UNIVERSITÀ
DI PARMA

DIPARTIMENTO DI SCIENZE
MATEMATICHE, FISICHE E INFORMATICHE

Corso di Laurea in Informatica

Tesi di Laurea

Da Specifiche Z a Programmi Java tramite JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Lorenzo De Santis

Anno Accademico 2016/2017

Indice

Introduzione

In questo lavoro di tesi presentiamo un approccio per la derivazione di un programma eseguibile dalla sua specifica formale. Precisamente, il nostro punto di partenza è il linguaggio di specifica Z, mentre il punto di arrivo è un programma in linguaggio Java.

Il linguaggio Z [1, 2] si appoggia sulla teoria matematica degli insiemi per modellare formalmente lo stato e i cambiamenti di stato di un qualsiasi sistema. Il sistema viene descritto come una macchina a stati mediante l'utilizzo di variabili matematiche, insiemi e relazioni fra questi.

In letteratura sono descritti vari approcci per derivare un programma dalle sue specifiche [3]. In generale la derivazione si basa sull'applicazione ripetuta di regole di riscrittura che permettono di trasformare la specifica (solitamente non eseguibile) in un programma eseguibile, semanticamente equivalente alla specifica, e corretto per costruzione (ammesso che le regole siano corrette).

Un esempio di tale tecnica è il B-Method [4], un metodo di sviluppo del software che prevede di partire dal linguaggio di specifica B, un linguaggio molto simile a Z, ma un po' più di basso livello rispetto a quest'ultimo, per arrivare a programmi nei linguaggi Ada, C o C++.

L'approccio che proponiamo in questo lavoro di tesi si basa sull'idea di ridurre il più possibile la distanza (il "*gap*") che esiste tra il linguaggio di specifica e il linguaggio di implementazione, arricchendo quest'ultimo di nuove possibilità che lo rendano molto più vicino al linguaggio di specifica utilizzato. L'aggiunta di queste nuove possibilità al linguaggio di implementazione avviene tramite la realizzazione di nuove librerie, sfruttando le capacità di astrazione del linguaggio di implementazione, in particolare quelle offerte dal paradigma Object-Oriented.

Nello specifico, in questa tesi utilizzeremo Java arricchito con le strutture presenti all'interno di JSetL [5, 6, 7], una libreria sviluppata all'Università di Parma, i cui servizi si avvicinano molto alla notazione utilizzata nel linguaggio Z. JSetL, infatti, nasce con l'intento di modellare un sistema di programmazione con vincoli, tipico della programmazione dichiarativa, mantenendo anche le caratteristiche di un linguaggio imperativo orientato agli oggetti come Java. In particolare, JSetL offre variabili insiemistiche ed intere, operazioni su insiemi, formulazione e

risoluzione di vincoli logici e non-determinismo.

Alle nuove possibilità offerte da JSetL ne verranno aggiunte altre, realizzate da un package Java, denominato `JSetL.z`, sviluppato appositamente, che permetteranno di avvicinare ulteriormente il codice eseguibile alla notazione Z usata nella specifica. In questo modo si otterrà un "*mapping*" preciso, facilmente automatizzabile, tra costrutti Z e corrispondenti frammenti di codice Java arricchito con JSetL e il package JSetL.z. Tale "mapping" verrà formalizzato tramite opportune *regole di riscrittura* che descrivano come passare, in modo algoritmico, da notazione Z a codice Java.

Osserviamo che non verrà fornita una dimostrazione della correttezza logica dei programmi generati: questa si basa piuttosto sull'assunzione di correttezza, da una parte della specifica Z e dei metodi di libreria Java forniti da JSetL, e dall'altra del procedimento di riscrittura utilizzato per trasformare la specifica Z in codice Java. La verifica di correttezza di quest'ultimo, comunque, è sicuramente facilitata dalla stretta corrispondenza stabilita tra i costrutti Z e il codice Java + JSetL generato.

Infine precisiamo che il presente lavoro di tesi non vuole essere una guida alla stesura di una buona specifica Z, ma che i documenti di specifica Z verranno solo descritti per *come sono*, non per *come vengono creati*. Il capitolo su Z ha quindi il solo scopo di introdurre la notazione e la sintassi tipiche del linguaggio, al fine di rendere maggiormente comprensibile il processo di conversione del documento e l'applicazione delle regole di riscrittura.

Capitolo I

Linguaggio di specifica Z

Z [1, 2] è un linguaggio di specifica per la formalizzazione di sistemi software, basato sui concetti e sulla sintassi della matematica, e in particolare della teoria degli insiemi. Pertanto, un documento Z contiene principalmente formule riconducibili alla notazione matematica e solo pochi simboli e notazioni grafiche propri del linguaggio. Forniamo al lettore un esempio concreto di specifica a scopo illustrativo nell'Appendice a pagina 78. Un documento Z si presenta come una serie di:

- insiemi dati
- variabili globali
- vincoli globali
- schemi

1 Insiemi dati

Gli insiemi dati sono collezioni di oggetti che si intende utilizzare, ma i cui componenti e valori non sono obbligatoriamente specificati al momento della loro introduzione.

Esempio I.1: Introduciamo l'insieme degli *STATI*, senza specificarne i componenti, e l'insieme dei *CONTINENTI*, di cui, invece, forniamo subito i componenti

$$[STATI]$$

$$CONTINENTI ::= EUROPA \mid AMERICA \mid ASIA \mid AFRICA \mid OCEANIA$$

2 Variabili globali

In Z le variabili sono di tipo matematico, ovvero non modificabili nè riassegnabili, al contrario di quelle dei linguaggi di programmazione imperativi. In particolare, si dicono *variabili globali* le variabili che non siano dichiarate all'interno di uno schema, poiché le si vuole rendere "visibili" ovunque nel documento.

Infatti, le variabili hanno regole di scope per le quali risultano visibili solo negli schemi nei quali sono dichiarate e negli schemi che utilizzino a loro volta tali schemi, con offuscamento temporaneo di variabili globali nel caso uno stesso nome venga riutilizzato.

Esempio I.2: Introduciamo una variabile globale di nome *MioContinente* e del tipo *CONTINENTI*

$$MioContinente : CONTINENTI$$

3 Vincoli globali

I vincoli globali sono predicati di vario tipo che esprimono condizioni sui dati, le quali devono sempre essere rispettate.

Esempio I.3: Vincoliamo la variabile *MioContinente* al valore *EUROPA*

$$MioContinente = EUROPA$$

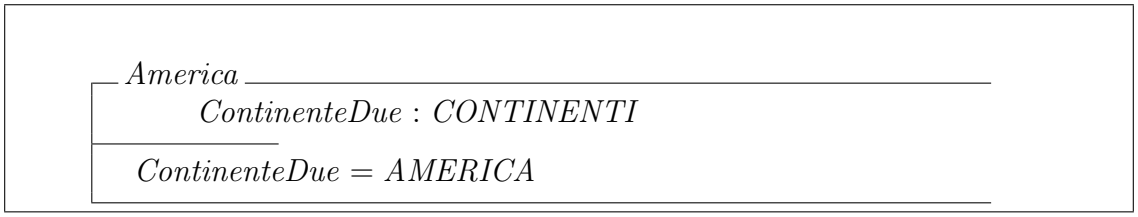
4 Schemi

Gli schemi rappresentano il componente principale di un documento Z e vengono utilizzati principalmente per tre funzioni:

- descrivere parte (o la totalità) di una struttura dati, con eventuali vincoli sui valori
- descrivere parte (o la totalità) dello stato del sistema
- descrivere parte (o la totalità) di un'operazione che può essere eseguita sul sistema.

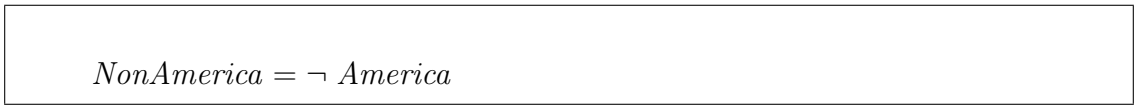
Ogni schema Z ha un nome univoco ed è formato da una prima parte, eventualmente vuota, contenente *definizioni* e una seconda parte, eventualmente vuota, contenente *predicati* logici.

Esempio I.4: Creiamo lo schema *America* avente come unica definizione la variabile *ContinenteDue* del tipo *CONTINENTI*, la quale è vincolata nella seconda parte al valore *AMERICA*



È inoltre possibile definire nuovi schemi tramite operazioni che combinino o modifichino schemi introdotti precedentemente: ciascuno di questi tuttavia si può ricondurre a uno schema Z corrispondente nella forma canonica (vedi pagina 33).

Esempio I.5: Lo schema *NonAmerica* è definito come risultato dell'operazione di negazione dello schema *America*



In Z, gli schemi e le variabili possono presentare una variante *decorata*: una *decorazione* è una componente grafica (come "?", "!" o "′") che può essere giustapposta a un nome. In base alla decorazione utilizzata, la variante dello schema o della variabile assumerà significati particolari, come descritto in seguito. In particolare, uno schema o una variabile decorati con "′" si dicono "*primed*" o "*dashed*".

5 Qualche classificazione per gli schemi

Gli schemi possono avere una costruzione gerarchica, ovvero uno schema può fare riferimento un altro. Perciò introduciamo una distinzione (detta per *contenimento*) fra schemi *esterni*, ovvero non riferiti da nessun altro schema della stessa classe funzionale e schemi *interni*, ovvero riferiti da almeno un altro schema della stessa classe funzionale. In particolare, gli schemi utilizzati come argomento di un'operazione fra schemi per la definizione di un nuovo schema si considerano interni a quest'ultimo.

È possibile utilizzare alcuni schemi per descrivere le interazioni del sistema con l'utente: differenziamo quindi gli schemi per *orientamento all'esterno*. In particolare, alcuni schemi possono indicare valori che il sistema necessita di ottenere o comunicare all'utente: si indica la richiesta di un valore per una certa variabile giustapponendovi la decorazione "?" e si indica la stampa del valore di una certa variabile giustapponendovi la decorazione "!". Definiamo pertanto schemi di *input* quelli in cui appaia almeno una variabile decorata con "?" (anche in uno schema interno ad esso) e schemi di *output* quelli in cui appaia almeno una variabile decorata con "!" (anche in uno schema interno ad esso). Solitamente uno schema di input è anche di output, in quanto è buona norma dare responso all'utente che abbia eseguito un'azione con un certo input.

Definiamo infine una classificazione per *funzione* più puntuale degli schemi Z:

- *schemi di struttura*: specificano un nuovo tipo di oggetto dotato di alcune proprietà, utile per descrivere il sistema. Le proprietà dell'oggetto sono introdotte tramite una collezione di campi nella prima parte di definizioni dello schema. È possibile riconoscerli perché vengono utilizzati in altri schemi per la definizione del tipo di una variabile in modo diretto, come tipo, o in modo indiretto, tramite il costrutto *PowerSet*(ovvero l'*insieme delle parti*) o le *Funzioni Parziali*. Si considera esterno ogni schema di struttura che appaia nella definizione del tipo di almeno una variabile.

Esempio I.6: Creiamo lo schema *Viaggio* che introduce una nuova struttura dati contenente i due campi *Partenza* ed *Arrivo*, entrambi di tipo CONTINENTE: inoltre i valori dei campi di questa struttura dati sono vincolati ad essere diversi per essere ammissibili

Viaggio

$Partenza : CONTINENTI$
$Arrivo : CONTINENTI$

$Partenza \neq Arrivo$

- *schemi di stato*: descrivono i dati tramite il loro tipo e le relazioni e le condizioni fra di essi, ovvero il "modello" del sistema. È possibile riconoscerli perché sono gli schemi oggetto delle operazioni (argomento dei simboli Ξ (Xi) e Δ (Delta)) o schemi esterni a questi ultimi e inoltre tutte le loro variabili non presentano nessuna decorazione.

Esempio I.7: Creiamo lo schema *DiarioDiBordo* che modella lo stato del sistema: in particolare, l'unica variabile *Viaggi* è un sottoinsieme di tutte le possibili strutture dati descritte nello schema *Viaggio*, e cioè un elemento del PowerSet ($\mathbb{P}$) di *Viaggio*. È inoltre presente il vincolo che la cardinalità di *Viaggi* non superi il valore 100

DiarioDiBordo

$$Viaggi : \mathbb{P}\ Viaggio$$

$$\#\ Viaggi \leq 100$$

- *schemi Delta*: sono gli schemi in cui si verifica un *cambiamento di stato*. Il cambiamento di stato avviene con la variazione del valore di una o più variabili di stato del sistema: si indica con il simbolo Δ (Delta) lo schema di cui si intende modificare le variabili e la modifica si esplicita associando a una variabile dello schema decorata con "′" il suo nuovo valore. È possibile riconoscere questi schemi dal simbolo Δ (anche in uno schema interno ad esso) e/o dall'associazione ad almeno una variabile decorata con "′" di un valore diverso da quello della versione non decorata della variabile stessa.

Esempio I.8: Creiamo lo schema Delta *AggiungiViaggio* che descrive l'operazione di modifica della variabile *Viaggi* presente nello schema *DiarioDiBordo*: in particolare, si richiede in input all'utente il Viaggio *Nuovo*? che viene poi unito all'insieme dei *Viaggi* già esistenti

AggiungiViaggio

$$\Delta DiarioDiBordo$$
$$Nuovo? : Viaggio$$

$$Viaggi' = Viaggi \cup \{Nuovo?\}$$

- *schemi Xi*: sono gli schemi in cui avviene una consultazione dello stato senza la modifica dello stesso: per indicare lo schema contenente la/e variabile/i da consultare si usa il simbolo Ξ (Xi). È possibile riconoscere questi schemi dal simbolo Ξ (anche in uno schema interno ad esso) e/o dal fatto che ogni variabile decorata con "′" viene associata al valore della versione non decorata della variabile stessa. Sono generalmente schemi di output o input/output.

Esempio I.9: Creiamo lo schema Xi *VisualizzaDiario* che descrive l'operazione di consultazione della variabile *Viaggi* presente nello schema *DiarioDiBordo*.

VisualizzaDiario

$$\Xi DiarioDiBordo$$
$$Visualizza! : \mathbb{P}\ Viaggio$$

$$Visualizza! = Viaggi$$

- *schemi di inizializzazione*: sono schemi non di input nè di output che descrivono un'associazione di variabili. È possibile riconoscerli perché presentano variabili con le decorazioni "′", ma nessuna variabile con le decorazioni "?" o "!".

Esempio I.10: Creiamo lo schema di inizializzazione *DiarioIniziale* che descrive l'operazione iniziale di associazione delle variabili nello schema *DiarioDiBordo*: in particolare, si vincola l'unica variabile *Viaggi* ad essere un insieme vuoto

DiarioIniziale

$$DiarioDiBordo'$$

$$Viaggi' = \varnothing$$

Raggruppiamo gli ultimi 3 tipi di schema sotto il nome di *schemi d'azione*, in quanto descrivono parte (o la totalità) di azioni che l'utente può compiere sul sistema.

I predicati presenti in uno schema di struttura riguardano le condizioni sui valori propri di ogni istanza di oggetto riconducibile allo schema.

I predicati presenti all'interno di uno schema di stato descrivono condizioni che devono essere sempre verificate (all'inizio del funzionamento del sistema e prima e

dopo l'applicazione di ogni schema d'azione) per garantire l'integrità e la coerenza dei dati che modellano il sistema.

I predicati presenti negli schemi Delta e Xi corrispondenti all'associazione di variabili che presentino le decorazioni "′" e !" descrivono lo svolgimento dell'operazione, mentre i restanti predicati descrivono le precondizioni da verificare prima dello svolgimento dell'operazione; uno schema di inizializzazione, invece, non ha precondizioni ma solo associazioni di variabili che implicano il *"set"* o il *"reset"* del sistema.

6 Standard Toolkit

Lo Standard Toolkit è un documento Z in cui vengono definiti alcuni insiemi, funzioni e altre strutture fondamentali, utili generalmente per la stesura di un qualsiasi documento Z.

Lo Standard Toolkit contiene, ad esempio, la definizione di $\mathbb{N}$, *dom*, *ran*,...

Esempio I.11: All'interno dello Standard Toolkit, l'insieme degli interi, $\mathbb{Z}$, è introdotto come insieme dato mentre l'insieme dei naturali, $\mathbb{N}$, è definito come un sottinsieme di $\mathbb{Z}$

$$[\mathbb{Z}]$$

$$\mathbb{N} : \mathbb{P}\,\mathbb{Z}$$

$$\mathbb{N} = \{n : \mathbb{Z} \mid n \geq 0\}$$

Di solito lo Standard Toolkit si considera un *"parent"* di ogni documento Z, ovvero le sue definizioni vengono incluse automaticamente nel documento Z come se fossero scritte all'inizio del documento. Pertanto, per funzionare correttamente, un programma Java che implementi un documento Z deve prevedere anche l'implementazione delle strutture Z dello Standard Toolkit.

Durante la trattazione, considereremo che lo Standard Toolkit appaia all'inizio del documento Z da convertire e quindi che venga convertito insieme a tutte le altre strutture della specifica.

Capitolo II

JSetL

JSetL [5, 6, 7] è una libreria Java che affianca alla programmazione ad oggetti alcune strutture per la programmazione dichiarativa tipiche della programmazione logica con vincoli: variabili logiche, liste, insiemi, unificazione, risoluzione di vincoli e non-determinismo. In particolare, sono supportati l'unificazione tra variabili logiche, liste ed insiemi e i vincoli di base sugli insiemi e sugli interi. Il non-determinismo viene utilizzato sia per la ricerca delle soluzioni che per i vincoli su insiemi come uguaglianza e unione. L'utente può inoltre espandere JSetL definendo nuovi vincoli propri, aggiungendoli quindi alla lista dei vincoli di base già forniti.

In questo capitolo introduciamo le principali strutture dati e funzioni offerte da JSetL ed utilizzate nella traduzione delle specifiche Z descritta nel capitolo successivo.

1 La Classe `LVar`

La classe `LVar` modella variabili logiche di tipo matematico come quelle presenti in Z. Esse non posseggono alcun valore modificabile, al contrario delle variabili di un linguaggio di programmazione imperativo. Tuttavia, si possono associare valori alle variabili logiche attraverso relazioni (o *vincoli*), che coinvolgono variabili logiche e valori da alcuni domini specifici.

