Università di Pisa

Linguaggi di Programmazione con Laboratorio

Seminario di fine corso

Puntatori di C: Implementazione ed operatori `&`, `*` e `new` in Small21

Daniele Canzoneri

Università di Pisa, Dipartimento di Matematica

26 ottobre 2021

Scopo del progetto

L'obiettivo di questo progetto è estendere Small21 aggiungendo i puntatori in stile C con i seguenti operatori:

- l'operatore `*`, detto *dereferenziamento*;
- l'operatore `&`, detto *immediato*;
- l'operatore `new`, detto *introduzione*.

Scopo del progetto

L'obiettivo di questo progetto è estendere Small21 aggiungendo i puntatori in stile C con i seguenti operatori:

- l'operatore `*`, detto *dereferenziamento*;
- l'operatore `&`, detto *immediato*;
- l'operatore `new`, detto *introduzione*.

Estenderemo inoltre il sistema dei tipi di Small21 introducendo i nuovi tipi puntatore.

Puntatori in C

I puntatori sono tipi di dato che servono a manipolare direttamente *l-valori*: sono realizzati utilizzando le locazioni di memoria della MA di C come valori di C.

Puntatori in C

I puntatori sono tipi di dato che servono a manipolare direttamente *l-valori*: sono realizzati utilizzando le locazioni di memoria della MA di C come valori di C.
Ogni puntatore ha un ben definito tipo che dipende dal tipo dell'oggetto puntato (detto *tipo base*): questo fa sì che un puntatore possa puntare solo a un oggetto del suo tipo base.

Puntatori in C

I puntatori sono tipi di dato che servono a manipolare direttamente *l-valori*: sono realizzati utilizzando le locazioni di memoria della MA di C come valori di C.
Ogni puntatore ha un ben definito tipo che dipende dal tipo dell'oggetto puntato (detto *tipo base*): questo fa sì che un puntatore possa puntare solo a un oggetto del suo tipo base.

I puntatori hanno molteplici utilizzi in C:

- vengono usati per costruire e accedere a strutture dati dinamiche come grafi, alberi, liste, ecc.

Puntatori in C

I puntatori sono tipi di dato che servono a manipolare direttamente *l-valori*: sono realizzati utilizzando le locazioni di memoria della MA di C come valori di C.
Ogni puntatore ha un ben definito tipo che dipende dal tipo dell'oggetto puntato (detto *tipo base*): questo fa sì che un puntatore possa puntare solo a un oggetto del suo tipo base.

I puntatori hanno molteplici utilizzi in C:

- vengono usati per costruire e accedere a strutture dati dinamiche come grafi, alberi, liste, ecc.
- sono indispensabili per emulare la trasmissione di parametri per reference nelle invocazioni di procedure e funzioni.

Vantaggi e svantaggi

Vantaggi:

- emulazione di strutture e meccanismi non presenti nel linguaggio;

Vantaggi e svantaggi

Vantaggi:

- emulazione di strutture e meccanismi non presenti nel linguaggio;
- migliori performance in termini di spazio e tempo per determinate operazioni;

Vantaggi e svantaggi

Vantaggi:

- emulazione di strutture e meccanismi non presenti nel linguaggio;
- migliori performance in termini di spazio e tempo per determinate operazioni;
- gestione di memoria dinamica.

Vantaggi e svantaggi

Vantaggi:

- emulazione di strutture e meccanismi non presenti nel linguaggio;
- migliori performance in termini di spazio e tempo per determinate operazioni;
- gestione di memoria dinamica.

Svantaggi:

- l'uso di puntatori può rendere il codice C meno leggibile e portare a errori molto difficili da individuare.

Cambiamenti in Small21

In Small21 implementiamo i puntatori utilizzando le locazioni di memoria della AM21 come valori di Small21 stesso.

Cambiamenti in Small21

In Small21 implementiamo i puntatori utilizzando le locazioni di memoria della AM21 come valori di Small21 stesso.
Estendiamo il sistema dei tipi con i nuovi tipi puntatore: aggiungiamo il tipo [ptr] t ai tipi Simple per ogni t $\in$ Simple.

Cambiamenti in Small21

In Small21 implementiamo i puntatori utilizzando le locazioni di memoria della AM21 come valori di Small21 stesso.
Estendiamo il sistema dei tipi con i nuovi tipi puntatore:
aggiungiamo il tipo [`ptr`] `t` ai tipi `Simple` per ogni `t` $\in$ `Simple`.

Aggiungiamo, inoltre, gli operatori precedentemente introdotti:

- l'operatore `new` applicato a un'espressione di tipo alloca nuova memoria e restituisce l'indirizzo della nuova memoria allocata.
  Esempio: `int* pt = new int;`

Cambiamenti in Small21

In Small21 implementiamo i puntatori utilizzando le locazioni di memoria della AM21 come valori di Small21 stesso.
Estendiamo il sistema dei tipi con i nuovi tipi puntatore:
aggiungiamo il tipo [`ptr`] `t` ai tipi `Simple` per ogni `t` $\in$ `Simple`.

