Programmazione "type" Safe

Angelo Gargantini

Materiale:

capitolo 6 del Mitchell + questi lucidi

Introduzione

- programmazione “safe” generalmente si intende come “type safe”.
- **tipo** = insieme di valori omogenei + operazioni che si possono fare
  - esempi: Integers, String,
    - int -> bool (*funzione* che da un intero mi dà un boolean)
  - esempi di non tipi: numeri dispari, array contenenti String e Integer, ...
  - dipende però dal linguaggio di programmazione

Ma cosa servono i tipi nei linguaggi di progr.?

1. organizzare e dare un nome ai concetti (documentazione)
   - spesso corrispondenti ai concetti nel dominio del problema che si vuole risolvere
   - indicare l'uso che si vorrà fare di certi identificatori (così il compilatore può controllare)
2. assicurarsi che sequenze di bit in memoria siano interpretate correttamente
   - per evitare errori come: `3 + true + "Angelo"`
3. dare informazioni al compilatore come manipolare i dati
   - ad esempio gli short richiedono meno bit dagli integer
   - se dichiaro un record, il compilatore saprà come accedere agli elementi del record

Tipi di errori: Hardware error

- confondere dati con i programmi
  - esempio: chiamata `x()` dove `x()` non è una procedura ma è un intero
- confondere tipi di dati semplici
  - esempio `float_add(3,4.5)`
    - `float_add`: operazione della CPU che chiama una routine della FPU
    - 3 sequenza di bit per rapprsentare 3: 0000 0000 0000 0011
    - 4.5 sequenza di bit per 4.5: 0000 0001 1100 0000 00001 0000 0000 ...
    - se la CPU prende 3 come sequenza di bit float, genera un errore hardware

Errori semantici

- Il programma fa qualcosa che non è quello che dovrebbe fare
  - esempio: `int_add(3,4.5)` 4.5 può essere interpretato come int ma cosa sarà uguale
- con struct, Object
  - esempio ereditarietà: `class A extends B`
  - `A a = new B()` **OK**
  - `B b = new A()` **NO**: b ha dei metodi in più che potrei invocare ma non trovare perchè b è un A

Type safety: sicurezza dei tipi

Un linguaggio di programmazione L si dice *type safe* se non esiste programma scritto in L che possa violare la distinzione di tipi in L

- confondere interi e float
- chiamare una fuzione attraverso un intero
- accedere ad una zona di memoria sbagliata

Safety	languaggio	spiegazione
Not safe	C and C++	Type casts, pointer arithmetic
Almost safe	Pascal	Explicit deallocation; dangling pointers
Safe	Lisp, ML, Smalltalk, Java	Complete type checking

Errori nei linguaggi non type safe

Type cast: permettere la conversione *non controllata* da un tipo ad un altro. In particolare da intero ad una funzione e cercare di eseguire una certa locazione di memoria che potrebbe non essere una istruzione corretta o fare qualcosa di non voluto

Pointer arithmetic: l'espressione *(p+i) ha tipo A se p è definito di tipo A*.
poichè il valore memorizzato a p+i potrebbe avere qualsiasi tipo,
l'assegnamento x = *(p+i) permette di memorizzare un valore di un tipo in un altro

C non è *memory safe*

```
void f(int* p, int i, int v) {

    p[i] = v;
}
```

- accedo all'indirizzo p+i
- p+i potrebbe contenere dati importanti o altro codice
  - posso modificare il return address di una chiamata di una procedura ed eseguire altro codice,
  - posso modificare dei diritti o leggere informazioni riservate

Deallocazione esplicita e Dangling Pointers.

- In Pascal, C, ... una locazione puntata da un putatore può essere deallocata dal programmatore: questo crea un “dangling pointer”
- esempio se p è un puntatore, dopo che viene deallocata la memoria puntata da p, il programma potrebbe allocare la stessa memoria nuovamente per memorizzare un altro tipo di valore (ad esempio una stringa). Posso continuare ad usare p per accedere a questa memoria e rompere la type safety
- altro esempio: uso di puntatori a valori sullo stack

Esempio

Funzione che converte un intero in stringa:

```
char * itoa(int i){
    char buf[20];
    sprintf(buf,"%d",i);
    return buf;
}
```

Dove è sbagliato?