Quando il dominio di una variabile è limitato ad un singolo valore, diciamo che la variabile è *legata* a (o *istanziata* con) questo valore. Altrimenti, la variabile è *libera*. Con abuso di terminologia, si può dire che il valore associato ad una variabile legata `x` è il *valore* di `x`.

La relazione di uguaglianza, in particolare, consente di associare un valore preciso a una variabile logica. Ad esempio, se `x` è una variabile logica, l'uguaglianza `x = 3` forza `x` ad essere vincolata al valore `3`. Il valore di una variabile logica è immutabile. Quindi non può essere modificato, ad esempio, da un assegnamento.

Una variabile logica può anche avere associato un *nome esterno*. Il nome esterno è un valore di stringa che può essere utile quando si stampa la variabile e i possibili vincoli che la coinvolgano.

La classe `LVar` fornisce costruttori per la creazione di variabili logiche e numerosi metodi semplici per il test e la manipolazione di variabili logiche, nonché vincoli di base sulle variabili logiche (vale a dire, uguaglianza e disuguaglianza, appartenenza e non appartenenza).

La classe `LVar` ha una serie di sottoclassi che trattano valori specifici di tipi specifici. Una distinzione primaria è tra valori *atomici* e *strutturati*. Le variabili che possono assumere valori atomici sono le *variabili logiche intere* (classe `IntLVar`, vedi più avanti). Viceversa, le variabili che possono assumere valori strutturati sono le *variabili logiche insiemistiche* (classe `LSet`, vedi pagina 14), i cui valori sono insiemi, eventualmente parzialmente specificati, di elementi di qualsiasi tipo.

Il metodo `output`

```
void output()
```

Il metodo `output` è implementato per ogni oggetto logico all'interno di JSetL, compresi gli oggetti `SubLVar` introdotti con il package JSetL.z (vedi pagina 27). Il metodo stampa il nome esterno dell'oggetto logico su cui è invocato, seguito da `"="`, seguito da una rappresentazione del valore dell'oggetto logico, se è legato, o da `"unknown"` se è libero (ad esempio, `x = 3` o `y = unknown`).

Vincoli per la Classe `LVar`

- Appartenenza

  `Constraint in(LSet ls)`

  Restituisce il vincolo atomico `this` $\in$ `ls`, che richiede che questa variabile logica sia un membro dell'insieme logico `ls`. Quando risolto, questo vincolo non unifica questa variabile logica con ogni valore in `ls`. Il vincolo avrà successo se almeno una unificazione riesce. Si noti che se questa variabile logica è libera, risolvendo il vincolo, la variabile verrà legata non-deterministicamente a ogni valore in `ls`.

- Non-appartenenza

  `Constraint nin(LSet ls)`

  Restituisce il vincolo atomico `this` $\notin$ `ls`, che richiede che questa variabile logica non sia un membro del set logico `ls`.

- Uguaglianza

  `Constraint eq(Object o)`

  Restituisce il vincolo atomico `this = o`, che unifica questa variabile logica con l'oggetto `o`. In particolare, se `o` è un oggetto diverso da una variabile logica e questa variabile non è legata, questa variabile viene legata ad o dopo che il vincolo è stato risolto.

- Disuguaglianza

  `Constraint neq(Object o)`

  Restituisce il vincolo atomico `this` $\neq$ `o`, che richiede che questa variabile logica sia diversa dall'oggetto `o`.

2 La Classe `LSet`

Un *set logico s* (o, semplicemente, un *l-set s*) è un particolare tipo di variabile logica il cui valore è una coppia `<elems, rest>`, dove *elems* è un insieme $\{e_0,\ldots,e_n\}$, `n` $\geq$ `0`, di oggetti di tipi arbitrari (il *valore* dell'insieme), e *rest* è un *l-set vuoto* o un *l-set libero* che rappresenta il resto di *s*. Quando *rest* è un l-set libero *r*, diciamo che *s* rappresenta un insieme *aperto* e usiamo la notazione (astratta) $\{e_0,\ldots,e_n \mid r\}$ per indicarlo; al contrario, quando *rest* è l'insieme vuoto, diciamo che *s* rappresenta un insieme *chiuso* e usiamo la notazione (astratta) $\{e_0,\ldots,e_n\}$. Quando *elems* è l'insieme vuoto e *rest* è nullo, *s* è l'insieme vuoto e usiamo {} per indicarlo.

I set logici sono simili a quelli della logica per molti aspetti. In particolare, gli l-set possono rappresentare delle collezioni *parzialmente specificate*. L'ordine degli elementi e le ripetizioni non hanno importanza in un l-set.

Si noti che la cardinalità di un set parzialmente specificato non è determinata in modo univoco (anche se l'insieme è chiuso). Ad esempio, la cardinalità dell'insieme `{1, x}`, dove `x` è una variabile libera, può essere 1 o 2 a seconda che `x` venga successivamente legata a un valore uguale a 1 o diverso da 1, rispettivamente.

In JSetL, un insieme logico è definito come un'istanza della classe `LSet`. I metodi forniti dalla classe `LSet` servono a creare nuovi l-set per trattare gli insiemi come variabili logiche e per gestire valori eventualmente associati agli insiemi. Inoltre, in quanto variabili logiche, gli l-set possono essere utilizzati per porre vincoli che implementano le operazioni insiemistiche di base.

Metodi per la Classe `LSet`

- Creazione di un insieme vuoto

  `static LSet empty()`

  Restituisce un l-set vuoto.

- Aggiunta di un elemento

  `LSet ins(Object o)`

  Se questo l-set è legato, restituisce l'insieme n (nuovo) , il cui valore viene ottenuto aggiungendo `o` come elemento dell'insieme legato a questa variabile. Altrimenti restituisce l'insieme n (nuovo), il cui valore è l'insieme contenente `o` come elemento e questo l-set come parte restante (cioè, {`o | ls`}, dove *ls* è questo l-set).

Vincoli per la Classe `LSet`

- Uguaglianza

  `Constraint eq(LSet ls)`

  Restituisce il vincolo atomico `this = ls`, che unifica questo l-set con l'insieme `ls`. Se questo l-set è libero ed `ls` è legato ad un insieme `s`, questo l-set viene legato ad `s` dopo che il vincolo è stato risolto; al contrario, se anche `ls` è libero, questo l-set rimane libero. Si noti che, risolvendo questo vincolo, le variabili libere eventualmente presenti in `ls` saranno vincolate ai valori necessari, come richiesto dall'unificazione dei due insiemi.

- Disuguaglianza

  `Constraint neq(LSet ls)`

  Restituisce il vincolo atomico `this` $\neq$ `s`, che richiede che questo l-set sia diverso da `ls`.

- Inclusione

  `Constraint subset(LSet ls)`

  Restituisce il vincolo atomico `this` $\subseteq$ `ls`, che richiede che questo l-set sia incluso in `ls`.

- Differenza insiemistica

  `Constraint diff(LSet ls1, LSet ls2)`

  Restituisce il vincolo atomico `this = ls1  ls2`, che richiede che questo l-set sia la differenza degli altri due l-set.

- Intersezione insiemistica

  `Constraint inters(LSet ls1, LSet ls2)`

  Restituisce il vincolo atomico `this = ls1` $\cap$ `ls2`, che richiede che questo l-set sia l'intersezione degli altri due l-set.

- Unione insiemistica

  `Constraint union(LSet ls1, LSet ls2)`

  Restituisce il vincolo atomico `this = ls1` $\cup$ `ls2`, che richiede che questo l-set sia l'unione degli altri due l-set.

- Cardinalità

  `Constraint size(IntLVar ilv)`

  Restituisce il vincolo atomico `|this| = i` (rispettivamente, `|this| = ilv`), che richiede che la variabile intera `i` (o la variabile logica intera `ilv`) sia uguale alla cardinalità di questo l-set.

- Quantificatore universale

  `Constraint forallElems(LVar y, Constraint c)`

  Se questo insieme è legato, restituisce una congiunzione di vincoli atomici $\texttt{c}_0 \wedge \texttt{c}_1 \wedge \ldots \wedge \texttt{c}_n$ per tutti gli elementi $\texttt{e}_0, \ldots, \texttt{e}_n$ dell'insieme legato a questo l-set. Ciascun vincolo $\texttt{c}_i$ è ottenuto da `c` sostituendo tutte le occorrenze di `y` in esso con $\texttt{e}_i$. Ad esempio, se `s` è l'insieme {`1, 2, z`}, dove `z` è una variabile logica non legata, `s.forallElems (y, y.neq (0))` genera una congiunzione di vincoli logicamente equivalente a $\texttt{1} \neq \texttt{0} \wedge \texttt{2} \neq \texttt{0} \wedge \texttt{z} \neq \texttt{0}$. Crea un'eccezione `NotInitVarException` se questo l-set non è legato.

La Classe `IntLSet`

Le *variabili logiche insiemistiche intere* (o, più brevemente, variabili insiemistiche intere) sono un caso speciale di variabili logiche insiemistiche, i cui valori sono limitati ad essere insiemi di interi. Il dominio di una variabile insiemistica intera `s` può essere specificato quando la variabile `s` viene creata, fornendo i valori del limite inferiore e superiore del dominio associato alla variabile.

In JSetL, una variabile insiemistica intera è un oggetto della classe `IntLSet`. Tale classe estende `LSet` e quindi ne mutua tutti i metodi e in più fornisce un costruttore per esplicitare il valore dei limiti inferiore e superiore del dominio.

- Costruttore

  `IntLSet(String n, int p, int q)`

  Un costruttore, per la classe `IntLSet`, in cui `n` rappresenta il nome esterno della variabile insiemistica intera e `p` e `q` rappresentano rispettivamente i limiti inferiore e superiore del dominio associato alla variabile.

3 La Classe `IntLVar`

Le *variabili logiche intere* sono un tipo particolare di variabili logiche, i cui valori sono limitati a numeri interi. Inoltre, una variabile logica intera ha un *dominio finito* e un *vincolo aritmetico intero* (eventualmente vuoto) associato ad esso.

Il dominio è rappresentato da un *multi-intervallo*, cioè l'unione di `n` ($\texttt{n} \geq 0$) *intervalli* disgiunti, ovvero insiemi numerici limitati superiormente e inferiormente.

Un dominio per una variabile logica intera `v` può essere specificato quando la variabile `v` viene creata e viene automaticamente aggiornato quando i vincoli eventualmente posti su `v` sono risolti, al fine di mantenere la coerenza dei vincoli. Ad esempio, se `x` e `y` sono variabili logiche intere entrambe di dominio `[1..10]` e aggiungiamo il vincolo `x > y`, allora il dominio di `x` viene aggiornato a `[2..10]` e il dominio di `y` a `[ 1..9]`. Quando il dominio di una variabile è limitato a un singolo valore `k` (cioè al singleton {`k`}), la variabile viene *legata* a questo valore. Al contrario, se il dominio viene ridotto all'insieme vuoto, significa che i vincoli che coinvolgono tale variabile non sono soddisfacibili.

I vincoli aritmetici associati a variabili logiche intere vengono generati mediante la valutazione di *espressioni logiche intere*, vale a dire le espressioni costruite utilizzando i soliti operatori aritmetici `sum`(`+`), `sub`(`-`), `mul`(`*`) e `div`(`/`), applicati a variabili logiche intere e costanti intere. La valutazione di un'espressione logica intera `e` fornisce una nuova variabile logica intera $\texttt{X}_1$ con un vincolo associato:

$$\texttt{X}_1 \texttt{ = e}_1 \texttt{ } \wedge \texttt{ X}_2 \texttt{ = e}_2 \texttt{ } \wedge \texttt{ ... } \wedge \texttt{ X}_n \texttt{ = e}_n$$

dove $\texttt{e}_1\texttt{,...,e}_n$ sono le sottoespressioni che appaiono in `e` e $\texttt{X}_1\texttt{,...,X}_n$ sono variabili logiche intere interne, che rappresentano una forma semplificata dell'espressione `e`.

Ad esempio, se `e` è l'espressione logica intera `x.sum (y.sub (1))`, dove `x` e `y` sono variabili logiche intere, la valutazione di `e` restituisce la variabile logica intera $\texttt{X}_1$ con il vincolo associato $\texttt{X}_1 \texttt{ = x + X}_2 \texttt{ } \wedge \texttt{ X}_2 \texttt{ = y - 1}$.

In JSetL, una variabile logica intera è un'istanza della classe `IntLVar`, che estende la classe `LVar`. I valori delle variabili logiche intere sono numeri interi. I vincoli eventualmente associati a variabili logiche intere sono istanze della classe `Constraint`.

Metodi per la Classe `IntLVar`

- Somma

  `IntLVar sum(Integer k)`

  Restituisce una variabile logica intera $\texttt{X}_1$ con un vincolo associato $\texttt{X}_1 = \texttt{X}_0 + \texttt{k} \wedge \texttt{C}_0$, dove $\texttt{X}_0$ è questa variabile logica e $\texttt{C}_0$ è il vincolo associato.

  `IntLVar sum(IntLVar v)`

  Restituisce una variabile logica intera $\texttt{X}_1$ con un vincolo associato $\texttt{X}_1 = \texttt{X}_0 + \texttt{v} \wedge \texttt{C}_\texttt{v} \wedge \texttt{C}_0$, dove $\texttt{X}_0$ è questa variabile logica, $\texttt{C}_0$ il suo vincolo associato e $\texttt{C}_\texttt{v}$ il vincolo associato alla variabile logica `v`.

- Sottrazione

  `IntLVar sub(Integer k)`

  `IntLVar sub(IntLVar v)`

  Esattamente come per la Somma, ma con il `+` rimpiazzato da `-` nel vincolo associato.

- Moltiplicazione

  `IntLVar mul(Integer k)`

  `IntLVar mul(IntLVar v)`

  Esattamente come per la Somma, ma con il `+` rimpiazzato da `*` nel vincolo associato.

- Divisione

  `IntLVar div(Integer k)`

  Restituisce una variabile logica intera $\texttt{X}_1$ con un vincolo associato $\texttt{X}_1 = \texttt{X}_0 / \texttt{k} \wedge \texttt{C}_0 \wedge \texttt{k} \neq 0$, dove $\texttt{X}_0$ è questa variabile logica e $\texttt{C}_0$ è il vincolo associato. Si noti che si fa riferimento all'operazione di divisione intera esatta, senza alcun tipo di arrotondamento o troncamento.

  `IntLVar div(IntLVar v)`

  Restituisce una variabile logica intera $\texttt{X}_1$ con un vincolo associato $\texttt{X}_1 = \texttt{X}_0 / \texttt{v} \wedge \texttt{C}_\texttt{v} \wedge \texttt{C}_0 \wedge \texttt{v} \neq 0$, dove $\texttt{X}_0$ è questa variabile logica, $\texttt{C}_0$ il suo vincolo associato e $\texttt{C}_\texttt{v}$ il vincolo associato alla variabile logica `v`.

Vincoli per la Classe `IntLVar`

- Equazione

  `Constraint eq(Integer k)`

  Restituisce il vincolo $\texttt{X}_0 = \texttt{k} \wedge \texttt{C}_0$, dove $\texttt{X}_0$ è questa variabile logica e $\texttt{C}_0$ è il relativo vincolo associato.

  `Constraint eq(IntLVar v)`

  Restituisce il vincolo $\texttt{X}_0 = \texttt{v} \wedge \texttt{C}_0 \wedge \texttt{C}_\texttt{v}$, dove $\texttt{X}_0$ è questa variabile logica, $\texttt{C}_0$ il suo vincolo associato e $\texttt{C}_\texttt{v}$ il vincolo associato alla variabile logica `v`.

- Disequazione

  `Constraint neq(Integer k)`

  `Constraint neq(IntLVar v)`

  Esattamente come per i vincoli di Equazione, ma con il simbolo = rimpiazzato da $\neq$ nel vincolo generato.

  `Constraint le(Integer k)`

  `Constraint le(IntLVar v)`

  Esattamente come per i vincoli di Equazione, ma con il simbolo = rimpiazzato da $\leq$ nel vincolo generato.

  `Constraint lt(Integer k)`

  `Constraint lt(IntLVar v)`

  Esattamente come per i vincoli di Equazione, ma con il simbolo = rimpiazzato da < nel vincolo generato.

  `Constraint ge(Integer k)`

  `Constraint ge(IntLVar v)`

  Esattamente come per i vincoli di Equazione, ma con il simbolo = rimpiazzato da $\geq$ nel vincolo generato.

  `Constraint gt(Integer k)`

  `Constraint gt(IntLVar v)`

  Esattamente come per i vincoli di Equazione, ma con il simbolo = rimpiazzato da > nel vincolo generato.

4 La Classe `LMap`

Le *funzioni parziali* sono un caso particolari degli insiemi logici, i cui elementi sono esclusivamente `coppie ordinate`. Le coppie ordinate sono particolari variabili logiche della forma `<key, value>` dove `key` rappresenta l'oggetto che rappresenta la coppia e `value` è il valore a cui viene associato. L'insieme a cui appartengono tutti gli oggetti `key` è detto *dominio*, mentre l'insieme a cui appartengono tutti gli oggetti `value` è detto *rango*.

In particolare, le funzioni parziali posseggono una proprietà detta *funzionalità*: questa proprietà prevede che, per ogni oggetto `key` appartenente al dominio, esista al massimo una coppia avente `key` come primo elemento, ovvero che esista al massimo un oggetto `value` associato ad ogni oggetto `key`.

In JSetL, le funzioni parziali sono rappresentate dalla classe `LMap` [8]. La classe `LMap` estende `LSet`, dalla quale eredita in particolare tutti i vincoli associati. In questo modo, una `LMap` è un insieme logico con la particolarità che i suoi elementi sono oggetti della classe `Pair`.

La classe `Pair` rappresenta un'astrazione per le coppie. Sulle coppie sono possibili tutti i vincoli e le proprietà delle variabili logiche, in quanto `Pair` estende la classe `LVar`. In particolare, possono essere o non essere legate ad un valore.

Metodi per la Classe `LMap`

- Creazione funzione parziale vuota

  `static LMap empty()`

  Restituisce la funzione parziale vuota.

- Aggiunta di una coppia

  `LMap ins(Pair p)`

  Se questa `LMap` è legata, restituisce la (nuova) `LMap` *lm* , il cui valore viene ottenuto aggiungendo `p` come coppia della funzione parziale legata a questa variabile. Altrimenti restituisce la (nuova) `LMap` *lm*, il cui valore è la funzione parziale contenente `p` come coppia e questo `LMap` come parte restante (cioè, {`p | this`}, dove *this* è questa `LMap`).