Aggiungiamo, inoltre, gli operatori precedentemente introdotti:

- l'operatore `new` applicato a un'espressione di tipo alloca nuova memoria e restituisce l'indirizzo della nuova memoria allocata.
  Esempio: `int* pt = new int;`
- l'operatore `*` applicato a un puntatore permette l'accesso al valore puntato.
  Esempio: `*pt = 3;` `int a = *pt + 1;`

Cambiamenti in Small21

In Small21 implementiamo i puntatori utilizzando le locazioni di memoria della AM21 come valori di Small21 stesso.
Estendiamo il sistema dei tipi con i nuovi tipi puntatore: aggiungiamo il tipo `[ptr] t` ai tipi `Simple` per ogni `t` $\in$ `Simple`.

Aggiungiamo, inoltre, gli operatori precedentemente introdotti:

- l'operatore `new` applicato a un'espressione di tipo alloca nuova memoria e restituisce l'indirizzo della nuova memoria allocata.
  Esempio: `int* pt = new int;`
- l'operatore `*` applicato a un puntatore permette l'accesso al valore puntato.
  Esempio: `*pt = 3;` `int a = *pt + 1;`
- l'operatore `&` applicato a un valore modificabile restituisce l'indirizzo di memoria a cui è associato il valore memorizzato.
  Esempio: `pt = &a;`

Puntatori in Small21: nuovi programmi

Con questa estensione di Small21 ci aspettiamo di essere in grado di eseguire un programma come il seguente:

```
Program esempio1 {
    int x = 20;
    int* ptx = &x;
    int y;
    *ptx = 10;
    ptx = new int;
    *ptx = 7;
    y = (x + *ptx);
}
```

Composizione operatori * e &

Gli operatori * e & sono uno "inverso" dell'altro: ci aspettiamo che

- `&*pt` restituisca l'indirizzo `pt`;

Composizione operatori * e &

Gli operatori * e & sono uno "inverso" dell'altro: ci aspettiamo che

- `&*pt` restituisca l'indirizzo `pt`;
- `*&var` abbia lo stesso comportamento di `var` (sia come l-espressione che come r-espressione).

Composizione operatori * e &

Gli operatori * e & sono uno "inverso" dell'altro: ci aspettiamo che

- `&*pt` restituisca l'indirizzo `pt`;
- `*&var` abbia lo stesso comportamento di `var` (sia come l-espressione che come r-espressione).

Ad esempio ci aspettiamo che il seguente programma

```
Program esempio2 {
    int a = 3;
    int b = (*&a + 7);
    int c;
    int* pta = &a;
    int* pt = &*pta;
    *&c = *pt;
}
```

sia un programma legale e abbia come stato finale `[a = 3; b = 10; c = 3]`.

Puntatori in Small21 - Espressività

Per semplicità di sviluppo i tipi puntatore sono limitati ai soli tipi `Simple`: pertanto non sono previsti puntatori array.

Puntatori in Small21 - Espressività

Per semplicità di sviluppo i tipi puntatore sono limitati ai soli tipi `Simple`: pertanto non sono previsti puntatori array.

Non è ancora possibile realizzare un programma come il seguente:

```
Program arraymultidim {
    (int[2]*)[2] A;
    int[2] A1;
    int[2] A2;

    A1[0] = 1;
    A1[1] = 0;
    A2[0] = 0;
    A2[1] = 1;
    A[0] = &A1;
    A[1] = &A2;
}
```

dove `int[2]*` rappresenta il tipo di un puntatore a un array di 2 elementi.

Puntatori in Small21 - Espressività

Non è ancora possibile eseguire un programma come il seguente che costruisce una lista di 10 elementi:

```
Program {
    record list {
        int val;
        (record list)* next;
    }
    int i = 0;
    (record list)* head;
    (record list)* curr = head;

    iter: if (i=10) goto end;
    curr = new (record list);
    curr.val = i;
    curr = curr.next;
    i = i + 1;
    goto iter;
    end: ;
}
```

Puntatori in Small21 - Espressività

Allo stato attuale di Small21 l'aggiunta di puntatori `Simple` non aumenta l'espressività perché non permette l'espressione di strutture e meccanismi non presenti nel linguaggio.

Puntatori in Small21 - Espressività

Allo stato attuale di Small21 l'aggiunta di puntatori `Simple` non aumenta l'espressività perché non permette l'espressione di strutture e meccanismi non presenti nel linguaggio.

In combinazione con altri meccanismi però:

Puntatori in Small21 - Espressività

Allo stato attuale di Small21 l'aggiunta di puntatori `Simple` non aumenta l'espressività perché non permette l'espressione di strutture e meccanismi non presenti nel linguaggio.

