Il linguaggio di specifica formale Z

Il linguaggio Z (Spivey, 1992)

- Sviluppato presso l'Università di Oxford (UK)
- Basato su FSM
- Applicato in ambito industriale
- Dotato di numerose estensioni (Object Z, Real time Z, ..)
- Esistono tool

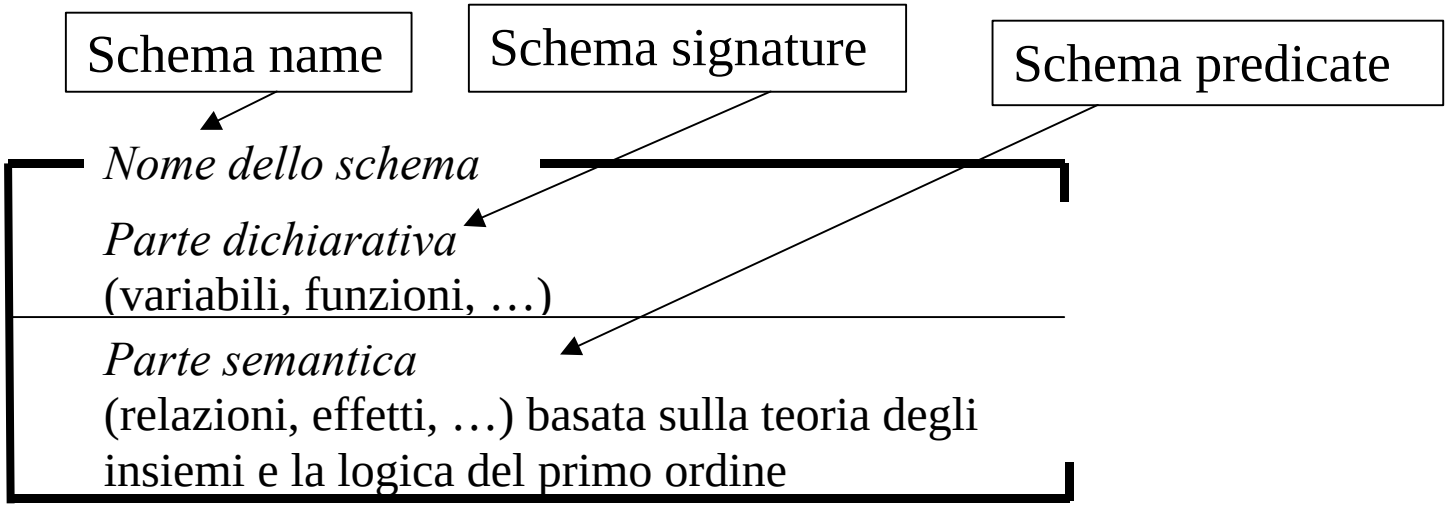
Specifica in Z

Ogni sistema sw è specificato mediante *schemi*, che definiscono:

- Proprietà statiche
  (a) Variabili di stato
  (b) Relazioni invarianti fra le variabili di stato

- Proprietà dinamiche
  (c) Operazioni legali
  (d) Effetti delle operazioni, sia in termini di I/O, sia di cambiamenti di stato

Uno schema Z


Schema name
Schema signature
Schema predicate
Nome dello schema
Parte dichiarativa
(variabili, funzioni, …)
Parte semantica
(relazioni, effetti, …) basata sulla teoria degli insiemi e la logica del primo ordine

Un esempio: l'elenco dei compleanni

Tipi base: [NAME, DATE] Non è necessario dettagliarli

```
_____BirthdayBook_____
known : ℙNAME                  // variabile di stato, rappresenta le persone conosciute
                               // ℙ indica il tipo set del tipo associato, qui NAME
                               // in alcune fonti, anziché ℙ si usa {}, es. {NAME}

birthday : NAME ↦ DATE         // funzione (a ogni elemento del dominio
                               // fa corrispondere un solo elemento del codominio)
_____________________________
dom(birthday) = known          // invariante: l'elenco contiene i compleanni
                               // solo di persone note
                               // dom indica il dominio, rng il codominio
_____________________________
```