Vincoli per la Classe `LMap`

- Dichiarazione di dominio

  `Constraint dom(LSet d)`

  Il metodo prende come argomento un oggetto `d` di tipo `LSet` che rappresenta il dominio della funzione e restituisce il relativo vincolo `Constraint`.

- Dichiarazione di rango

  `Constraint ran(LSet r)`

  Il metodo prende come argomento un oggetto `r` di tipo `LSet` che rappresenta il codominio della funzione e restituisce il relativo vincolo `Constraint`.

5 La Classe `Constraint`

I vincoli rappresentano condizioni che possono essere imposte su variabili logiche e collezioni logiche, nonché sugli oggetti creati dalle loro sottoclassi derivate. Queste condizioni possono essere applicate anche se gli oggetti logici coinvolti non hanno alcun valore preciso associato ad essi.

Un vincolo in JSetL è un'espressione che può assumere una delle forme:

- vincoli atomici
  - il *vincolo vuoto*, indicato con `[]`
  - e_0`.op(`e_1`,...,`e_n`)`, con `n = 0,...,3`
  - `op(`e_0`,`e_1`,...,`e_n`)`, con `n = 0,...,3`

  dove `op` è il nome del vincolo e e_i $(0 \leq i \leq 3)$ sono espressioni il cui tipo e numero dipende da `op`. In particolare `op` può essere una collezione di metodi predefiniti che implementano operazioni come l'uguaglianza, la disuguaglianza, il confronto intero, nonché le operazioni insiemistiche di base (come l'appartenenza, l'unione, l'intersezione, ...).

- vincoli composti:
  - C_1`.and (`C_2`)` (congiunzione)
  - C_1`.or (`C_2`)` (disgiunzione)
  - C_1`.impliesTest (`C_2`)` (implicazione)

  dove C_1 e C_2 sono vincoli JSetL e `and, or, impliesTest` rappresentano rispettivamente la congiunzione logica ($C_1 \wedge C_2$), la disgiunzione logica ($C_1 \vee C_2$) e l'implicazione logica ($C_1 \Rightarrow C_2$) tra C_1 e C_2.

- vincoli di negazione:
  - C_1`.notTest()` (negazione)

  dove C_1 è un vincolo JSetL e `notTest` rappresenta la negazione logica di C_1 ($\neg C_1$).

I vincoli in JSetL sono definiti come istanze della classe `Constraint`. Gli oggetti di classe `Constraint` vengono creati utilizzando costruttori e altri metodi della classe `Constraint` (come `and,or,impliesTest,...`), nonché come risultato di chiamate alle *funzioni di vincolo* fornite dalle classi che implementano i vari oggetti logici.

6 La Classe `SolverClass`

I vincoli vengono risolti utilizzando un *risolutore di vincoli*. In JSetL un risolutore di vincoli può essere creato come un'istanza della classe `SolverClass`: questa classe fornisce metodi per porre nuovi vincoli, ovvero aggiungerli all'attuale raccolta di vincoli (*store di vincoli*), nonché la consultazione, la verifica di soddisfacibilità e la ricerca di (tutte le) soluzioni per i vincoli accumulati.

I vincoli possono essere accumulati in un risolutore specifico aggiungendoli al suo store. La raccolta di vincoli in uno store è interpretata logicamente come una congiunzione di vincoli: pertanto, ogni inserimento di un nuovo vincolo, crea un nuovo congiunto nello store in cui viene aggiunto.

Metodi per la Classe `SolverClass`

- Aggiunta di un vincolo

  `void add(Constraint c)`

  Aggiunge un vincolo `c` (atomico o composto) allo store di questo risolutore. In questa fase non viene eseguita alcuna elaborazione del vincolo aggiunto.

- Test di soddisfacibilità

  `boolean check()`

  Controlla se il vincolo `C` attualmente nello store di questo risolutore è soddisfacibile oppure no e restituisce rispettivamente `true` o `false`. Se `C` è soddisfacibile, viene calcolata una soluzione, se possibile. Per soluzione, si intende un insieme di sostituzioni per le variabili logiche libere presenti nel vincolo. La computazione della soluzione può comportare lo sfruttamento del non-determinismo. Lo store dei vincoli risultante conterrà eventualmente una forma semplificata del vincolo `C`. Al contrario, se `C` è insoddisfacibile, tutte le variabili libere in `C` e lo store dei vincoli rimangono invariati.

Capitolo III

Il package `JSetL.z`

Per mantenere semplici le regole di conversione dai documenti Z ai sorgenti Java, in modo che quest'ultimi assomiglino il più possibile ai primi, i servizi offerti dalla libreria JSetL sono stati espansi con l'introduzione del sotto-package `JSetL.z`. All'interno di questo package sono state definite interfacce e classi astratte che descrivono i servizi necessari al funzionamento dei sorgenti Java generati dalla conversione. Inoltre, per ciascun componente è stata fornita un' implementazione minimale, in modo che il risultato della conversione sia da subito eseguibile ed utilizzabile.

In particolare, all'interno del package `JSetL.z` sono presenti:

- la classe astratta `ZDocument`, che definisce gli oggetti e i metodi per la conservazione dello stato e l'esecuzione delle operazioni di un generico documento Z.
- l'interfaccia `ZInterface`, che definisce i metodi per l'interazione fra il sistema definito nel documento Z e l'utente.
- la classe astratta `SubLVar`, che definisce i metodi per modellare il generico schema di struttura.
- l'interfaccia `ZDynamicState`, che definisce i metodi per modellare i vincoli dinamici dello stato del sistema.
- l'interfaccia `ZStaticState`, che definisce i metodi per modellare i vincoli statici dello stato del sistema.

1 La classe astratta `ZDocument`

La classe astratta `ZDocument` si occupa di creare e gestire tutte le strutture necessarie a permettere il funzionamento del sistema descritto nel documento di specifica.

Per quanto riguarda i campi dati, la classe `ZDocument` contiene i riferimenti a:

- un oggetto `ZInterface`, che serve per aprire l'interfaccia interattiva nel `main` della classe principale.
- un oggetto `ZDynamicState`, che serve per eseguire il `rebound`, ovvero la *riassociazione*, delle variabili di stato e tenere traccia dei vincoli dinamici.
- un oggetto `ZStaticState`, che serve per aggiungere e tenere traccia dei vincoli statici.
- un oggetto `SolverClass`, che serve per trovare le soluzioni ai vincoli posti durante le singole operazioni.
- un oggetto `Constraint`, che funge da accumulatore per i vincoli temporanei.
- quattro liste, che contengono rispettivamente i riferimenti agli insiemi dati e alle variabili temporanee di input, modifica e output.

Per quanto riguarda le operazioni, la classe `ZDocument` offre i seguenti metodi:

- Costruttore

  `protected ZDocument()`

  Crea ed inizializza tutti i campi dati. Viene invocato nel metodo `main` della classe principale.

- Aggiunta di un vincolo temporaneo

  `protected void tempPost(Constraint c)`

  Aggiunge il vincolo `c` a un accumulatore di vincoli temporaneo, ovvero l'oggetto `Constraint` nei campi dati. Come le liste, anche l'accumulatore è temporaneo e il suo valore è utilizzato solo per la durata di un'operazione e reinizializzato al suo termine.

- Aggiunta di un vincolo statico

  `public void statPost(Constraint c)`

  Aggiunge il vincolo `c` allo stato statico, ovvero l'oggetto `ZStaticState` nei campi dati.

- Esecuzione di un'operazione Z

  `protected void execute()`

  Permette l'esecuzione dei metodi delegando all'interfaccia le operazioni di input/output degli oggetti nelle liste temporanee, aggiungendo con il metodo `add` i vincoli statici, dinamici e temporanei all'oggetto `SolverClass`, verificando le soluzioni con il metodo `check`, aggiornando i vincoli dinamici con il metodo `rebound` e infine reinizializzando tutte le strutture temporanee, compreso l'oggetto `SolverClass`.

- Dichiarazione di un nuovo insieme dato

  `protected void givenSet(LSet ls)`

  Aggiunge alla rispettiva lista nei campi dati l'oggetto `LSet` fornito come parametro. Viene inoltre aggiunto un vincolo statico `disj` che disgiunge l'oggetto `LSet` da ogni altro insieme dato già appartenente alla lista.

- Dichiarazione di variabili temporanee

  `protected void askVar(Object o)`

  `protected void prmVar(Object o, Object oPrm)`

  `protected void ansVar(Object o)`

  Aggiungono alle rispettive liste nei campi dati gli oggetti passati. Le liste sono temporanee poiché i valori al loro interno vengono utilizzati solo durante lo svolgimento di un'operazione (all'interno del metodo `execute`) e le liste vengono reinizializzate al termine dell'operazione.

- Interfacce funzionali per le relazioni

  `protected IntLVar size(LSet s)`

  `protected LSet union(LSet s1, LSet s2)`

  `protected LSet diff(LSet s1, LSet s2)`

  `protected LSet inters(LSet s1, LSet s2)`

  Questi metodi permettono di utilizzare come funzioni le omonime relazioni presenti in JSetL per gli oggetti di classe `LSet`. Questo permette la riscrittura "*in loco*" delle relative funzioni matematiche all'interno di un'espressione insiemistica. Ciò avviene mediante la creazione di una nuova variabile temporanea che funge da risultato della funzione, su cui viene applicata la relazione prima di essere restituita.

2 L'interfaccia `ZInterface`

L'interfaccia `ZInterface` modella l'interazione fra il sistema descritto in una specifica Z e gli utenti di tale sistema. Per interazione intendiamo la capacità dell'utente di eseguire operazioni, ricevere messaggi e fornire dati, secondo quanto previsto dalla specifica del sistema.

In particolare, le operazioni possibili sono quelle descritte dagli schemi d'azione, mentre le informazioni sui dati da comunicare in entrata e in uscita sono contenute rispettivamente negli schemi di input e di output presenti nel documento di specifica.

L'interfaccia `ZInterface` prevede quindi i seguenti metodi:

- Avvio interfaccia

  ```
  public void open(ZDocument doc)
  ```

  Inizializza ed avvia l'interfaccia interattiva fra il sistema e l'utente, permettendo a quest'ultimo di eseguire operazioni sul sistema. È il metodo richiamato nel `main` della classe principale.

- Output del valore di una variabile

  ```
  public void output(LVar lv)
  public void output(IntLVar ilv)
  public void output(LSet ls)
  public void output(SubLVar slv)
  public void output(LMap lm)
  ```

  Permette al sistema di mostrare all'utente il valore di una variabile, ovvero di un oggetto di un qualsiasi tipo supportato da JSetL, compresi gli oggetti `SubLVar` introdotti nel package `JSetL.z`.

- Input del valore di una variabile

  ```
  public void input(LVar lv)
  public void input(IntLVar ilv)
  public void input(LSet ls)
  public void input(SubLVar slv)
  public void input(LMap lm)
  ```

  Permette all'utente di fornire al sistema il valore di una variabile, ovvero di un oggetto di un qualsiasi tipo supportato da JSetL, compresi gli oggetti `SubLVar`.

- Errore dell'interfaccia

  `public void inError()`

  `public void rebError()`

  `public void outError()`

  `public void subError(SubLVar slv)`

  `public void solError(SolverClass sc)`

  Permettono all'interfaccia di avvisare l'utente di un malfunzionamento dovuto a un errore implementativo delle interfacce e/o delle classi astratte presenti in JSetL.z, secondo cinque possibili classi di errore. Le cinque classi di errore riguardano, rispettivamente: errori durante l'input, errori durante il rebound, errori durante l'output, errori dovuti a un oggetto `SubLVar` ed errori dell'oggetto `SolverClass`. Si noti che queste classi di errore non sono da confondersi con gli errori eventualmente presenti nelle specifiche che il sistema modellato potrebbe necessitare di mostrare all'utente, ma errori che dipendono solo dall'implementazione che è stata fornita per la specifica. Perciò diciamo che la sorgente di tali messaggi non è il sistema modellato ma l'interfaccia.

All'interno del package `JSetL.z` viene fornita con la classe `TextInterface` un'implementazione testuale dell'interfaccia `ZInterface`. Ogni interazione è gestita mediante gli stream di input e di output tipici del linguaggio Java.

L'utente, digitando la stringa corretta da linea di comando, è in grado di eseguire un'operazione del sistema modellato oppure di disattivare l'interfaccia, terminando il programma. Per quanto riguarda le interazioni di output, sono state demandate all'utilizzo del metodo `output`, fornito dagli oggetti della libreria JSetL e ridefinito per gli oggetti `SubLVar`, mentre le interazioni di input sono state riscritte per ogni tipo di oggetto supportato da JSetL.

3 La classe astratta `SubLVar`

La classe astratta `SubLVar` modella un nuovo tipo di variabile logica non ancora presente in JSetL, in grado di avere un certo numero di campi dotati di nome, costituiti da altrettante strutture della libreria JSetL.

Nella classe viene dichiarato il campo statico pubblico `type` di classe `LSet` che rappresenta l'insieme di tutte le possibili istanze di oggetti specificati da ogni particolare `SubLVar`, ovvero il dominio logico della variabile rappresentata. Per quanto riguarda le operazioni, la classe astratta `SubLVar` presenta:

- Costruttore

  `public SubLVar(String s, ZDocument doc)`

  È un costruttore con parametro stringa per creare un oggetto `SubLVar` fornito di nome e parametro `ZDocument` per avere un riferimento statico all'oggetto `ZDocument` di cui la classe principale è sottoclasse. Tale riferimento è necessario per il funzionamento di alcuni metodi. Viene richiamato da ogni classe implementante, ovvero dalle classi frutto della conversione di uno schema di struttura (vedi pagina 52).

- Output del valore dei campi

  `public Constraint output(ZInterface zi)`

  Tramite la libreria `Reflection` di Java, permette di ottenere i campi dati della generica classe implementante `SubLVar` e, per ciascuno di essi, demandare l'output di un valore all'oggetto `ZInterface` stesso che ha richiesto l'input dell'intera `SubLVar`.

- Input del valore dei campi

  `public void input(ZInterface zi)`

  Tramite la libreria `Reflection` di Java, permette di ottenere i campi dati della generica classe implementante `SubLVar` e, per ciascuno di essi, demandare l'input di un valore all'oggetto `ZInterface` stesso che ha richiesto l'output l'intera `SubLVar`.

- Uguaglianza

  `public Constraint eq(SubLVar slv)`

  Tramite la libreria `Reflection` di Java, permette di ottenere i campi dati della generica classe implementante `SubLVar` e di fornire la congiunzione di vincoli che impone l'uguaglianza per ciascuno di essi con i relativi campi dati della seconda `SubLVar` fornita.

- Interfacce funzionali per le relazioni

  `protected IntLVar size(LSet s)`

  `protected LSet union(LSet s1, LSet s2)`

  `protected LSet diff(LSet s1, LSet s2)`

  `protected LSet inters(LSet s1, LSet s2)`

  Come `ZDocument` a pagina 25.

4 L'interfaccia `ZDynamicState`

L'interfaccia `ZDynamicState` modella l'insieme dei vincoli dinamici, ovvero le associazioni delle variabili di stato. Tali vincoli devono essere mantenuti in forma non risolta poiché devono poter essere aggiunti, rimossi o rimpiazzati, senza lasciare traccia della precedente versione di un vincolo.

L'interfaccia `ZDynamicState` prevede quindi i seguenti metodi:

- Riassociazione delle variabili di stato

  ```
  public boolean rebound(ZDocument doc, List<Pair> changeList)
  ```

  Lega la prima variabile di ogni coppia della lista al valore della seconda, la quale è solo temporanea. Non deve rimanere traccia del precedente valore della prima variabile di ogni coppia.

- Stato dinamico

  ```
  public Constraint state()
  ```

  Restituisce un vincolo contenente la congiunzione di tutti i vincoli momentaneamente presenti nello stato dinamico, che rappresenta il valore attuale delle variabili di stato.

All'interno del package `JSetL.z` viene fornita con la classe `DynStateImpl` un'implementazione dell'interfaccia `ZDynamicState`. All'interno di `DynStateImpl`, esiste un campo dati di classe `HashMap`, nel quale ciascuna variabile di stato legata è associata ad un vincolo `eq` che ne specifica il valore.

Quando viene chiamato il metodo `rebound`, per ogni coppia presente nella lista passata, viene invocato il metodo privato `free` su ogni variabile presente nella prima posizione della coppia, poi si ottiene il valore della variabile presente nella seconda posizione con il metodo `getValue`, si lega la prima variabile a quel valore e infine si invoca il metodo `free` anche sulla variabile presente nella seconda posizione della coppia. Il metodo `free` della classe `DynStateImpl` elimina dall'oggetto `HashMap` la voce relativa alla variabile passata e rimpiazza tale variabile con una nuova avente lo stesso nome all'interno della classe principale.

Il metodo `state` restituisce un vincolo contenente la congiunzione di tutti i vincoli presenti nella seconda posizione di ogni coppia dell'oggetto `HashMap`.

5 L'interfaccia `ZStaticState`

L'interfaccia `ZStaticState` modella l'insieme dei vincoli statici, ovvero l'insieme di vincoli che garantiscono l'integrità dei dati. Tali vincoli non possono mai essere violati, quindi, al contrario di quelli dinamici, essi non possono essere rimossi, ma solo aggiunti. Inoltre, devono essere mantenuti in forma non risolta poiché devono poter essere ricalcolati durante l'esecuzione di ogni operazione, in congiunzione con i vincoli dinamici.

L'interfaccia `ZStaticState` prevede quindi i seguenti metodi:

- Aggiunta di un vincolo statico

  `public void add(Constraint c)`

  Aggiunge un nuovo vincolo `c` alla congiunzione dei vincoli statici.

- Stato statico

  `public Constraint state()`

  Restituisce un vincolo contenente la congiunzione di tutti i vincoli presenti nello stato statico.

All'interno del package `JSetL.z` viene fornita con la classe `StatStateImpl` un'implementazione dell'interfaccia `ZDynamicState`. All'interno di `StatStateImpl`, esiste un campo dati di classe `Constraint` inizialmente contenente il vincolo vuoto: i nuovi vincoli aggiunti tramite il metodo `add` vengono via via congiunti all'interno dell'oggetto `Constraint`, in modo da mantenerlo sempre aggiornato. Il metodo `state` restituisce semplicemente l'oggetto `Constraint`.