Esempio

Funzione che converte un intero in stringa:

```
char * itoa(int i){
    char buf[20];
    sprintf(buf,"%d",i);
    return buf;
}
s
```

- dove è buf ? buf ora punta ad una zona dellocata

Esempio2: allocazione sullo stack in C/C++

```
struct Point {int x; int y;};
struct Point *newPoint(int x,int y) {
   struct Point result = {x,y};
   return &result;
}
void bar() {
    struct Point *p = newPoint(1,2);
    p->y = 1234;
}
```

Dove è sbagliato?

Esempio: allocazione sullo stack in C/C++

```
struct Point {int x; int y;};
struct Point *newPoint(int x,int y) {
   struct Point result = {x,y};
   return &result;
}
void bar() {
    struct Point *p = newPoint(1,2);
    p->y = 1234;
}
```

- Errore: p punta ad una zona di memoria che è stata liberata all'uscita da newPoint
- bisogna usare la malloc:
  `struct Point * result = malloc(sizeof(struct Point))`
- non sempre ci si accorge di questi errori

Esercizio

1. scrivi un codice in C che dà errore in esecuzione dovuto all'uso improprio dei puntatori (dell'aritmetica, dereferenziazione di null o simili)
2. scrivi un codice simile all'esempio precedente con un errore dovuto a dangling pointer ed eseguilo. Fai in modo di effettivamente osservare l'errore: ad esempio accedete al dangling pointer dopo aver riutiliazzato la memoria. (vedi esempio2.c)
3. riscrivi il codice di cui al punto 2 senza l'errore e osserva che il risultato è corretto. (vedi esempio3.c)

Compile time vs run time type checking

Tra i linguaggi **safe**, distinguiamo

- Lisp usa run-time (l'esecuzione) type checking
  - `(car x)` controlla prima che x sia una lista (`car` restituisce il primo elemento di una lista)
- ML usa compile-time type checking
  - f(x) è ok se f : A -> B e x : A

Java

- Java usa compile time
  - considera la seguente istruzione
    `T a = (Z) expr`
  - javac controlla durante la compilazione che `Z` sia un (sottotipo di) `T`
  - durante la compilazione introduce però anche un controllo da fare durante l'esecuzione che `expr` sia covertibile in tipo `Z`
  - posso sempre sapere il tipo di un oggetto mediante instanceof

Esercizio

Scrivi in Java

1. una conversione di tipo corretta sia in compilazione che in esecuzione
2. una conversione sempre sbagliata
3. una conversione corretta in compilazione che però dia errore in esecuzione

Pro e contro

Entrambi gli approcci (run-time e compile time) prevengono errori di tipo, però:

- Run-time checking rallenta l'esecuzione
- Compile-time checking limita la flessibilità dei programmi
  - Lisp list: elementi possono avere diversi tipi
  - java array: tutti gli elementi devono avere lo stesso tipo (dichiarato)
- ***nota: alcuni programmi run time corretti non lo sono compile time***

run time chkng scarta solo i programmi erronei

In Lisp, possiamo scrivere

```
(cond ((< x 10) x) (else (car x)))
```

- alcune volte ci sara' errore altre no - se x e' una lista

In Java, se scrivo

```
int x;
if (0>-1) { x++;} else { x = "ciao";}
```

- con compile time t.c. da' errore di tipo perchè assegna ad x int una String
- eppure questo programma è type safe, perchè nessuna esecuzione causa errori di tipo (0 è sempre > -1)
- posso scrivere un compile type checker che accetti tutti i programmi type safe a run-time?
  (ad esempio non scarti il programma Java sopra)

compile time chkng è conservativo

La risposta è no: il compile time tc è sempre conservativo: per quanto intelligente dovrà scartare dei programmi che in esecuzione non darebbero problemi

- fondamenti teorici: problema dell'indecidibilità
- intuitivamente:

```
if (espressione-MOLTO-complicata)
then (expression-with-type-error)
else (no-type-error)
```

- Per decidere se si verifica errore o no il compilatore dovrebbe essere in grado di risolvere `espressione-MOLTO-complicata` (es: x^2 >0, x^2 -5xy +y^2 = 10, x^5 +y^5 >55, ...)

Cosa fare per avere safe software?

Q Se vogliamo scrivere codice safe cosa possiamo fare?

A Scrivere attentamente, progettare prima, documentare, etc.

Q Se vogliamo essere sicuri che il nostro codice è safe?

A Due soluzioni possibili

1. usare linguaggi type safe (Java, lisp;..) linguaggi + astratti
2. usare C (like) e dei tools che ci aiutano a rendere i programmi C safe

Prezzo da pagare per usare linguaggi tipo Java

1. Performance overhead
   - array-bound checks, garbage collection
2. Memory overhead:
   - Dynamic typing information
3. Type annotations:
   - Verbose types and compiler hints
4. Efforts of porting legacy C code:
   - Similarity to C: syntax, grammar, code organization, data representation, etc

Vantaggio del C

C ancora usato per molte cose
OS, device driver, ...

