 \sim_{N(l)} \sigma_2$ con $l = init(S)$ e dimostriamo che esiste un σ_2' tale che $\langle S, \sigma_2\rangle \to \sigma_2'$ e $\sigma_1' \sim_{X(l)} \sigma_2'$. Lo facciamo per casi sulla semantica di $\mathcal{A}[\![a]\!]\sigma$ (da notare che se S è un assegnamento, allora vale $\langle [x := a]^l, \sigma_1\rangle \to \sigma_1[x \to \mathcal{A}[\![a]\!]\sigma]$).

Caso $a = (a_1 \text{ op } a_2)$. Per come è definita la regola $[ass]$ possiamo scrivere che

$$\langle [x := a_1 \text{ op } a_2], \sigma_1\rangle \to \sigma_1' = \sigma_1[x \leftarrow \mathcal{A}[\![a_1 \text{ op } a_2]\!]\sigma_1].$$

Vale anche $\sigma_1 \sim_{N(l)} \sigma_2$, cioè $\forall (x, y, \{l\}) \in N(l), \eta_{\sigma_1\sigma_2}(x, y)$. Se prendo un $\sigma_2' = \sigma_2[x \leftarrow \mathcal{A}[\![a]\!]\sigma_2]$ allora vale $\langle [x := a_1 \text{ op } a_2], \sigma_2\rangle \to \sigma_2'$. Per com'è definita la copy propagation (e dato che questo comando uccide variabili copia), $N(l) \supseteq X(l)$ quindi vale che $\sigma_1 \sim_{X(l)} \sigma_2$. Segue banalmente che $\sigma_1' \sim_{X(l)} \sigma_2'$ perché in σ_1' e σ_2' si modifica solo il valore della variabile x, mentre in $X(l)$ la stessa variabile non è presente (perché in questo caso il comando di assegnamento uccide variabili copia che contengono x).

Caso $a = y$, per una qualche variabile y. Sempre come nel caso precedente, per come è definita la regola $[ass]$ possiamo scrivere, che $\langle [x := y], \sigma_1 \rangle \to \sigma_1' = \sigma_1[x \leftarrow \sigma_1(y)]$. Vale anche $\sigma_1 \sim_{N(l)} \sigma_2$, cioè $\forall (x, y, \{l\}) \in N(l), \eta_{\sigma_1 \sigma_2}(x, y)$. Se prendo un $\sigma_2' = \sigma_2[x \leftarrow \sigma_2(y)]$ allora concludo che $\langle [x := y], \sigma_2 \rangle \to \sigma_2'$. In questo caso,

$$X(l) = (N(l) - (k, x, \{l'\}) - (x, g, \{l''\})) \cup (x, y, \{l\}) = K(l) \cup (x, y, \{l\}))$$

per qualche $k, g \in \mathbf{Var}_*$ e $l, l', l'' \in \mathbf{Lab}_*$.

Per verificare che $\sigma_1' \sim_{X(l)} \sigma_2'$ è necessario verificare che $\forall (z, w, \{l\}) \in K(l), \eta_{\sigma_1' \sigma_2'}(z, w)$ e che $\eta_{\sigma_1' \sigma_2'}(x, y)$.

La prima parte è banalmente verificata perché da $N(l) \supseteq K(l)$ segue che $\sigma_1 \sim_{K(l)} \sigma_2$ e quindi $\sigma_1' \sim_{K(l)} \sigma_2'$ perché negli stati σ_1' e σ_2' solo la variabile x viene modificata mentre in $K(l)$ non se ne fa riferimento.

La seconda parte impone di verificare che valga $\eta_{\sigma_1' \sigma_2'}(x, y)$, cioè che $\sigma_1'(x) = \sigma_1'(y) = \sigma_2'(x) = \sigma_2'(y)$. Per come sono state definite σ_1' e σ_2', segue che $\sigma_1'(x) = \sigma_1(y) = \sigma_1'(y)$ e $\sigma_2'(x) = \sigma_2(y) = \sigma_2'(y)$. Per ottenere l'enunciato, basta dimostrare che $\sigma_1(y) = \sigma_2(y)$. Questo segue banalmente se $y \in N(l)$, mentre se y non è copia di nessun'altra variabile, vale ugualmente grazie al fatto che la computazione parte dallo stesso comando e viene applicata la stessa regola. □