Capitolo IV

Da Z a Java con JSetL

In questo capitolo descriviamo la procedura da seguire per derivare un programma (eseguibile) in linguaggio Java + JSetL dalla sua specifica formale Z. Con linguaggio Java + JSetL intendiamo codici sorgenti Java in cui venga importata la libreria JSetL, nei quali siano quindi presenti tutti i metodi e le classi offerte da JSetL.

Nel corso della trattazione considereremo che i documenti di specifica Z oggetto della conversione appartengano a un sottinsieme di documenti con le seguenti caratteristiche:

- documenti privi dei costrutti Z descritti a pagina 74, per i quali JSetL non offre ancora un supporto adeguato o le cui regole di riscrittura risulterebbero eccessivamente complesse e poco utilizzabili.
- documenti corretti, che descrivano correttamente il sistema da modellare e privi di qualsiasi errore, in quanto esistono già strumenti per supportare la stesura e la verifica di documenti Z [9].
- documenti normalizzati, ovvero in cui i tipi di tutte le variabili siano insiemi dati, schemi di struttura o derivati dei due: eventuali altri insiemi utilizzati verranno sostituiti dal tipo in Z corrispondente e i vincoli sull'insieme verranno espressi come predicati.

La conversione vera e propria è preceduta da una fase preliminare di trasformazione del documento Z in modo da ottenerne uno equivalente ma più vicino alla struttura finale del programma Java, e quindi più facile da convertire. Di fatto, si tratta di eliminare le abbreviazioni e le schematizzazioni tipiche del linguaggio Z che lo rendono più utilizzabile per chi scrive i documenti di specifica.

Una volta eseguite queste operazioni preliminari, è necessario applicare le regole descritte in questo capitolo per tutti gli insiemi dati, variabili globali, vincoli globali, schemi di stato, schemi d'azione e schemi di struttura contenuti nel documento. Queste porteranno alla creazione di nuove strutture JSetL, nuove classi, all'accumulo di vincoli e alla creazione di metodi che sarà possibile eseguire sul sistema così costituito.

1 Operazioni preliminari per la conversione

Prima di procedere con la conversione, un documento Z deve attraversare le seguenti fasi di trasformazione:

1. analisi del documento di specifica espresso con la sintassi del linguaggio Z, al fine di estrarne una lista organizzata dei componenti Z presenti e delle loro caratteristiche. Per svolgere questa operazione, è possibile utilizzare il parser fornito da CZT [10], che in particolare, è anche in grado di riconoscere il tipo in Z di ciascuna variabile ed espressione ed effettuare il typechecking.

2. classificazione di ciascuno schema riconosciuto dal parser per contenimento, per orientamento all'esterno e per funzione.

3. espansione di ogni insieme dato definito per enumerazione, di ogni operazione fra schemi e di ogni schema esterno, secondo le modalità descritte in seguito.

4. rinominazione, in ogni loro occorrenza, di tutte le eventuali variabili globali in tutti gli schemi esterni espansi che non dichiarino una variabile locale con lo stesso nome: il nuovo nome sarà il risultato della giustapposizione di quello vecchio con la stringa "*glbl*" (che sta per *globale*). Qualora la rinominazione generasse nuovi conflitti di nome in qualche schema, sarà sufficiente giustapporre nuovamente la stessa stringa il numero di volte necessario.

Espansione degli insiemi dati enumerati

Qualora un insieme dato sia definito per enumerazione, esso deve essere espanso nel relativo codice Z equivalente. All'enumerazione viene sostituita una terna formata da:

- un insieme dato con lo stesso nome dell'insieme dato precedente.

- variabili globali nello stesso numero e nome di quelle elencate per enumerazione e aventi come tipo l'insieme dato appena introdotto.

- vincoli globali tali che, per ogni coppia di elementi enumerati, venga posto il vincolo di disuguaglianza fra i due e l'insieme dato appena introdotto sia eguagliato all'enumerazione insiemistica di tutti gli elementi enumerati.

Esempio IV.1: L'insieme dato definito per enumerazione *CONTINENTI* viene espanso nell'insieme dato [*CONTINENTI*], nella definizione delle variabili globali dei suoi elementi e nei vincoli di disuguaglianza fra le coppie di elementi e di uguaglianza con l'enumerazione dei suoi elementi

Prima dell'espansione:

$$CONTINENTI ::= EUROPA \mid AMERICA \mid ASIA \mid AFRICA \mid OCEANIA$$

Dopo l'espansione:

$$[CONTINENTI]$$

$$\begin{array}{|l} EUROPA, \\ \quad AMERICA, ASIA, AFRICA, OCEANIA : CONTINENTI \\ \hline EUROPA \neq AMERICA \land EUROPA \neq ASIA \land \\ \quad EUROPA \neq AFRICA \land EUROPA \neq OCEANIA \\ AMERICA \neq ASIA \land AMERICA \neq AFRICA \land \\ \quad AMERICA \neq OCEANIA \\ ASIA \neq AFRICA \land ASIA \neq OCEANIA \\ AFRICA \neq OCEANIA \\ CONTINENTI = \{EUROPA, AMERICA, ASIA, AFRICA, OCEANIA\} \end{array}$$

Espansione delle operazioni fra schemi

Alcuni schemi possono essere definiti come il risultato di operazioni fra schemi a partire da altri schemi già definiti. In particolare, in questa trattazione presenteremo solo tre operazioni fondamentali: la negazione, la congiunzione e la disgiunzione.

- Il risultato della congiunzione di due schemi consiste in uno schema avente come parte di definizioni l'unione delle definizioni dei due schemi e come parte dei predicati la congiunzione logica di tutti i vincoli di entrambi gli schemi.

- Il risultato della negazione di uno schema consiste in uno schema avente la stessa parte di definizioni dello schema negato e la negazione logica della congiunzione di tutti i suoi vincoli.
- Il risultato della disgiunzione di due schemi consiste in uno schema avente come parte di definizioni l'unione delle definizioni dei due schemi e come parte dei predicati la disgiunzione logica delle congiunzioni dei vincoli dei due schemi.

Esempio IV.2: Lo schema *NonAmerica* è definito come risultato dell'operazione di negazione dello schema *America*(vedi pagina 7) e pertanto l'unico predicato presente è la negazione dell'unico predicato presente nello schema *America*

Prima dell'espansione:

$NonAmerica = \neg\ America$

Dopo l'espansione:

NonAmerica

$ContinenteDue : CONTINENTI$

$ContinenteDue \neq AMERICA$

Esempio IV.3: Lo schema *Mondo* è definito come risultato dell'operazione di disgiunzione fra lo schema *America*(vedi pagina 7) e lo schema *NonAmerica*

Prima dell'espansione:

$Mondo = America \lor NonAmerica$

Dopo l'espansione:

Mondo

$ContinenteDue : CONTINENTI$

$ContinenteDue = AMERICA \lor ContinenteDue \neq AMERICA$

Espansione degli schemi esterni

Qualora uno schema presenti uno o più riferimenti ad altri schemi, ovvero la presenza del nome di un altro schema al suo interno, ciascuno di questi deve essere espanso fino all'esaurimento di tutti i riferimenti. Non si considera un riferimento l'utilizzo di uno schema di struttura per la definizione del tipo di una variabile. L'operazione di espansione di uno schema produce uno schema equivalente a quello iniziale e non ne modifica in alcun modo la classificazione.

Esistono quattro tipi di riferimenti ad altri schemi: il riferimento decorato con Δ, il riferimento decorato con Xi, il riferimento decorato con "′" e il riferimento non decorato. Inoltre, un riferimento può apparire nella parte delle definizioni o nella parte dei predicati di uno schema.

- un riferimento decorato con Δ a uno schema si espande sostituendolo con due riferimenti, rispettivamente decorato con "′" e non decorato, allo stesso schema.

- un riferimento decorato con Xi a uno schema si espande sostituendolo con un riferimento decorato con Δ allo stesso schema e inserendo nello schema in cui appare un vincolo di uguaglianza fra la versione decorata con "′" e non decorata di ogni variabile definita nello schema riferito.

- un riferimento decorato con "′" a uno schema, se posizionato nella parte di definizione, si espande ricopiando nello schema in cui appare la definizione della versione decorata con "′" di tutte le variabili definite nello schema riferito e tutti i vincoli presenti nello schema riferito, ma con ogni variabile definita nello schema riferito sostituita dalla sua versione decorata con "′". Se invece il riferimento è posizionato nella parte dei predicati, esso si espande ricopiando solo i vincoli.

- un riferimento non decorato a uno schema, se posizionato nella parte di definizione, si espande ricopiando nello schema in cui appare la definizione di tutte le variabili definite nello schema riferito e tutti i vincoli presenti nello schema riferito. Se invece il riferimento è posizionato nella parte dei predicati, esso si espande ricopiando solo i vincoli.

Esempio IV.4: Lo schema *VisualizzaDiario* contiene un riferimento *Xi* allo schema *DiarioDiBordo* (vedi pagina 43): in 4 passaggi questo diventerà prima un riferimento Δ a *DiarioDiBordo* unitamente al primo vincolo, poi due riferimenti a *DiarioDiBordo*, rispettivamente non decorato e decorato con "′", poi il primo di questi verrà espanso nelle relative definizioni e vincoli e infine anche il secondo verrà espanso in definizioni e vincoli, terminando la lista dei riferimenti presenti.

Prima dell'espansione:

VisualizzaDiario

$$\Xi DiarioDiBordo$$
$$Visualizza! : \mathbb{P}\ Viaggio$$

$$Visualizza! = Viaggi$$

Dopo l'espansione:

VisualizzaDiario

$$Viaggi : \mathbb{P}\ Viaggio$$
$$Viaggi' : \mathbb{P}\ Viaggio$$
$$Visualizza! : \mathbb{P}\ Viaggio$$

$$Viaggi' = Viaggi$$
$$\#\ Viaggi \leq 100$$
$$\#\ Viaggi' \leq 100$$
$$Visualizza! = Viaggi$$

2 Funzioni di traduzione

Descriveremo le regole di conversione introducendo alcune funzioni di traduzione aventi come argomento un qualche costrutto Z e come risultato il relativo codice Java+JSetL.

Funzioni ρ e P. La coppia di funzioni ρ e P (Rho, la "R" dell'alfabeto Greco rispettivamente minuscola e maiuscola, che sta per *Rinominazione*) si applica ai nomi presenti nel documento, restituendo il codice Java relativo al nome stesso rispettivamente di sole lettere minuscole e di sole maiuscole. Ad esempio:

$$\rho(\mathit{Variabile}) = \texttt{variabile}$$

mentre:

$$\mathrm{P}(\mathit{Variabile}) = \texttt{VARIABILE}$$

Funzione τ. La funzione τ (Tau, la "T" dell'alfabeto Greco che sta per *Tipo*) si applica a insiemi dati e variabili, restituendo la relativa struttura JSetL che li modella correttamente. Il valore restituito da τ è quindi una classe Java e dipende dal tipo rilevato dal parser CZT (primitivo, intero, schema o derivato di questi).

In particolare, la funzione τ restituirà:

- `IntLVar`: se applicata ad una variabile intera, avente quindi come tipo $\mathbb{Z}$ o un suo sottinsieme
- `LVar`: se applicata ad una variabile di tipo primitivo, avente quindi come tipo un altro insieme dato
- `IntLSet`: se applicata ad un insieme di interi, avente quindi come tipo $\mathbb{P}\,\mathbb{Z}$ o un suo sottinsieme
- `LSet`: se applicata ad un insieme di non soli interi, avente quindi come tipo $\mathbb{P}$ di un altro insieme dato
- `LMap`: se applicata ad una funzione, avente quindi come tipo il prodotto cartesiano di due insiemi o un suo sottinsieme
- P(*Schema*): se applicata ad una variabile il cui tipo è definito da *Schema*, ovvero uno schema di struttura, il cui tipo è quindi l'insieme di tutte le possibili istanze generate a partire da quello schema

Inoltre, essendo τ la funzione scelta per gestire i tipi, essa è stata sovraccaricata di un significato particolare se applicata ad uno schema di struttura, descritto nella relativa sezione a pagina 52.

Funzione δ. La funzione δ (Delta, la "D" dell'alfabeto Greco che sta per *Definizione*) si applica a insiemi dati, variabili (globali e non) e schemi, restituendo il codice Java contenente tutte le relative dichiarazioni di variabili necessarie alla traduzione.

Ad esempio, la funzione δ applicata ad uno schema restituisce il codice Java in cui vengono dichiarate tutte le variabili relative allo schema, ovvero traduce tipicamente la parte di definizioni dello schema stesso.

Funzione γ. La funzione γ (Gamma, la "C" dell'alfabeto Greco, che sta per *Conversione*) si applica all'intero documento Z, agli insiemi dati, a predicati e definizioni (anche globali), alle espressioni e agli schemi, restituendo il codice Java che li implementa. Talvolta, utilizzeremo delle regole di riscrittura per descrivere questa funzione più direttamente. Se, ad esempio, vale che:

$$\gamma(\textit{codice in Z}) = \texttt{codice in Java + JSetL}$$

potrà essere utilizzata la seguente schematizzazione in modo equivalente:

$$\frac{\gamma(\textit{codice in Z})}{\texttt{codice in Java + JSetL}}$$

La funzione γ avrà carattere principalmente ricorsivo in quanto la traduzione in codice Java di molte strutture dipende dai componenti che formano la struttura.

Durante la trattazione verranno introdotte alcune varianti delle funzioni γ e δ per differenziarne leggermente il comportamento rispetto al contesto in cui vengono applicate.

Diamo inoltre per induzione la definizione di γ sulle liste di elementi: se `V` è la lista vuota `[]`, allora γ(`V`)=`""`, dove `""` è la stringa di codice Java vuota. Se invece `L` è la lista avente `E` come primo elemento e `R` come resto della coda, ovvero `L = [E|R]`, γ(`L`)=γ(`E`) $\bullet$ γ(`R`), dove $\bullet$ è l'operazione di concatenazione di codice Java(che spesso nelle regole rimarrà sottintesa). La stessa definizione induttiva sulle liste vale anche per δ e per le varianti di entrambe le funzioni.

3 Conversione del Documento Z

Convertire un documento Z significa eseguire le operazioni preliminari di trasformazione del documento ed applicare la funzione γ al documento risultante.

Il generico documento Z, a seguito delle operazioni preliminari, avrà la forma:

section *Specification* **parents** *standard_toolkit*
$INSIEMIDATI =$
$[INSIEME1 \mid ALTRIIINSIEMI]$
$VARIABILIGLOBALI =$
$[VARIABILEGLOBALE1 \mid ALTREVARIABILIGLOBALI]$
$VINCOLIGLOBALI =$
$[VINCOLOGLOBALE1 \mid ALTREVINCOLIGLOBALI]$
$SCHEMI =$
$[SCHEMISTRUTTURA \mid SCHEMISTATO \mid SCHEMIAZIONE]$
$SCHEMISTATO =$
$[SCHEMASTATO1 \mid ALTRISCHEMISTATO]$
$SCHEMIAZIONE =$
$[SCHEMAAZIONE1 \mid ALTRISCHEMIAZIONE]$
$SCHEMISTRUTTURA =$
$[SCHEMASTRUTTURA1 \mid ALTRISCHEMISTRUTTURA]$

Il documento Z verrà convertito in una o più classi del linguaggio Java: almeno una classe principale per il documento complessivo, più una classe per ogni schema di struttura esterno. Queste classi dovranno appartenere a un qualsiasi stesso package.

Precisamente, se *DocumentoZ* è il documento ottenuto a seguito delle operazioni preliminari sulla specifica Z originaria, il risultato della applicazione:

$$\gamma(DocumentoZ)$$

sarà il seguente programma Java:

```
import JSetL.*;
import JSetL.z.*;
@SuppressWarnings("rawtypes")// per le sottoclassi (parametriche)
   di SubLVar
public class P(DocumentoZ) extends ZDocument
{
  //definizioni di insiemi dati(strutture JSetL pubbliche)
  δ(INSIEMIDATI)

  //definizioni di variabili globali e di stato(strutture JSetL
     pubbliche)
  δ(VARIABILIGLOBALI)
  δ(SCHEMISTATO)

  P(DocumentoZ)()
  {
    super();
    //vincoli di variabili globali e di stato, vincoli globali di
       stato (usare statPost)
    γ(VARIABILIGLOBALI)
    γ(VINCOLIGLOBALI)
    γ(SCHEMISTATO)

    //vincoli di insiemi dati (usare givenSet)
    γ(INSIEMIDATI)
    τ(SCHEMISTRUTTURA)
  }

  public static void main(String[] args)
  {
    //crea il documento e lo apre tramite l'interfaccia
    P(DocumentoZ) doc = new P(DocumentoZ)();
    intz.open(doc);
  }

  γ(SCHEMIAZIONE)
}
In altri sorgenti Java:
γ(SCHEMISTRUTTURA)
```

4 Insiemi dati

Per ogni insieme dato $[SET]$, nella classe principale viene definito e creato un attributo pubblico `ρ(SET)` di classe `τ(SET)` (`IntLSet` per insiemi di interi, `LSet` altrimenti) avente come nome esterno `"P(SET)"` (ovvero il nome e il nome esterno dell'oggetto LSet sono lo stesso dell'insieme dato, rispettivamente in lettere minuscole e maiuscole).

Viene inoltre effettuata una chiamata al metodo `givenSet` avente come argomento la variabile creata `ρ(SET)`, per informare il sistema che tale `LSet` (o `IntLSet`) rappresenterà uno dei tipi possibili.

Come abbiamo visto in precedenza, le definizioni (ovvero il risultato della funzione δ) e i vincoli (ovvero il risultato della funzione γ) derivanti da un insieme dato vengono posizionati in parti diverse della classe principale: definiremo quindi separatamente il risultato dell'applicazione delle due funzioni su un insieme dato.

Pertanto, la definizione della funzione δ applicata ad un insieme dato è la seguente:

$$\frac{\delta([SET])}{\texttt{public } \tau\texttt{(SET) } \rho\texttt{(SET) = new } \tau\texttt{(SET)("P(SET)");}}$$

Invece, la definizione della funzione γ applicata ad un insieme dato è la seguente:

$$\frac{\gamma([SET])}{\texttt{givenSet(}\rho\texttt{(SET));}}$$

Esempio IV.5: Il tipo *STATI* viene convertito nella definizione e creazione dell'`LSet stati`, con nome esterno `"STATI"` e nella chiamata al metodo `givenSet` avente `stati` come argomento.

