UNIVERSITÀ DEGLI STUDI DI PARMA

Facoltà di Scienze
Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea Triennale

Interpretazione astratta di operatori per la manipolazione di bit in linguaggi imperativi

Candidato:
Alessandro Vincenzi

Relatore:
Prof. Roberto Bagnara

Correlatore:
Prof. Enea Zaffanella

Anno Accademico 2005/2006

We are like dwarfs sitting upon the shoulders of giants,
and so able to see more and see farther than the ancients.

— Bernard of Chartres (1130)

Indice

1 Introduzione **7**

2 Preliminari **9**
2.1 Insiemi e funzioni 9
2.2 Operatori 10

3 Il linguaggio CMM **13**
3.1 La sintassi del linguaggio 13
3.2 Semantica statica 15
3.2.1 Identificatori definiti e identificatori liberi 16
3.2.2 Ambienti di tipo 16
3.2.3 Predicati di semantica statica 17
3.3 Semantica dinamica 19
3.3.1 Locazioni assolute 19
3.3.2 Ambienti di esecuzione concreti 19
3.3.3 Strutture di memoria, stati con valore e stati con eccezione 20
3.3.4 Funzione di conversione 23
3.3.5 Operatori aritmetici e *bitwise* 24
3.3.6 Configurazioni 24
3.3.7 Relazioni di valutazione concreta 25
3.3.8 Computazioni infinite 30

4 Interpretazione astratta **31**
4.1 Domini concreti e domini astratti 31
4.2 Approssimazione di domini composti 32
4.2.1 Approssimazione dei prodotti cartesiani 32
4.2.2 Approssimazione di unioni disgiunte 33
4.3 Approssimazione degli elementi di CMM 33
4.3.1 Filtri astratti 37
4.3.2 Cast astratto 38

4.3.3 Configurazioni astratte 38
4.3.4 Espressioni, dichiarazioni e statement supportati 39
4.3.5 Relazioni di valutazione astratta 40

5 Proprietà aritmetiche degli operatori *bitwise* 45
5.1 Numeri interi e loro rappresentazione 45
5.2 Complemento a uno 46
5.3 Congiunzione bit-a-bit 47
5.4 Disgiunzione inclusiva bit-a-bit 52
5.5 Disgiunzione esclusiva bit-a-bit 57

6 Implementazione degli operatori *bitwise* 63
6.1 Il dominio degli intervalli 63
6.2 Complemento a uno 64
6.3 Congiunzione bit-a-bit 65
6.4 Disgiunzione inclusiva bit-a-bit 71
6.5 Disgiunzione esclusiva bit-a-bit 76
6.6 Scorrimento a sinistra 77
6.7 Scorrimento a destra 78

7 Conclusioni 81

Capitolo 1

Introduzione

In informatica, l'*interpretazione astratta* è una teoria di approssimazione della semantica dei programmi, basata sull'utilizzo di funzioni monotone su insiemi parzialmente ordinati come, ad esempio, i reticoli. Può essere vista come l'esecuzione non standard dei sorgenti di un programma, eseguita con l'intento di ottenere informazioni sulla semantica del codice ma senza eseguire fedelmente tutte le operazioni.

Questa tesi fornisce un esempio di interpretazione astratta degli operatori aritmetici utilizzati per la manipolazione di bit nei linguaggi imperativi (come C, C++, Java e Phyton).

Saranno quindi presi in considerazione gli operatori bit-a-bit:

- $\sim$ complemento ad uno,
- $\wedge$ congiunzione,
- $\vee$ disgiunzione inclusiva,
- $\veebar$ disgiunzione esclusiva,
- $\ll$ scorrimento a sinistra,
- $\gg$ scorrimento a destra.

Tali operatori considerano i numeri interi (positivi e negativi) come una semplice sequenza di bit: i primi quattro operano in parallelo sui bit nella stessa posizione ed eseguono l'operazione logica che rappresentano; gli ultimi due operatori, invece, fanno scorrere i bit nelle due direzioni destra e sinistra. Una descrizione dettagliata della loro semantica è riportata nella sezione 2.2.

Nel capitolo 3 verrà introdotto un linguaggio imperativo, che per comodità chiameremo CMM (C-meno-meno) in quanto i tipi e le categorie sintattiche considerate sono un sottoinsieme del C. Tale linguaggio fornisce il

contesto in cui utilizzare gli operatori per la manipolazione di bit. In particolare, CMM mette a disposizione tipi di dato distinti per gli interi illimitati, gli interi limitati con segno e gli interi limitati senza segno, consentendo quindi una significativa trattazione delle problematiche associate alla formalizzazione della semantica degli operatori sopra menzionati, nonchè alla loro successiva approssimazione. Di CMM verrà specificata la sintassi (sezione 3.1), la semantica statica (sezione 3.2) e la semantica dinamica (sezione 3.3). Le specifiche fanno riferimento ai modelli definiti ed utilizzati in [1].

Il capitolo 4 introdurrà il concetto di interpretazione astratta e spiegherà quale modello è stato preso in considerazione per approssimare tutti gli elementi introdotti per la definizione del linguaggio CMM e dei suoi comportamenti dinamici: per ognuno di essi, e per ogni costrutto utilizzato, verranno fornite regole che specificano quali caratteristiche devono essere garantite per realizzare un'interpretazione corretta del linguaggio CMM. Anche in questo caso sono stati utilizzati i sistemi già presenti in [1].

Nel capitolo 5 si dimostreranno alcune proprietà aritmetiche degli operatori di complemento a uno, congiunzione, disgiunzione esclusiva e inclusiva *bitwise*.

Nel capitolo 6, infine, si prenderà in considerazione il dominio degli intervalli di numeri interi. Per ogni operatore di manipolazione dei bit sarà descritto un algoritmo che utilizza gli intervalli per individuare gli estremi superiori ed inferiori dei valori ottenuti eseguendo le operazioni *bitwise* su insiemi di numeri invece che su singoli numeri interi come accade nell'esecuzione concreta di un programma.

Capitolo 2

Preliminari

2.1 Insiemi e funzioni

Gli insiemi di numeri sono indicati con i seguenti simboli:

- $\mathbb{N}$ insieme dei numeri naturali, zero incluso;
- $\mathbb{Z}$ insieme dei numeri interi;
- $\bar{\mathbb{Z}}$ insieme degli interi estesi con i due simboli di più e meno infinito ($\mathbb{Z} \cup \{+\infty, -\infty\}$).
- $\mathbb{R}$ insieme dei numeri reali.

Siano S e T due insiemi. La notazione $S \subseteq_{\mathrm{f}} T$ indica che S è un *sottoinsieme finito* di T. Scriviamo $S \uplus T$ per indicare l' unione $S \cup T$ quando $S \cap T = \varnothing$. L' insieme delle funzioni totali (risp., parziali) da S a T è indicato con $S \to T$ (risp., $S \rightarrowtail T$) e nel seguito indicheremo con $\mathrm{dom}(f)$ il *dominio* della corrispondente funzione $f \colon S \to T$ (risp., $f \colon S \rightarrowtail T$), ovvero l'insieme $\mathrm{dom}(f) = S$ (risp., $\mathrm{dom}(f) \subseteq S$).

Sia $S = \{s_1, \ldots, s_n\}$ un insieme finito di cardinalità $n \geq 0$. Allora la notazione $\{s_1 \mapsto t_1, \ldots, s_n \mapsto t_n\}$, dove $\{t_1, \ldots, t_n\} \subseteq T$, indica la funzione $f \colon S \to T$ tale che $f(s_i) = t_i$, per ogni $i = 1, \ldots, n$. Si osservi che, nel caso in cui il codominio T sia identificabile dal contesto, l'insieme vuoto $\varnothing$ indica la funzione $f \colon \varnothing \to T$.

Quando denotiamo l'applicazione di una funzione $f \colon (S_1 \times \cdots \times S_n) \to T$, ometteremo, per convenzione, le parentesi che racchiudono la ennupla degli argomenti e scriveremo $f(s_1, \ldots, s_n)$ per indicare $f\big((s_1, \ldots, s_n)\big)$.

Siano $f_0 \colon S_0 \rightarrowtail T_0$ e $f_1 \colon S_1 \rightarrowtail T_1$ due funzioni parziali. La funzione $f_0[f_1] \colon (S_0 \cup S_1) \rightarrowtail (T_0 \cup T_1)$ è definita, per ogni $x \in \mathrm{dom}(f_0) \cup \mathrm{dom}(f_1)$, da

$$\big(f_0[f_1]\big)(x) \stackrel{\text{def}}{=} \begin{cases} f_1(x), & \text{se } x \in \mathrm{dom}(f_1); \\ f_0(x), & \text{se } x \in \mathrm{dom}(f_0) \setminus \mathrm{dom}(f_1). \end{cases}$$

(Notare che, se f_0 e f_1 sono funzioni totali, allora $f_0[f_1]$ è a sua volta totale.)

Sia $f \colon S \rightarrowtail T$ una funzione parziale e $S' \subseteq S$ un insieme: la notazione $f\,|_{S'}$ indica la restrizione di f su S', ovvero la funzione $f\,|_{S'} \colon S' \rightarrowtail T$ definita, per ogni $x \in S' \cap \mathrm{dom}(f)$, da $f\,|_{S'}(x) = f(x)$. (Notare che, se f è una funzione totale, allora $f\,|_{S'}$ è a sua volta totale.)

Sia $f \colon D_1 \times \cdots \times D_n \to D_0$, una funzione il cui domino è composto da n reticoli limitati definiti come $(D_i, \sqsubseteq_i, \bot_i, \sqcup_i)$, per ogni $i = 0, \ldots, n$. Allora la funzione f è *stretta sull'i-simo argomento* se $d_i = \bot_i$ implica $f(d_1, \ldots, d_n) = \bot_0$.