- Performance
- Explicit memory management
- Control over low-level data representation

Come rendere C safe?

- usare librerie safe:
  - http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx
  - http://www.zork.org/safestr/
- tools per l'analisi statica e dinamica per trovare safety violations
  - esempio: purify, ...
- tools e tecniche per prevenire safety violation
  - making C programs safe: SafeC, CCured
  - safe variants of C: Cyclone, Vault

SafeC

approccio traduttore da C a C

input programmi C (qualsiasi)

output programmi C sicuri

metodo garantisce cattura le violazioni di memoria e vari errori run time inserendo dei controlli e aggiungendo informazioni (ad esempio ai puntatori)

pro si applica a C code già esistente

contro rallenta l'esecuzione e aumenta la memoria necessaria

CCured

approccio traduttore da C a C

input programmi C con annotazioni particolari (opzionali)

output programmi C sicuri

metodo abbina analisi statica, controlli dinamici e garbage collector

pro si applica a C code già esistente con minime modifiche

contro rallenta l'esecuzione

Cyclone

approccio safe C-like language

input programmi C modificati

output programmi C sicuri

metodo controlli dinamici solo dove necessario e garbage collector

pro minimo overhead di tempo e di memoria

contro richiede di modificare i programmi originali

Sommario Introduzione

1. concetto di type safe
2. C non è type safe: aritmetica dei puntatori, conversioni, dangling pointers
3. linguaggi type safe possono controllare il tipi a compile-time o a run-time
4. il controllo compile-time è sempre più severo
5. diversi modi per usare linguaggi come C in modo sicuro

Cyclone - Materiale

- queste slides
- la documentazione che si trova sul sito di cyclone: http://www.research.att.com/projects/cyclone/
- sul sito web c'è un manuale in pdf

Installazione per linux - opzionale

1. scaricare i sorgenti dal sito di cyclone
   - meglio prendere l'ultima versione dal repository cvs:

     ```
     cvs -d
     :pserver:anonymous@cvs.eecs.harvard.edu:/home/cyclone/
     co cyclone
     ```
2. (se non si usa cvs, scompattare con tar zxf)
3. aprire una shell e fare cd nelle directory dove c'è cyclone
4. ./configure
   (a) se vuoi metterlo in una directory particolare esistente, usa ./configure –prefix=~/cyclone
5. make
6. make install

Installazione per windows

come per linux però:

- installare cygwin (http://www.cygwin.com/)
- in windows salvarli in una directory senza spazi

Eseguire cyclone

- aggiungi la directory cyclone/bin nel tuo path o chiama cyclone con tutto il percorso
- compilatore e altri tool
- il il compilatore funziona più o meno come un normale compilatore:

```
cyclone opzioni <program.cyc>
-help Print a short description of the
      command-line options.
-v Print compilation stages verbosely.
--version Print version number and exit.
-o file Set the output file name to file.
-O Optimize.
-O2 A higher level of optimization.
-O3 Even more optimization.
...
```

Primo Esempio

Se vogliamo compilare il solito esempio in C:

```
#include <stdio.h>
int main() {
    printf("hello, world\n");
    return 0;
}
```

Per compilarlo ed eseguirlo, salva il programma in un file hello.cyc, ed esegui

```
cyclone -o hello hello.cyc
```

Nota che il return in Cyclone è obbligatorio

Puntatori

- Cyclone accetta il normale uso dei puntatori:

```
#include <stdio.h>
int main() {
    int x = 3;
    int *y = &x; // y punta alla cella di y
    *y = *y + 1;
    printf("%d\n",x);
    return 0;
}
```

in Cyclone è OK: che rischi ci sono?