$[STATI]$

$\delta([STATI])$ `= public LSet stati = new LSet("STATI");`
$\gamma([STATI])$ `= givenSet(stati);`

5 Variabili globali

Le variabili globali vengono trattate allo stesso modo delle definizioni di uno schema di stato (vedi pagina 43).

Esempio IV.6: La variabile globale *MioContinente* viene convertita nella definizione e creazione dell'`LVar miocontinente`, con nome esterno `"MIOCONTINENTE"` e nella chiamata al metodo `statPost` fornendo come argomento il vincolo di tipo della variabile, ovvero l'appartenenza all'insieme `continenti`.

$$MioContinente : CONTINENTI$$

δ(*MioContinente* : *CONTINENTI*) =

```
    public LVar miocontinente = new LVar("MIOCONTINENTE");
```

γ(*MioContinente* : *CONTINENTI*) =

```
    statPost(miocontinente.in(continenti));
```

6 Vincoli globali

I vincoli globali vengono trattati allo stesso modo dei predicati di uno schema di stato (vedi pagina 43).

Esempio IV.7: Il vincolo di uguaglianza viene tradotto nel relativo vincolo equivalente in Java+JSetL e poi aggiunto ai vincoli statici tramite la chiamata al metodo `statPost`

$$MioContinente = EUROPA$$

γ(*MioContinente* = *EUROPA*) =

```
    statPost(miocontinente.eq(europa));
```

7 Schemi di stato

Come abbiamo visto in precedenza a pagina 40, i codici delle definizioni (ovvero il risultato della funzione δ) e dei vincoli (ovvero il risultato della funzione γ) derivanti da uno schema di stato vengono posizionati in punti diversi della classe principale: definiremo quindi separatamente il risultato dell'applicazione delle due funzioni su uno schema di stato.

Ogni schema di stato esterno espanso si può considerare come una lista di definizioni e una lista di predicati provenienti rispettivamente dalla parte delle definizioni e dalla parte dei predicati. Questo tipo di schema avrà perciò la seguente forma:

$$\begin{array}{|l} \textit{SCHEMA\_DI\_STATO} \\ \hline \textit{DEFINIZIONI} = [\textit{DEFINIZIONE}1 \mid \textit{ALTREDEFINIZIONI}] \\ \hline \textit{PREDICATI} = [\textit{PREDICATO}1 \mid \textit{ALTRIPREDICATI}] \\ \hline \end{array}$$

Definiamo quindi le regole di conversione dello schema tramite la conversione di queste due liste.

Le regole di conversione applicate alla lista di definizioni genereranno per ogni voce la definizione e la creazione di un attributo pubblico della classe principale (catturata quindi dalla funzione δ), unitamente ai vincoli di tipo da aggiungere allo stato statico tramite il metodo `statPost` nel costruttore presente nella classe principale (catturati quindi dalla funzione γ).

Le regole di conversione applicate alla lista di predicati, invece, genereranno per ogni predicato un vincolo JSetL, anch'esso da aggiungere allo stato statico tramite il metodo `statPost` nel costruttore presente nella classe principale (catturata quindi dalla funzione γ).

Essendo l'utilizzo del vincolo `statPost` tipico degli schemi di stato, introdurremo solo per la lista dei vincoli di stato una variante della funzione γ, ovvero γ_{stato}, così definita:

$$\frac{\gamma_{stato}(\textit{VINCOLO})}{\texttt{statPost(}\gamma\texttt{(VINCOLO));}}$$

Le regole di conversione di uno schema di stato saranno quindi le seguenti:

$$\frac{\delta(SCHEMA\_DI\_STATO)}{\delta\texttt{(DEFINIZIONI)}}$$

$$\frac{\gamma(SCHEMA\_DI\_STATO)}{\gamma_{stato}\texttt{(DEFINIZIONI)}\,\gamma_{stato}\texttt{(PREDICATI)}}$$

Le regole di conversione delle singole definizioni e predicati che compongono le liste sono descritte rispettivamente a pagina 55 e a pagina 58.

Esempio IV.8: Lo schema *DiarioDiBordo* viene convertito nell'attributo pubblico `viaggi` di classe `LSet` e nome esterno `"VIAGGI"` e in due vincoli che appariranno nel costruttore. Il primo esprime il tipo della variabile `viaggi`, ovvero un sottinsieme dell'insieme di tutti i possibili oggetti di classe `Viaggio`, rappresentato dal campo statico `type` dell'omonima classe. Il secondo vincola la cardinalità di `viaggi` a non superare il valore 100

DiarioDiBordo

$$Viaggi : \mathbb{P}\ Viaggio$$

$$\#\ Viaggi \leq 100$$

δ(*DiarioDiBordo*) =

```
    public LSet viaggi = new LSet("VIAGGI");
```

γ(*DiarioDiBordo*) =

```
    statPost(viaggi.subset(Viaggio.type));
    statPost(size(viaggi).le(100));
```

8 Schemi d'azione

Per ogni schema d'azione esterno espanso verrà generato un metodo pubblico nella classe principale che rappresenta la relativa operazione in Z e che l'utente finale del sistema potrà essere in grado di eseguire attraverso l'interfaccia.

Ogni schema d'azione esterno espanso si può considerare come una lista di definizioni e una lista di predicati provenienti rispettivamente dalla parte delle definizioni e dalla parte dei predicati. Tuttavia, la lista delle definizioni di uno schema d'azione è partizionata in quattro differenti liste di definizioni:

- le definizioni di variabili di stato, non decorate.
- le definizioni della versione decorata con "′" delle variabili di stato.
- le definizioni di variabili di input, ovvero decorate con "?".
- le definizioni di variabili di output, ovvero decorate con "!".

In particolare, gli schemi non di input avranno la lista delle variabili di input vuota e viceversa: lo stesso discorso vale per gli schemi di output.

Il generico schema d'azione avrà perciò la seguente forma:

SCHEMA_DI_AZIONE

$$
\begin{array}{l}
DEFINIZIONI = \\
\quad [DEFINIZIONISTATO \mid DEFINIZIONIPRIMED \mid \\
\quad DEFINIZIONIINPUT \mid DEFINIZIONIOUTPUT] \\
DEFINIZIONISTATO = \\
\quad [DEFINIZIONESTATO1 \mid ALTREDEFINIZIONISTATO] \\
DEFINIZIONIPRIMED = \\
\quad [DEFINIZIONEPRIMED1 \mid ALTREDEFINIZIONIPRIMED] \\
DEFINIZIONIINPUT = \\
\quad [DEFINIZIONEINPUT1 \mid ALTREDEFINIZIONIINPUT] \\
DEFINIZIONIOUTPUT = \\
\quad [DEFINIZIONEOUTPUT1 \mid ALTREDEFINIZIONIOUTPUT] \\
\hline
PREDICATI = [PREDICATO1 \mid ALTRIPREDICATI]
\end{array}
$$

La conversione dello schema sarà quindi data dalla conversione delle quattro liste di definizioni e della lista dei vincoli.

Per ciascuna delle quattro liste di definizioni sono previste regole di conversione leggermente diverse, che noi esprimeremo creando quattro versioni della funzione δ, rispettivamente δ_{stt} (da *state*), δ_{prm} (da *primed*), δ_{ask} (da *ask*) e δ_{ans} (da *answer*) e quattro versioni della funzione γ, rispettivamente γ_{stt}, γ_{prm}, γ_{ask} e γ_{ans}.

Funzioni δ_{stt} e γ_{stt}. Queste due funzioni semplicemente restituiscono la stringa vuota di codice Java `""`, poiché i metodi creati per tradurre gli schemi d'azione appartengono alla classe principale e quindi hanno visibilità sugli attributi del documento, ovvero le variabili di stato:

$$\frac{\gamma_{stt}(VARIABILE : TIPO)}{\texttt{""}}$$

$$\frac{\delta_{stt}(VARIABILE : TIPO)}{\texttt{""}}$$

Funzioni δ_{prm}, δ_{ask} e δ_{ans}. Queste funzioni si applicano su variabili decorate per simulare l'aggiunta delle decorazioni Z "′", "?" e "!", apponendo al loro posto rispettivamente le stringhe `prm`, `ask` e `ans`:

$$\frac{\delta_{prm}(VARIABILE')}{\texttt{VARIABILEPRM}}$$

$$\frac{\delta_{ask}(VARIABILE?)}{\texttt{VARIABILEASK}}$$

$$\frac{\delta_{ans}(VARIABILE!)}{\texttt{VARIABILEANS}}$$

Le tre funzioni si possono inoltre applicare durante la definizione della variabile decorata:

$$\frac{\delta_{prm}(VARIABILE' : TIPO)}{\delta\texttt{(}\delta_{prm}(VARIABILE')\texttt{:TIPO)}}$$

$$\frac{\delta_{ask}(VARIABILE? : TIPO)}{\delta\texttt{(}\delta_{ask}(VARIABILE?)\texttt{:TIPO)}}$$

$$\frac{\delta_{ans}(VARIABILE! : TIPO)}{\delta\texttt{(}\delta_{ans}(VARIABILE!)\texttt{:TIPO)}}$$

Funzioni γ_{prm}, γ_{ask} e γ_{ans}. Queste funzioni si occupano di chiamare metodi per informare il sistema della tipologia delle variabili che vengono definite e create. I metodi invocati sono quelli della classe astratta `ZDocument`, più precisamente:

- per le variabili di modifica dello stato, ovvero decorate con "′", avviene una chiamata al metodo `prmVar`, che richiede come primo argomento la variabile di stato non decorata (il cui valore verrà modificato) e come secondo argomento la variabile di stato primed (il cui valore sarà sostituito al precedente valore della variabile).

- per le variabili di input, ovvero decorate con "?", avviene una chiamata al metodo `askVar` che richiede come argomento la variabile di cui il sistema richiederà un valore in input all'utente.

- per le variabili di output, ovvero decorate con "!", avviene una chiamata al metodo `ansVar` che richiede come argomento la variabile di cui il sistema mostrerà il valore in output all'utente.

Ciascuna funzione, oltre ad invocare i metodi sopra citati, aggiunge inoltre i vincoli di tipo per la variabile:

$$\frac{\gamma_{prm}(VARIABILE' : TIPO)}{\begin{array}{l}\texttt{prmVar(}\rho\texttt{(VARIABILE),}\rho(\delta_{prm}\texttt{(VARIABILE)));}\\ \gamma(\delta_{prm}(VARIABILE')\texttt{:TIPO)}\end{array}}$$

$$\frac{\gamma_{ask}(VARIABILE? : TIPO)}{\begin{array}{l}\texttt{askVar(}\rho(\delta_{ask}\texttt{(VARIABILE)));}\\ \gamma(\delta_{ask}(VARIABILE?)\texttt{:TIPO)}\end{array}}$$

$$\frac{\gamma_{ans}(VARIABILE! : TIPO)}{\begin{array}{l}\texttt{ansVar(}\rho(\delta_{ans}\texttt{(VARIABILE)));}\\ \gamma(\delta_{ans}(VARIABILE!)\texttt{:TIPO)}\end{array}}$$

Per quanto riguarda la lista dei vincoli, essa viene tradotta in modo molto simile alla lista dei vincoli di uno schema di stato: infatti l'unica differenza risiede nell'utilizzo del metodo `tempPost` invece di `statPost`, poiché i vincoli di un'operazione sono temporanei e quindi vanno aggiunti ad un accumulatore temporaneo invece che allo stato statico.

Introdurremo quindi in modo analogo solo per la lista dei vincoli di azione una variante della funzione γ, ovvero γ_{azione}, così definita:

$$\frac{\gamma_{azione}(VINCOLO)}{\texttt{tempPost}(\gamma(\texttt{VINCOLO}));}$$

Precisamente, dato lo schema d'azione *SCHEMA_DI_AZIONE*, il risultato della applicazione:

$$\gamma(SCHEMA\_DI\_AZIONE)$$

sarà il seguente metodo Java della classe principale:

```
//metodo per lo schema d'azione "ρ(SCHEMA_DI_AZIONE)"
public void ρ(SCHEMA_DI_AZIONE) ()
{
  //variabili di input? (usare askVar)
  δ_ask(DEFINIZIONIINPUT)
  γ_ask(DEFINIZIONIINPUT)

  //variabili di modifica' (usare prmVar)
  δ_prm(DEFINIZIONIPRIMED)
  γ_prm(DEFINIZIONIPRIMED)

  //variabili di output! (usare ansVar)
  δ_ans(DEFINIZIONIOUTPUT)
  γ_ans(DEFINIZIONIOUTPUT)

  //accumulo vincoli (usare post)
  γ_azione(PREDICATI)

  //esecuzione operazione
  execute();
}
```

Esempio IV.9: Lo schema Delta di input *AggiungiViaggio*, dopo l'espansione, viene convertito nell'omonimo metodo, nel quale sono presenti le definizioni delle due variabili `nuovoask` e `viaggiprm` rispettivamente di input e di modifica, con i relativi vincoli di tipo e le chiamate ad `askVar` e `prmVar`. Non essendo uno schema di output, non sono definite variabili di output. Seguono la versione normale e decorata con "′" dei vincoli di stato, dati dall'espansione del riferimento $\Delta DiarioDiBordo$. Da ultimo appare la traduzione del vincolo che ridefinisce il valore della variabile $Viaggi'$

AggiungiViaggio

$\Delta DiarioDiBordo$
$Nuovo? : Viaggio$

$Viaggi' = Viaggi \cup \{Nuovo?\}$

```
//metodo per lo schema d'azione "aggiungiviaggio"
public void aggiungiviaggio (){
//variabili di input?  (usare askVar)
VIAGGIO nuovoask = new VIAGGIO("NUOVOASK");
askVar(nuovoask);
tempPost(nuovoask.in(VIAGGIO.type));

//variabili di modifica' (usare prmVar)
LSet viaggiprm = new LSet("VIAGGIPRM");
prmVar(viaggi,viaggiprm);
tempPost(viaggiprm.subset(VIAGGIO.type));

//variabili di output!  (usare ansVar)

//accumulo vincoli (usare post)
tempPost(size(viaggi).le(100));
tempPost(size(viaggiprm).le(100));
tempPost(viaggiprm.eq(union(viaggi,LSet.empty().ins(nuovoask))));

//esecuzione operazione
execute();
}
```

Esempio IV.10: Lo schema Xi di output *VisualizzaDiario*, dopo l'espansione, viene convertito nell'omonimo metodo, nel quale sono presenti le definizioni delle due variabili `viaggiprm` e `visualizzaans` rispettivamente di modifica e di output, con i relativi vincoli di tipo e le chiamate a `prmVar` e `ansVar`. Non essendo uno schema di input, non sono definite variabili di input. Seguono il vincolo di uguaglianza fra la versione non decorata e decorata con "′" della variabile di stato *Viaggi* e la versione non decorata e decorata con "′" dei vincoli di stato, dati dall'espansione del riferimento Ξ*DiarioDiBordo* (vedi pagina 35). Da ultimo appare la traduzione del vincolo che lega il valore della variabile di output *Visualizza*!

VisualizzaDiario

$$\Xi DiarioDiBordo$$
$$Visualizza! : \mathbb{P}\ Viaggio$$

$$Visualizza! = Viaggi$$

```
//metodo per lo schema d'azione "visualizzadiario"
public void visualizzadiario (){
//variabili di input?  (usare askVar)

//variabili di modifica' (usare prmVar)
LSet viaggiprm = new LSet("VIAGGIPRM");
tempPost(viaggiprm.subset(VIAGGIO.type));
prmVar(viaggi,viaggiprm);

//variabili di output!  (usare ansVar)
LSet visualizzaans = new LSet("VISUALIZZAANS");
tempPost(visualizzaans.subset(VIAGGIO.type));
ansVar(visualizzaans);

//accumulo vincoli (usare post)
tempPost(viaggiprm.eq(viaggi));
tempPost(size(viaggi).le(100));
tempPost(size(viaggiprm).le(100));
tempPost(visualizzaans.eq(viaggi));

//esecuzione operazione
execute();
}
```

Esempio IV.11: Lo schema di inizializzazione *DiarioIniziale*, dopo l'espansione, viene convertito nell'omonimo metodo, nel quale è presente la definizione della variabile di modifica `viaggiprm`, con il relativo vincolo di tipo e la chiamata a `prmVar`. Non essendo uno schema di input nè di output, non sono definite variabili di input nè di output. Segue la versione decorata con "′" dei vincoli di stato, dati dall'espansione del riferimento $DiarioDiBordo'$. Da ultimo appare la traduzione del vincolo che lega il valore della variabile di modifica $Viaggi'$

DiarioIniziale

$DiarioDiBordo'$

$Viaggi' = \varnothing$

```
//metodo per lo schema d'azione "diarioiniziale"
public void diarioiniziale (){
//variabili di input?  (usare askVar)

//variabili di modifica' (usare prmVar)
LSet viaggiprm = new LSet("VIAGGIPRM");
tempPost(viaggiprm.subset(VIAGGIO.type));
prmVar(viaggi,viaggiprm);

//variabili di output!  (usare ansVar)

//accumulo vincoli (usare post)
tempPost(size(viaggiprm).le(100));
tempPost(viaggiprm.eq(LSet.empty()));

//esecuzione operazione
execute();
}
```

9 Schemi di struttura

Ogni schema di struttura esterno espanso si può considerare come una lista di definizioni e una lista di predicati provenienti rispettivamente dalla parte delle definizioni e dalla parte dei predicati, esattamente come uno schema di stato. Questo tipo di schema avrà perciò la seguente forma:

$$\begin{array}{|l} \textit{SCHEMA\_DI\_STRUTTURA} \\ \hline \textit{DEFINIZIONI} = [\textit{DEFINIZIONE}1 \mid \textit{ALTREDEFINIZIONI}] \\ \hline \textit{PREDICATI} = [\textit{PREDICATO}1 \mid \textit{ALTRIPREDICATI}] \\ \hline \end{array}$$

Per ogni schema di struttura esterno espanso, sarà necessario creare una nuova classe in un nuovo file sorgente Java nello stesso package della classe principale, contenente il risultato della conversione. Tale classe sarà parametrica rispetto alla classe principale ed estenderà la classe astratta `SubLVar`, in modo che i suoi campi pubblici rispecchino le variabili definite nella prima lista e che il costruttore aggiunga allo stato statico i vincoli relativi ai predicati della seconda lista. Conterrà inoltre un campo pubblico e statico `type` di classe `LSet` che rappresenta l'insieme di tutti i possibili oggetti istanziabili con questa classe.