Corollario 1. Se $cp \models CP^{\subseteq}(S)$ (con S label consistent) allora:

- se $\langle S, \sigma_1 \rangle \to^* \langle S', \sigma_1' \rangle$ e $\sigma_1 \sim_{N(init(S))} \sigma_2$ allora esiste σ_2' tale che $\langle S, \sigma_2 \rangle \to^* \langle S', \sigma_2' \rangle$ e $\sigma_1' \sim_{N(init(S'))} \sigma_2'$.
- se $\langle S, \sigma_1 \rangle \to^* \sigma_1'$ e $\sigma_1 \sim_{N(init(S))} \sigma_2$ allora esiste σ_2' tale che $\langle S, \sigma_2 \rangle \to^* \sigma_2'$ e $\sigma_1' \sim_{X(l)} \sigma_2'$ per qualche $l \in final(S)$.

Dimostrazione. La dimostrazione è sulla lunghezza della sequenza di derivazione usando il teorema 1. □

Per completezza, è possibile dimostrare che l'analisi è corretta in un modo più semplice. L'idea è quella di dimostrare che l'analisi proposta è un'istanza del general framework, sfruttando quindi la conoscenza che questo framework è corretto. Vediamo brevemente quali sono gli elementi necessari che ci permettono di affermare che la copy propagation è un'istanza del general framework:

- (L,$\trianglelefteq$) è un reticolo completo (ogni sottoinsieme Y di L ha lub e glb, vedi figura 2, pag. 9).
- (L,$\trianglelefteq$) ha altezza finita perché tutte le catene sono finite (da notare che l'insieme delle variabili è finito). Grazie al lemma A.6 di pag 397 di [4] possiamo concludere che il reticolo soddisfa le condizioni sulle catene ascendenti/discendenti.

- L'insieme $\mathcal{F} = \{f : L \to L \mid \exists l_k, l_g \in L : f(l) = (l - l_k) \cup l_g\}$ è una famiglia di funzioni monotone, contiene l'identità (con l_k e l_g uguali a $\emptyset$) ed è chiuso sotto composizione.
- È possibile definire una relazione di flusso **F** (in questo caso è la funzione pred).
- Possiamo definire l'insieme delle etichette iniziali **E** (nel nostro caso, il nodo iniziale del grafo di flusso di un programma).
- Il valore iniziale ι per le etichette iniziale **E** ($\emptyset$).
- Una funzione che mappa le etichette dei nodi in **F** o **E** nelle corrispondenti funzioni di transfert in $\mathcal{F}$ (la funzione dovrà, data un'etichetta, rilevare un comando di assegnamento e quindi restituire la funzione di transfert associata; quando rileva altri comandi, la funzione corrisponde alla funzione identità).

La terminazione dell'algoritmo che calcola la soluzione minima è assicurata dal fatto che le analisi presentate in questa sezione, sono tutte istanze del general framework (come abbiamo appena dimostrato) e per questo è già stato proposto, per esempio in [4], un algoritmo che calcola in tempo finito la soluzione minima.

3 CP con Program Analyzer Generator

Per poter analizzare programmi più complessi rispetto a quello presentato nella sezione 2.5, è stato scelto di utilizzare uno strumento automatico. Vediamo brevemente in che cosa consiste. PAG/WWW è il porting web di un analizzatore statico di codice sorgente. Nella versione completa è possibile analizzare qualsiasi linguaggio di programmazione, mentre nella versione web, è consentito l'utilizzo solamente del linguaggio WHILE (è lo stesso di quello presentato in 2.1 con l'aggiunta della gestione delle parentesi).

Sono solamente due gli elementi che l'utente deve fornire a PAG/WWW per calcolare i risultati di un'analisi: una *specifica* dell'analisi ed un qualsiasi *codice sorgente*. Un'analisi in PAG consiste di quattro sezioni:

- TYPE, in cui si specificano i tipi di dati nuovi che vengono costruiti a partire dai tipi base (sezione facoltativa);
- PROBLEM, in cui si specifica il framework in cui l'analisi viene eseguita (come per esempio la direzione dell'analisi, il dominio astratto su cui si basa l'analisi, ecc...);
- TRANSFERT, in cui si definiscono le funzioni di transfert in base al tipo di comando presente nel nodo;

- SUPPORT, in cui si definiscono alcune funzioni generiche di supporto che possono essere utilizzate in altre sezioni.