Sia D un insieme e $A, B \subseteq D$ due suoi sottoinsiemi; l'*estensione puntuale* di un operatore binario $\cdot \colon D \times D \to D$ è definita come l'operatore $\odot \colon \wp(D) \times \wp(D) \to \wp(D)$ tale che $A \odot B = \{\, a \cdot b \in D \;.\; a \in A, b \in B \,\}$.

$S^\star$ indica l'insieme di tutte le stringhe (anche vuote) ottenute concatenando i simboli in S. La stringa vuota è indicata con ϵ. Se $w, z \in S \cup S^\star$, la concatenazione di w e z è un elemento di $S^\star$ indicato con wz (oppure con $w \cdot z$).

La funzione totale *parte intera* $\mathrm{int} \colon \mathbb{R} \to \mathbb{Z}$ è definita come

$$\forall\, x \in \mathbb{R}, \quad \mathrm{int}(x) \stackrel{\text{def}}{=} \begin{cases} \lfloor x \rfloor, & \text{se } x \geq 0, \\ \lceil x \rceil, & \text{se } x < 0, \end{cases}$$

La funzione totale $\neg \colon \{0,1\} \to \{0,1\}$ corrisponde con la negazione logica di un singolo bit, ovvero $\neg 0 = 1$ e $\neg 1 = 0$.

2.2 Operatori

In CMM si possono utilizzare le principali operazioni aritmetiche disponibili nei linguaggi imperativi: in questa sezione si introdurranno le specifiche degli operatori. Non è restrittivo assumere che tutte le operazioni si eseguono solamente tra operandi dello stesso tipo. I tipi considerati per la definizione della semantica degli operatori di CMM sono i tipi base in Type definiti nella successiva sezione 3.1.

Tutte le operazioni aritmetiche sono sempre definite se eseguite sugli interi illimitati. Come espresso nello standard C in [4, Sezione 6.2.5, punto 9], le

operazioni aritmetiche eseguite tra tipi senza segno sono sempre definite e nel caso in cui il risultato dell'operazione non è rappresentabile, tale valore verrà ridotto modulo un numero pari al massimo valore rappresentabile nel tipo considerato più uno. Discriminiamo quindi il comportamento degli operatori di CMM nel caso in cui vengano valutati su interi limitati e con segno.

Meno unario: il risultato del meno unario è l'opposto del suo operando. Se il risultato non è rappresentabile nel tipo con segno, allora tale operatore può generare un numero casuale o segnalare l'evento con il lancio di un'eccezione.

Somma: il risultato della somma binaria è la somma dei due operandi. Se il valore ottenuto non è rappresentabile nel tipo del risultato allora questo operatore può generare un numero casuale o lanciare un'eccezione.

Differenza: il risultato della differenza binaria è il valore ottenuto sottraendo il secondo operando dal primo. Se il valore ottenuto non è rappresentabile nel tipo del risultato allora la valutazione della differenza può generare un numero casuale o segnalare l'evento con il lancio di un'eccezione.

Prodotto: il risultato del prodotto binario è il prodotto due operandi. Se il valore ottenuto non è rappresentabile nel tipo del risultato tale operatore può generare un numero casuale o segnalare l'evento con il lancio di un'eccezione.

Divisione intera: il risultato della divisione intera è la parte intera della divisione del primo operando per il secondo. Se il secondo operando è nullo o se il valore ottenuto non è rappresentabile, allora tale operatore può generare un numero casuale o segnalare l'evento con il lancio di un'eccezione.

Modulo: il modulo restituisce il resto della divisione intera tra il primo e il secondo operando. Se il secondo operando è nullo allora questo operatore può generare un numero casuale o segnalare l'evento con il lancio di un'eccezione.

Complemento a uno: il risultato è la negazione bit-a-bit dell'operando: ogni bit nel risultato sarà 1 se e solo se il bit corrispondente nell'operando è pari a 0.

Congiunzione: il risultato è l'and bit-a-bit dei due operandi: ogni bit nel risultato sarà 1 se e solo se ognuno dei corrispondenti bit negli operandi è pari a 1.

Disgiunzione inclusiva: il risultato è l'or bit-a-bit dei due operandi: ogni bit nel risultato sarà 1 se e solo se almeno uno dei due corrispondenti bit negli operandi è pari a 1.

Disgiunzione esclusiva: il risultato è l'or esclusivo bit-a-bit dei due operandi: ogni bit nel risultato sarà 1 se e solo se esattamente uno dei corrispondenti bit negli operandi è pari a 1.

Scorrimento a sinistra: i bit dell'operando a sinistra scorrono verso

sinistra di un valore pari a quello indicato dall'operando a destra; altrettanti zeri saranno inseriti da destra. In termini aritmetici, per ogni $x, y \in \mathbb{Z}$,

$$x \ll y \stackrel{\text{def}}{=} x \cdot 2^y.$$

Se il valore dell'operando sinistro è non negativo ed il risultato è rappresentabile in tale tipo allora lo scorrimento sinistro restituirà $x \cdot 2^y$ come espresso nella definizione. Altrimenti, se il valore del risultato non è rappresentabile o l'operando sinistro è negativo, lo *shift* può generare un valore intero arbitrario o lanciare un'eccezione.

Scorrimento a destra: i bit dell'operando a sinistra scorrono verso destra di un valore pari a quello indicato dall'operando a destra; altrettanti zeri saranno inseriti da sinistra. In termini aritmetici, per ogni $x, y \in \mathbb{Z}$,

$$x \gg y \stackrel{\text{def}}{=} x \div 2^y.$$

Se l'operando sinistro ha un valore non negativo, allora lo scorrimento destro restituirà $x \div 2^y$ come espresso nella definizione. Altrimenti lo scorrimento a destra può generare un valore intero arbitrario o lanciare un'eccezione.

Per entrambi gli operatori di scorrimento occorre precisare, in aggiunta a quanto precedentemente specificato, che l'esecuzione di queste due operazioni restituisce un valore intero arbitrario o lancia un'eccezione nel caso in cui:

- l'operando a destra è un valore negativo;
- l'operando destro è maggiore o uguale alla lunghezza in bit dell'operando sinistro.

La definizione della semantica degli operatori per la manipolazione di bit fa riferimento alla semantica degli operatori del C, come indicato in [4, Sezione 6.5.3.3, punto 4], [4, Sezioni 6.5.10-12] e [4, Sezione 6.5.7].

Capitolo 3

Il linguaggio CMM

In questo capitolo verrà introdotto il linguaggio imperativo CMM. Tale linguaggio offre la possibilità di dichiarare e utilizzare un tipo di dato booleano, un tipo intero illimitato e infiniti tipi di dati interi con e senza segno. Utilizza inoltre un modello di memoria non banale, che permette di mantenere traccia dei nomi delle variabili (allocate in modo automatico), della loro posizione nella memoria, del loro tipo e del loro valore. La semantica dinamica di CMM è stata definita in modo da considerare il verificarsi di comportamenti eccezionali, come un errore di memoria o una divisione per zero: nel caso in cui si verificano degli errori tutte le istruzioni che si incontrano a partire dal punto in cui si sono verificati fino alla fine del programma saranno ignorate.

La definizione del linguaggio inizierà con l'introduzione della sintassi e delle regole di semantica statica. Verranno poi introdotti tutti i costrutti e gli operatori utili alla creazione del modello di memoria con cui eseguire i programmi: grazie ad essi sarà possibile definire in modo preciso i comportamenti dinamici che si verificano durante l'esecuzione.

3.1 La sintassi del linguaggio

Nel seguito indicheremo con k la dimensione della rappresentazione binaria degli interi limitati. Tale valore è da considerarsi come un multiplo della lunghezza di una *word*, ovvero un valore intero non nullo che può assumere, ad esempio, i valori nell'insieme $\mathsf{K} = \{8, 16, 32, 64, \ldots\}$.

Gli insiemi sintattici di base del linguaggio CMM sono:

Identificatori

$$\mathrm{id} \in \mathrm{Id} \stackrel{\text{def}}{=} \{\mathrm{id}_0, \mathrm{id}_1, \mathrm{id}_2, \ldots\};$$

Tipi aritmetici con segno

$$\text{sig} \in \text{Signed} \stackrel{\text{def}}{=} \{\text{integer}\} \cup \{\text{sint}_\text{k} \,.\, \mathsf{k} \in \mathsf{K}\};$$

Tipi aritmetici senza segno

$$\text{unsig} \in \text{Unsigned} \stackrel{\text{def}}{=} \{\text{uint}_\text{k} \,.\, \mathsf{k} \in \mathsf{K}\};$$

Tipi aritmetici

$$\text{aT} \in \text{aType} \stackrel{\text{def}}{=} \text{Signed} \uplus \text{Unsigned};$$

Tipo booleano

$$\text{bT} \in \text{bType} \stackrel{\text{def}}{=} \{\text{boolean}\};$$

Tipi base

$$T \in \text{Type} \stackrel{\text{def}}{=} \text{aType} \uplus \text{bType};$$

Interi illimitati

$$m^{\mathsf{i}_\infty} \in \text{Integers} \stackrel{\text{def}}{=} \{(m, \mathsf{i}_\infty) \,.\, m \in \mathbb{Z}\};$$

Interi limitati con segno

$$m^{\mathsf{s}_\mathsf{k}} \in \text{SInt}_\text{k} \stackrel{\text{def}}{=} \{(m, \mathsf{s}_\mathsf{k}) \,.\, m \in \mathbb{Z},\ (-2^{k-1}) \leq m \leq (2^{k-1} - 1)\};$$

Interi limitati senza segno

$$m^{\mathsf{u}_\mathsf{k}} \in \text{UInt}_\text{k} \stackrel{\text{def}}{=} \{(m, \mathsf{u}_\mathsf{k}) \,.\, m \in \mathbb{N},\ m \leq 2^k - 1\};$$