In combinazione con altri meccanismi però:

- aggiungendo i puntatori array possiamo esprimere array multidimensionali: possiamo eseguire un programma come il precedente;

Puntatori in Small21 - Espressività

Allo stato attuale di Small21 l'aggiunta di puntatori `Simple` non aumenta l'espressività perché non permette l'espressione di strutture e meccanismi non presenti nel linguaggio.

In combinazione con altri meccanismi però:

- aggiungendo i puntatori array possiamo esprimere array multidimensionali: possiamo eseguire un programma come il precedente;
- aggiungendo record e puntatori record possiamo esprimere liste e strutture dinamiche in generale: possiamo eseguire un programma come il precedente;

Puntatori in Small21 - Espressività

Allo stato attuale di Small21 l'aggiunta di puntatori `Simple` non aumenta l'espressività perché non permette l'espressione di strutture e meccanismi non presenti nel linguaggio.

In combinazione con altri meccanismi però:

- aggiungendo i puntatori array possiamo esprimere array multidimensionali: possiamo eseguire un programma come il precedente;
- aggiungendo record e puntatori record possiamo esprimere liste e strutture dinamiche in generale: possiamo eseguire un programma come il precedente;

Grande aumento di espressività

Sintassi concreta di Small21

- **Sintassi concreta:**

  $\texttt{Simple} ::= \texttt{int} \mid \texttt{bool} \mid \texttt{Simple}*$

  $\texttt{ExpT} \rightarrow \ldots \mid \texttt{\&DExp} \mid \texttt{new Simple}$

  $\texttt{DExp} \rightarrow \ldots \mid *\texttt{Exp}$

Sintassi concreta di Small21

- **Sintassi concreta:**
  $\texttt{Simple} ::= \texttt{int} \mid \texttt{bool} \mid \texttt{Simple}*$
  $\texttt{ExpT} \rightarrow \ldots \mid \texttt{\&DExp} \mid \texttt{new Simple}$
  $\texttt{DExp} \rightarrow \ldots \mid *\texttt{Exp}$

Osservazione: gli operatori $*$ e & possono essere applicati a espressioni e non solo a identificatori per permettere la composizione con altri operatori (come nell'esempio precedente).

Sintassi astratta di Small21

- **Sintassi astratta:**
  `Type` ::= ... | [`ptr`] `Type`
  `DExp` ::= ... | [*] `Exp`
  `Exp` ::= ... | [&] `Exp` | [`new`] `Type`

Sintassi astratta di Small21

- **Sintassi astratta:**

  Type ::= ... | [ptr] Type
  DExp ::= ... | [*] Exp
  Exp ::= ... | [&] Exp | [new] Type

  In OCaml:

```
type tye =
    ...
    | Ptr of tye
    ...
```

```
exp =
    ...
    | Deref of exp
    | Addr of exp
    | New of tye
    ...
```

Modifiche alla AM21

Introduciamo le seguenti modifiche alla AM21:

```
type mval =
    ...
    | MvalL of loc
    ...
```

```
type aval =
    ...
    | Lval of loc
    ...
```

In questo modo introduciamo le locazioni della AM21 come valori memorizzabili (da assegnare alle variabili puntatore) e come valori atomici (per essere trattate dalle operazioni introdotte).

Modifiche alla AM21

Introduciamo le seguenti modifiche alla AM21:

```
type mval =
    ...
    | MvalL of loc
    ...
```

```
type aval =
    ...
    | Lval of loc
    ...
```

In questo modo introduciamo le locazioni della AM21 come valori memorizzabili (da assegnare alle variabili puntatore) e come valori atomici (per essere trattate dalle operazioni introdotte).

Modifichiamo inoltre la funzione `isSimple` per tenere conto dei nuovi tipi puntatore introdotti:

```
isSimple = function
    | Int -> true
    | Bool -> true
    | Ptr ty -> isSimple ty      (** added **)
    | _ -> false
```

Sistema Y: regole per $*$

$\texttt{Exp} ::= \ldots \mid [*]\ \texttt{Exp}$

L'operatore $*$ ha un doppio comportamento: utilizzato in una *l-espressione* (per esempio a sinistra di un assegnamento) calcola un valore modificabile, utilizzato in una *r-espressione* (per esempio a destra di un assegnamento) calcola un valore memorizzabile. Questo dà luogo a regole d'inferenza diverse a seconda del suo utilizzo.

Sistema Y: regole per $*$

$\texttt{Exp} ::= \ldots \mid [*]\ \texttt{Exp}$

L'operatore $*$ ha un doppio comportamento: utilizzato in una *l-espressione* (per esempio a sinistra di un assegnamento) calcola un valore modificabile, utilizzato in una *r-espressione* (per esempio a destra di un assegnamento) calcola un valore memorizzabile. Questo dà luogo a regole d'inferenza diverse a seconda del suo utilizzo.