Situazioni in cui l'uso dei puntatori è "unsafe"

1. assegno un int ad un puntatore e faccio puntare quello che voglio
2. con l'aritmetica porto il puntatore fuori memoria

NO Cast da int a puntatore

- in C posso scrivere, compilare ed eseguire

```
int main(){
    char * PTR; PTR = 1000; *PTR = 'a';
}
code/cyclone> gcc -o error1 error_1.c
error_1.c: In function 'main': error_1.c:4:
warning: assignment makes pointer from integer
                without a cast
code/cyclone> error1
Segmentation fault
```

- In Cyclone NO: *you can't cast an integer to a pointer. Cyclone prevents this because it would let you overwrite arbitrary memory locations. In Cyclone, NULL is a keyword suitable for situations where you would use a (casted) 0 in C.*

NO aritmetica dei puntatori

- in C posso scrivere

```
int main(){
    int x[100];
    int * ptr = x;
    ptr +=20000;
    *ptr = 2;
}
code/cyclone> gcc -o error2 error_2.c
code/cyclone> error2
Segmentation fault
```

In cyclone NO: *You can't do pointer arithmetic on a * pointer. Pointer arithmetic in C can take a pointer out of bounds, so that when the pointer is eventually dereferenced, it corrupts memory or causes a crash.*

(auto) NO update di puntatore null

```
int main(){
    int * ptr = 0;
    *ptr = 2;
}
```

There is one other way to crash a C program using pointers: you can dereference the NULL pointer or try to update the NULL location. Cyclone prevents this by inserting a null check whenever you dereference a * pointer (that is, whenever you use the *, ->, or subscript operation on a pointer.)

Cosa puoi fare in Cyclone

vedremo alcune caratteristiche di cyclone dando esempi di violazione di sicurezza e spiegheremo che cyclone previene tale problema.

1. NULL pointers
2. buffer overflow
3. zero terminated strings
4. bounded pointers
5. dangling pointers

NULL pointers

Considera la funzione `int getc(FILE *);`

- cosa succede se chiamo `int getc(NULL)?`
- non è specificato
  - se getc è scritta in modo che controlla ogni volta che non sia null, ho un rallentamento dell'esecuzione. Se getc non fa controlli (come in genere fa) posso avere errori
- in cyclone ho due alternative
  - se non faccio nulla, cyclone usa un suo getc definito nella sua libreria e che inserisce un controllo
  - oppure introduce un nuovo tipo di puntatori:

Puntatori non nulli

- Un puntatore non-NULL è indicato da @notnull, o brevemente mettendo @ al posto *

```
int * @notnull ptrA;
int @ ptrB; // forma breve
```

- Un puntatore @notnull non è mai NULL:
  - cyclone controlla quando lo usi (ad esempio non puoi assegnargli NULL)
  - quando dereferenzi un puntatore @notnull o accedi al suo contenuto, cyclone può evitare di controllare che sia non nullo
- utili per documentazione ed efficienza

Esempio di @notnull pointer: i file

getc in cyclone è dichiarato

```
int getc(FILE *@notnull); oppure int getc(FILE @)
```

Se apro un file e poi leggo

```
FILE *f = fopen("/etc/passwd","r");
//f will be NULL if the file /etc/passwd
// doesn't exist or can't be read
int c = getc(f)
```

cyclone inserisce un controllo che non sia nullo e dà un warning.
Se voglio evitare il warning (ma non il check)

```
int c = getc((FILE *@notnull)f);
```

Esempio di @notnull pointer: i file, alternative

Oppure, con check prima del getc

```
FILE @f = (FILE @) fopen("/etc/passwd","r");
int c = getc(f)
```

Oppure, con controllo esplicito (cyclone non mette check)

```
if (f == NULL) {
   fprintf(stderr,"cannot open passwd file!");
   exit(-1);
}
int c = getc(f); //OK so che f non è null
```

Buffer overflows

- per prevenire buffer overflows, cyclone limita l'aritmetica dei puntatori su *-pointer e su @-pointer
- si può però fare su "fat" puntatori, che mantengono un'informazione aggiuntiva sulla dimensione dell'array (a cui si può accedere)
- fat pointers sono denotati con `@fat` o brevemente con ?

```
int strlen(const char ?s){
   int i,n;
   if (!s) return 0;
   n = s.size;
   for (i = 0; i <n; i++,s++)
     if (!*s) return i;
   return n;
}
```

Coversioni

- gli array vengono convertiti a ?-pointers.

```
char a[20]=”pippo”;
strlen(a);
```

- si possono convertire puntatori * a ? con size 1;
- viceversa (da ? a *) non c’è problema

puntatori a puntatori

Il seguente programma scrive i comandi dati sulla stringa di comando
in C

```
int main(int argc, char ** argv) {
```

argv è un char **, a pointer to a pointer to a character, which is thought of as an array of an array of characters. In Cyclone:

```
#include <stdio.h>
int main(int argc, char *@fat *@fat argv) {
  while (argc > 0) { /* print args space */
    printf("%s ",*argv);
    argc--;
    argv++; // <--- aritmetica OK
  }
  printf("\n"); return 0;
}
```