Booleani

$$m^{\mathsf{b}} \in \text{Bool} \stackrel{\text{def}}{=} \{(0, \mathsf{b}), (1, \mathsf{b})\};$$

Eccezioni run-time

$$\chi \in \text{RTSExcept} \stackrel{\text{def}}{=} \{\texttt{stkovflw}, \texttt{divbyzero}, \texttt{converror}, \ldots\}.$$

Partendo dagli insiemi di base, possiamo quindi definire le principali categorie sintattiche, specificate con le seguenti regole espresse in forma BNF:

Costanti

$$\text{Con} \ni \text{con} ::= m^{\mathsf{i}_\infty} \mid m^{\mathsf{s}_\mathsf{k}} \mid m^{\mathsf{u}_\mathsf{k}} \mid m^{\mathsf{b}}$$

Espressioni

$$\begin{aligned}\text{Exp} \ni e ::= {} & \text{con} \mid \text{id} \mid (T)\, e \\ & \mid -e \mid e_0 + e_1 \mid e_0 - e_1 \mid e_0 * e_1 \mid e_0 \,/\, e_1 \mid e_0 \,\%\, e_1 \\ & \mid \sim e \mid e_0 \wedge e_1 \mid e_0 \vee e_1 \mid e_0 \veebar e_1 \mid e_0 \ll e_1 \mid e_0 \gg e_2 \\ & \mid e_0 = e_1 \mid e_0 \neq e_1 \mid e_0 < e_1 \mid e_0 \leq e_1 \mid e_0 \geq e_1 \mid e_0 > e_1 \\ & \mid \textbf{not}\, e \mid e_0 \,\textbf{and}\, e_1 \mid e_0 \,\textbf{or}\, e_1\end{aligned}$$

Dichiarazioni

$$\text{Decl} \ni d ::= \textbf{nil} \mid \textbf{var}\, \text{id} : T = e \mid d_0; d_1$$

Statement

$$\begin{aligned}\text{Stmt} \ni s ::= {} & \textbf{nop} \mid \text{id} := e \mid d; s \mid s_0; s_1 \\ & \mid \textbf{if}\, e\, \textbf{then}\, s_0\, \textbf{else}\, s_1 \mid \textbf{while}\, e\, \textbf{do}\, s\end{aligned}$$

Un programma scritto in CMM è un singolo statement $s \in \text{Stmt}$ e per questo motivo non vi è la necessità di definire la categoria sintattica dei programmi.

La funzione type: Con $\to$ Type, che riconduce una costante numerica o booleana al nome del suo tipo, è definita da:

$$\text{type}(\text{con}) \stackrel{\text{def}}{=} \begin{cases} \text{integer}, & \text{se con} = m^{\mathsf{i}\infty}; \\ \text{sint}_\text{k}, & \text{se con} = m^{\mathsf{s}_\mathsf{k}}; \\ \text{uint}_\text{k}, & \text{se con} = m^{\mathsf{u}_\mathsf{k}}; \\ \text{boolean}, & \text{se con} = m^{\mathsf{b}}. \end{cases}$$

3.2 Semantica statica

La semantica statica del linguaggio CMM stabilisce le regole che permettono di verificare che il codice sorgente sia scritto in modo corretto nel rispetto della grammatica che lo definisce. Assumiamo che il processo di verifica della semantica statica sia svolto in due fasi: in un primo momento il codice sorgente sarà elaborato da tools di trasformazione del codice che introdurrà, ove necessario, cast impliciti in modo da avere operazioni aritmetiche, *bitwise* e confronti solamente tra operandi dello stesso tipo come è stato assunto nella precedente sezione 2.2. Un tool adatto a questo scopo è [5]. In un secondo momento si eseguirà il controllo dei tipi le cui regole sono riportate in questa sezione. I predicati di semantica dinamica definiti nella successiva sezione 3.3 saranno applicati solamente a programmi ben tipati.

3.2.1 Identificatori definiti e identificatori liberi

L'insieme degli identificatori introdotti dalle dichiarazioni è definito come segue:

$$\begin{aligned} \mathrm{DI}(\mathbf{nil}) &\stackrel{\text{def}}{=} \varnothing; \\ \mathrm{DI}(\mathbf{var}\,\mathrm{id} : T = e) &\stackrel{\text{def}}{=} \{\mathrm{id}\}; \\ \mathrm{DI}(d_0; d_1) &\stackrel{\text{def}}{=} \mathrm{DI}(d_0) \cup \mathrm{DI}(d_1). \end{aligned}$$

L'insieme di identificatori liberi che appaiono in espressioni, dichiarazioni e statement è invece definito da:

$$\begin{aligned} \mathrm{FI}(\mathrm{con}) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{nop}) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{nil}) &\stackrel{\text{def}}{=} \varnothing; \\ \mathrm{FI}(\mathbf{var}\,\mathrm{id} : T = e) \stackrel{\text{def}}{=} \mathrm{FI}((T)\,e) &\stackrel{\text{def}}{=} \mathrm{FI}(e); \\ \mathrm{FI}(-e) \stackrel{\text{def}}{=} \mathrm{FI}(\sim e) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{not}\,e) &\stackrel{\text{def}}{=} \mathrm{FI}(e); \\ \mathrm{FI}(e_0\ \mathrm{op}\ e_1) &\stackrel{\text{def}}{=} \mathrm{FI}(e_0) \cup \mathrm{FI}(e_1); \\ \mathrm{FI}(\mathrm{id}) &\stackrel{\text{def}}{=} \{\mathrm{id}\}; \\ \mathrm{FI}(\mathrm{id} := e) &\stackrel{\text{def}}{=} \{\mathrm{id}\} \cup \mathrm{FI}(e); \\ \mathrm{FI}(d_0; d_1) &\stackrel{\text{def}}{=} \mathrm{FI}(d_0) \cup \big(\mathrm{FI}(d_1) \setminus \mathrm{DI}(d_0)\big); \\ \mathrm{FI}(d; s) &\stackrel{\text{def}}{=} \mathrm{FI}(d) \cup \big(\mathrm{FI}(s) \setminus \mathrm{DI}(d)\big); \\ \mathrm{FI}(s_0; s_1) &\stackrel{\text{def}}{=} \mathrm{FI}(s_0) \cup \mathrm{FI}(s_1); \\ \mathrm{FI}(\mathbf{if}\,e\,\mathbf{then}\,s_0\,\mathbf{else}\,s_1) &\stackrel{\text{def}}{=} \mathrm{FI}(e) \cup \mathrm{FI}(s_0) \cup \mathrm{FI}(s_1); \\ \mathrm{FI}(\mathbf{while}\,e\,\mathbf{do}\,s) &\stackrel{\text{def}}{=} \mathrm{FI}(e) \cup \mathrm{FI}(s); \end{aligned}$$

con $\mathrm{op} \in \{+, \ldots, \%, \wedge, \ldots, \gg, =, \ldots, >, \mathbf{and}, \mathbf{or}\}$.

3.2.2 Ambienti di tipo

Definiamo la nuova categoria sintattica dei

TIPI DENOTABILI

$$\mathrm{dType} \ni \mathrm{dT} ::= T\ \mathrm{loc}$$

Un ambiente di tipo è quindi definibile come una funzione che associa ad ogni identificatore di un dato insieme finito il suo corrispondente tipo denotabile.

Definizione 3.2.1 (TEnv_I, TEnv) *Per ogni* $I \subseteq_{\mathrm{f}} \mathrm{Id}$*, l'insieme degli* ambienti di tipo su I *è* $\mathrm{TEnv}_I \stackrel{\text{def}}{=} I \to \mathrm{dType}$*; l'insieme di tutti gli ambienti di tipo è definito da* $\mathrm{TEnv} \stackrel{\text{def}}{=} \biguplus_{I \subseteq_{\mathrm{f}} \mathrm{Id}} \mathrm{TEnv}_I$*. Gli ambienti di tipo saranno in seguito rappresentati con i simboli* β*,* β_0*,* β_1*, ... Per convenzione, la notazione* $\beta : I$ *indicherà in modo più conciso che* $\beta \in \mathrm{TEnv}_I$*.*

3.2.3 Predicati di semantica statica

Siano $I \subseteq_{\mathrm{f}} \mathrm{Id}$ e $\beta \in \mathrm{TEnv}_I$. I seguenti predicati, elencati insieme al loro significato informale, specificano la correttezza rispetto ai tipi di un programma i cui identificatori liberi sono contenuti in I.

$\beta \vdash_I e : T$, e è ben formata ed ha tipo T in β;
$\beta \vdash_I d : \delta$, d è ben formata e produce l'ambiente di tipo δ in β;
$\beta \vdash_I s$, s è ben formato in β.

