Barletta Massimiliano 802646
Gazzi Renzo 802479

Putting Static Analysis to Work for Verification: A Case Study

T. Lev-Ami, T. Reps, M. Sagiv, R. Wilhelm

Obiettivi

- Provare la correttezza parziale di programmi corretti
- Scoprire, individuare e diagnosticare bugs in programmi non corretti

- IDEA: Algoritmo che analizza procedure di ordinamento che manipolano liste
- Cenni sul prototipo: TVLA

Introduzione (I)

Si vuole dimostrare che:

1. versioni corrette di bubble-sort e insertion-sort producono liste ordinate
2. La proprietà della lista "essere ordinata" è mantenuta da queste operazioni:
   - Inserzione di un elemento
   - Cancellazione di un elemento
   - Merge di due liste ordinate
   - Inversione di una lista ordinata

Introduzione (II)

- Dimostriamo inoltre che:
  - E' possibile creare algoritmi di analisi sufficientemente precisi tale da poter stabilire la correttezza parziale attraverso l'informazione contenuta in "state-descriptors" che si ottengono con analisi statica

Shape Analysis (I)

- Per i nostri obiettivi si utilizza una versione estesa di Shape Anaysis

  (Parametric Shape Analysis via 3-valued logic):

  - Determina l'informazione circa le strutture dati allocate nello heap alle quali le variabili possono far riferimento (strutture dati allocate dinamicamente)
  - Offre un metodo per generare diversi tools di program analysis (Prototipo TVLA).
  - Per i nostri esempi "shape-descriptors" mantengono un'informazione relativa all'ordinamento rispetto al campo valore di elementi vicini della lista.

Shape Analysis (II)

- Difficoltà:

  – Aggiornamenti tramite puntatori
    - Es. **`x` → `next = y`**

  – Allocazione dinamica della memoria
    - Non è noto un limite superiore alla dimensione delle strutture presenti a run time.

Dichiarazione tipo lista

Ordinamento

```
/* list.h */
typedef struct node {
  int d;
  struct node *n;
} *L;
```


2-valued logical structure

Strutture logiche per rappresentare stati di memoria.

Una struttura logica consiste di:

1. Un universo di elementi (heap cells);
2. Una famiglia di predicati base.

Una struttura logica a 2 valori S è l'insieme di nodi (individuals) chiamati universo (U^S) e l'interpretazione per un insieme di simboli predicati su questo universo. L'interpretazione di un simbolo predicato p in S è denotato con p^S. Per ciascun predicato p di arietà k, p^S è una funzione

$$P^S : (U^S)^k \rightarrow \{0,1\}$$

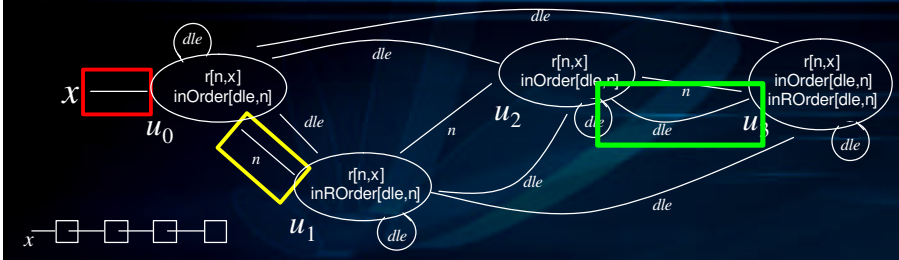
Core e Instrumentation

- L'insieme dei simboli predicati è diviso in due parti disgiunte:
  1. **Core**: parte della semantica a puntatori. Memorizzano proprietà atomiche dello stato di memoria;
  2. **Intrumentation**: memorizzano proprietà derivate. Hanno una formula in termini di predicati core.

Core predicates (I)

- x(u): vale 1 se una variabile x punta ad un elemento della lista rappresentato da u. Corrisponde all'arco che va da x all'elemento. In assenza dell'arco x(u)=0;
- n(u1,u2): vale 1 se u2 è successore di u1 o detto in altro modo se la componente *n* di u1 punta a u2;
- dle(u1,u2): vale 1 se il valore *d* di u1 <= valore d di u2.