Aggiunta di operazioni

- AddBirthday

Inserisce una nuova voce nell'elenco → modifica lo stato

- FindBirthday

Trova un compleanno, dato il nome di una persona → è una interrogazione (e, come tale, non modifica lo stato)

- Remind

Trova i compleanni che cadono in una data assegnata → è una interrogazione

- Replace

Sostituisce una voce preesistente in elenco, che si riferisce alla medesima persona → modifica lo stato

- Delete

Cancella una voce in elenco → modifica lo stato

Operazioni di modifica

```
_____AddBirthday_______
Δ BirthdayBook                  // Δ significa che produce una modifica dello stato
                                // dello schema associato, qui BirthdayBook

Name? : NAME                    // ? significa che è una variabile d'ingresso
Date? : DATE
_____________________________
Name? ∉ known                   // precondizioni sulle variabili
                                // prima di eseguire l'operazione

birthday' =                                 // postcondizioni sulle variabili
   birthday ∪ {Name? ↦ Date?}               // dopo l'esecuzione dell'operazione
                                            // X (prima), X' (dopo)
_____________________________
```

Osservazione

Le specifiche di un'operazione che cambia lo stato non dicono cosa succede alle variabili di stato non menzionate nelle postcondizioni, ad es. `known`, perché le loro variazioni possono essere provate a partire dall'invariante

```
known' = dom(birthday')                          // invariante
known' = dom(birthday ∪ {Name? ↦ Date?})  //postcondizione
known' = dom(birthday) ∪ dom({Name? ↦ Date?})
known' = known ∪ dom({Name? ↦ Date?})     // invariante
known' = known ∪ {Name?}
```

Operazioni di interrogazione

```
_____FindBirthday_______
Ξ BirthdayBook                  //  Ξ significa che è una query
Name? : NAME
Date! : DATE                    // ! significa che è una variabile d'uscita
_____________________________
Name? ∈ known                   // precondizione
Date! = birthday(Name?)         // postcondizione
_____________________________
```

```
_____Remind_____________
Ξ BirthdayBook
Today? : DATE
Cards! : ℙNAME
_____________________________
                                                  // precondizione
Cards! = {n ∈ known | birthday(n) = Today?}       // postcondizione
_____________________________
```

Operatori

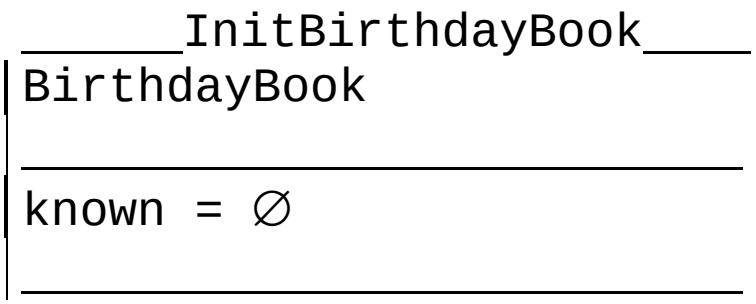
```
_____Replace_____________
ΔBirthdayBook
Name? : NAME
Date? : DATE
_____________________________
Name? ∈ known
birthday' =
    birthday ⊕ {Name? ↦ Date?}    // ⊕ è l'operatore di overriding
    // i cui operandi sono due funzioni: per ogni coppia di valori della seconda funzione
    // (a) sostituisce la coppia corrispondente della prima, se presente
    // (b) aggiunge la coppia alla prima, altrimenti
_____________________________
```

Operatori (cont.)

```
_____Delete_____________
Δ BirthdayBook
Name? : NAME
_____________________________
Name? ∈ known

birthday' = {Name?} ⩤ birthday   //  ⩤ è l'operatore di sottrazione dal
    // dominio: l'operando di sx è un insieme che appartiene al dominio della
    // funzione che costituisce l'operando di dx. Dalla funzione a dx vengono
    // rimosse tutte le coppie il cui primo elemento appartiene all'insieme a sx
_____________________________
```

Stato iniziale



Commenti

Le specifiche di un'operazione (sia di cambiamento, sia di interrrogazione) non dicono cosa succede se le precondizioni non sono verificate. Ad es.