Nella specifica dei nuovi tipi, nella definizione delle funzioni di transfert e nella definizione dei parametri dell'analisi, è necessario utilizzare il linguaggio funzionale FULA, la cui specifica è possibile trovarla ampiamente descritta in [1].

3.1 Implementazione analisi con Pag

Rispetto all'analisi definita nella sezione 2.2, presentiamo il programma FULA creato per effettuare i test. Per una migliore leggibilità non verrà presentato tutto il codice in un unico blocco. Verranno presentate singolarmente le sezioni che lo compone, aggiungendo per ciascuno una breve motivazione delle scelte implementative effettuate. Inoltre, delle due analisi proposte, verrà inserito solo il codice dell'analisi eager, mentre per l'analisi lazy è stato inserito solamente un confronto rispetto alla precedente in quando semplice riduzione.

3.2 Sezione Type e Problem

In questa sezione l'utente ha la possibilità di definire tipi che non sono built-in. Nel caso dell'analisi di copy propagation (eager), è necessario definire un nuovo tipo che rappresenta il reticolo L che è un lift (cioè un reticolo con top e bottom) del powerset della tripla `Var,Var,ListaLabel`. Dato che l'ultimo elemento della tripla della definizione di $\bigtriangledown$ (2, pag 7) è unione di altri elementi, allora si è deciso di usare l'elemento lista al posto del powerset.

```
TYPE
ListaLabel=list(Label)

VarPairLabel = Var * Var * ListaLabel

VarPairLabelSet = set(VarPairLabel)

VarPairLabelSetLifted = lift(VarPairLabelSet)

PROBLEM Copy_Propagation
direction  : forward
carrier    : VarPairLabelSetLifted
init       : top
init_start : lift({})
```

```
combine    : newglb
```

Per quanto riguarda la sezione Problem, come si può immaginare vengono specificate direzione, reticolo su cui si basa l'analisi, valore di inizializzazione degli insiemi entry ed exit, valore dell'entry del nodo iniziale e operatore $\bigtriangledown$ che specifica come l'informazione deve essere combinata nei nodi di confluenza di diversi archi. Per la spiegazione della funzione *newglb* si rimanda alla sezione 3.4. L'analisi lazy creerà un tipo `VarPairLabel` che sarà semplicemente una tripla formata da due variabili ed una label. Per quanto riguarda l'analisi lazy, rispetto a quanto precedentemente asserito, utilizzerà una funzione `glb` (o `intersection`) al posto di `newglb` definita sul carrier adeguato.

3.3 Sezione Tranfer

```
ASSIGN(var, exp) =  let entry <= @ in (

  if (expType(exp) = "VAR") then
    /* variabile:=variabile */
    if (var!=expVar(exp)) then
      lift(entry - kill(entry,var) + gen(var, expVar(exp), label))
    else
      @
    endif
  else
    /*variabile:=espressione*/
    lift(entry-kill(entry,var))
  endif
)
```

Nella sezione Transfert vengono inserite le specifiche delle funzioni di transfert per ogni tipologia di comando. Nel caso dell'analisi CP, è utile trattare solo il caso del comando di assegnamento [`x:=a`]. Se a è uguale a x, quindi $FV(a) \neq \emptyset$, si restituisce direttamente l'entry, senza effettuare nessuna modifica. Viceversa, se a è una variabile, si segue la definizione generale (vedi la definizione dei vincoli dell'analisi). Negli altri casi, si uccidono le variabili copia che sono uguali a x. La versione dell'analisi lazy risulta uguale nella sezione Transfert.