Core predicates (II)



	$x(u)$
u_0	1
u_1	0
u_2	0
u_3	0

n	u_0	u_1	u_2	u_3
u_0	0	1	0	0
u_1	0	0	1	0
u_2	0	0	0	1
u_3	0	0	0	0

dle	u_0	u_1	u_2	u_3
u_0	1	1	0	0
u_1	0	1	0	0
u_2	1	1	1	1
u_3	1	1	0	1

Instrumentation predicates (I)

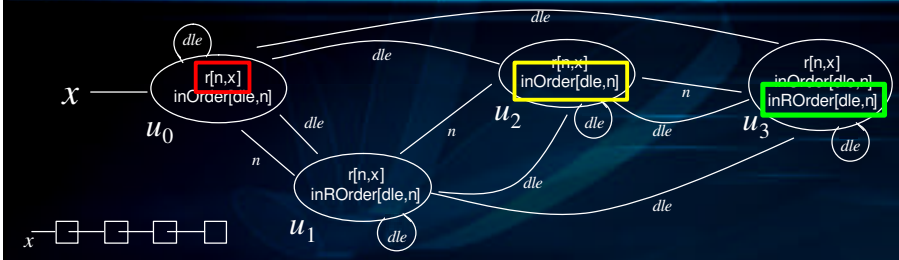
- r[n,x](u): vale 1 se un elemento della lista è raggiungibile da x possibilmente usando una sequenza di accessi attraverso la componente *n* dell'elemento
- c[n](u): vale 1 se l'elemento fa parte di una lista ciclica
- is[n](u): vale 1 se più puntatori fanno riferimento all'elemento

Instrumentation predicates (II)

Per rappresentare l'ordinamento degli elementi sono stati introdotti due ulteriori instrumentation predicates:

- inOrder[dle,n](u): vale 1 se il valore del campo *d* è minore o uguale a quello dei successori
- inROrder[dle,n](u): vale 1 se il valore del campo *d* è maggiore o uguale a quello dei predecessori

Instrumentation predicates (III)



	r[n,x]	*is[n]*	*c[n]*	*inOrder[dle,n]*	*inROrder[dle,n]*
u_0	1	0	0	1	0
u_1	1	0	0	0	1
u_2	1	0	0	1	0
u_3	1	0	0	1	1

3-valued logical structure

- In maniera equivalente alla 2-valued logical structure esposta precedentemente, possiamo definire una 3-valued logical structure.

  Utilizziamo strutture logiche a 3 valori per rappresentare stati di memoria presenti durante l'esecuzione

  $p^S: (U^S)^K \rightarrow \{0,1,1/2\}$

  1 e 0 sono valori definiti (vero e falso)

  1/2 è indefinito

Rappresentazione conservativa (I)

Sia $S^\#$ una struttura a due valori, S una struttura a 3 valori e sia f una funzione suriettiva:
$f: U^{S\#} \rightarrow U^S$

- diciamo che f include $S^\#$ in S per ogni predicato p di arietà k e $u_1, \ldots, u_k$ appartenenti a $U^{S\#}$ sia se

  $p^{S\#}(u_1, \ldots, u_k) = p^S(f(u_1), \ldots, (u_k))$

  o

  $p^S(f(u_1), \ldots, (u_k)) = 1/2.$

- diciamo che S rappresenta tutte le strutture a 2 valori che possono essere "embedded" in esso per mezzo di qualche funzione f. S quindi può rappresentare in maniera compatta molte strutture dati.

Summary nodes

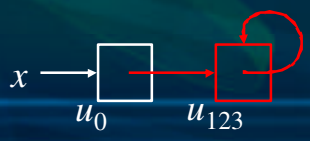
Nodi in strutture 3-valued che possono rappresentare più nodi di una struttura 2-valued.

S_m è il predicato che usiamo per rappresentare questa informazione.

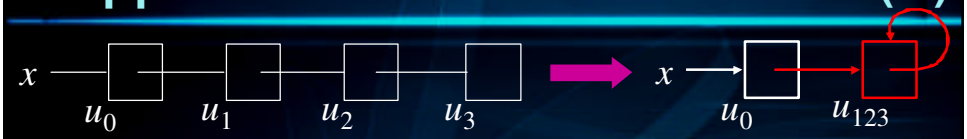
Dato un summary node *w*,

- $S_m(w)=1/2$ indica che il nodo può rappresentare più di un nodo della struttura a 2 valori.

- $S_m(w)=0$ indica che *w* rappresenta (con certezza) un solo nodo.



Rappresentazione conservativa (II)



	$x(u)$
u_0	1
u_1	0
u_2	0
u_3	0

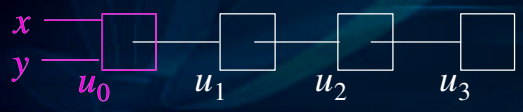
	$x(u)$
u_0	1
u_{123}	0

n	u_0	u_1	u_2	u_3
u_0	0	1	0	0
u_1	0	0	1	0
u_2	0	0	0	1
u_3	0	0	0	0

n	u_0	u_{123}
u_0	0	1/2
u_{123}	0	1/2

Formule

- Permettono di osservare le proprietà delle strutture: es. $\exists v$: $x(u) \wedge y(u)$ $= 1$