I seguenti predicati sono definiti induttivamente dalle seguenti regole:

Espressioni

Costante

$$\frac{}{\beta \vdash_I \mathrm{con} : T} \quad \text{se } \mathrm{type}(\mathrm{con}) = T$$

Identificatore

$$\frac{}{\beta \vdash_I \mathrm{id} : T} \quad \text{se } \beta(\mathrm{id}) = T\ \mathrm{loc}$$

Cast esplicito

$$\frac{\beta \vdash_I e : T_0}{\beta \vdash_I (T)\, e : T}$$

Operazioni aritmetiche

$$\frac{\beta \vdash_I e : \mathrm{aT}}{\beta \vdash_I -e : \mathrm{aT}}$$

$$\frac{\beta \vdash_I e_0 : \mathrm{aT} \quad \beta \vdash_I e_1 : \mathrm{aT}}{\beta \vdash_I e_0 \boxdot e_1 : \mathrm{aT}} \quad \text{se } \boxdot \in \{+, -, *, /, \%\}$$

Operazioni *bitwise*

$$\frac{\beta \vdash_I e : \mathrm{aT}}{\beta \vdash_I \sim e : \mathrm{aT}}$$

$$\frac{\beta \vdash_I e_0 : \mathrm{aT} \quad \beta \vdash_I e_1 : \mathrm{aT}}{\beta \vdash_I e_0 \boxdot e_1 : \mathrm{aT}} \quad \text{se } \boxdot \in \{\wedge, \vee, \veebar, \ll, \gg\}$$

Test aritmetici

$$\frac{\beta \vdash_I e_0 : \text{aT} \quad \beta \vdash_I e_1 : \text{aT}}{\beta \vdash_I e_0 \circledast e_1 : \text{boolean}} \quad \text{se } \circledast \in \{=, \neq, <, \leq, \geq, >\}$$

Espressioni booleane

$$\frac{\beta \vdash_I e : \text{boolean}}{\beta \vdash_I \textbf{not } e : \text{boolean}}$$

$$\frac{\beta \vdash_I e_0 : \text{boolean} \quad \beta \vdash_I e_1 : \text{boolean}}{\beta \vdash_I e_0 \diamond e_1 : \text{boolean}} \quad \text{se } \diamond \in \{\textbf{and}, \textbf{or}\}$$

Dichiarazioni

Nil

$$\frac{}{\beta \vdash_I \textbf{nil} : \varnothing}$$

Dichiarazione di variabile

$$\frac{\beta \vdash_I e : T}{\beta \vdash_I \textbf{var}\, \text{id} : T = e : \{\text{id} \mapsto T \text{ loc}\}}$$

Concatenazione di dichiarazioni

$$\frac{\beta \vdash_I d_0 : \beta_0 \quad \beta[\beta_0] \vdash_{I \cup \text{DI}(d_0)} d_1 : \beta_1}{\beta \vdash_I d_0; d_1 : \beta_0[\beta_1]}$$

Statement

Nop

$$\frac{}{\beta \vdash_I \textbf{nop}}$$

Assegnamento

$$\frac{\beta \vdash_I e : T}{\beta \vdash_I \text{id} := e} \quad \text{se } \beta(\text{id}) = T \text{ loc}$$

Blocco

$$\frac{\beta \vdash_I d : \beta_0 \quad \beta[\beta_0] \vdash_{I \cup \text{DI}(d)} s}{\beta \vdash_I d; s}$$

Concatenazione di statement

$$\frac{\beta \vdash_I s_0 \quad \beta \vdash_I s_1}{\beta \vdash_I s_0; s_1}$$

Test

$$\frac{\beta \vdash_I e : \text{boolean} \quad \beta \vdash_I s_0 \quad \beta \vdash_I s_1}{\beta \vdash_I \textbf{if}\, e\, \textbf{then}\, s_0\, \textbf{else}\, s_1}$$

Loop

$$\frac{\beta \vdash_I e : \text{boolean} \quad \beta \vdash_I s}{\beta \vdash_I \textbf{while}\, e\, \textbf{do}\, s}$$

Un programma s è ritenuto *valido* se e solo se $\varnothing \vdash_\varnothing s$.

3.3 Semantica dinamica

3.3.1 Locazioni assolute

Una *locazione assoluta* (o, semplicemente, *locazione*) è un identificatore univoco di un'area di memoria di dimensione non specificata. L'insieme (possibilmente infinito) di tutte le locazioni è indicato con Loc, mentre le singole locazioni sono indicate con l, l_0, l_1, ...

3.3.2 Ambienti di esecuzione concreti

Definizione 3.3.1 (dVal, Env_I) *L'insieme dei* valori concreti denotabili *è*

$$\text{dVal} \stackrel{\text{def}}{=} (\text{Loc} \times \text{Type}).$$

Per $I \subseteq_\text{f} \text{Id}$, $\text{Env}_I \stackrel{\text{def}}{=} I \to \text{dVal}$ *è l'insieme degli* ambienti concreti su I. *L'insieme di tutti gli ambienti concreti di esecuzione (in seguito denotati semplicemente con 'ambienti') è definito da* $\text{Env} \stackrel{\text{def}}{=} \biguplus_{I \subseteq_\text{f} \text{Id}} \text{Env}_I$. *Gli ambienti in* Env_I *saranno denotati con* ρ, ρ_0, ρ_1, ... *Per convenzione, la notazione* $\rho : I$ *indicherà in modo più conciso che* $\rho \in \text{Env}_I$. *Dati* $\rho : I$ *e* $\beta : I$, *scriveremo* $\rho : \beta$ *per indicare che*

$$\forall \text{id} \in I : \big(\exists (l, T) \in \text{Loc} \times \text{Type} \;.\; \beta(\text{id}) = T\,\text{loc} \wedge \rho(\text{id}) = (l, T)\big).$$

3.3.3 Strutture di memoria, stati con valore e stati con eccezione

Una *struttura di memoria* è composta da due elementi: uno *stack* ed una mappa di memoria. La mappa di memoria sarà modellata con una funzione parziale che associa ad ogni coppia 'locazione-tipo' del proprio dominio a un corrispondente valore numerico. Il nostro linguaggio utilizza solamente variabili automatiche, quindi la struttura di memoria dovrà occuparsi di gestire un unico insieme di variabili, e la coppia '*stack*-mappa' di memoria è sufficiente per i nostri scopi. Come sarà chiaro dalle definizioni seguenti, il nostro concetto di struttura di memoria non sarà specificato completamente: definiremo il modello e gli operatori in modo da poter descrivere la semantica dei programmi ma lasceremo libera l'implementazione evitando di scendere nei dettagli della sua costruzione. Inoltre, saranno definiti degli operatori specifici che si occuperanno di allocare/deallocare, leggere, scrivere e aggiornare i dati indicizzati dalle locazioni.

Le strutture di memoria saranno in seguito usate per descrivere il risultato di computazioni i cui unici comportamenti osservabili sono dati dai loro *side effect*. Computazioni che producono come risultato un valore proprio saranno descritte da uno *stato con valore*, che raccoglie in esso sia il valore prodotto che la struttura di memoria il cui stato è indice del *side effect* dell'esecuzione. Il risultato di un comportamento eccezionale sarà descritto da una coppia (ovvero uno *stato con eccezione*) definita con la struttura di memoria e l'eccezione stessa.

Definizione 3.3.2 (Map, Stack, Mem) *L'insieme di tutte le mappe assolute è*

$$\mathrm{Map} \stackrel{\mathrm{def}}{=} (\mathrm{Loc} \times \mathrm{Type}) \rightarrowtail \mathrm{Con}.$$

Le mappe assolute sono indicate con μ, μ_0, μ_1, ...

Sia $W = \big(\mathrm{Loc} \cup \{\dagger\}\big)^\star$. *Un elemento* $w \in W$ *è uno* stack *se e solo se ogni locazione appare in esso al più una volta. L'insieme di tutti gli* stack *è* Stack *ed il simbolo '*$\dagger$*' è chiamato* stack marker.

Una struttura di memoria *è un elemento del prodotto cartesiano* $\mathrm{Mem} \stackrel{\mathrm{def}}{=} \mathrm{Map} \times \mathrm{Stack}$. *Le strutture di memoria sono indicate con* σ, σ_0, σ_1, ...

Con queste definizioni possiamo definire uno *stato con valore* come un generico elemento in $\mathrm{ValState} \stackrel{\mathrm{def}}{=} \mathrm{Con} \times \mathrm{Mem}$ e denotarlo con i simboli υ, υ_0, υ_1, ... Uno *stato con eccezione* è invece un generico elemento in $\mathrm{ExceptState} \stackrel{\mathrm{def}}{=} \mathrm{Mem} \times \mathrm{RTSExcept}$ e lo denotiamo con ε, ε_0, ε_1, ...

Definiamo ora una funzione parziale che ci permette di operare sulle mappe assolute per modificarne i valori: si tratta della funzione di aggiornamento

$$\cdot[\cdot := \cdot]\colon \big(\mathrm{Map} \times (\mathrm{Loc} \times \mathrm{Type}) \times \mathrm{Con}\big) \rightarrowtail \mathrm{Map}$$

definita, per ogni $\mu \in \mathrm{Map}$, $(l, T) \in \mathrm{Loc} \times \mathrm{Type}$ tale che $(l, T) \in \mathrm{dom}(\mu)$ e $\mathrm{con} \in \mathrm{Con}$ tale che $T = \mathrm{type}(\mathrm{con})$, da

$$\mu\big[(l, T) := \mathrm{con}\big] \stackrel{\mathrm{def}}{=} \mu',$$

dove $\mu' \in \mathrm{Map}$ è una qualsiasi mappa assoluta che soddisfi le seguenti condizioni:

1. $\mathrm{dom}(\mu') = \mathrm{dom}(\mu)$;
2. $\mu'(l, T) = \mathrm{con}$;
3. $\mu'(l', T') = \mu(l', T')$, per ogni $(l', T') \in \mathrm{dom}(\mu)$ tali che $l' \neq l$.

In pratica si tratta di una funzione che, data una mappa di memoria, una coppia 'locazione-tipo' ed una costante, restituisce una nuova mappa di memoria con lo stesso dominio e le stesse "conoscenze" della mappa precedente, ma in grado di "rispondere" con la costante data se interrogata proprio sulla coppia locazione-tipo usata dall'operatore. Grazie a tale funzione possiamo definire gli operatori per la gestione della memoria di cui abbiamo bisogno.