Per la conversione della prima lista utilizzeremo la funzione δ esattamente come per gli schemi di stato, con la differenza che questa volta le variabili saranno definite e create all'interno della classe relativa allo schema di struttura e non nella classe principale. Per la conversione della seconda lista avremo invece bisogno di aggiungere i vincoli allo stato statico usando il metodo `statPost` che però necessita di avere un riferimento alla classe principale per essere invocato. Utilizzeremo quindi il riferimento `d` della classe parametrica `T` derivante dal costruttore. Introdurremo quindi solo per la lista dei vincoli di struttura una variante della funzione γ, ovvero $\gamma_{struttura}$, così definita:

$$\frac{\gamma_{struttura}(\textit{VINCOLO})}{\texttt{d.statPost(}\gamma\texttt{(VINCOLO));}}$$

Ricordiamo inoltre che la funzione τ per la gestione dei tipi è stata sovraccaricata di un significato particolare se applicata ad uno schema di struttura (non una variabile avente uno schema di struttura come tipo): essa restituirà l'invocazione del metodo `givenSet` sul campo statico `type` di tipo `LSet` dello schema di struttura su cui viene invocata.

Pertanto, questa invocazione andrà inserita nel costruttore della classe principale:

$$\frac{\tau(SCHEMA\_DI\_STRUTTURA)}{\texttt{givenSet(P(SCHEMA\_DI\_STRUTTURA).type);}}$$

La regola di conversione di uno schema di struttura sarà quindi la seguente:

$$\gamma(SCHEMA\_DI\_STRUTTURA)$$

```
import JSetL.*;
import JSetL.z.*;

@SuppressWarnings("serial")
// T deve estendere il tipo del documento
public class P(SCHEMA_DI_STRUTTURA)<T extends P(DocumentoZ)>
    extends SubLVar
{
  //LSet che rappresenta il tipo di questa struttura
  static public LSet type = new LSet("P(SCHEMA_DI_STRUTTURA)TYPE");

  //strutture JSetL come campi (pubblici)
  δ(DEFINIZIONI)

  @SuppressWarnings("unchecked")
  public P(SCHEMA_DI_STRUTTURA)(ZDocument d, String name)
  {
    super(name,d);
    init((T) d);
  }

  void init(T d)
  {
    //vincoli di tipizzazione della struttura (usare d.statPost)
    d.statPost(this.in(type));
    γ_struttura(DEFINIZIONI)

    //vincoli, parte dei predicati (usare d.statPost)
    γ_struttura(VINCOLI)
  }
}
```

Esempio IV.12: Lo schema di struttura *Viaggio* viene convertito nell'omonima classe parametrica che estende `SubLVar`. La classe presenta il campo statico `type` che ne rappresenta il tipo, oltre ai campi pubblici `partenza` e `arrivo` dati dalla parte di definizioni. Seguono il costruttore comune ad ogni classe per schemi di struttura e il metodo `init` che pone i vincoli di tipo relativi ai campi, oltre al vincolo risultato della traduzione della parte di predicati.

Viaggio

$Partenza : CONTINENTI$
$Arrivo : CONTINENTI$

$Partenza \neq Arrivo$

```
import JSetL.*;
import JSetL.z.*;

@SuppressWarnings("serial")
// T deve estendere il tipo del documento
public class VIAGGIO<T extends DOCUMENTOZ> extends SubLVar {
    //LSet che rappresenta il tipo di questa struttura
    static public LSet type = new LSet("VIAGGIOTYPE");
    //strutture JSetL come campi (pubblici)
    public LVar partenza = new LVar ("PARTENZA");
    public LVar arrivo = new LVar ("ARRIVO");

    @SuppressWarnings("unchecked")
    public VIAGGIO(ZDocument d, String name){
    super(name,d);
    init((T) d);
    }

    void init(T d){
    //vincoli di tipizzazione della struttura (usare d.statPost)
    d.statPost(this.in(type));
    d.statPost(partenza.in(d.continenti));
    d.statPost(arrivo.in(d.continenti));
    //vincoli, parte dei predicati (usare d.statPost)
    d.statPost(partenza.neq(arrivo));
    }
}
```

10 Definizioni

Ogni definizione in Z ha la forma:

$$VAR : TIPO$$

Per ogni definizione, il parser CZT rileverà un tipo Z: usando la funzione τ (vedi pagina 37) è possibile ottenere la classe di JSetL che modella correttamente il tipo Z rilevato dal parser. La regola di conversione per la funzione δ dovrà quindi generare un nuovo campo pubblico ρ`(VAR)` avente come classe il risultato di τ e un costruttore associato che inizializzi tale campo con il nome `"P(VAR)"` della variabile.

Esiste tuttavia un'eccezione per le variabili aventi un tipo definito in uno schema di struttura: in quel caso il costruttore necessiterà come primo argomento aggiuntivo del riferimento alla classe principale. Quindi, ancora una volta, sovraccaricheremo la funzione τ di un significato particolare qualora le venga fornito come parametro un tipo Z. La funzione τ restituirà rispettivamente la stringa di codice Java `"this,"` se il tipo sarà quello definito da uno schema di struttura e la stringa vuota di codice Java `""`, altrimenti.

Pertanto la regola di riscrittura per una definizione è la seguente:

$$\frac{\delta(VAR : TIPO)}{\texttt{public } \tau\texttt{(VAR) } \rho\texttt{(VAR) = new } \tau\texttt{(VAR) (}\tau\texttt{(TIPO)"P(VAR)");}}$$

Per quanto riguarda l'applicazione della funzione γ su una definizione, essa dovrà restituire i vincoli che esprimano il tipo della variabile definita. Qualora il tipo sia primitivo (ovvero non composto), il risultato della funzione γ applicata su una definizione sarà, in base al tipo della variabile, rispettivamente:

- tipo primitivo non intero:

$$\frac{\gamma(VAR : TIPO)}{\rho\texttt{(VAR).in(}\rho\texttt{(TIPO))}}$$

- tipo primitivo intero:

$$\frac{\gamma(VAR : TIPO)}{\rho\texttt{(VAR).dom(}\rho\texttt{(TIPO))}}$$

- tipo definito in schema di struttura:

$$\frac{\gamma(VAR : TIPO)}{\rho\texttt{(VAR).in(P(TIPO).type)}}$$

Qualora, invece, il tipo sia composto e definito mediante un costrutto di PowerSet, allora avrà la forma generica:

$VAR : \mathbb{P}\ TIPO$

Pertanto, il risultato della funzione γ applicata sulla definizione sarà, in base al tipo della variabile, rispettivamente:

- tipo definito tramite uno schema di struttura:

$$\frac{\gamma(VAR : TIPO)}{\rho\texttt{(VAR).subset(P(TIPO).type)}}$$

- tipo non definito tramite uno schema di struttura:

$$\frac{\gamma(VAR : TIPO)}{\rho\texttt{(VAR).subset(}\rho\texttt{(TIPO))}}$$

Qualora, infine, il tipo sia composto e definito mediante una Funzione Parziale, allora avrà la forma generica:

$VAR : TIPO1 \nrightarrow TIPO2$

Per esprimere il vincolo di tipo saranno quindi necessari i due vincoli sul dominio e sul rango della funzione. Pertanto, il risultato della funzione γ applicata sulla definizione sarà una congiunzione di due vincoli JSetL.

Il primo congiunto rappresenterà il vincolo sul dominio e sarà, in base al primo tipo della variabile, rispettivamente:

- tipo definito tramite uno schema di struttura:

$$\frac{\gamma(VAR : TIPO1 \nrightarrow ????)}{\rho\texttt{(VAR).dom(P(TIPO1).type)}}$$

- tipo non definito tramite uno schema di struttura:

$$\frac{\gamma(VAR : TIPO1 \nrightarrow ????)}{\rho\texttt{(VAR).dom(}\rho\texttt{(TIPO1))}}$$

Per quanto riguarda il secondo congiunto, esso rappresenterà il vincolo sul rango e sarà, in base al secondo tipo della variabile, rispettivamente:

- tipo definito tramite uno schema di struttura:

$$\frac{\gamma(\mathit{VAR} :???? \nrightarrow \mathit{TIPO2})}{\rho\texttt{(VAR).ran(P(TIPO2).type)}}$$

- tipo non definito tramite uno schema di struttura:

$$\frac{\gamma(\mathit{VAR} :???? \nrightarrow \mathit{TIPO2})}{\rho\texttt{(VAR).ran(}\rho\texttt{(TIPO2))}}$$

Quindi, complessivamente, la regola di conversione sarà:

$$\frac{\gamma(\mathit{VAR} : \mathit{TIPO1} \nrightarrow \mathit{TIPO2})}{\texttt{(}\gamma\texttt{ (VAR: TIPO1 }\nrightarrow\texttt{ ????)).and(}\gamma\texttt{(VAR: ???? }\nrightarrow\texttt{ TIPO2)}}$$

Esempio IV.13: La funzione *listino* viene convertita nell'omonimo campo di classe `LMap` e nella congiunzione di vincoli sul tipo riguardanti il dominio e il rango della funzione.

$$\mathit{listino} : \mathit{Viaggio} \nrightarrow \mathbb{Z}$$

δ(*listino* : *Viaggio* $\nrightarrow$ $\mathbb{Z}$) =

```
    public LMap listino = new LMap("LISTINO");
```

γ(*listino* : *Viaggio* $\nrightarrow$ $\mathbb{Z}$) =

```
    statPost((listino.dom(VIAGGIO.type)).and(listino.ran(z)));
```

11 Predicati

Ogni predicato Z può essere tradotto nel suo corrispondente vincolo JSetL o insieme di vincoli corrispondenti, qualora un solo vincolo JSetL non sia sufficiente. Le regole di conversione che presenteremo si applicano su predicati ed espressioni già elaborati dal parser e, pertanto, non ci occuperemo di questioni come le parentesi, la precedenza o particolari abbreviazioni di alcun genere. Ad esempio, considereremo che ogni quantificatore si riferisca ad una sola variabile, non ad una lista, e che un'eventuale lista sia riconosciuta dal parser come una serie di quantificazioni singole contenute l'una nell'altra, in modo da avere una scrittura equivalente.

Esistono due tipi di predicati atomici: i predicati insiemistici e i predicati algebrici, distinguibili grazie al tipo rilevato dal parser CZT. In Z, un predicato atomico esprime una relazione fra due espressioni. Pertanto, la forma generica di un predicato atomico è:

$$\mathit{ESPRESSIONE}1 \qquad \mathit{OPERATORE} \qquad \mathit{ESPRESSIONE}2$$

Inoltre, i predicati atomici possono essere combinati tramite l'operatore logico unario di negazione, gli operatori logici binari di disgiunzione, congiunzione, implicazione e coimplicazione e gli operatori logici ternari di quantificazione universale e quantificazione esistenziale. Pertanto, il risultato della funzione γ applicata ad un predicato sarà, in base all'operatore presente, rispettivamente:

- OPERATORI INSIEMISTICI
  - uguaglianza insiemistica:

$$\frac{\gamma(\mathit{ESPRESSIONE}1 = \mathit{ESPRESSIONE}2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{eq}(\gamma(\texttt{ESPRESSIONE2}))}$$

  - disuguaglianza insiemistica:

$$\frac{\gamma(\mathit{ESPRESSIONE}1 \neq \mathit{ESPRESSIONE}2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{neq}(\gamma(\texttt{ESPRESSIONE2}))}$$

  - appartenenza:

$$\frac{\gamma(\mathit{ESPRESSIONE}1 \in \mathit{ESPRESSIONE}2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{in}(\gamma(\texttt{ESPRESSIONE2}))}$$

  - non-appartenenza:

$$\frac{\gamma(\mathit{ESPRESSIONE}1 \notin \mathit{ESPRESSIONE}2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{nin}(\gamma(\texttt{ESPRESSIONE2}))}$$

– inclusione:

$$\frac{\gamma(ESPRESSIONE1 \subseteq ESPRESSIONE2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{subset}(\gamma(\texttt{ESPRESSIONE2}))}$$

– inclusione stretta:

$$\frac{\gamma(ESPR1 \subset ESPR2)}{(\gamma(\texttt{ESPR1}).\texttt{subset}(\gamma(\texttt{ESPR2}))).\texttt{and}(\gamma(\texttt{ESPR1}).\texttt{neq}(\gamma(\texttt{ESPR2})))}$$

- OPERATORI ALGEBRICI

– equazione:

$$\frac{\gamma(ESPRESSIONE1 = ESPRESSIONE2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{eq}(\gamma(\texttt{ESPRESSIONE2}))}$$

– disequazione:

$$\frac{\gamma(ESPRESSIONE1 \neq ESPRESSIONE2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{neq}(\gamma(\texttt{ESPRESSIONE2}))}$$

$$\frac{\gamma(ESPRESSIONE1 \leq ESPRESSIONE2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{le}(\gamma(\texttt{ESPRESSIONE2}))}$$

$$\frac{\gamma(ESPRESSIONE1 < ESPRESSIONE2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{lt}(\gamma(\texttt{ESPRESSIONE2}))}$$

$$\frac{\gamma(ESPRESSIONE1 \geq ESPRESSIONE2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{ge}(\gamma(\texttt{ESPRESSIONE2}))}$$

$$\frac{\gamma(ESPRESSIONE1 > ESPRESSIONE2)}{\gamma(\texttt{ESPRESSIONE1}).\texttt{gt}(\gamma(\texttt{ESPRESSIONE2}))}$$

- OPERATORI LOGICI

– negazione:

$$\frac{\gamma(\neg\ VINCOLO)}{(\gamma(\texttt{VINCOLO})).\texttt{notTest}()}$$

– disgiunzione:

$$\frac{\gamma(VINCOLO1 \vee VINCOLO2)}{\gamma(\texttt{VINCOLO1}).\texttt{or}(\gamma(\texttt{VINCOLO2}))}$$

– congiunzione:

$$\frac{\gamma(\mathit{VINCOLO1} \wedge \mathit{VINCOLO2})}{\gamma(\texttt{VINCOLO1}).\texttt{and}(\gamma(\texttt{VINCOLO2}))}$$

– implicazione:

$$\frac{\gamma(\mathit{VINCOLO1} \Rightarrow \mathit{VINCOLO2})}{\gamma(\texttt{VINCOLO1}).\texttt{impliesTest}(\gamma(\texttt{VINCOLO2}))}$$

– coimplicazione:

$$\frac{\gamma(\mathit{VIN1} \Leftrightarrow \mathit{VIN2})}{(\gamma(\texttt{VIN1}).\texttt{impliesTest}(\gamma(\texttt{VIN2}))).\texttt{and}(\gamma(\texttt{VIN2}).\texttt{impliesTest}(\gamma(\texttt{VIN1})))}$$

– quantificazione universale:

$$\frac{\gamma(\forall\, \mathit{VAR} : \mathit{INSIEME} \bullet \mathit{VINCOLO})}{\gamma(\texttt{INSIEME}).\texttt{forallElems}(\gamma(\texttt{VAR}),\gamma(\texttt{VINCOLO}))}$$

– quantificazione esistenziale:

$$\frac{\gamma(\exists\, \mathit{VAR} : \mathit{INSIEME} \bullet \mathit{VINCOLO})}{\gamma(\texttt{INSIEME}).\texttt{forallElems}(\gamma(\texttt{VAR}),(\gamma(\texttt{VINCOLO})).\texttt{notTest}()).\texttt{notTest}()}$$

Esempio IV.14: Viene convertita la disgiunzione di un predicato algebrico e un predicato insiemistico.

$$\frac{\gamma(\#\ \mathit{viaggi} > 0 \vee \mathit{viaggi} = \varnothing)}{(\texttt{size(viaggi).gt(0)}).\texttt{or(viaggi.eq(LSet.empty()))}}$$

Esempio IV.15: Viene convertito un predicato composto da una quantificazione universale.

$$\frac{\gamma(\forall\, x : \mathit{viaggi} \bullet x.\mathit{partenza} \neq x.\mathit{arrivo})}{\texttt{viaggi.forAllElems(x, x.partenza.neq(x.arrivo))}}$$

12 Espressioni

In Z, le espressioni sono tipate ed il parser CZT è in grado di rilevarne il tipo. In particolare, distinguiamo fra espressioni insiemistiche ed espressioni algebriche. Le regole di conversione per le espressioni Z sono quindi:

- ESPRESSIONI INSIEMISTICHE
  - insieme vuoto:

$$\frac{\gamma(\varnothing)}{\texttt{LSet.empty()}}$$

  - costante:

$$\frac{\gamma(COSTANTE)}{\texttt{"COSTANTE"}}$$

  - insieme enumerato:

$$\frac{\gamma(\{e_1, e_2, \ldots, e_n\})}{\texttt{LSet.empty().ins(}\gamma\texttt{(}e_1\texttt{)).ins(}\gamma\texttt{(}e_2\texttt{))....ins(}\gamma\texttt{(}e_n\texttt{))}}$$

  - intervallo:

$$\frac{\gamma(ESPRESSIONE1..ESPRESSIONE2)}{\texttt{new IntLSet(}\gamma\texttt{(ESPRESSIONE1),}\gamma\texttt{(ESPRESSIONE2))}}$$

  - variabile insiemistica, in uno schema di struttura:

$$\frac{\gamma(VARIABILE)}{\texttt{d.}\rho\texttt{(VARIABILE)}}$$

  - variabile insiemistica, fuori da uno schema di struttura:

$$\frac{\gamma(VARIABILE)}{\rho\texttt{(VARIABILE)}}$$

  - variabile insiemistica decorata con "′":

$$\frac{\gamma(VARIABILE')}{\gamma(\delta_{prm}(VARIABILE'))}$$

  - variabile insiemistica decorata con "?":

$$\frac{\gamma(VARIABILE?)}{\gamma(\delta_{ask}(VARIABILE?))}$$

– variabile insiemistica decorata con "!":

$$\frac{\gamma(VARIABILE!)}{\gamma(\delta_{ans}(VARIABILE!))}$$

– accesso al campo insiemistico di una variabile:

$$\frac{\gamma(VARIABILE.CAMPO)}{\gamma(\texttt{VARIABILE}).\rho(\texttt{CAMPO})}$$

– unione:

$$\frac{\gamma(ESPRESSIONE1 \cup ESPRESSIONE2)}{\texttt{union}(\gamma(\texttt{ESPRESSIONE1}),\gamma(\texttt{ESPRESSIONE2}))}$$

– intersezione:

$$\frac{\gamma(ESPRESSIONE1 \cap ESPRESSIONE2)}{\texttt{inters}(\gamma(\texttt{ESPRESSIONE1}),\gamma(\texttt{ESPRESSIONE2}))}$$