3.4 Sezione Support

In questa sezione sono state inserite le funzioni a supporto dell'analisi di copy propagation. La funzione `newglb` definisce l'operatore di combine nel seguente modo:

```
newglb :: VarPairLabelSetLifted * VarPairLabelSetLifted -> VarPairLabelSetLifted
newglb(bot,_)=bot
newglb(_,bot)=bot
newglb(top,second)=second
newglb(first,top)=first
newglb(first,second)=lift({ e | a in drop(first); b in drop(second);
                      a#1=b#1 && a#2=b#2; let e=(a#1,a#2,merge(a#3,b#3))})

kill :: VarPairSet*Var -> VarPairLabelSet
kill(entry,var) = {varlab | varlab in entry;
                   varlab#1 = var  || varlab#2 = var; varlab#1!=varlab#2}


gen :: Var*Var*Label -> VarPairLabelSet
gen(var, exp,label) = {(var, exp,[label]) | var!=exp}

merge :: ListaLabel * ListaLabel -> ListaLabel
merge([],[])=[]
merge([],l)=l
merge(l,[])=l
merge(h1:t1,h2:t2) = if (isMember(h1,h2:t2)) then merge(t1,h2:t2)
                      else h1:merge(t1,h2:t2) endif

isMember :: Label * ListaLabel -> bool
isMember(l,[])=false
isMember(l,h1:t1)=if (l=h1) then true else isMember(l,t1) endif
```

Per quanto riguarda l'analisi lazy quello che c'è da dire è che contiene le stesse funzioni gen e kill dell'analisi eager, definite su tipi adeguati. Le altre funzioni (`merge` e `isMember`) non sono necessarie perché sono funzioni di supporto a `newglb`.

Per uno scopo puramente didattico e per approfondire la conoscenza dello strumento PAG si è pensato di inglobare nell'analisi la logica che gestisce espressioni complesse, quali ad esempio $[x := y + 0]$ o $[x := 1 * (y + (0 * 0))]$, ma che equivalgono ad assegnamenti $[x := y]$. Come si può immaginare, queste ottimizzazioni fanno parte degli obiettivi delle *peephole optimization* e fanno uso della constant folding. Per ragioni di completezza è stato inserito il codice prodotto nell'appendice A.

4 Testing

Passiamo ora alla fase di testing dell'analisi definita nelle sezioni precedenti. Dato che due sono le analisi proposte, due sano i risultati prodotti. Il programma utilizzato per i testi su PAG/WWW è il seguente:

```
/*  i: */  program test1
/*  0: */  begin
/*  1: */  y:=4;
/*  2: */  a:=b;
/*  3: */  if (x>3) then
/*  4: */      x:=y;
           else
/*  5: */      c:=a+3;
/*  6: */      x:=y;
/*  7: */      k:=3/x;
/*  8: */  c:=4+a*x;
/*  9: */  while (x>3) do
/* 10: */      a:=a-x;
/* 11: */      a:=b;
/* 12: */  x:=x;
/* 13: */  a:=x+1;
/*  f: */  end
```

4.1 Analisi Lazy

Nella tabella 5(b) si può vedere il risultato prodotto da PAG/WWW rispetto al programma presentato precedentemente e nella figura 5 una rappresentazione grafica. Il risultato ci dice che solamente una serie di sostituzioni possono essere effettuate nella zona alta del codice, come per esempio [`c:=a+3`]^5[b/a] e [`k:=3/x`]^7[y/x] o [`c:=4+a*x`]^8[b/a]. È da notare che in quest'ultima sostituzione, è possibile cambiare solo la variabile `a` mentre sarebbe corretto sostituire anche la variabile `x`.