	$x(u)$	$y(u)$	$t(u)$
u_1	1	1	0
u_2	0	0	0
u_3	0	0	0
u_4	0	0	0

Formule

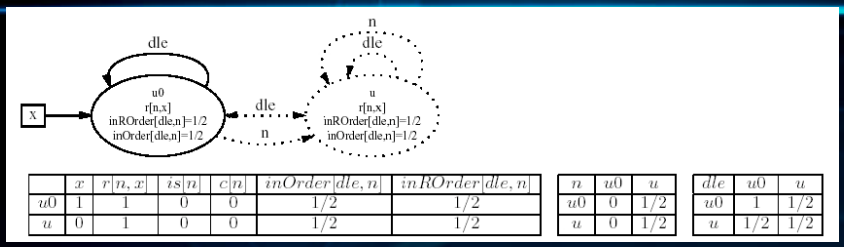
- Esempi di applicazione delle formule
  - L'elemento puntato da *x* è diverso da NULL?
    $\exists u$: $x(u)$
  - Le variabili *x* ed *y* puntano allo stesso elemento?
    $\exists u$: $x(u) \wedge y(u)$
  - L'elemento a cui fa riferimento la variabile *x* ha un riferimento a se stesso? (riferimento ciclico)
    $\exists u$: $x(u) \wedge n(u,u)$

Formule

Definizione:

Una formula ? è potenzialmente soddisfatta su una struttura S se esiste un assegnamento per cui ? è valutata 1 o 1/2 su S

Esempio



	x	$r[n,x]$	$is[n]$	$c[n]$	$inOrder[dle,n]$	$inROrder[dle,n]$
$u0$	1	1	0	0	1/2	1/2
u	0	1	0	0	1/2	1/2

n	$u0$	u
$u0$	0	1/2
u	0	1/2

dle	$u0$	u
$u0$	1	1/2
u	1/2	1/2

Valutando la formula: $\exists a$: ($x(a) \wedge n^*(a,b)$

u_0 raggiungibile da x per cui la formula sarà valutata 1
u (summary node) non è puntato direttamente da x; il nodo rappresentato da u può o non può essere raggiunto da x, per cui 1/2

Il risultato finale è dunque $1 \wedge 1/2 = 1/2$

Ma r[n,x](u) = 1 che è piu' preciso. L'informazione memorizzata per un predicato di tipo instrumentation può essere più precisa del risultato della valutazione della formula. (Instrumentation principle)

Predicate-update formulae

- Ad ogni riga di codice è associata una predicate-update formulae che specifica come cambia il valore di un predicato quando quel codice viene eseguito, cioè i cambiamenti delle strutture dati in memoria
- Le p-u formulae sono date e ne hanno dimostrato la correttezza

Condition	Precondition formula
`x == y`	$\exists v : x(v) \wedge y(v)$
`x != y`	$\neg\exists v : x(v) \wedge y(v)$
`x == NULL`	$\neg\exists v : x(v)$
`x != NULL`	$\exists v : x(v)$
`x->d <= y->d`	$\exists v_1, v_2 : x(v_1) \wedge y(v_2) \wedge dle(v_1, v_2)$
`x->d == y->d`	$\exists v_1, v_2 : \begin{array}{l} x(v_1) \wedge y(v_2) \\ \wedge\ dle(v_1, v_2) \wedge dle(v_2, v_1) \end{array}$
`x->d < y->d`	$\exists v_1, v_2 : x(v_1) \wedge y(v_2) \wedge \neg dle(v_2, v_1)$
uninterpreted	$1/2$

Procedura parzialmente corretta

Definizione

Una procedura di ordinamento è parzialmente corretta, se ogni volta che termina, la lista in output che essa produce è in ordine non decrescente

Valutazione della correttezza (I)

Ordinamenti con liste

- **Per ogni u: r[n,x](u) → inOrder[dle,n](u) (1)**
  Se la formula è valutata 1 allora i nodi raggiungibili da *x* devono essere in ordine non decrescente.

PROBLEMA:

Una procedura di ordinamento che restituisce sempre NULL sarà soddisfatta.

Per evitare ciò, è necessario che sia soddisfatta anche una seconda proprietà: la lista in output deve essere una permutazione della lista in input.

Utilizziamo un altro predicato:

- orig[n,x](u): indelebile marcatore sugli elementi inizialmente raggiungibili da x.