– sottrazione:

$$\frac{\gamma(ESPRESSIONE1 \setminus ESPRESSIONE2)}{\texttt{diff}(\gamma(\texttt{ESPRESSIONE1}),\gamma(\texttt{ESPRESSIONE2}))}$$

- ESPRESSIONI ALGEBRICHE

– numero intero:

$$\frac{\gamma(NUMERO)}{\texttt{NUMERO}}$$

– variabile intera, in uno schema di struttura:

$$\frac{\gamma(VARIABILE)}{\texttt{d}.\rho(\texttt{VARIABILE})}$$

– variabile intera, fuori da uno schema di struttura:

$$\frac{\gamma(VARIABILE)}{\rho(\texttt{VARIABILE})}$$

– variabile intera decorata con "′":

$$\frac{\gamma(VARIABILE')}{\gamma(\delta_{prm}(VARIABILE'))}$$

– variabile intera decorata con "?":

$$\frac{\gamma(VARIABILE?)}{\gamma(\delta_{ask}(VARIABILE?))}$$

– variabile intera decorata con "!":

$$\frac{\gamma(VARIABILE!)}{\gamma(\delta_{ans}(VARIABILE!))}$$

– accesso al campo intero di una variabile:

$$\frac{\gamma(VARIABILE.CAMPO)}{\gamma(\texttt{VARIABILE}).\rho(\texttt{CAMPO})}$$

– somma:

$$\frac{\gamma(ESPRESSIONE1 + ESPRESSIONE2)}{(\gamma(\texttt{ESPRESSIONE1})).\texttt{sum}(\gamma(\texttt{ESPRESSIONE2}))}$$

– sottrazione:

$$\frac{\gamma(ESPRESSIONE1 - ESPRESSIONE2)}{(\gamma(\texttt{ESPRESSIONE1})).\texttt{sub}(\gamma(\texttt{ESPRESSIONE2}))}$$

– moltiplicazione:

$$\frac{\gamma(ESPRESSIONE1 * ESPRESSIONE2)}{(\gamma(\texttt{ESPRESSIONE1})).\texttt{mul}(\gamma(\texttt{ESPRESSIONE2}))}$$

– divisione esatta:

$$\frac{\gamma(ESPRESSIONE1/ESPRESSIONE2)}{(\gamma(\texttt{ESPRESSIONE1})).\texttt{div}(\gamma(\texttt{ESPRESSIONE2}))}$$

– cardinalità di un insieme:

$$\frac{\gamma(\#\, ESPRESSIONE)}{\texttt{size}(\gamma(\texttt{ESPRESSIONE}))}$$

Esempio IV.16: Un insieme dato per enumerazione viene convertito in un insieme vuoto a cui poi sono aggiunti tutti gli elementi enumerati.

$$\frac{\gamma(\{EUROPA, AMERICA, ASIA, AFRICA, OCEANIA\})}{\begin{array}{l}\texttt{LSet.empty().ins(europa).ins(america)}\\ \quad\texttt{.ins(asia).ins(africa).ins(oceania)}\end{array}}$$

Esempio IV.17: Un'espressione algebrica viene convertita usando i relativi metodi JSetL.

$$\frac{\gamma((x+1)*2)}{\texttt{x.sum(1).mul(2)}}$$

Capitolo V

Un esempio completo

In questo capitolo mostriamo un semplice esempio di documento Z e il processo completo per convertirlo in un programma Java esteso con JSetL.

Modelleremo un semplice gioco, in cui il sistema genera un numero casuale e l'utente ha la possibilità di cercare di indovinarlo: se lo indovina vince, altrimenti può riprovare. È possibile avviare una nuova partita in due diverse modalità, facile e difficile, per le quali esistono intervalli rispettivamente più piccolo e più grande di numeri dai quali viene estratto il numero casuale.

1 La specifica Z

section *Gioco* **parents** *standard_toolkit*

//MESSAGGIO contiene le possibili comunicazioni del sistema per l'utente

$MESSAGGIO ::= VITTORIA \mid RIPROVA$

//MODALITA contiene le possibili modalità di svolgimento del gioco

$MODALITA ::= FACILE \mid DIFFICILE$

//Variabili e vincoli globali stabiliscono delle costanti per il gioco

$$\begin{array}{|l} limitefacile : \mathbb{N} \\ limitedifficile : \mathbb{N} \\ \hline limitefacile = 3 \\ limitedifficile = 5 \end{array}$$

//PartitaFacile è uno schema di stato che descrive, appunto, la modalità facile del gioco

$$
\begin{array}{|l}
\hline
\textit{PartitaFacile} \\
\textit{difficolta} : \textit{MODALITA} \\
\textit{casuale} : \mathbb{N} \\
\hline
\textit{difficolta} = \textit{FACILE} \\
\textit{casuale} < \textit{limitefacile} \\
\hline
\end{array}
$$

//PartitaDifficile è uno schema di stato che descrive, appunto, la modalità difficile del gioco

$$
\begin{array}{|l}
\hline
\textit{PartitaDifficile} \\
\textit{difficolta} : \textit{MODALITA} \\
\textit{casuale} : \mathbb{N} \\
\hline
\textit{difficolta} = \textit{DIFFICILE} \\
\textit{casuale} < \textit{limitedifficile} \\
\hline
\end{array}
$$

//Partita è uno schema di stato esterno che descrive globalmente lo stato del sistema

$$\textit{Partita} == \textit{PartitaFacile} \lor \textit{PartitaDifficile}$$

//PartitaIniziale è uno schema di inizializzazione esterno che setta la partita di default alla modalità facile

$$
\begin{array}{|l}
\hline
\textit{PartitaIniziale} \\
\textit{Partita}' \\
\hline
\textit{PartitaFacile}' \\
\hline
\end{array}
$$

//NuovaPartita è uno schema Delta esterno che permette di avviare una nuova partita selezionando la modalità

$$
\begin{array}{|l}
\hline
\textit{NuovaPartita} \\
\Delta \textit{Partita} \\
\textit{difficolta}? : \textit{MODALITA} \\
\hline
\textit{difficolta}' = \textit{difficolta}? \\
\textit{casuale}' \neq \textit{casuale} \\
\hline
\end{array}
$$

//ProvaGiusto è uno schema Xi che descrive il caso in cui il numero tentato sia corretto

ProvaGiusto

$\Xi Partita$
$casuale? : \mathbb{N}$
$messaggio! : MESSAGGIO$

$casuale = casuale?$
$messaggio! = VITTORIA$

//ProvaSbagliato è uno schema Xi che descrive il caso in cui il numero tentato non sia corretto

ProvaSbagliato

$\Xi Partita$
$casuale? : \mathbb{N}$
$messaggio! : MESSAGGIO$

$casuale \neq casuale?$
$messaggio! = RIPROVA$

//Prova è uno schema Xi esterno che descrive globalmente il tentativo di indovinare il numero casuale

$Prova == ProvaGiusto \lor ProvaSbagliato$

2 Operazioni preliminari

Analisi del documento All'interno del documento vengono rilevati i seguenti costrutti Z:

- i due insiemi dati *MESSAGGIO* e *MODALITA*, entrambi definiti per enumerazione
- le due variabili globali *limitefacile* e *limitedifficile*
- due vincoli globali
- gli otto schemi *PartitaFacile*, *PartitaDifficile*, *Partita*, *PartitaIniziale*, *NuovaPartita*, *ProvaGiusto*, *ProvaSbagliato*, *Prova*

Degli insiemi, delle definizioni e delle espressioni viene inoltre riconosciuto il tipo Z, che verrà utilizzato successivamente per l'applicazione delle regole di conversione.

Classificazione degli schemi Gli schemi vengono classificati come segue:

- tre schemi di stato, di cui *PartitaFacile* e *PartitaDifficile* interni e *Partita* esterno e descritto tramite un'operazione
- lo schema di inizializzazione esterno *PartitaIniziale*
- lo schema Delta esterno di input *NuovaPartita*
- tre schemi Xi di input e di output, di cui *ProvaGiusto* e *ProvaSbagliato* interni e *Prova* esterno e descritto tramite un'operazione

Espansione Vengono espansi i due insiemi dati definiti per enumerazione, le due operazioni fra schemi e i quattro schemi esterni.

Mostriamo inoltre un esempio di come il tipo $\mathbb{N}$ delle variabili *limitefacile*, *limitedifficile* e *casuale* può essere normalizzato diventando l'insieme dato $\mathbb{Z}$ con l'aggiunta del vincolo legato all'insieme $\mathbb{N}$, ovvero che i suoi elementi siano maggiori o uguali di `0`. Questa procedura non è prevista fra le regole di espansione illustrate in questa tesi, in quanto è una procedura standard nell'ambito del codice Z e pertanto ha il solo scopo illustrativo per il lettore.

Il risultato ottenuto è il seguente:

section *Gioco* **parents** *standard_toolkit*

$[MESSAGGIO]$

$[MODALITA]$

$$
\begin{array}{|l}
limitefacile : \mathbb{Z} \\
limitedifficile : \mathbb{Z} \\
VITTORIA : MESSAGGIO \\
RIPROVA : MESSAGGIO \\
FACILE : MODALITA \\
DIFFICILE : MODALITA \\
\hline
limitefacile = 3 \\
limitedifficile = 5 \\
limitefacile \geq 0 \\
limitedifficile \geq 0 \\
VITTORIA \neq RIPROVA \\
FACILE \neq DIFFICILE \\
MESSAGGIO = \{VITTORIA, RIPROVA\} \\
MODALITA = \{FACILE, DIFFICILE\}
\end{array}
$$

Partita

$$
\begin{array}{l}
difficolta : MODALITA \\
casuale : \mathbb{Z} \\
\hline
casuale \geq 0 \\
(difficolta = FACILE \land casuale < limitefacile) \lor \\
\quad (difficolta = DIFFICILE \land casuale < limitedifficile)
\end{array}
$$

PartitaIniziale

$$
\begin{array}{l}
difficolta' : MODALITA \\
casuale' : \mathbb{Z} \\
\hline
casuale' \geq 0 \\
difficolta' = FACILE \\
casuale' < limitefacile \\
(difficolta' = FACILE \land casuale' < limitefacile) \lor \\
\quad (difficolta' = DIFFICILE \land casuale' < limitedifficile)
\end{array}
$$

NuovaPartita

$$
\begin{array}{l}
difficolta : MODALITA \\
casuale : \mathbb{Z} \\
difficolta' : MODALITA \\
casuale' : \mathbb{Z} \\
difficolta? : MODALITA \\
\hline
casuale \geq 0 \\
(difficolta = FACILE \land casuale < limitefacile) \lor \\
\quad (difficolta = DIFFICILE \land casuale < limitedifficile) \\
casuale' \geq 0 \\
(difficolta' = FACILE \land casuale' < limitefacile) \lor \\
\quad (difficolta' = DIFFICILE \land casuale' < limitedifficile) \\
difficolta' = difficolta? \\
casuale' \neq casuale
\end{array}
$$

Prova

$$
\begin{array}{l}
difficolta : MODALITA \\
casuale : \mathbb{Z} \\
difficolta' : MODALITA \\
casuale' : \mathbb{Z} \\
casuale? : \mathbb{Z} \\
messaggio! : MESSAGGIO \\
\hline
difficolta' = difficolta \\
casuale' = casuale \\
casuale \geq 0 \\
(difficolta = FACILE \land casuale < limitefacile) \lor \\
\quad (difficolta = DIFFICILE \land casuale < limitedifficile) \\
casuale' \geq 0 \\
(difficolta' = FACILE \land casuale' < limitefacile) \lor \\
\quad (difficolta' = DIFFICILE \land casuale' < limitedifficile) \\
casuale? \geq 0 \\
(casuale = casuale? \land messaggio! = VITTORIA) \lor \\
\quad (casuale \neq casuale? \land messaggio! = RIPROVA)
\end{array}
$$

È facile vedere come il risultato ottenuto dall'espansione sia molto meno leggibile dell'originale, ovvero ridotto a liste di definizioni e predicati; tuttavia, questa è la forma più semplice per l'applicazione delle regole di riscrittura.

Rinominazione delle variabili globali Non essendo definite variabili con lo stesso nome in nessuno schema, *limitefacile*, *limitedifficile*, *VITTORIA*, *RIPROVA*, *FACILE* e *DIFFICILE* vengono rimpiazzate rispettivamente da *limitefacileglbl*, *limitedifficileglbl*, *VITTORIAglbl*, *RIPROVAglbl*, *FACILEglbl* e *DIFFICILEglbl* in ogni loro occorrenza.

3 Il codice generato

Non essendo presente alcuno schema di struttura, viene generata la sola classe `GIOCO` mostrata in seguito:

```
import JSetL.*;
import JSetL.z.*;

@SuppressWarnings("rawtypes")// per le sottoclassi (parametriche) di SubLVar
public class GIOCO extends ZDocument
{
  //definizioni di insiemi dati(strutture JSetL pubbliche)
  public IntLSet z = new IntLSet("Z",0,5);// approssimazione di Z, Standard
      Toolkit
  public LSet messaggio = new LSet("MESSAGGIO");
  public LSet modalita = new LSet("MODALITA");

  //definizioni di variabili globali e di stato(strutture JSetL pubbliche)
  public IntLVar limitefacileglbl = new IntLVar("LIMITEFACILEGLBL");
  public IntLVar limitedifficileglbl = new IntLVar("LIMITEDIFFICILEGLBL");
  public LVar vittoriaglbl = new LVar("VITTORIAGLBL");
  public LVar riprovaglbl = new LVar("RIPROVAGLBL");
  public LVar facileglbl = new LVar("FACILEGLBL");
  public LVar difficileglbl = new LVar("DIFFICILEGLBL");
  public LVar difficolta = new LVar("DIFFICOLTA");
  public IntLVar casuale = new IntLVar("CASUALE");

  GIOCO()
  {
    super();

    //vincoli di variabili globali e di stato, vincoli globali di stato (usare
        statPost)
    statPost(limitefacileglbl.in(z));
    statPost(limitedifficileglbl.in(z));
    statPost(vittoriaglbl.in(messaggio));
    statPost(riprovaglbl.in(messaggio));
    statPost(facileglbl.in(modalita));
    statPost(difficileglbl.in(modalita));
    statPost(limitefacileglbl.eq(3));
    statPost(limitedifficileglbl.eq(5));
    statPost(limitefacileglbl.ge(0));
    statPost(limitedifficileglbl.ge(0));
    statPost(vittoriaglbl.neq(riprovaglbl));
    statPost(facileglbl.neq(difficileglbl));
    statPost(messaggio.eq(LSet.empty().ins(vittoriaglbl).ins(riprovaglbl)));
    statPost(modalita.eq(LSet.empty().ins(facileglbl).ins(difficileglbl)));
    statPost(difficolta.in(modalita));
    statPost(casuale.in(z));
```

```
  statPost(casuale.ge(0));
  statPost((difficolta.eq(facileglbl).and(casuale.le(limitefacileglbl)))
    .or(difficolta.eq(difficileglbl).and(casuale.le(limitedifficileglbl))));

  //vincoli di insiemi dati (usare givenSet)
  givenSet(z);
  givenSet(messaggio);
  givenSet(modalita);
}

public static void main(String[] args)
{
  //crea il documento e lo apre tramite l'interfaccia
  GIOCO doc = new GIOCO();
  intz.open(doc);
}

//operazione per lo schema d'azione "partitainiziale"
public void partitainiziale ()
{
  //variabili di input? (usare askVar)

  //variabili di modifica' (usare prmVar)
  LVar difficoltaPrm = new LVar("DIFFICOLTAPRM");
  IntLVar casualePrm = new IntLVar("CASUALEPRM");
  prmVar(difficolta,difficoltaPrm);
  tempPost(difficoltaPrm.in(modalita));
  prmVar(casuale,casualePrm);
  tempPost(casualePrm.in(z));

  //variabili di output! (usare ansVar)

  //accumulo vincoli (usare tempPost)
  tempPost(casualePrm.ge(0));
  tempPost(difficoltaPrm.eq(facileglbl));
  tempPost(casualePrm.le(limitefacileglbl));
  tempPost((difficoltaPrm.eq(facileglbl)
    .and(casualePrm.le(limitefacileglbl)))
    .or(difficoltaPrm.eq(difficileglbl)
    .and(casualePrm.le(limitedifficileglbl))));

  //esecuzione operazione
  execute();
}
```

```
//operazione per lo schema d'azione "nuovapartita"
public void nuovapartita ()
{
  //variabili di input? (usare askVar)
  LVar difficoltaAsk = new LVar("DIFFICOLTAASK");
  askVar(difficoltaAsk);
  tempPost(difficoltaAsk.in(modalita));

  //variabili di modifica' (usare prmVar)
  LVar difficoltaPrm = new LVar("DIFFICOLTAPRM");
  IntLVar casualePrm = new IntLVar("CASUALEPRM");
  prmVar(difficolta,difficoltaPrm);
  tempPost(difficoltaPrm.in(modalita));
  prmVar(casuale,casualePrm);
  tempPost(casualePrm.in(z));

  //variabili di output! (usare ansVar)

  //accumulo vincoli (usare tempPost)
  tempPost(casuale.ge(0));
  tempPost((difficolta.eq(facileglbl).and(casuale.le(limitefacileglbl)))
    .or(difficolta.eq(difficileglbl).and(casuale.le(limitedifficileglbl))));
  tempPost(casualePrm.ge(0));
  tempPost((difficoltaPrm.eq(facileglbl)
    .and(casualePrm.le(limitefacileglbl)))
    .or(difficoltaPrm.eq(difficileglbl)
    .and(casualePrm.le(limitedifficileglbl))));
  tempPost(difficoltaPrm.eq(difficoltaAsk));
  tempPost(casualePrm.neq(casuale));

  //esecuzione operazione
  execute();
}

//operazione per lo schema d'azione "prova"
public void prova()
{
  //variabili di input? (usare askVar)
  IntLVar casualeAsk = new IntLVar("CASUALEASK");
  askVar(casualeAsk);
  tempPost(casualeAsk.in(z));

  //variabili di modifica' (usare prmVar)
  LVar difficoltaPrm = new LVar("DIFFICOLTAPRM");
  IntLVar casualePrm = new IntLVar("CASUALEPRM");
  prmVar(difficolta,difficoltaPrm);
  tempPost(difficoltaPrm.in(modalita));
  prmVar(casuale,casualePrm);
  tempPost(casualePrm.in(z));
```

```
    //variabili di output! (usare ansVar)
    LVar messaggioAns = new LVar("MESSAGGIOANS");
    ansVar(messaggioAns);
    tempPost(messaggioAns.in(messaggio));

    //accumulo vincoli (usare tempPost)
    tempPost(difficoltaPrm.eq(difficolta));
    tempPost(casualePrm.eq(casuale));
    tempPost(casuale.ge(0));
    tempPost((difficolta.eq(facileglbl).and(casuale.le(limitefacileglbl)))
      .or(difficolta.eq(difficileglbl).and(casuale.le(limitedifficileglbl))));
    tempPost(casualePrm.ge(0));
    tempPost((difficoltaPrm.eq(facileglbl)
      .and(casualePrm.le(limitefacileglbl)))
      .or(difficoltaPrm.eq(difficileglbl)
      .and(casualePrm.le(limitedifficileglbl))));
    tempPost(casualeAsk.ge(0));
    tempPost((casuale.eq(casualeAsk).and(messaggioAns.eq(vittoriaglbl)))
      .or(casuale.neq(casualeAsk).and(messaggioAns.eq(riprovaglbl))));

    //esecuzione operazione
    execute();
  }
}
```

Conclusioni e sviluppi futuri

Nel presente lavoro di tesi è stato descritto un procedimento per passare da una specifica in linguaggio Z a un programma in linguaggio Java esteso con JSetL.