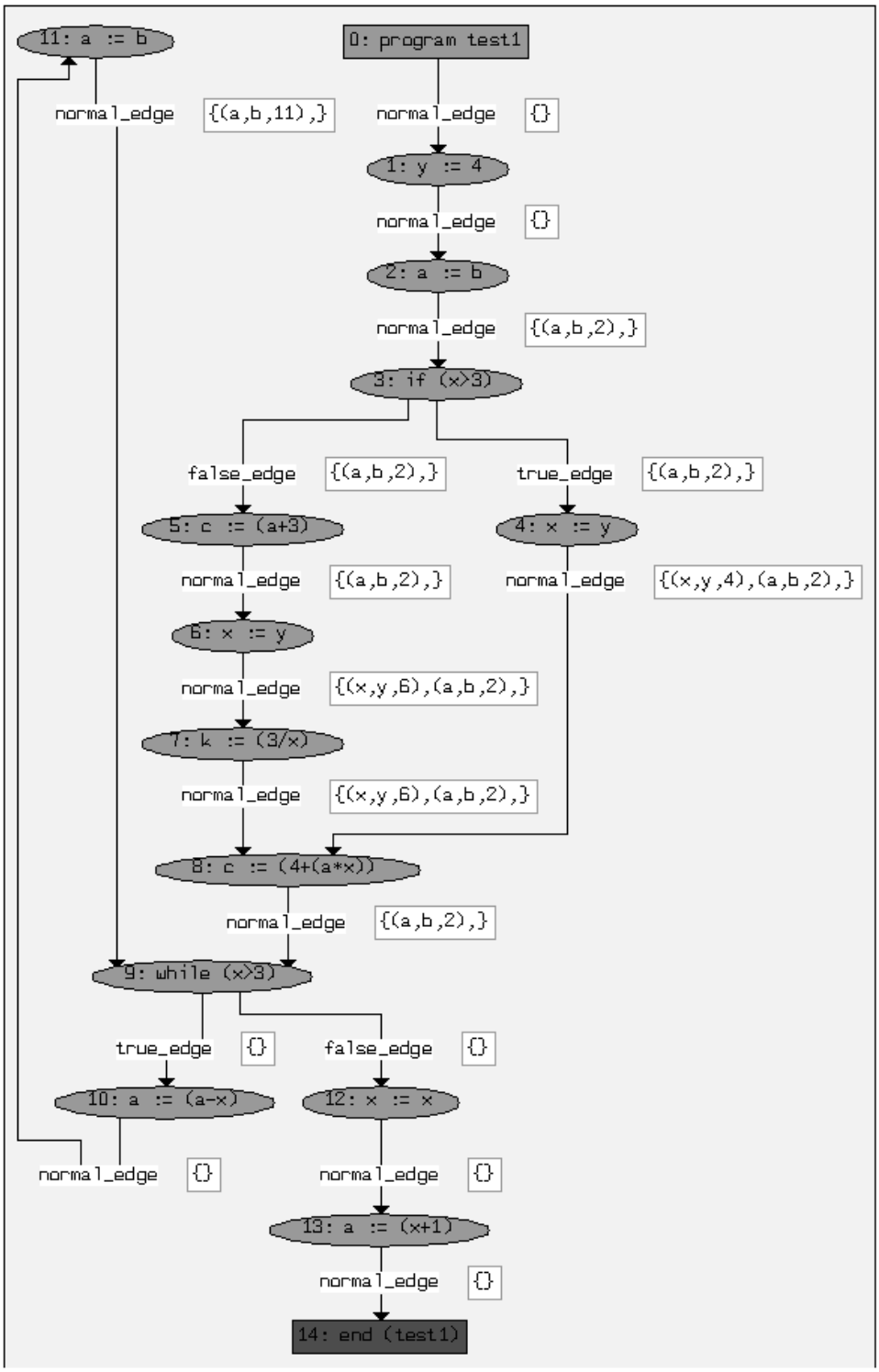


Figura 5: Rappresentazione grafica delle informazioni in entrata ed in uscita ad ogni nodo al termine dell'analisi lazy

4.2 Analisi Eager

Per quanto riguarda l'analisi eager, si vede subito dalla tabella 5 e dalla figura 6 che durante l'analisi sono state raccolte più informazioni rispetto a prima e questo fa dell'analisi eager un'analisi estremamente più precisa, soprattutto nel caso di programmi complessi rispetto a quello utilizzato in questi test. Proviamo a vedere passo passo quali sono i punti in cui l'analisi eager consente una migliore raccolta di informazioni sulla copy propagation.

Fino al nodo 7, le due analisi si comportano allo stesso modo per cui le stesse sostituzioni possono essere effettuate. Il nodo 8 è il nodo di confluenza tra i due rami dell'if. In questo caso, l'analisi eager si comporta meglio, infatti raccoglie l'informazione sulla copy propagation delle variabili x e y che sono state dichiarate al nodo 6 e 4. L'analisi lazy, come ci si poteva aspettare, non propaga l'informazione per cui nei nodi successivi nessuna sostituzione della variabile x è possibile. Per quanto riguarda le altre due variabili copia, a e b, non c'è niente da dire essendo stata dichiarata prima dell'ingresso dell'if.

Si può sottolineare il fatto che in tutte e due le analisi, quando viene rilevato un comando [x:=x], nessuna copia viene uccisa. L'analizzatore si comporta in modo tale da rilevare questa situazione e propagare l'informazione in ingresso direttamente a quella in uscita senza apportare nessuna modifica.