Che succede se si tenta di inserire due volte il compleanno della stessa persona?

Oppure

Cosa succede se si ricerca il compleanno di una persona non in elenco?

Si possono produrre specifiche aggiuntive per rispondere a queste domande (le specifiche in linguaggio Z possono essere date *incrementalmente*), che verranno poi combinate con le specifiche originali (Z è *compositivo*)

Incrementi

Definizione di un nuovo tipo

```
REPORT ::= ok | already_known | not_known
```

Success

$$result! : REPORT$$

$$result! = ok$$

AlreadyKnown

$$\Xi BirthdayBook$$
$$Name? : NAME$$
$$result! : REPORT$$

$$Name? \in known$$
$$result! = already\_known$$

Incrementi (cont.)

NotKnown

Ξ BirthdayBook
Name? : NAME
result! : REPORT

Name? $\notin$ known
result! = not_known

Composizione degli schemi

RAddBirthday

(AddBirthday ∧ Success) ∨ AlreadyKnown // versione robusta di AddBirthday

che equivale a

RAddBirthday

```
Δ BirthdayBook
Name? : NAME
Date? : DATE
result! : REPORT

(Name? ∉ known
birthday' = birthday ∪ {Name? ↦ Date?}
result! = ok) ∨

(Name? ∈ known
birthday' = birthday
result! = already_known)
```

Composizione degli schemi (cont.)

RFindBirthday

$(\text{FindBirthday} \wedge \text{Success}) \vee \text{NotKnown}$ // versione robusta di FindBirthday

RRemind

$\text{Remind} \wedge \text{Success}$ // versione robusta di Remind

Ulteriori operatori

- Sequenza: es. `seq CHAR`
  Funzione che ha come dominio un intervallo di numeri naturali, ad es.
  `S = {1 ↦ a, 2 ↦ b, 3 ↦ c}`, oppure, più sinteticamente `S = <a,b,c>`

- Lunghezza di una sequenza: es. `#S` (vale 3 nel caso dell'esempio precedente)
  Elemento `i`-esimo di una sequenza `S`: `S(i)`, dove `1 ≤ i ≤ #S`

- Campo: es., dato `b: BirthdayBook`
  `b.known` oppure
  `b.birthday(n)`, dove `n ∈ b.known`

- Concatenazione di sequenze: es. `<a,b> ⌢ <c> = <a,b,c>`

- Restrizione del dominio di una funzione: es. `{Name?} ◁ birthday`
  Conserva solo le coppie della funzione a dx il cui primo elemento appartiene all'insieme a sx

Ulteriori operatori (cont.)

- Restrizione del codominio di una funzione:
  es. `{10/08, 09/11}` $\rhd$ `birthday`
  Conserva solo le coppie della funzione a dx il cui secondo elemento appartiene all'insieme a sx

- Sottrazione dal codominio di una funzione:
  es. `{10/08, 09/11}` $\rhd$ `birthday`
  Rimuove dalla funzione a dx tutte le coppie il cui secondo elemento appartiene all'insieme a sx

Z: vantaggi e non

Vantaggi

- Prove (semi-automatiche) di correttezza
- Verifica automatica di completezza e consistenza
- Spesso usato per le specifiche di sistemi safety-critical (sono utili a tal fine le condizioni invarianti)

Problemi

- Mancanza di costrutti di scope / strutturazione
- Consente di rappresentare bene le specifiche di basso livello ma non quelle più generali