Sistema Y:

$$[\text{Y101}]\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \to_{\texttt{Y}} ([\texttt{ptr}]\,\texttt{t}, \texttt{Y}_\rho) \qquad \texttt{t} \in \texttt{Simple}}{\langle [*]\,\texttt{e}, \texttt{Y}_\rho \rangle \to_{\texttt{DY}} ([\texttt{mut}]\,\texttt{t}, \texttt{Y}_\rho)}$$

$$[\text{Y102}]\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \to_{\texttt{Y}} ([\texttt{ptr}]\,\texttt{t}, \texttt{Y}_\rho) \qquad \texttt{t} \in \texttt{Simple}}{\langle [*]\,\texttt{e}, \texttt{Y}_\rho \rangle \to_{\texttt{Y}} (\texttt{t}, \texttt{Y}_\rho)}$$

Sistema Y: regole per $*$

$\mathtt{Exp} ::= \ldots \mid [*]\ \mathtt{Exp}$

Gestione errori di tipo:

$$[\text{E101}]\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{ptr}]\ \mathtt{t}, \mathtt{Y}_\rho) \quad \mathtt{t} \notin \mathtt{Simple}}{\langle [*]\,\mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{DY}} ([\mathtt{terr}], \mathtt{Y}_\rho)} \qquad [\text{E101.1}]\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho) \quad \mathtt{t} \neq [\mathtt{ptr}]\ \mathtt{t}'}{\langle [*]\,\mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{DY}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

$$[\text{E102}]\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{ptr}]\ \mathtt{t}, \mathtt{Y}_\rho) \quad \mathtt{t} \notin \mathtt{Simple}}{\langle [*]\,\mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)} \qquad [\text{E102.1}]\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho) \quad \mathtt{t} \neq [\mathtt{ptr}]\ \mathtt{t}'}{\langle [*]\,\mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

Sistema Y: regole per &

$\texttt{Exp} ::= \ldots \mid [\&]\ \texttt{Exp}$

Sistema Y:

$$[\text{Y103}]\ \frac{\begin{array}{c}\langle \texttt{e}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{DY}} (\texttt{[mut]}\,\texttt{t}, \texttt{Y}_\rho) \\ \texttt{t} \in \texttt{Simple}\end{array}}{\langle \texttt{[\&]}\,\texttt{e}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{Y}} (\texttt{[ptr]}\,\texttt{t}, \texttt{Y}_\rho)}$$

Sistema Y: regole per &

$\texttt{Exp} ::= \ldots \mid \texttt{[\&] Exp}$

Sistema Y:

$$[\text{Y103}]\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \to_{\texttt{DY}} (\texttt{[mut] t}, \texttt{Y}_\rho) \quad \texttt{t} \in \texttt{Simple}}{\langle \texttt{[\&] e}, \texttt{Y}_\rho \rangle \to_{\texttt{Y}} (\texttt{[ptr] t}, \texttt{Y}_\rho)}$$

Gestione errori di tipo:

$$[\text{E103}]\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \to_{\texttt{DY}} (\texttt{[mut] t}, \texttt{Y}_\rho) \quad \texttt{t} \notin \texttt{Simple}}{\langle \texttt{[\&] e}, \texttt{Y}_\rho \rangle \to_{\texttt{Y}} (\texttt{[terr]}, \texttt{Y}_\rho)} \qquad [\text{E103.1}]\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \to_{\texttt{DY}} (\texttt{t}, \texttt{Y}_\rho) \quad \texttt{t} \neq \texttt{[mut] t}'}{\langle \texttt{[\&] e}, \texttt{Y}_\rho \rangle \to_{\texttt{Y}} (\texttt{[terr]}, \texttt{Y}_\rho)}$$

Sistema Y: regole per &

$\mathtt{Exp} ::= \ldots \mid [\&]\ \mathtt{Exp}$

Sistema Y:

$$[\mathrm{Y103}]\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{DY}} ([\mathtt{mut}]\,\mathtt{t}, \mathtt{Y}_\rho) \quad \mathtt{t} \in \mathtt{Simple}}{\langle [\&]\,\mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{ptr}]\,\mathtt{t}, \mathtt{Y}_\rho)}$$

Gestione errori di tipo:

$$[\mathrm{E103}]\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{DY}} ([\mathtt{mut}]\ \mathtt{t}, \mathtt{Y}_\rho) \quad \mathtt{t} \notin \mathtt{Simple}}{\langle [\&]\,\mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)} \qquad [\mathrm{E103.1}]\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{DY}} (\mathtt{t}, \mathtt{Y}_\rho) \quad \mathtt{t} \neq [\mathtt{mut}]\ \mathtt{t}'}{\langle [\&]\,\mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

Osservazione: l'operatore & è applicabile solo a valori modificabili. In particolare, l'uso con costanti è segnalato dall'errore E103.1 (in realtà tale errore è superfluo in quanto l'inferenza di tipo $\to_{\mathtt{DY}}$ deduce sempre un valore modificabile).