In definitiva, il programma di test può essere riscritto come:

```
/*  i: */  program test1
/*  0: */  begin
/*  1: */  y:=4;
/*  3: */  if (x>3) then
/*  4: */     skip;
           else
/*  5: */     c:=b+3;
/*  7: */     k:=3/y;
/*  8: */  c:=4+b*y;
/*  9: */  while (y>3) do
/* 10: */     a:=a-y;
/* 13: */  a:=y+1;
/*  f: */  end
```

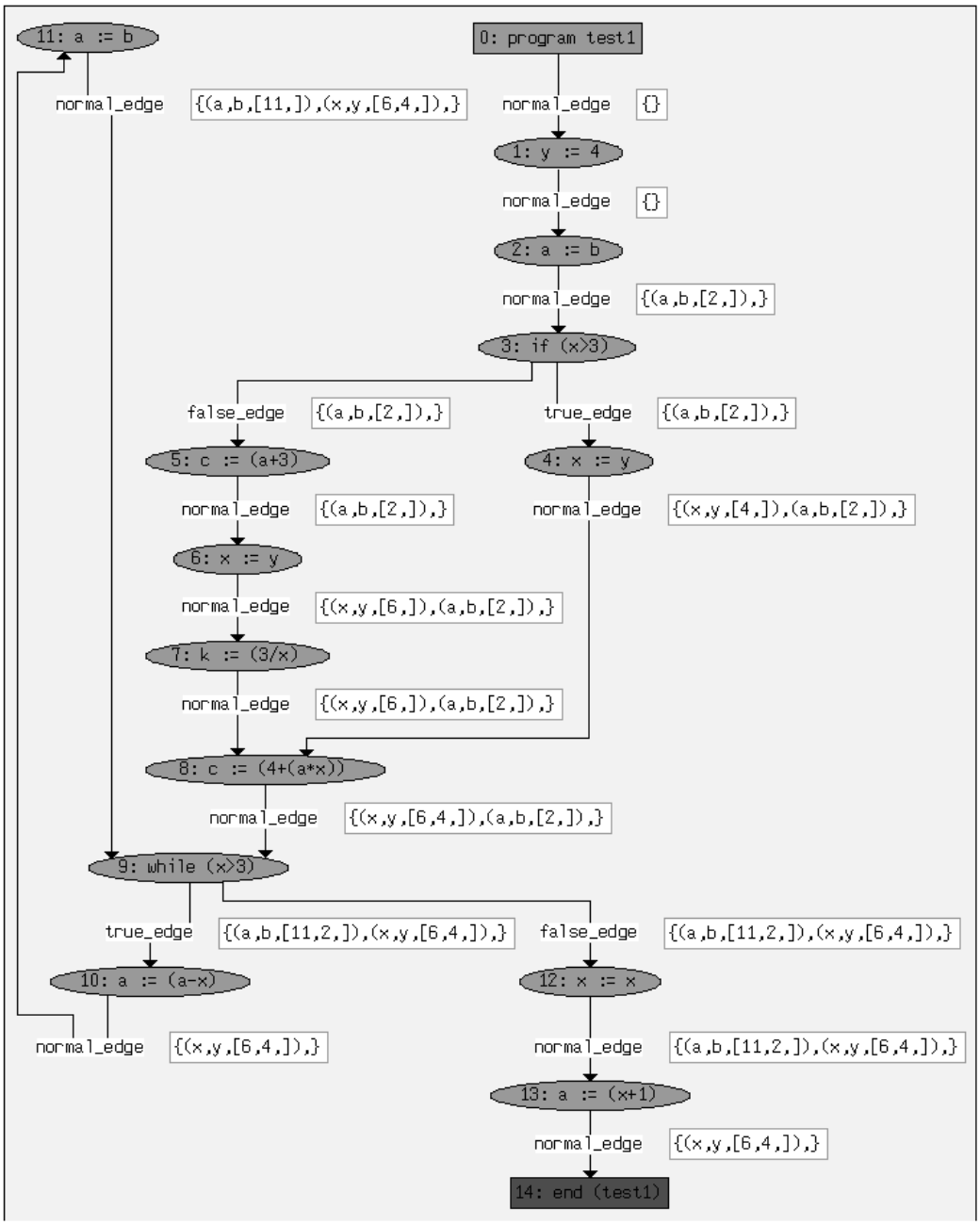


Figura 6: Rappresentazione grafica delle informazioni in entrata ed in uscita ad ogni nodo al termine dell'analisi eager