Gli operatori di lettura e aggiornamento per una struttura di memoria

$$\begin{aligned} \cdot[\cdot, \cdot]\colon \big(\mathrm{Mem} \times \mathrm{Loc} \times \mathrm{Type}\big) &\to (\mathrm{ValState} \uplus \mathrm{ExceptState}), \\ \cdot[\cdot := \cdot]\colon \big(\mathrm{Mem} \times (\mathrm{Loc} \times \mathrm{Type}) \times \mathrm{Con}\big) &\to (\mathrm{Mem} \uplus \mathrm{ExceptState}) \end{aligned}$$

sono rispettivamente definiti, per ogni $\sigma = (\mu, w) \in \mathrm{Mem}$, $l \in \mathrm{Loc}$, $T \in \mathrm{Type}$ e $\mathrm{con} \in \mathrm{Con}$, come segue: sia $d = (l, T)$; allora

$$\sigma[l, T] \stackrel{\mathrm{def}}{=} \begin{cases} \big(\mu(d), \sigma\big), & \text{se } d \in \mathrm{dom}(\mu); \\ (\sigma, \texttt{memerror}), & \text{altrimenti;} \end{cases}$$

$$\sigma\big[(l, T) := \mathrm{con}\big] \stackrel{\mathrm{def}}{=} \begin{cases} \big(\mu[d := \mathrm{con}], w\big), & \text{se } d \in \mathrm{dom}(\mu) \text{ e } T = \mathrm{type}(\mathrm{con}); \\ (\sigma, \texttt{memerror}), & \text{altrimenti.} \end{cases}$$

Entrambi gli operatori sono definiti nei termini delle operazioni di lettura e aggiornamento proprie della mappa assoluta e rispondono con uno stato con valore o con eccezione a seconda del risultato ottenuto.

La funzione di allocazione

$$\text{new}\colon \text{ValState} \to \big((\text{Mem} \times \text{Loc}) \uplus \text{ExceptState}\big)$$

è invece definita, per ogni $\upsilon = (\text{con}, \sigma) \in \text{ValState}$, con $\sigma = (\mu, w)$, da

$$\text{new}(\upsilon) \stackrel{\text{def}}{=} \begin{cases} ((\mu', w'), l), & \text{se lo stack di } \sigma \text{ può essere esteso;} \\ (\sigma, \texttt{stkovflw}), & \text{altrimenti;} \end{cases}$$

dove $w' \in \text{Stack}$, $l \in \text{Loc}$ e $\mu' \in \text{Map}$ sono tali che:

1. $w' = w \cdot l$;
2. per ogni $T \in \text{Type}$, $(l, T) \notin \text{dom}(\mu)$;
3. per ogni $(l', T') \in \text{dom}(\mu)$, $\mu'(l', T') = \mu(l', T')$;
4. $\mu'\big(l, \text{type}(\text{sval})\big) = \text{sval}$.

La funzione che si occupa di marcare lo *stack*, $\text{mark}\colon \text{Mem} \to \text{Mem}$, è definita, per $(\mu, w) \in \text{Mem}$, da

$$\text{mark}\big((\mu, w)\big) \stackrel{\text{def}}{=} (\mu, w\dagger).$$

La funzione parziale $\text{unmark}\colon \text{Mem} \rightarrowtail \text{Mem}$ che, al contrario della precedente, si occupa di rimuovere l'ultimo segmento dello *stack* è definita, per ogni $\sigma = (\mu, w'\dagger w'') \in \text{Mem}$ tali che $w'' \in \text{Loc}^\star$, da

$$\text{unmark}\big((\mu, w'\dagger w'')\big) \stackrel{\text{def}}{=} (\mu', w'),$$

dove la mappa assoluta $\mu' \in \text{Map}$ soddisfa:

1. $\text{dom}(\mu') = \big\{\, (l, T) \in \text{dom}(\mu) \;.\; l \text{ non appare in } w'' \,\big\}$;
2. $\mu' = \mu\,|_{\text{dom}(\mu')}$.

Per comodità di notazione, la funzione parziale unmark è definita anche in caso di stato con eccezione. In pratica, per ogni $\varepsilon = (\sigma, \chi) \in \text{ExceptState}$,

$$\text{unmark}(\varepsilon) \stackrel{\text{def}}{=} \big(\text{unmark}(\sigma), \chi\big).$$

Intuitivamente, le due funzioni mark e unmark utilizzano lo *stack marker* '†' per implementare l'allocazione e la deallocazione automatica di variabili dallo *stack*. Si osservi però che tale meccanismo di allocazione non è direttamente gestito dalle funzioni che sono in diretto contatto con la memoria. Il

meccanismo, infatti, segue la seguente dinamica: ogni volta che si incontra un blocco (ovvero una coppia dichiarazione-statement), lo *stack* viene automaticamente marcato e la funzione new aggiunge nuovi dati nella mappa assoluta ogni volta che si incontra una singola dichiarazione di variabile all'interno di tale blocco. In seguito alle dichiarazioni si possono modificare i valori delle variabili tramite l'assegnamento. All'uscita del blocco, infine, viene automaticamente chiamata la funzione unmark che elimina dallo *stack* tutte le locazioni aggiunte con la new dalla marcatura fino a questo punto. Per garantire la corretta deallocazione delle variabili anche quando lo statement termina restituendo uno stato con eccezione si è estesa la definizione di unmark.

Assumiamo inoltre che, per evitare problemi di *aliasing*, non vi siano sovrapposizioni tra le celle di memoria associate a (l_0, T_0) e le celle associate a (l_1, T_1), se non nel caso in cui $l_0 = l_1$. Inoltre, non specifichiamo quale sia la relazione tra il risultato di $\mu(l, T_0)$ e $\mu(l, T_1)$ nel caso in cui $T_0 \neq T_1$.

3.3.4 Funzione di conversione

Assumiamo l'esistenza di una funzione totale per la conversione dei valori costanti nel caso in cui si debbano eseguire delle operazioni di cast. Tale funzione, definita come

$$\text{cast}\colon \big(\text{Type} \times \text{Con}\big) \to \text{Con} \uplus \text{RTSExcept},$$

opera come segue: per ogni costante $\text{con} \in \text{Con}$ tale che $\text{con} = (m, \mathsf{t})$,

$$\begin{aligned}
\text{cast}(\text{boolean}, \text{con}) &\stackrel{\text{def}}{=} \begin{cases} (0, \mathsf{b}), & \text{se } m = 0; \\ (1, \mathsf{b}), & \text{se } m \neq 0 \end{cases} \\
\text{cast}(\text{integer}, \text{con}) &\stackrel{\text{def}}{=} (m, \mathsf{i}_\infty); \\
\text{cast}(\text{sint}_\mathrm{k}, \text{con}) &\stackrel{\text{def}}{=} \begin{cases} (m, \mathsf{s}_\mathsf{k}), & \text{se } (m, \mathsf{s}_\mathsf{k}) \in \text{SInt}_\mathrm{k} \\ \surd, & \text{altrimenti;} \end{cases} \\
\text{cast}(\text{uint}_\mathrm{k}, \text{con}) &\stackrel{\text{def}}{=} \begin{cases} (m, \mathsf{u}_\mathsf{k}), & \text{se } (m, \mathsf{u}_\mathsf{k}) \in \text{UInt}_\mathrm{k} \\ (m \wedge 2^k, \mathsf{u}_\mathsf{k}), & \text{altrimenti.} \end{cases}
\end{aligned}$$

dove $\surd \in \text{Con} \uplus \text{RTSExcept}$ può coincidere con $(m_0, \mathsf{s}_\mathsf{k})$ dove m_0 è un numero intero scelto in modo casuale o con l'eccezione `converror` $\in$ RTSExcept.

3.3.5 Operatori aritmetici e *bitwise*

Sia $\boxdot$ un operatore concreto in $\{+, -, *, /, \%, \ll, \gg\}$. Ognuno di tali operatori è definito come una funzione

$$\boxdot : \mathrm{Con} \times \mathrm{Con} \to \mathrm{Con} \uplus \mathrm{RTSExcept}$$

I rimanenti operatori concreti per la manipolazione di bit $\boxdot \in \{\sim, \wedge, \vee, \veebar\}$ restituiscono sempre una costante e sono invece definiti come funzioni

$$\boxdot : \mathrm{Con} \times \mathrm{Con} \to \mathrm{Con}$$

3.3.6 Configurazioni

La semantica dinamica di CMM viene espressa attraverso una *relazione di valutazione (o riduzione)*, che specifica come una *configurazione non-terminale* viene ridotta ad una *configurazione terminale*. Gli insiemi di configurazioni non-terminali sono parametrici nel rispetto di un ambiente di tipo in modo da poter associare ogni identificatore al suo tipo corrispondente.

Definizione 3.3.3 (Configurazioni non-terminali) *Gli insiemi di* configurazioni concrete non-terminali *per le espressioni, le dichiarazioni e gli statement, sono dati, per ogni* $\beta \in \mathrm{TEnv}_I$*, da*

$$\Gamma_{\mathrm{e}}^{\beta} \stackrel{\mathrm{def}}{=} \left\{ \langle e, \sigma \rangle \in \mathrm{Exp} \times \mathrm{Mem} \;.\; \exists T \in \mathrm{Type} \;.\; \beta \vdash_I e : T \right\},$$
$$\Gamma_{\mathrm{d}}^{\beta} \stackrel{\mathrm{def}}{=} \left\{ \langle d, \sigma \rangle \in \mathrm{Decl} \times \mathrm{Mem} \;.\; \exists \delta \in \mathrm{TEnv} \;.\; \beta \vdash_I d : \delta \right\},$$
$$\Gamma_{\mathrm{s}}^{\beta} \stackrel{\mathrm{def}}{=} \left\{ \langle s, \sigma \rangle \in \mathrm{Stmt} \times \mathrm{Mem} \;.\; \beta \vdash_I s \right\}.$$

Ogni tipo di configurazione terminale deve permettere la possibilità di computare sia in uno stato con valore che in uno stato con eccezione.