Sistema Y: regole per new

$\texttt{Exp} ::= \ldots \mid [\texttt{new}]\ \texttt{tye}$

Sistema Y:

$$[\text{Y104}]\ \frac{\texttt{t} \in \texttt{Simple}}{\langle [\texttt{new}]\,\texttt{t}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{Y}} ([\texttt{ptr}]\,\texttt{t}, \texttt{Y}_\rho)}$$

Gestione errori di tipo:

$$[\text{E104}]\ \frac{\texttt{t} \notin \texttt{Simple}}{\langle [\texttt{new}]\,\texttt{t}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{Y}} ([\texttt{terr}], \texttt{Y}_\rho)}$$

Nuovi programmi esprimibili

Con la sintassi introdotta possiamo esprimere i programmi precedentemente discussi:

Sintassi concreta:

```
Program esempio1 {
    int x = 20;
    int* ptx = &x;
    int y;
    *ptx = 10;
    ptx = new int;
    *ptx = 7;
    y = (x + *ptx);
}
```

Sintassi astratta:

```
let esempio1 = Prog ("esempio1",
  Block (
    SeqD (
      SeqD(
        Var (Int, "x", N 20),
        Var (Ptr Int, "ptx", Addr (Val "x"))),
      Var (Int, "y", EE)),
    UnL (
      SeqS (
        SeqS (
          SeqS (
            Upd (Deref (Val "ptx"), N 10),
            Upd (Val "ptx", New Int)),
          Upd (Deref (Val "ptx"), N 7)),
        Upd (Val "y", Plus(Val "x", Deref (Val "ptx")))
      )
    )
  )
);;
```

Nuovi programmi esprimibili

Sintassi concreta:

```
Program esempio2 {
    int a = 3;
    int b = (*&a + 7);
    int c;
    int* pta = &a;
    int* pt = &*pta;
    *&c = *pt;
}
```

Sintassi astratta:

```
let esempio2 = Prog ("esempio2",
  Block (
    SeqD (
      SeqD (
        SeqD (
          SeqD (
            Var (Int, "a", N 3),
            Var (Int, "b",
              Plus (Deref (Addr (Val "a")), N 7))),
          Var (Int, "c", EE)),
        Var (Ptr Int, "pta", Addr (Val "a"))),
      Var (Ptr Int, "pt", Addr (Deref (Val "pta")))),
    UnL (
      Upd (
        Deref (Addr (Val "c")),
        Deref (Val "pt")
      )
    )
  )
);
```

Regole di inferenza - $\mathtt{SEM}_{\mathtt{EXP}}$ e $\mathtt{SEM}_{\mathtt{DEN}}$

Coerentemente con le nuove regole del sistema Y introdotte introduciamo le seguenti regole di inferenza:

Regole di inferenza - $\mathrm{SEM}_{\mathrm{EXP}}$ e $\mathrm{SEM}_{\mathrm{DEN}}$

Coerentemente con le nuove regole del sistema Y introdotte introduciamo le seguenti regole di inferenza:

$\mathtt{Exp} ::= \ldots \mid [*]\ \mathtt{Exp}$

$$[\mathrm{X101}]\ \frac{\langle \mathtt{e}, \sigma\rangle \to_{\mathtt{Y}} ([\mathtt{ptr}]\,\mathtt{t},\ \mathtt{loc_t},\ \sigma_{\mathtt{F}}) \quad \mathtt{t} \in \mathtt{Simple}}{\langle [*]\,\mathtt{e}, \sigma\rangle \to_{\mathtt{Den}} \lfloor [\mathtt{mut}]\,\mathtt{t},\ \mathtt{loc_t},\ \sigma_{\mathtt{F}}\rfloor}$$

$$[\mathrm{X102}]\ \frac{\langle \mathtt{e}, \sigma\rangle \to_{\mathtt{Y}} ([\mathtt{ptr}]\,\mathtt{t},\ \mathtt{loc_t},\ \sigma_{\mathtt{F}}) \quad \mathtt{t} \in \mathtt{Simple} \qquad \sigma_{\mathtt{F}} = (\Delta_{\mathtt{F}}, \mu_{\mathtt{F}}) \quad \mu_{\mathtt{F}}(\mathtt{loc_t}) = \mathtt{v_t}}{\langle [*]\,\mathtt{e}, \sigma\rangle \to \lfloor \mathtt{t},\ \mathtt{v_t},\ \sigma_{\mathtt{F}}\rfloor}$$

Regole di inferenza - $\mathrm{SEM}_{\mathrm{EXP}}$ e $\mathrm{SEM}_{\mathrm{DEN}}$

Coerentemente con le nuove regole del sistema Y introdotte introduciamo le seguenti regole di inferenza:

$\mathtt{Exp} ::= \ldots \mid [*]\ \mathtt{Exp}$

$$[\mathrm{X101}]\ \frac{\langle \mathtt{e}, \sigma\rangle \to_{\mathtt{Y}} ([\mathtt{ptr}]\,\mathtt{t},\ \mathtt{loc_t},\ \sigma_{\mathtt{F}}) \qquad \mathtt{t} \in \mathtt{Simple}}{\langle [*]\,\mathtt{e}, \sigma\rangle \to_{\mathtt{Den}} \lfloor [\mathtt{mut}]\,\mathtt{t},\ \mathtt{loc_t},\ \sigma_{\mathtt{F}}\rfloor}$$

$$[\mathrm{X102}]\ \frac{\langle \mathtt{e}, \sigma\rangle \to_{\mathtt{Y}} ([\mathtt{ptr}]\,\mathtt{t},\ \mathtt{loc_t},\ \sigma_{\mathtt{F}}) \qquad \mathtt{t} \in \mathtt{Simple} \qquad \sigma_{\mathtt{F}} = (\Delta_{\mathtt{F}}, \mu_{\mathtt{F}}) \qquad \mu_{\mathtt{F}}(\mathtt{loc_t}) = \mathtt{v_t}}{\langle [*]\,\mathtt{e}, \sigma\rangle \to \lfloor \mathtt{t},\ \mathtt{v_t},\ \sigma_{\mathtt{F}}\rfloor}$$

Osservazione: nelle regole X101 e X102 il vincolo $\mathtt{t} \in \mathtt{Simple}$ è superfluo in quanto il tipo $[\mathtt{ptr}]\ \mathtt{t}$, secondo le regole attuali, ha necessariamente tipo $\mathtt{t} \in \mathtt{Simple}$.

Regole di inferenza - SEM_{EXP}

$$\text{Exp} ::= \ldots \mid [\&]\ \text{Exp}$$

$$[\text{X103}]\ \frac{\langle \mathtt{e}, \sigma \rangle \rightarrow_{\text{DY}} ([\mathtt{mut}]\,\mathtt{t},\ \mathtt{loc_t},\ \sigma_{\text{F}}) \qquad \mathtt{t} \in \mathtt{Simple}}{\langle [\&]\,\mathtt{e}, \sigma \rangle \rightarrow \lfloor [\mathtt{ptr}]\,\mathtt{t},\ \mathtt{loc_t},\ \sigma_{\text{F}} \rfloor}$$

Regole di inferenza - SEM_{EXP}

$\texttt{Exp} ::= \ldots \mid \texttt{[\&] Exp}$

$$[\text{X103}]\ \frac{\langle \texttt{e}, \sigma\rangle \rightarrow_{\texttt{DY}} (\texttt{[mut]}\,\texttt{t},\ \texttt{loc}_\texttt{t},\ \sigma_\texttt{F}) \qquad \texttt{t} \in \texttt{Simple}}{\langle \texttt{[\&]}\,\texttt{e}, \sigma\rangle \rightarrow \lfloor \texttt{[ptr]}\,\texttt{t},\ \texttt{loc}_\texttt{t},\ \sigma_\texttt{F}\rfloor}$$

$\texttt{Exp} ::= \ldots \mid \texttt{[new] tye}$

$$[\text{X104}]\ \frac{\sigma = (\Delta, \mu) \qquad \texttt{t} \in \texttt{Simple} \qquad \triangleright(\mu, \texttt{t}, 1) = (\texttt{loc}_\texttt{t}, \mu_\texttt{F}) \qquad (\Delta, \mu_\texttt{F}) = \sigma_\texttt{F}}{\langle \texttt{[new]}\,\texttt{t}, \sigma\rangle \rightarrow \lfloor \texttt{[ptr]}\,\texttt{t},\ \texttt{loc}_\texttt{t},\ \sigma_\texttt{F}\rfloor}$$

Regole di inferenza - SEM_{EXP}

$\texttt{Exp} ::= \ldots \mid \texttt{[\&] Exp}$

$$[\text{X103}]\ \frac{\langle \texttt{e}, \sigma\rangle \rightarrow_{\texttt{DY}} (\texttt{[mut]\,t}, \texttt{loc}_\texttt{t}, \sigma_\texttt{F}) \qquad \texttt{t} \in \texttt{Simple}}{\langle \texttt{[\&]\,e}, \sigma\rangle \rightarrow \lfloor \texttt{[ptr]\,t}, \texttt{loc}_\texttt{t}, \sigma_\texttt{F}\rfloor}$$

$\texttt{Exp} ::= \ldots \mid \texttt{[new] tye}$

$$[\text{X104}]\ \frac{\sigma = (\Delta, \mu) \qquad \texttt{t} \in \texttt{Simple} \qquad \triangleright(\mu, \texttt{t}, 1) = (\texttt{loc}_\texttt{t}, \mu_\texttt{F}) \qquad (\Delta, \mu_\texttt{F}) = \sigma_\texttt{F}}{\langle \texttt{[new]\,t}, \sigma\rangle \rightarrow \lfloor \texttt{[ptr]\,t}, \texttt{loc}_\texttt{t}, \sigma_\texttt{F}\rfloor}$$

Osservazione: a differenza delle altre espressioni presenti in Small21, l'operatore new modifica lo stato allocando nuova memoria nello store di AM21.