Zero-Terminated

- Per rappresentare le stringhe si possono usare questo tipo di puntatori

```
char *@zeroterm
```

- @zeroterm qualifier indicates that the pointer points to a zero-terminated sequence.
- The qualifier is orthogonal to other qualifiers, such as @fat or @notnull, so you can freely combine them.
- puoi fare aritmetica: quando fai x+i controlla che in x tra 0 a i-1 non ci siano null
- attenzione ad usarli: può diventare dispendioso

Buonded pointers

- A pointer type can also specify that it points to a sequence of a particular (statically known) length using the @numelts qualifier

```
void foo(int *@numelts(4) arr);
```

- Bounded pointers are most often constructed from arrays.
- when you pass an array as a parameter to a function, it is promoted automatically to a pointer,This pointer will have a sequence bound that is the same as the length of the array:

```
int x[4] = {1,2,3,4};
foo(x);
```

- the parameter x being passed to foo is automatically cast to type int *@numelts(4), which is the type expected by foo.

Conversioni

```
void foo(int *@numelts(4) arr);
int y[8] = {1,2,3,4,5,6,7,8};
foo(y);
```

- the type of y is automatically cast to type int *@numelts(8). Since 8 >= 4, the call is safe and so Cyclone accepts it but emits a warning “implicit cast to shorter array.”
- the following code will be rejected, because the pointer being passed is too short:

```
int bad[2] = {1,2};
foo(bad); // does not typecheck
```

Array cui lunghezza non è nota

- bounded pointers can also be used to correlate a pointer to an array whose length is not known statically with a variable that defines it. in C (num is the length of the array pointed at by p)

```
int sum(int num, int *p) {
    int a = 0;
    for (unsigned i = 0; i < num; i++) a += p[i];
}
```

- In Cyclone, this relationship can be expressed by giving sum the following type (the body of the function is the same):

```
int sum(tag_t<‘n> num,
        int *@notnull @numelts(valueof(‘n)) p) {
```

The type of num is specified as tag_t<‘n>. This simply means that num holds an integer value, called ‘n, and the number of elements of p is equal to n

In breve

in C:

```
int sum(int num, int *p)
```

in cyclone

```
int sum(tag_t num, int p[num]);
```

Regions: dangling pointers

- dangling pointer = un puntatore che punta ad una zona di memoria che è stata deallocata (ad esempio una zona dello stack che viene liberata da un pop)
- Se dereferenzio un dangling pointer ho un errore (ma non sempre me ne accorgo)
- Esempio tipico

```
struct Point {int x; int y;};
struct Point *newPoint(int x,int y) {
   struct Point result = {x,y};
   return &result;
}
void bar() {
    struct Point *p = newPoint(1,2);
    p->y = 1234;
}
```

BUG!!!

- per trovare questi errori cyclone associa una regione ad ogni puntatore.

The code has an obvious bug: the function newPoint returns a pointer to a locally-defined variable (result), even though the storage for that variable is deallocated upon exit from the function. That storage may be re-used (e.g., by a subsequent procedure call) leading to subtle bugs or security problems. For instance, in the code above, after bar calls newPoint, the storage for the point is reused to store information for the activation record of the call to foo. This includes a copy of the pointer p and the return address of foo. Therefore, it may be that p->y actually points to the return address of foo. The assignment of the integer 1234 to that location could then result in foo "returning" to an arbitrary hunk of code in memory. Nevertheless, the C type-checker readily admits the code.
In Cyclone, this code would be rejected by the type-checker to avoid the kind of problems mentioned above. The reason the code is

Altre caratteristiche

vedi la documentazione

Esempi/esercizi - opzionale

Dato il seguente codice, riscriverlo in cyclone

```
#include <stdio.h>
void foo(char *s) {
   printf(s);
}
int main(int argc, char **argv) {
  argv++;
  for (argc--; argc >= 0; argc--, argv++)
        foo(*argv);
}
```

Esempi

Dato il seguente codice, riscriverlo in cyclone

```
#include <stdio.h>
void foo(char ?s) {
   printf(s);
}
int main(int argc, char ??argv) {
  argv++;
  for (argc--; argc >= 0; argc--, argv++)
        foo(*argv);
}
```

Esercizio / Progetto

- prendi un programma scritto da te e convertilo in cyclone
- oppure scrivi un programma che copia un array di caratteri in un altro array di caratteri e restituisce il nuovo array (clone)