(a)

nodo
$[y := 4]^1$
$[a := b]^2$
if $[x > 3]^3$
$\quad [x := y]^4$
else $[c := a + 3]^5$
$\quad [x := y]^6$
$\quad [k := 3/x]^7$
$[c := 4 + a * x]^8$
while $[x > 3]^9$
$\quad [a := a - x]^{10}$
$\quad [a := b]^{11}$
$[x := x]^{12}$
$[a := x + 1]^{13}$

(b)

CP_{entry}	CP_{exit}
$\emptyset$	$\emptyset$
$\emptyset$	(a,b,2)
(a,b,2)	(a,b,2)
(a,b,2)	(a,b,2),(x,y,4)
(a,b,2)	(a,b,2)
(a,b,2)	(a,b,2),(x,y,6)
(a,b,2),(x,y,6)	(a,b,2),(x,y,6)
(a,b,2)	(a,b,2)
$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$
$\emptyset$	(a,b,11)
$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$

(c)

CP_{entry}	CP_{exit}
$\emptyset$	$\emptyset$
$\emptyset$	(a,b,{2})
(a,b,{2})	(a,b,{2})
(a,b,{2})	(a,b,{2}),(x,y,{4})
(a,b,{2})	(a,b,{2})
(a,b,{2})	(a,b,{2}),(x,y,{6})
(a,b,{2}),(x,y,{6})	(a,b,{2}),(x,y,{6})
(x,y,{6,4}),(a,b,{2})	(x,y,{6,4}),(a,b,{2})
(x,y,{6,4}),(a,b,{11,2})	(x,y,{6,4}),(a,b,{11,2})
(x,y,{6,4}),(a,b,{11,2})	(x,y,{6,4})
(x,y,{6,4})	(a,b,{11}),(x,y,{6,4})
(a,b,{11,2}),(x,y,{6,4})	(a,b,{11,2}),(x,y,{6,4})
(a,b,{11,2}),(x,y,{6,4})	(x,y,{6,4})

Tabella 5: In (a) codice sorgente di test. Insiemi CP_{entry} e CP_{exit} dell'ultimo run dell'algoritmo per l'analisi di copy propagation eager (c) e lazy (b).

5 Future Work

L'approccio fino ad ora adottato alla copy propagation è stato abbastanza semplice. Quando si pensa ai miglioramenti da apportare, si potrebbero proporre due alternative, anche se in ognuna si tende a fuorviare rispetto alla definizione principale di copy propagation. Si rischia quindi di creare analisi ibride che attingono caratteristiche da diverse altre analisi.

Un possibile miglioramento all'analisi consiste nell'utilizzo di una struttura che permette di rilevare quando un *insieme* di variabili hanno lo stesso valore, per cui L può essere così definito:

$$L = \mathcal{P}(\mathcal{P}(\mathbf{Var}_*) \times \mathcal{P}(\mathbf{Lab}_*))$$

in modo tale che quando troviamo un codice:

$$[\texttt{y:=1}]^1; [\texttt{x:=y}]^2; [\texttt{k:=3}]^3; [\texttt{y:=g}]^4; [\texttt{m:=g*x+3}]^5;$$

si possa rilevare che nel cammino $2 \to 3 \to 4 \to 5$ le variabili $\{x, y, g\}$ hanno lo stesso valore ed ottenere il nuovo codice semanticamente equivalente:

$$[\texttt{y:=1}]^1; [\texttt{k:=3}]^3; [\texttt{m:=y*y+3}]^5;$$

In alternativa, si potrebbe pensare all'utilizzo del seguente dominio astratto:

$$L = \mathcal{P}(\mathcal{P}(\mathbf{Var}_*) \to \mathbb{Z}^\top)_\perp$$

in modo tale da rilevare quando più variabili hanno lo stesso valore (e quindi sono delle copie) ed identificare anche il valore corrispondente (in un certo senso, questa struttura riprende quella utilizzata per la constant propagation). Questa analisi, a differenza di quella presentata nella sezione 2 dato che considera i valori delle variabili è un esempio di problema non distributivo la cui risoluzione risulta in qualche modo più difficile.

6 Conclusioni

L'analisi di copy propagation sviluppata con PAG/WWW ha permesso di capire lo stato dell'arte per quanto concerne gli strumenti automatici di analisi dataflow e la facilità con cui data una proprietà da osservare sia facile definire un'analisi completamente automatizzabile.