Modifiche all'interprete di Small21

dexpSem - Implementazione della regola X101 con relativi controlli di tipo:

```
dexpSem dexp (sk,mu) = match dexp with
    ...
    | Deref exp -> (match expSem exp (sk,mu) with
        | (Ptr ty, Lval loct, sgF) when (isSimple ty) ->
            (Mut ty, loct, sgF)
        | (Ptr ty, _, _) ->
            raise (TypeErrorE("E101 expSem: simple Type? - ", exp))
        | _ ->
            raise (TypeErrorE("E101.1 expSem - ", dexp))
```

Modifiche all'interprete di Small21

expSem - Implementazione della regola X102 con relativi controlli di tipo:

```
expsem exp (sk,mu) = match exp with
    ...
    | Deref exp1 -> (match expSem exp1 (sk,mu) with
        | (Ptr ty, Lval loct, (skF,muF)) when (isSimple ty) ->
            let vt = mTOe (getStore muF loct) in
            (ty, vt, (skF,muF))
        | (Ptr ty, _, _) ->
            raise (TypeErrorE("E102 expSem: simple Type? - ", exp1))
        | _ ->
            raise (TypeErrorE("E102.1 expSem - ", exp)))
```

Modifiche all'interprete di Small21

expSem - Implementazione delle regole X103, X104 con relativi controlli di tipo:

```
expsem exp (sk,mu) = match exp with
    ...
    | Addr dexp ->
        (match dexpSem dexp (sk,mu) with
            | (Mut ty, loct, sgF) when isSimple ty ->
                (Ptr ty, Lval loct, sgF)
            | (Mut ty, _, _) ->
                raise (TypeErrorE("E103 expSem: simple Type? - ", dexp))
            | _ ->
                raise (TypeErrorE("E103.1 expSem - ", exp)))
    | New ty when isSimple ty ->
        let (loct, muF) = allocate mu 1 in
        let sgF = (sk, muF) in
        (Ptr ty, Lval loct, sgF)
    | New ty ->
        raise (TypeErrorE("E104 expSem: Simple Type? - ", exp))
```

Modifiche all'interprete di Small21

expSem - Implementazione delle regole X103, X104 con relativi controlli di tipo:

```
expsem exp (sk,mu) = match exp with
    ...
    | Addr dexp ->
        (match dexpSem dexp (sk,mu) with
            | (Mut ty, loct, sgF) when isSimple ty ->
                (Ptr ty, Lval loct, sgF)
            | (Mut ty, _, _) ->
                raise (TypeErrorE("E103 expSem: simple Type? - ", dexp))
            | _ ->
                raise (TypeErrorE("E103.1 expSem - ", exp)))
    | New ty when isSimple ty ->
        let (loct, muF) = allocate mu 1 in
        let sgF = (sk, muF) in
        (Ptr ty, Lval loct, sgF)
    | New ty ->
        raise (TypeErrorE("E104 expSem: Simple Type? - ", exp))
```

Esecuzione esempi - Nuovi operatori

```
Stack:
 >{esempio1,0,[y/(Mint,L2);
  ptx/(Mint*,L1);
  x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-20,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-7]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-17,L3<-7]
```

```
Program esempio1 {
    int x = 20;
    int* ptx = &x;
    int y;

    *ptx = 10;

    ptx = new int;

    *ptx = 7;

    y = (x + *ptx);
}
```

Esecuzione esempi - Nuovi operatori

```
Stack:
 >{esempio1,0,[y/(Mint,L2);
  ptx/(Mint*,L1);
  x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-20,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-7]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-17,L3<-7]
```

```
Program esempio1 {
    int x = 20;
    int* ptx = &x;
    int y;

    *ptx = 10;

    ptx = new int;

    *ptx = 7;

    y = (x + *ptx);
}
```

Esecuzione esempi - Nuovi operatori

```
Stack:
 >{esempio1,0,[y/(Mint,L2);
  ptx/(Mint*,L1);
  x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-20,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-7]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-17,L3<-7]
```

```
Program esempio1 {
    int x = 20;
    int* ptx = &x;
    int y;

    *ptx = 10;

    ptx = new int;

    *ptx = 7;

    y = (x + *ptx);
}
```

Esecuzione esempi - Nuovi operatori

```
Stack:
 >{esempio1,0,[y/(Mint,L2);
  ptx/(Mint*,L1);
  x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-20,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-7]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-17,L3<-7]
```

```
Program esempio1 {
    int x = 20;
    int* ptx = &x;
    int y;

    *ptx = 10;

    ptx = new int;

    *ptx = 7;

    y = (x + *ptx);
}
```