Valutazione della correttezza (II)

Per ogni u: orig[n,x](u)<->r[n,x](u) (2)

- Se la formula è valutata 1 allora gli elementi raggiungibili da *x* dopo l'esecuzione della procedura sono esattamente gli stessi di quelli raggiungibili all'inizio della procedura, e di conseguenza la procedura effettua una permutazione degli elementi

Nota: in un programma può essere paragonato ad una variabile ausiliaria utilizzata per denotare l'iniziale valore di un'altra variabile

Esempio: Insertion_Sort (I)

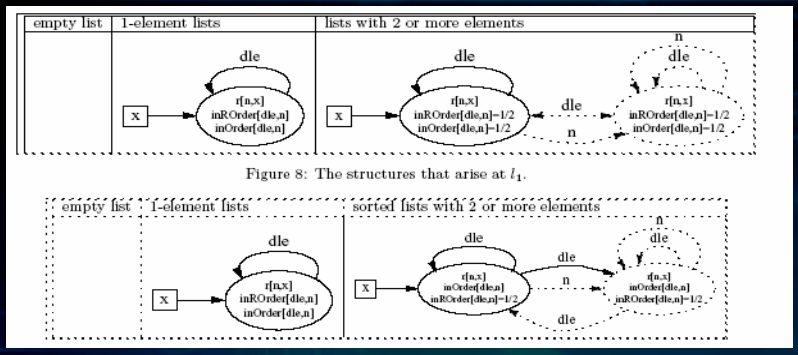


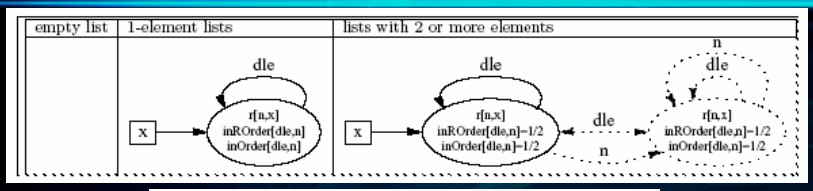
Figure 8: The structures that arise at l_1.

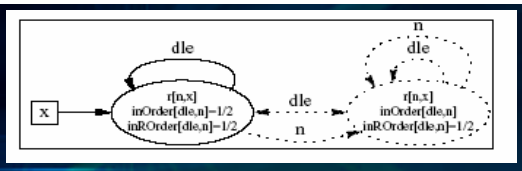
Le strutture descrivono tutti i possibili stati di memoria in cui la variabile *x* punta ad una lista aciclica. Le formule descritte precedentemente sono valutate 1 per cui la procedura insertion_sort agisce correttamente su tutti gli input accettabili

Compile-time debugging of programs

- Dimostriamo come l'output dell'algoritmo, quando applicato a programmi non corretti, provveda a dare informazioni utili per catturare bugs a tempo di compilazione.
- Comuni errori:
  - dimenticare il confronto degli elementi agli estremi
  - lista ordinata non secondo l'ordine specificato

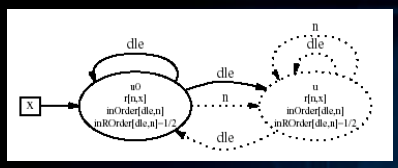
Esempio: Insertion_Sort (II)



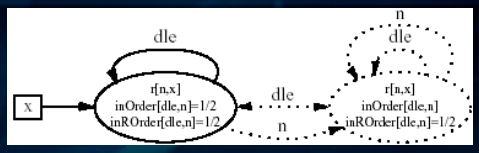


- Le due immagini rappresentano lo stato iniziale (sopra) e quello finale (sotto) di una procedura di insertion_sort ERRATA. Non considera infatti il primo elemento della lista

Esempio: Insertion_Sort (III)



Corretta



Errata

- Questa struttura offre i seguenti indizi circa la natura del bug:
- inOrder[dle,n]=1 per sn, significa che molti elementi della lista potrebbero non essere nell'ordine corretto.
- inOrder[dle,n]=1/2 per il primo nodo, indica che può essere o non essere nella posizione corretta rispetto al resto degli elementi della lista.

Prototipo: TVLA

- Versione del sistema in JAVA
  - Correntemente:
    - Gestisce programmi che usano puntatori
    - Supporta solo analisi intraprocedurale, cioè non sono permesse chiamate a funzioni o procedure
    - Non supportati casting e aritmetica dei puntatori
    - Analizza solo piccoli programmi
- Per questa analisi è stato utilizzato questo programma

Correttezza parziale

- Per dimostrare la correttezza parziale di operazioni su ADT, l'utente deve fornire al TVLA:
  - il grafo di flusso della procedura (control flow)
  - un insieme di strutture 3-valued che caratterizzano gli input accettabili della procedura (input validi)
  - formule che caratterizzano gli output accettabili di una procedura corretta (formule)

Work in progress

- Possibili sviluppi per generare automaticamente formule di predicate-update corrette per instrumentation predicates
- Estendere TVLA, superando le limitazioni attuali specie sulla dimensione dei programmi oggi supportati
- Estendere questo tipo di analisi a qualsiasi tipo di struttura non limitandosi ad un particolare datatype