Definizione 3.3.4 (Configurazioni terminali) *Gli insiemi di* configurazioni concrete terminali *per le espressioni, le dichiarazioni e gli statement sono dati, rispettivamente, da*

$$T_{\mathrm{e}} \stackrel{\mathrm{def}}{=} \mathrm{ValState} \uplus \mathrm{ExceptState},$$
$$T_{\mathrm{d}} \stackrel{\mathrm{def}}{=} (\mathrm{Env} \times \mathrm{Mem}) \uplus \mathrm{ExceptState},$$
$$T_{\mathrm{s}} \stackrel{\mathrm{def}}{=} \mathrm{Mem} \uplus \mathrm{ExceptState}.$$

In seguito, scriveremo N per indicare una configurazione concreta non-terminale e η per indicare una configurazione concreta terminale.

3.3.7 Relazioni di valutazione concreta

Le relazioni di valutazione concreta che completano la definizione della semantica concreta di CMM sono definite per induzione strutturale da un insieme di regole di valutazione. Le relazioni di valutazione sono del tipo

$$\rho \vdash_\beta N \to \eta,$$

dove $\beta \in \mathrm{TEnv}_I$, $\rho \in \mathrm{Env}_J$, $\rho : \beta\,|_J$ e, per qualche $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{s}\}$, $N \in \Gamma_q^\beta$ e $\eta \in T_q$.

Espressioni

Costante

$$\frac{}{\rho \vdash_\beta \langle \mathrm{con}, \sigma\rangle \to \langle \mathrm{con}, \sigma\rangle} \tag{3.1}$$

Identificatore

$$\frac{}{\rho \vdash_\beta \langle \mathrm{id}, \sigma\rangle \to \sigma\big[\rho(\mathrm{id})\big]} \tag{3.2}$$

Cast esplicito

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle (T)\, e, \sigma\rangle \to \varepsilon} \tag{3.3}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{con}, \sigma_0\rangle}{\rho \vdash_\beta \langle (T)\, e, \sigma\rangle \to \eta} \tag{3.4}$$

dove

$$\eta = \begin{cases} \langle \mathrm{con}_0, \sigma_0\rangle & \text{se } \mathrm{cast}(T, \mathrm{con}) = \mathrm{con}_0; \\ \langle \sigma_0, \texttt{converror}\rangle & \text{se } \mathrm{cast}(T, \mathrm{con}) = \texttt{converror}. \end{cases}$$

Operazioni aritmetiche

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle -e, \sigma\rangle \to \varepsilon} \tag{3.5}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{con}, \sigma_0\rangle}{\rho \vdash_\beta \langle -e, \sigma\rangle \to \eta} \tag{3.6}$$

dove

$$\eta = \begin{cases} \langle \mathrm{con}_0, \sigma_0 \rangle & \text{se } -\mathrm{con} = \mathrm{con}_0; \\ \langle \sigma_0, \chi \rangle & \text{se } -\mathrm{con} = \chi. \end{cases}$$

Indichiamo con $\boxdot$ ogni operatore sintattico astratto in $\{+, -, *, /, \%\}$ e con '$\circ$' la corrispondente operazione aritmetica in $\{+, -, \cdot, \div, \mathrm{mod}\}$. Allora le regole per addizione, sottrazione, moltiplicazione, divisione intera e modulo sono date dai seguenti schemi:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \varepsilon} \tag{3.7}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle \mathrm{con}_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \varepsilon} \tag{3.8}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle \mathrm{con}_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \langle \mathrm{con}_1, \sigma_1 \rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \eta} \tag{3.9}$$

dove

$$\eta = \begin{cases} \langle \mathrm{con}_2, \sigma_1 \rangle & \text{se } \mathrm{con}_0 \circ \mathrm{con}_1 = \mathrm{con}_2; \\ \langle \sigma_1, \chi \rangle & \text{se } \mathrm{con}_0 \circ \mathrm{con}_1 = \chi; \end{cases}$$

Operazioni *bitwise* Sia $\backsim$ l'operatore sintattico astratto corrispondente all'operazione $\sim$; allora la regola per il complemento a uno è

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle \sim e, \sigma \rangle \to \varepsilon} \tag{3.10}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \langle \mathrm{con}, \sigma_0 \rangle}{\rho \vdash_\beta \langle \sim e, \sigma \rangle \to \langle \backsim \mathrm{con}, \sigma_0 \rangle} \tag{3.11}$$

Indichiamo ora con $\boxdot$ ogni operatore sintattico astratto in $\{\wedge, \vee, \veebar\}$ e con '$\circ$' la corrispondente operazione per la manipolazione dei bit in $\{\wedge, \vee, \veebar\}$. Allora le regole per le operazioni di congiunzione, disgiunzione inclusiva ed esclusiva bit-a-bit sono date dallo schema seguente:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \varepsilon} \tag{3.12}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle \mathrm{con}_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \varepsilon} \tag{3.13}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle \mathrm{con}_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \langle \mathrm{con}_1, \sigma_1 \rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \langle \mathrm{con}_0 \circ \mathrm{con}_1, \sigma_1 \rangle} \tag{3.14}$$

Indichiamo ora con $\boxdot$ ogni operatore sintattico astratto in $\{\ll, \gg\}$ e con '$\circ$' la corrispondente operazione per lo scorrimento dei bit in $\{\circledcirc, \circledcirc\}$. Le regole che descrivono gli operatori di scorrimento sono:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma\rangle \to \varepsilon} \tag{3.15}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle \mathrm{con}_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma\rangle \to \varepsilon} \tag{3.16}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle \mathrm{con}_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \langle \mathrm{con}_1, \sigma_1\rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma\rangle \to \eta} \tag{3.17}$$

$$\text{dove } \eta = \begin{cases} \langle \mathrm{con}_2, \sigma_1\rangle, & \text{se } \mathrm{con}_0 \circ \mathrm{con}_1 = \mathrm{con}_2; \\ \langle \sigma_1, \texttt{shifterror}\rangle, & \text{se } \mathrm{con}_0 \circ \mathrm{con}_1 = \texttt{shifterror}. \end{cases}$$

Test aritmetici Sia $\circledast \in \{=, \neq, <, \leq, \geq, >\}$ un operatore sintattico astratto e sia '$\lesseqgtr$' la corrispondente operazione di test in $\mathbb{Z} \times \mathbb{Z} \to \mathrm{Bool}$. Le regole per i test aritmetici sono date dagli schemi seguenti:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma\rangle \to \varepsilon} \tag{3.18}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle \mathrm{con}_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma\rangle \to \varepsilon} \tag{3.19}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle \mathrm{con}_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \langle \mathrm{con}_1, \sigma_1\rangle}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma\rangle \to \langle \mathrm{con}_0 \lesseqgtr \mathrm{con}_1, \sigma_1\rangle} \tag{3.20}$$

Espressioni booleane

$$\frac{\rho \vdash_\beta \langle b, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \textbf{not}\ \ b, \sigma\rangle \to \varepsilon} \tag{3.21}$$

$$\frac{\rho \vdash_\beta \langle b, \sigma\rangle \to \langle t, \sigma_0\rangle}{\rho \vdash_\beta \langle \textbf{not}\ \ b, \sigma\rangle \to \langle \neg\, t, \sigma_0\rangle} \tag{3.22}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle b_0 \textbf{ and } b_1, \sigma\rangle \to \varepsilon} \tag{3.23}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle}{\rho \vdash_\beta \langle b_0 \textbf{ and } b_1, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle} \tag{3.24}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle \quad \rho \vdash_\beta \langle b_1, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle b_0 \textbf{ and } b_1, \sigma\rangle \to \eta} \tag{3.25}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle b_0 \textbf{ or } b_1, \sigma\rangle \to \varepsilon} \tag{3.26}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle}{\rho \vdash_\beta \langle b_0 \textbf{ or } b_1, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle} \tag{3.27}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle \quad \rho \vdash_\beta \langle b_1, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle b_0 \textbf{ or } b_1, \sigma\rangle \to \eta} \tag{3.28}$$

Dichiarazioni

Nil

$$\frac{}{\rho \vdash_\beta \langle \textbf{nil}, \sigma\rangle \to \langle \varnothing, \sigma\rangle} \tag{3.29}$$

Dichiarazione di variabile

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \textbf{var}\, \mathrm{id} : T = e, \sigma\rangle \to \varepsilon} \tag{3.30}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \upsilon}{\rho \vdash_\beta \langle \textbf{var}\, \mathrm{id} : T = e, \sigma\rangle \to \varepsilon} \quad \text{se } \mathrm{new}(\upsilon) = \varepsilon \tag{3.31}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \upsilon}{\rho \vdash_\beta \langle \textbf{var}\, \mathrm{id} : T = e, \sigma\rangle \to \big\langle \{\mathrm{id} \mapsto (l, T)\}, \sigma_1\big\rangle} \quad \text{se } \mathrm{new}(\upsilon) = (\sigma_1, l) \tag{3.32}$$

Concatenazione di dichiarazioni

$$\frac{\rho \vdash_\beta \langle d_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle d_0; d_1, \sigma\rangle \to \varepsilon} \tag{3.33}$$

$$\frac{\rho \vdash_\beta \langle d_0, \sigma\rangle \to \langle \rho_0, \sigma_0\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle d_1, \sigma_0\rangle \to \varepsilon}{\rho \vdash_\beta \langle d_0; d_1, \sigma\rangle \to \varepsilon} \tag{3.34}$$

se $\beta \vdash_{\mathrm{FI}(d_0)} d_0 : \beta_0$

$$\frac{\rho \vdash_\beta \langle d_0, \sigma\rangle \to \langle \rho_0, \sigma_0\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle d_1, \sigma_0\rangle \to \langle \rho_1, \sigma_1\rangle}{\rho \vdash_\beta \langle d_0; d_1, \sigma\rangle \to \big\langle \rho_0[\rho_1], \sigma_1\big\rangle} \tag{3.35}$$

se $\beta \vdash_{\mathrm{FI}(d_0)} d_0 : \beta_0$

Statement

NOP

$$\frac{}{\rho \vdash_\beta \langle \mathbf{nop}, \sigma\rangle \to \sigma} \tag{3.36}$$