Esecuzione esempi - Nuovi operatori

```
Stack:
 >{esempio1,0,[y/(Mint,L2);
  ptx/(Mint*,L1);
  x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-20,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L0,L2<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-Undef]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-Undef,L3<-7]

Stack:
>{esempio1,0,[y/(Mint,L2);
 ptx/(Mint*,L1);
 x/(Mint,L0)],:cmdNext:,[N]}]
Store:
[L0<-10,L1<-L3,L2<-17,L3<-7]
```

```
Program esempio1 {
    int x = 20;
    int* ptx = &x;
    int y;

    *ptx = 10;

    ptx = new int;

    *ptx = 7;

    y = (x + *ptx);
}
```

Esecuzione esempi - Composizione operatori

```
Stack:
>{esempio2,0,[pt/(Mint*,L4);
 pta/(Mint*,L3);
 c/(Mint,L2);
 b/(Mint,L1);
 a/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-10,L2<-Undef,L3<-L0,L4<-L0]


Stack:
>{esempio2,0,[pt/(Mint*,L4);
 pta/(Mint*,L3);
 c/(Mint,L2);
 b/(Mint,L1);
 a/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-10,L2<-3,L3<-L0,L4<-L0]
```

```
Program esempio2 {
    int a = 3;
    int b = (*&a + 7);
    int c;
    int* pta = &a;
    int* pt = &*pta;

    *&c = *pt;
}
```

Esecuzione esempi - Composizione operatori

```
Stack:
>{esempio2,0,[pt/(Mint*,L4);
 pta/(Mint*,L3);
 c/(Mint,L2);
 b/(Mint,L1);
 a/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-10,L2<-Undef,L3<-L0,L4<-L0]


Stack:
>{esempio2,0,[pt/(Mint*,L4);
 pta/(Mint*,L3);
 c/(Mint,L2);
 b/(Mint,L1);
 a/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-10,L2<-3,L3<-L0,L4<-L0]
```

```
Program esempio2 {
    int a = 3;
    int b = (*&a + 7);
    int c;
    int* pta = &a;
    int* pt = &*pta;

    *&c = *pt;
}
```

Esecuzione esempi - Composizione operatori

```
Stack:
>{esempio2,0,[pt/(Mint*,L4);
 pta/(Mint*,L3);
 c/(Mint,L2);
 b/(Mint,L1);
 a/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-10,L2<-Undef,L3<-L0,L4<-L0]


Stack:
>{esempio2,0,[pt/(Mint*,L4);
 pta/(Mint*,L3);
 c/(Mint,L2);
 b/(Mint,L1);
 a/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-10,L2<-3,L3<-L0,L4<-L0]
```

```
Program esempio2 {
    int a = 3;
    int b = (*&a + 7);
    int c;
    int* pta = &a;
    int* pt = &*pta;

    *&c = *pt;
}
```

Osserviamo che lo stato finale coincide con quello atteso:

`[a = 3; b = 10; c = 3]`

Esempio - Doppio puntatore

Il seguente programma mostra la possibilità di utilizzo di puntatori "multipli":

Sintassi concreta:

```
Program esempio3 {
    int** dp;
    dp = new int*;
    *dp = new int;
    **dp = 5;
}
```

Sintassi astratta:

```
let esempio3 = Prog ("esempio3",
  Block (
    Var (Ptr (Ptr Int), "dp", EE),
    UnL (
      SeqS (
        SeqS (
          Upd (Val "dp", New (Ptr Int)),
          Upd (Deref (Val "dp"), New Int)),
        Upd (Deref (Deref (Val "dp")), N 5)
      )
    )
  )
);;
```

Esecuzione esempi - Doppio puntatore

```
>{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-L2,L2<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-L2,L2<-5]
```

```
Program esempio3 {
    int** dp;

    dp = new int*;

    *dp = new int;

    **dp = 5;
}
```

Esecuzione esempi - Doppio puntatore

```
>{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-L2,L2<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-L2,L2<-5]
```

```
Program esempio3 {
    int** dp;

    dp = new int*;

    *dp = new int;

    **dp = 5;
}
```

Esecuzione esempi - Doppio puntatore

```
>{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-L2,L2<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-L2,L2<-5]
```

```
Program esempio3 {
    int** dp;

    dp = new int*;

    *dp = new int;

    **dp = 5;
}
```

Esecuzione esempi - Doppio puntatore

```
>{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-L2,L2<-Undef]

Stack:
 >{esempio3,0,[dp/(Mint**,L0)],:cmdNext:,[N]}
]
Store:
[L0<-L1,L1<-L2,L2<-5]
```

```
Program esempio3 {
    int** dp;

    dp = new int*;

    *dp = new int;

    **dp = 5;
}
```

Riferimenti bibliografici


Marco Bellia
Materiale delle lezioni del corso di LPL: laboratori 2,3,4,5,6,7
Aprile-Maggio 2020


Marco Bellia
Materiale delle lezioni del corso di LPL: Small21 - Definizione
Maggio 2020


Maurizio Gabbrielli, Simone Martini
Programming Languages: Principles and Paradigms
Undergraduate topics in computer science, Springer London, 2010


Brian W. Kernighan, Dennis M. Ritchie
The C Programming Language
2^{nd} ed., Prentice Halls, 1988