Il processo descritto è stato fortemente generalizzato al fine di supportare correttamente un maggior numero di documenti di specifica Z. Tuttavia, il prezzo da pagare per tale generalità è una ricaduta delle prestazioni nei programmi generati, a cui durante lo svolgimento di questo lavoro non è stata data particolare importanza.

Piuttosto, l'obiettivo principale che ci eravamo preposti era quello di mantenere semplici le regole di riscrittura; pertanto, siamo soddisfatti del risultato ottenuto in quanto è stato possibile individuare un procedimento di conversione algoritmico e quindi automatizzabile.

Ciò è stato possibile grazie alla stretta somiglianza fra le specifiche Z e i servizi JSetL, ulteriormente accentuata dall'introduzione del package `JSetL.z`. Tale package è stato sviluppato in modo modulare al fine di renderlo robusto, facilmente espandibile e modificabile nei suoi diversi moduli.

Tuttavia, non ancora tutti i costrutti presenti nel linguaggio Z sono al momento supportati da JSetL e non sono state fornite regole per tutti i costrutti di Z.

In particolare, non sono ancora supportati:

- schemi/definizioni con tipo parametrico
- promozioni
- relazioni e altri tipi di funzione
- definizioni intensionali di insiemi
- tipo per le sequenze logiche

Per alcuni costrutti sintattici, ritenuti non fondamentali, JSetL non fornisce al momento supporto diretto e perciò eventuali regole di conversione sarebbero risultate eccessivamente complesse, con un risultato poco apprezzabile.

Si preferisce quindi posticipare la stesura delle regole relative ai seguenti costrutti in vista di un ampliamento dei servizi offerti da JSetL.

Si vuole tuttavia precisare che il sottinsieme delle specifiche Z trattato in questo lavoro di tesi non è da considerarsi riduttivo: ad esempio, le sequenze in Z sono trattate come funzioni parziali aventi come dominio i numeri naturali e come rango l'insieme degli oggetti presenti nella sequenza.

Pertanto, utilizzando la notazione funzionale al posto di quella solita delle sequenze, è possibile vedere che, nonostante le regole di riscrittura per le sequenze non siano state fornite, non sono da considerarsi esclusi i documenti Z che fanno uso della notazione sequenziale.

Nella maggior parte dei casi, quindi, tramite un lavoro di espansione e riscrittura della notazione Z è possibile produrre un documento Z equivalente che sia completamente compatibile con il procedimento di conversione mostrato in questa tesi, similmente a quanto descritto nelle operazioni preliminari all'inizio del capitolo 3.

Una volta ultimata la completa formalizzazione delle regole di riscrittura, sarebbe possibile progettare e realizzare un tool automatico in grado di convertire, tramite le regole, una qualsiasi specifica Z direttamente in un programma Java funzionante. Ciò permetterebbe di automatizzare il lavoro di implementazione e codifica di un progetto, riducendolo alla sola attività di specifica.

Infine, come specificato nell'introduzione, non è presente la dimostrazione della correttezza delle regole fornite. Tale dimostrazione rappresenterebbe un passo aggiuntivo verso la dimostrazione di correttezza dei programmi generati.

Ringraziamenti

Ringrazio per l'assoluta disponibilità, la dedizione e la costanza

il Professor Maximiliano Cristià

che mi ha indicato sin dal principio la strada da percorrere per la stesura del documento di specifica Z sul sistema bibliotecario, fornendomi documentazione, consigli e la Sua esperienza durante l'intera durata del mio lavoro di tirocinio e tesi.

Ringrazio per avermi offerto l'opportunità di questa collaborazione internazionale in veste di Relatore

il Professor Gianfranco Rossi

che mi ha fornito con precisione e tempestività ogni informazione utile allo svolgimento del tirocinio e mi ha guidato nella stesura di questa tesi.

Ringrazio infine per il supporto e la comprensione dimostrati nei miei confronti

i miei genitori, la mia fidanzata e i miei amici

che mi stanno sempre vicini creando intorno a me un ambiente piacevole anche durante il lavoro.

Riferimenti bibliografici

[1] Maximiliano Cristiá
Introducción a la notación Z
Ingeniería de Software I, Universidad Nacional de Rosario, Marzo de 2015

[2] Mike McMorran, Steve Powell
Z - Guide for Beginners
Blackwell Scientific Publications

[3] R.G. Dromey
Program derivation: the development of programs from specifications
Addison-Wesley, 1989

[4] Méthode B Home Page
http://methode-b.com/

[5] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.

[6] Gianfranco Rossi, Roberto Amadini
JSetL User's Manual
http://cmt.math.unipr.it/jsetl/JSetLUserManual-v.2.3.pdf

[7] JSetL Home Page
http://cmt.math.unipr.it/jsetl.html

[8] Gianluca Lutero, Gianfranco Rossi
Estensione della libreria Java JSetL con vincoli su funzioni parziali
Tesi di laurea, Università degli Studi di Parma, A.A. 2014-2015

[9] CZT: Community Z Tools
http://czt.sourceforge.net/

[10] CZT Parsers
http://czt.sourceforge.net/parser/

Appendice

Al fine di mostrare al lettore un esempio concreto di un documento di specifica Z, alleghiamo la specifica di un sistema per la gestione dei libri, dei prestiti e degli utenti di una biblioteca in cui sia possibile eseguire le seguenti operazioni:

- consultare la lista dei prestiti attivi
- aggiungere un nuovo libro
- aggiungere un nuovo utente
- verificare la presenza di un libro nella biblioteca e la sua disponibilità
- creare un prestito di un libro per un utente
- terminare un prestito alla riconsegna del libro

section *Specification* **parents** *standard_toolkit*

//ISBN rappresenta un codice identificativo univoco per ogni singolo libro

[*ISBN*]

//COPIA rappresenta un valore per distinguere le varie copie di libri con lo stesso ISBN

[*COPIA*]

//DESCRIZIONE contiene informazioni utili a ricercare un libro come: titolo, sottotitolo, autore ed edizione

[*DESCRIZIONE*]

//UTENTE contiene un identificativo univoco (nome//matricola//codice cliente) dell'utente

[*UTENTE*]

//MESSAGGIO contiene le possibili comunicazioni sulle operazioni per l'utente del sistema

$$MESSAGGIO ::= OK \mid UTENTE\_NON\_VALIDO \mid LIBRO\_NON\_ESISTENTE \mid LIBRO\_OCCUPATO \mid PRESTITO\_NON\_REGISTRATO \mid UTENTE\_NON\_REGISTRATO \mid UTENTE\_GIA\_REGISTRATO \mid LIBRO\_GIA\_PRESTATO$$

//Definizione assiomatica per le funzioni nuovaCopia e copiaEsistente

$$nuovaCopia : \mathbb{P}\, COPIA \pfun COPIA$$
$$copiaEsistente : \mathbb{P}\, COPIA \pfun COPIA$$

$$\forall A : \mathbb{P}\, COPIA \bullet nuovaCopia(A) \notin A$$
$$\forall A : \mathbb{P}\, COPIA \bullet copiaEsistente(A) \in A$$

//Libri è uno schema di stato che descrive i libri presenti nella Biblioteca, dove per lo stesso ISBN si ha una stessa DESCRIZIONE del Libro

Libri

$$lib : ISBN \times COPIA \pfun DESCRIZIONE$$

$$\forall id1, id2 : ISBN \times COPIA \bullet id1.1 = id2.1 \Rightarrow lib(id1) = lib(id2)$$

//LibriIniziali è uno schema di inizializzazione che resetta lo stato dei Libri

LibriIniziali

$$Libri'$$

$$lib' = \varnothing$$

//Prestiti è uno schema di stato che descrive i prestiti attivi della Biblioteca, ad ogni utente può essere prestata al massimo una copia di un libro

Prestiti

$$pres : ISBN \times COPIA \pfun UTENTE$$

$$\forall ut : UTENTE;\ cod : ISBN \bullet \#(\{cod\} \ndres \mathrm{dom}(pres \rhd \{ut\})) = 1$$

//PrestitiIniziali è uno schema di inizializzazione che resetta lo stato dei Prestiti

PrestitiIniziali

$$Prestiti'$$

$$pres' = \varnothing$$

//Biblioteca è uno schema di stato che modella l'intera biblioteca

Biblioteca

$$Libri$$
$$Prestiti$$
$$utenti : \mathbb{P}\ UTENTE$$

$$\text{dom}\ pres \subseteq \text{dom}\ lib$$
$$\text{ran}\ pres \subseteq utenti$$

//BibliotecaIniziale è uno schema di inizializzazione che resetta l'intera Biblioteca

BibliotecaIniziale

$$LibriIniziali$$
$$PrestitiIniziali$$
$$Biblioteca'$$

$$utenti' = \varnothing$$

//Seguono gli schemi d'azione per le seguenti operazioni: StampaPrestiti, AggiungiLibro, AggiungiUtente, CercaLibro, PrestaLibro, RestituisciLibro
//StampaPrestiti è uno schema xi che permette di visualizzare tutti i prestiti attivi

StampaPrestiti

$$\Xi Biblioteca$$
$$prestiti! : \mathbb{P}(UTENTE \times DESCRIZIONE)$$

$$prestiti! = \{id1 : \text{dom}(pres) \bullet (pres(id1), lib(id1))\}$$

//AggiungiLibro1 è il caso che il libro sia nuovo, e quindi necessiti di una descrizione

*AggiungiLibro*1

$$\Delta Biblioteca$$
$$cod? : ISBN$$
$$des? : DESCRIZIONE$$
$$res! : MESSAGGIO$$

$$cod? \notin \text{dom}(\text{dom}(lib))$$
$$lib' = lib \cup \{(cod?, nuovaCopia(\varnothing)) \mapsto des?\}$$
$$pres' = pres$$
$$utenti' = utenti$$
$$res! = OK$$

//AggiungiLibro2 è il caso che il libro non sia nuovo, e quindi non necessiti di una descrizione

*AggiungiLibro*2

$\Delta Biblioteca$
$cod? : ISBN$
$res! : MESSAGGIO$

$cod? \in \mathrm{dom}(\mathrm{dom}(lib))$
$lib' = lib \cup \{(cod?, nuovaCopia(\mathrm{ran}(\{cod?\} \lhd \mathrm{dom}(lib)))) \mapsto$
$\quad lib(cod?, copiaEsistente(\mathrm{ran}(\{cod?\} \lhd \mathrm{dom}(lib))))\}$
$pres' = pres$
$utenti' = utenti$
$res! = OK$

//AggiungiLibro è uno schema delta che permette di introdurre un nuovo libro nel sistema fornendo il suo ISBN e una sua DESCRIZIONE

$AggiungiLibro == AggiungiLibro1 \lor AggiungiLibro2$

//AggiungiUtenteOK è il caso nominale dell'operazione AggiungiUtente

AggiungiUtenteOK

$\Delta Biblioteca$
$ut? : UTENTE$
$res! : MESSAGGIO$

$ut? \notin utenti$
$utenti' = utenti \cup \{ut?\}$
$pres' = pres$
$lib' = lib$
$res! = OK$

//AggiungiUtenteERR è il caso di errore dell'operazione AggiungiUtente

AggiungiUtenteERR

$\Xi Biblioteca$
$ut? : UTENTE$
$res! : MESSAGGIO$

$ut? \in utenti$
$res! = UTENTE\_GIA\_REGISTRATO$

//AggiungiUtente è uno schema delta che permette di aggiungere un nuovo UTENTE nel sistema

$AggiungiUtente == AggiungiUtenteOK \lor AggiungiUtenteERR$

//CercaLibroOK è il caso nominale dell'operazione CercaLibro

CercaLibroOK

$\Xi Biblioteca$
$des? : DESCRIZIONE$
$cod! : ISBN$
$cop! : COPIA$
$res! : MESSAGGIO$

$(cod!, cop!) \in \mathrm{dom}(lib \rhd \{des?\})$
$\{(cod!, cop!)\} \lhd pres = \varnothing$
$res! = OK$

//CercaLibroERR1 è un caso d'errore dell'operazione CercaLibro

CercaLibroERR1

$\Xi Biblioteca$
$des? : DESCRIZIONE$
$res! : MESSAGGIO$

$lib \rhd \{des?\} = \varnothing$
$res! = LIBRO\_NON\_ESISTENTE$

//CercaLibroERR2 è un caso d'errore dell'operazione CercaLibro

CercaLibroERR2

$\Xi Biblioteca$
$des? : DESCRIZIONE$
$res! : MESSAGGIO$

$\mathrm{dom}(lib \rhd \{des?\}) \subseteq \mathrm{dom}(pres)$
$res! = LIBRO\_OCCUPATO$

//CercaLibro è uno schema d'azione che permette di individuare un libro dalla sua descrizione, se presente e disponibile

$CercaLibro == CercaLibroOK \lor CercaLibroERR1 \lor CercaLibroERR2$

//PrestaLibroOK è il caso nominale dell'operazione PrestaLibro

PrestaLibroOK

$\Delta Biblioteca$
$cod? : ISBN$
$ut? : UTENTE$
$res! : MESSAGGIO$

$ut? \in utenti$
$(\{cod?\} \lhd \text{dom}(lib)) \setminus \text{dom}(pres) \neq \varnothing$
$\{cod?\} \lhd \text{dom}(pres \rhd \{ut?\}) = \varnothing$
$pres' = pres \oplus \{(cod?, copiaEsistente(\text{ran}((\{cod?\} \lhd \text{dom}(lib)) \setminus$
$\text{dom}(pres)))) \mapsto ut?\}$
$lib' = lib$
$utenti' = utenti$
$res! = OK$

//PrestaLibroERR1 è un caso d'errore dell'operazione PrestaLibro

PrestaLibroERR1

$\Xi Biblioteca$
$cod? : ISBN$
$res! : MESSAGGIO$

$cod? \notin \text{dom}(\text{dom}(lib))$
$res! = LIBRO\_NON\_ESISTENTE$

//PrestaLibroERR2 è un caso d'errore dell'operazione PrestaLibro

PrestaLibroERR2

$\Xi Biblioteca$
$ut? : UTENTE$
$res! : MESSAGGIO$

$ut? \notin utenti$
$res! = UTENTE\_NON\_REGISTRATO$

//PrestaLibroERR3 è un caso d'errore dell'operazione PrestaLibro

PrestaLibroERR3

$\Xi Biblioteca$
$cod? : ISBN$
$res! : MESSAGGIO$

$(\{cod?\} \lhd \text{dom}(lib)) \subseteq \text{dom}(pres)$
$res! = LIBRO\_OCCUPATO$

//PrestaLibroERR4 è un caso d'errore dell'operazione PrestaLibro

PrestaLibroERR4

$\Xi Biblioteca$
$cod? : ISBN$
$ut? : UTENTE$
$res! : MESSAGGIO$

$\{cod?\} \lhd \text{dom}(pres \rhd \{ut?\}) \neq \varnothing$
$res! = LIBRO\_GIA\_PRESTATO$

//PrestaLibro è uno schema d'azione che permette di creare un prestito verso un utente di un libro esistente e disponibile nella biblioteca

$PrestaLibro == PrestaLibroOK \lor PrestaLibroERR1 \lor PrestaLibroERR2 \lor PrestaLibroERR3 \lor PrestaLibroERR4$

//RestituisciLibroOK è il caso nominale dell'operazione RestituisciLibro

RestituisciLibroOK

$\Delta Biblioteca$
$cod? : ISBN$
$ut? : UTENTE$
$res! : MESSAGGIO$

$ut? \in utenti$
$\{cod?\} \lhd \text{dom}(pres \rhd \{ut?\}) \neq \varnothing$
$pres' = (\{cod?\} \lhd \text{dom}(pres \rhd \{ut?\})) \lhd pres$
$lib' = lib$
$utenti' = utenti$
$res! = OK$

//RestituisciLibroERR1 è un caso d'errore dell'operazione RestituisciLibro

RestituisciLibroERR1

$\Xi Biblioteca$
$cod? : ISBN$
$res! : MESSAGGIO$

$cod? \notin \text{dom}(\text{dom}(lib))$
$res! = LIBRO\_NON\_ESISTENTE$

//RestituisciLibroERR2 è un caso d'errore dell'operazione RestituisciLibro

RestituisciLibroERR2

$\Xi Biblioteca$
$ut? : UTENTE$
$res! : MESSAGGIO$

$ut? \notin utenti$
$res! = UTENTE\_NON\_REGISTRATO$

//RestituisciLibroERR3 è un caso d'errore dell'operazione RestituisciLibro

RestituisciLibroERR3

$\Xi Biblioteca$
$cod? : ISBN$
$ut? : UTENTE$
$res! : MESSAGGIO$

$\{cod?\} \lhd \text{dom}(pres \rhd \{ut?\}) = \varnothing$
$res! = PRESTITO\_NON\_REGISTRATO$

//RestituisciLibro è uno schema d'azione che permette di eliminare un prestito precedentemente creato, in seguito alla restituzione di un libro

$RestituisciLibro == RestituisciLibroOK \lor RestituisciLibroERR1 \lor RestituisciLibroERR2 \lor RestituisciLibroERR3$