ASSEGNAMENTO

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathrm{id} := e, \sigma\rangle \to \varepsilon} \tag{3.37}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{con}, \sigma_0\rangle}{\rho \vdash_\beta \langle \mathrm{id} := e, \sigma\rangle \to \sigma_0\big[\rho(\mathrm{id}) := \mathrm{con}\big]} \tag{3.38}$$

BLOCCO

$$\frac{\rho \vdash_\beta \big\langle d, \mathrm{mark}(\sigma)\big\rangle \to \varepsilon}{\rho \vdash_\beta \langle d; s, \sigma\rangle \to \mathrm{unmark}(\varepsilon)} \tag{3.39}$$

$$\frac{\rho \vdash_\beta \big\langle d, \mathrm{mark}(\sigma)\big\rangle \to \langle \rho_0, \sigma_0\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle s, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle d; s, \sigma\rangle \to \mathrm{unmark}(\eta)} \tag{3.40}$$

se $\beta \vdash_{\mathrm{FI}(d)} d : \beta_0$

CONCATENAZIONE DI STATEMENT

$$\frac{\rho \vdash_\beta \langle s_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle s_0; s_1, \sigma\rangle \to \varepsilon} \tag{3.41}$$

$$\frac{\rho \vdash_\beta \langle s_0, \sigma\rangle \to \sigma_0 \quad \rho \vdash_\beta \langle s_1, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle s_0; s_1, \sigma\rangle \to \eta} \tag{3.42}$$

TEST

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma\rangle \to \varepsilon} \tag{3.43}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle \quad \rho \vdash_\beta \langle s_0, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma\rangle \to \eta} \tag{3.44}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle \quad \rho \vdash_\beta \langle s_1, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma\rangle \to \eta} \tag{3.45}$$

LOOP

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \to \varepsilon} \tag{3.46}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \to \sigma_0} \tag{3.47}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle \quad \rho \vdash_\beta \langle s, \sigma_0\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \to \varepsilon} \tag{3.48}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle \quad \rho \vdash_\beta \langle s, \sigma_0\rangle \to \sigma_1 \quad \rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma_1\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \to \eta} \tag{3.49}$$

3.3.8 Computazioni infinite

Le normali computazioni possono non terminare: lo statement iterativo, infatti, nella regola 3.49 reintroduce ricorsivamente l'elemento della conclusione nelle sue premesse e, nel caso in cui non si verifichino mai comportamenti eccezionali, come nelle regole 3.46 o 3.48, o la guardia non venga mai valutata ad un valore falso come in 3.47, la computazione potrebbe portare alla costruzione di un albero infinito.

In questo contesto non ci preoccuperemo di tale eventualità ed assumeremo che gli alberi ottenuti fondendo le regole della sezione 3.3.7 siano sempre finiti. Per una trattazione che consideri anche alberi infiniti si veda [1, Sezione 5.6].

Capitolo 4

Interpretazione astratta

4.1 Domini concreti e domini astratti

Il concetto chiave dell'interpretazione astratta è che l'insieme dei valori numerici assunti dalle variabili durante l'esecuzione di un programma possono essere sostituiti da una proprietà o da un valore astratto, in modo da poter definire la semantica astratta dell'intero programma. Le definizioni che seguono fanno riferimento al modello proposto in [6].

Un dominio di valori concreti è modellato come un reticolo completo

$$(C, \sqsubseteq, \bot, \top, \sqcap, \sqcup).$$

La relazione di approssimazione concreta $c_1 \sqsubseteq c_2$ vale se c_1 è una proprietà più forte di c_2. Un dominio di valori astratti è modellato come un semi-reticolo limitato

$$(D^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp)$$

dotato di un elemento *bottom* ($\bot^\sharp \in D^\sharp$) e per cui, per ogni $d_1^\sharp, d_2^\sharp \in D^\sharp$, esista il least upper bound $d_1^\sharp \sqcup^\sharp d_2^\sharp$. Il dominio astratto $D^\sharp$ è legato a quello concreto C da una funzione monotona $\gamma\colon D^\sharp \to C$ di concretizzazione. L'approssimazione è detta essere stretta se γ è una funzione stretta.

Per approssimare specifici oggetti concreti, assumiamo l'esistenza di una funzione di astrazione parziale $\alpha\colon C \rightarrowtail D^\sharp$ tale che, per ogni $c \in C$, se $\alpha(c)$ è definita, allora $c \sqsubseteq \gamma\big(\alpha(c)\big)$. In particolare, assumiamo che $\alpha(\bot) = \bot^\sharp$ sia sempre definita.

I nostri domini concreti sono ottenuti come il reticolo delle parti

$$\big(\wp(D), \subseteq, \varnothing, D, \cap, \cup\big)$$

di qualche insieme di oggetti concreti D, implicitamente ordinato con l'operazione di inclusione insiemistica. In tale situazione, assumiamo che ci sia

anche un elemento *top* ($\top^\sharp \in D^\sharp$) tale che $\alpha\big(\wp_{\mathrm{f}}(D)\big) \sqsubseteq \top^\sharp$. Vista l'esistenza di questo elemento, indicheremo il semi-reticolo degli elementi astratti con

$$(D^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp).$$

In aggiunta, per ogni oggetto concreto $d \in D$ e ogni elemento astratto $d^\sharp \in D^\sharp$ tale che i corrispondenti domini siano legati dalla funzione di concretizzazione $\gamma\colon D^\sharp \to \wp(D)$, scriviamo $d \propto d^\sharp$ e $d \not\propto d^\sharp$ per indicare, rispettivamente, che $d \in \gamma(d^\sharp)$ e $d \notin \gamma(d^\sharp)$. Per una notazione più leggibile, indicheremo $\sqsubseteq^\sharp$, $\bot^\sharp$, $\top^\sharp$ e $\sqcup^\sharp$ con, rispettivamente, $\sqsubseteq$, $\bot$, $\top$ e $\sqcup$.

4.2 Approssimazione di domini composti

Le approssimazioni di domini concreti composti sono di solito ottenute combinando le approssimazioni già definite per i loro componenti base. Per $i = 1$, 2, si considerino l'insieme di oggetto concreti D_i ed il corrispondente reticolo $\big(\wp(D_i), \subseteq, \varnothing, D_i, \cap, \cup\big)$; sia anche $D_i^\sharp$ un dominio astratto legato a $\wp(D_i)$ dalla funzione di concretizzazione $\gamma_i\colon D_i^\sharp \to \wp(D_i)$.

4.2.1 Approssimazione dei prodotti cartesiani

I valori del prodotto cartesiano $D_1 \times D_2$ possono essere approssimati da elementi del prodotto cartesiano $D_1^\sharp \times D_2^\sharp$. In pratica, il dominio astratto $\big(D_1^\sharp \times D_2^\sharp, \sqsubseteq, \bot, \top, \sqcup\big)$ è legato al reticolo concreto $\big(\wp(D_1 \times D_2), \subseteq, \varnothing, D_1 \times D_2, \cap, \cup\big)$ dalla funzione di concretizzazione $\gamma\colon (D_1^\sharp \times D_2^\sharp) \to \wp(D_1 \times D_2)$ definita, per ogni $(d_1^\sharp, d_2^\sharp) \in D_1^\sharp \times D_2^\sharp$, da

$$\gamma(d_1^\sharp, d_2^\sharp) \stackrel{\text{def}}{=} \big\{\, (d_1, d_2) \in D_1 \times D_2 \;.\; d_1 \in \gamma_1(d_1^\sharp), d_2 \in \gamma_2(d_2^\sharp) \,\big\}. \tag{4.1}$$

Quindi l'approssimazione $(d_1, d_2) \propto (d_1^\sharp, d_2^\sharp)$ vale se e solo se $d_1 \propto d_1^\sharp$ e $d_2 \propto d_2^\sharp$.

In tale approssimazione, $D_1^\sharp$ e $D_2^\sharp$ sono entrambe strette, quindi un migliore schema di approssimazione può essere ottenuto usando il costrutto *smash product*, che realizza una semplice forma di riduzione collassando $(d_1^\sharp, d_2^\sharp)$ all'elemento *bottom* ogni qualvolta che $d_1^\sharp = \bot$ o $d_2^\sharp = \bot$. Per esempio,

$$D_1^\sharp \otimes D_2^\sharp \stackrel{\text{def}}{=} \big\{\, (d_1^\sharp, d_2^\sharp) \in D_1^\sharp \times D_2^\sharp \;.\; d_1^\sharp = \bot \text{ se e solo se } d_2^\sharp = \bot \,\big\}.$$

La funzione di concretizzazione è definita esattamente come per (4.1). La funzione di costruzione $\cdot \otimes \cdot\colon (D_1^\sharp \times D_2^\sharp) \to (D_1^\sharp \otimes D_2^\sharp)$ è definita da

$$d_1^\sharp \otimes d_2^\sharp \stackrel{\text{def}}{=} \begin{cases} (d_1^\sharp, d_2^\sharp), & \text{se } d_1^\sharp \neq \bot \text{ e } d_2^\sharp \neq \bot; \\ \bot, & \text{altrimenti.} \end{cases}$$

4.2.2 Approssimazione di unioni disgiunte

Supponiamo che $D_1 \cap D_2 = \varnothing$. Allora i valori dell'unione disgiunta $D = D_1 \uplus D_2$ possono essere approssimati dagli elementi del prodotto cartesiano $D^\sharp = D_1^\sharp \times D_2^\sharp$. In questo caso, il dominio astratto $D^\sharp$ è legato al reticolo concreto $\big(\wp(D), \subseteq, \varnothing, D, \cap, \cup\big)$ attraverso la funzione di concretizzazione $\gamma\colon (D_1^\sharp \times D_2^\sharp) \to \wp(D_1 \uplus D_2)$ definita, per ogni $(d_1^\sharp, d_2^\sharp) \in D_1^\sharp \times D_2^\sharp$, da

$$\gamma(d_1^\sharp, d_2^\sharp) \stackrel{\text{def}}{=} \gamma_1(d_1^\sharp) \uplus \gamma_2(d_2^\sharp).$$