Riferimenti bibliografici

[1] Experiencing Program Analysis. Pag/www. `http://pag.cs.uni-sb.de/`.

[2] Agostino Cortesi. Dispense del corso di analisi e verifica di programmi. http://www.dsi.unive.it/~avp.

[3] Flemming Nielson and Hanne Riis Nielson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, 1999.

[4] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

A Appendice

```
TRANSFER
ASSIGN(var, exp) =  let entry <= @ in (

if (expType(exp) = "VAR") then
  /* variabile:=variabile */
  if (var!=expVar(exp)) then
    lift(entry - kill(entry,var) + gen(var, expVar(exp), label))
  else
    @
  endif

else
  /*variabile:=espressione*/
  if (noVariables(exp)>=2) || (noVariables(exp)=0)then
    lift(entry-kill(entry,var))
  else
    /* c'e' una sola variabile */
    if eval(var,exp)=false then
      if (var=parsingVariable(onlyVariable(exp))) then
        lift(entry - kill(entry,var))
      else
        lift(entry - kill(entry,var)
        + gen(var, parsingVariable(onlyVariable(exp)), label))
      endif
    else
      @
    endif
  endif
endif)

SUPPORT
/* questa funzione restituisce true se l'espressione passata come parametro
è riconducibile ad un comando x:=y */
eval :: Var * Expression -> bool
eval(var, exp) = case expType(exp) of
```

```
      /*x:=x, caso base*/
      "VAR" => (var=expVar(exp));
      "ARITH_BINARY" => case expOp(exp) of
             "+" =>
                if ((expType(expSubLeft(exp))="CONST")
                    && ( expVal(expSubLeft(exp))=0 )) then
                    eval(var, expSubRight(exp))
                else
                  if ((expType(expSubRight(exp))="CONST")
                    && ( expVal(expSubRight(exp))=0 )) then
                    eval(var,expSubLeft(exp))
                  else
                    eval(var,expSubLeft(exp)) && eval(var,expSubRight(exp))
                  endif
                endif;
             endcase;
      "CONST" => (expVal(exp)=0);
      endcase

/* calcola il numero di variabili presenti in una espressione */
noVariables :: Expression -> snum
noVariables(exp)=case expType(exp) of
      "VAR"  => 1;
      "CONST" => 0;
      "TRUE" => 0;
      "FALSE" => 0;
      "ARITH_BINARY" => noVariables(expSubLeft(exp))+noVariables(expSubRight(exp));
      "ARITH_UNARY" => noVariables(expSub(exp));
      "BOOL_BINARY" => noVariables(expSubLeft(exp))+noVariables(expSubRight(exp));
      "BOOL_UNARY" => noVariables(expSub(exp));
endcase

/* restituisce la prima variabile trovata nell'espressione */
/*vincolo: deve esserci una sola variabile */
onlyVariable :: Expression -> ListOfVariables
onlyVariable(exp)=case expType(exp) of
      "VAR"  => expVar(exp):[];
      "CONST" => [];
      "TRUE" => [];
      "FALSE" => [];
      "ARITH_BINARY" => if LeastOne(expSubLeft(exp))=true then
                           onlyVariable(expSubLeft(exp))
                        else
                           onlyVariable(expSubRight(exp))
                        endif;
      "ARITH_UNARY" => onlyVariable(expSub(exp));
endcase
```

```
/* restituisce l'elemento in testa della lista */
/* vincolo: deve esserci una variabile sola */
parsingVariable :: ListOfVariables -> Var
parsingVariable(hd1:[])=hd1

/* restituisce true se l'espressione passata come parametro ha almeno una variabile */
LeastOne :: Expression -> bool
LeastOne(exp)=case expType(exp) of
      "VAR"  => true;
      "CONST" => false;
      "TRUE" => false;
      "FALSE" => false;
      "ARITH_BINARY" => LeastOne(expSubLeft(exp)) || LeastOne(expSubRight(exp));
      "ARITH_UNARY" => LeastOne(expSub(exp));
      "BOOL_BINARY" => LeastOne(expSubLeft(exp)) || LeastOne(expSubRight(exp));
      "BOOL_UNARY" => LeastOne(expSub(exp));
endcase
```