Di conseguenza, l'approssimazione fornita da $D^\sharp$ è stretta se lo sono anche quelle di $D_1^\sharp$ e $D_2^\sharp$. Per semplificare la notazione, se $d_1^\sharp \in D_1^\sharp$ allora scriveremo $d_1^\sharp$ per indicare anche l'elemento astratto $(d_1^\sharp, \bot) \in D^\sharp$; similmente, $d_2^\sharp \in D_2^\sharp$ indica anche l'elemento astratto $(\bot, d_2^\sharp) \in D^\sharp$. Come al solito, per ogni $i = 1$, 2 e $d_i \in D_i$, la notazione $d_i \propto (d_1^\sharp, d_2^\sharp)$ indica l'asserzione $d_i \in \gamma(d_1^\sharp, d_2^\sharp)$, che è equivalente a $d_i \in \gamma_i(d_i^\sharp)$. Ricordiamo che $D_1^\sharp \uplus^\sharp D_2^\sharp \neq D_1^\sharp \uplus D_2^\sharp$. e che il dominio astratto $D^\sharp$, come specificato in precedenza, sarà indicato con $D_1^\sharp \uplus^\sharp D_2^\sharp$.

4.3 Approssimazione degli elementi di CMM

Approssimazione dei tipi aritmetici

Assumiamo che per ognuno dei tipi aritmetici definiti nella sezione 3.1 esista il corrispondente dominio concreto. Per gli interi illimitati avremo quindi il dominio $\big(\wp(\text{Integers}), \subseteq, \varnothing, \text{Integers}, \cap, \cup\big)$ che sarà associato al corrispondente dominio astratto $\big(\text{Integers}^\sharp, \sqsubseteq, \bot, \top, \sqcup\big)$ tramite la funzione γ, che si suppone essere stretta. Per gli interi limitati con segno e, rispettivamente, senza segno, avremo, al variare di $k \in \mathbb{N} \setminus \{0\}$, i domini $\big(\wp(\text{SInt}_\text{k}), \subseteq, \varnothing, \text{SInt}_\text{k}, \cap, \cup\big)$ e $\big(\wp(\text{UInt}_\text{k}), \subseteq, \varnothing, \text{UInt}_\text{k}, \cap, \cup\big)$, che si possono approssimare correttamente con i corrispondenti domini astratti $\big(\text{SInt}_\text{k}{}^\sharp, \sqsubseteq, \bot, \top, \sqcup\big)$ e $\big(\text{UInt}_\text{k}{}^\sharp, \sqsubseteq, \bot, \top, \sqcup\big)$. Anche in questi casi si suppone l'esistenza della funzione γ tra le coppie dominio concreto e dominio astratto introdotti in precedenza. Gli elementi di $\text{Integers}^\sharp$ sono indicati con i simboli $m^{\sharp\mathsf{i}\infty}$, $m_0^{\sharp\mathsf{i}\infty}$, $m_1^{\sharp\mathsf{i}\infty}$, ... mentre gli elementi di $\text{SInt}_\text{k}{}^\sharp$ (risp., $\text{UInt}_\text{k}{}^\sharp$) sono indicati con i simboli $m^{\sharp\mathsf{s}_\mathsf{k}}$, $m_0^{\sharp\mathsf{s}_\mathsf{k}}$, $m_1^{\sharp\mathsf{s}_\mathsf{k}}$, ... (risp., $m^{\sharp\mathsf{u}_\mathsf{k}}$, $m_0^{\sharp\mathsf{u}_\mathsf{k}}$, $m_1^{\sharp\mathsf{s}_\mathsf{k}}$ ...). Assumiamo che le funzioni γ siano strette e che le funzioni parziali $\alpha\colon \wp(\text{Integers}) \rightarrowtail \text{Integers}^\sharp$, $\alpha\colon \wp(\text{SInt}_\text{k}) \rightarrowtail \text{SInt}_\text{k}{}^\sharp$, $\alpha\colon \wp(\text{UInt}_\text{k}) \rightarrowtail \text{UInt}_\text{k}{}^\sharp$, siano tutte definite su tutti i singoletti dei corrispondenti domini.

Assumiamo che esistano le operazioni binarie astratte '$\oplus$', '$\ominus$', '$\odot$', '$\oslash$', '$\oplus$', '$\oslash$', '$\otimes$', '$\otimes$', '$\otimes$' definite sui domini astratti $\text{Integers}^\sharp$, $\text{SInt}_\text{k}{}^\sharp$ e

$\text{UInt}_k{}^\sharp$. Si richiede che tali operazioni siano strette su entrambi gli argomenti e che siano corrette nel rispetto delle corrispondenti operazioni sui domini $\wp(\text{Integers})$, $\wp(\text{SInt}_k)$ e $\wp(\text{UInt}_k)$ che, rispettivamente, sono le ovvie estensioni puntuali dell'addizione, sottrazione, moltiplicazione, divisione intera, modulo, congiunzione bit-a-bit, disgiunzione inclusiva ed esclusiva bit-a-bit e scorrimento sinistro e destro sui diversi tipi di interi. Consideriamo per esempio l'operatore di somma. Richiediamo che

$$\gamma(m_0^{\sharp\mathsf{t}} \oplus m_1^{\sharp\mathsf{t}}) \supseteq \left\{ m_0^{\mathsf{t}} + m_1^{\mathsf{t}} \;.\; m_0^{\mathsf{t}} \in \gamma(m_0^{\sharp\mathsf{t}}), m_1^{\mathsf{t}} \in \gamma(m_1^{\sharp\mathsf{t}}) \right\}$$

per ogni $m_0^{\sharp\mathsf{t}}$ e $m_1^{\sharp\mathsf{t}}$ entrambi appartenenti allo stesso dominio astratto $\text{Integers}^\sharp$, $\text{SInt}_k{}^\sharp$ o $\text{UInt}_k{}^\sharp$. Lo stesso deve valere per '$\ominus$', '$\odot$', '$\oslash$', '$\otimes$' e '$\oplus$' nel rispetto dell'implementazione scelta per le corrispondenti operazioni concrete. Per la divisione intera astratta '$\oslash$' richiediamo che per ogni $m_0^{\sharp\mathsf{t}}$ e $m_1^{\sharp\mathsf{t}}$, entrambi nello stesso dominio intero astratto, valga

$$\gamma(m_0^{\sharp\mathsf{t}} \oslash m_1^{\sharp\mathsf{t}}) \supseteq \left\{ m_0^{\mathsf{t}} \div m_1^{\mathsf{t}} \;.\; m_0^{\mathsf{t}} \in \gamma(m_0^{\sharp\mathsf{t}}), m_1^{\mathsf{t}} \in \gamma(m_1^{\sharp\mathsf{t}}), \text{e } m_0^{\mathsf{t}} \div m_1^{\mathsf{t}} \text{ è definita} \right\}.$$

Lo stesso valga per la operazione astratta '$\oplus$' nel rispetto dell'implementazione dell'operazione operazione di modulo 'mod'. Per lo scorrimento di bit a destra '$\oslash$', richiediamo la correttezza nei confronti dell'operatore concreto $\gg$. Deve infatti valere che

$$\gamma(m_0^{\sharp\mathsf{t}} \oslash m_1^{\sharp\mathsf{t}}) \supseteq \left\{ m_0^{\mathsf{t}} \gg m_1^{\mathsf{t}} \;\middle|\; \begin{array}{l} m_0^{\mathsf{t}} \in \gamma(m_0^{\sharp\mathsf{t}}), m_1^{\mathsf{t}} \in \gamma(m_1^{\sharp\mathsf{t}}), \\ m_1^{\mathsf{t}} \geq 0, \\ \mathsf{t} \in \{\mathsf{s}_k, \mathsf{u}_k\} \text{ e } m_1^{\mathsf{t}} < k, \\ \text{type}(m_0^{\mathsf{t}}) \in \text{Signed e } m_0^{\mathsf{t}} \geq 0 \end{array} \right\}.$$

Lo stesso valga per la operazione astratta '$\oslash$', nel rispetto dell'operazione di scorrimento a destra '$\ll$'. Assumiamo anche che vi sia l'operazione astratta unaria e l'operazione astratta di complemento a uno, indicate rispettivamente con '$\ominus$' e '$\oslash$'. Per '$\ominus$' richiediamo che sia stretta sull'operando e definita in modo corretto nel rispetto della corrispondente operazione concreta, ovvero

$$\gamma(\ominus m^{\sharp\mathsf{t}}) \supseteq \left\{ -m \;.\; m \in \gamma(m^{\sharp\mathsf{t}}) \right\}.$$

Lo stesso valga per l'operazione di complemento a uno astratta '$\oslash$'.

Approssimazione dei booleani

Assumiamo l'esistenza del semi-reticolo completo $\left(\text{Bool}^\sharp, \sqsubseteq, \bot, \top, \sqcap, \sqcup\right)$, legato al dominio concreto dei booleani $\left(\wp(\text{Bool}), \subseteq, \varnothing, \text{Bool}, \cap, \cup\right)$ attraverso

una connessione di Galois. Gli elementi di $\text{Bool}^\sharp$ sono indicati con $t^{\sharp\mathsf{b}}$, $t_0^{\sharp\mathsf{b}}$, $t_1^{\sharp\mathsf{b}}$, ... Assumiamo che ci siano operazioni astratte '$\ominus$', '$\triangledown$' e '$\triangle$' su $\text{Bool}^\sharp$ che siano strette su tutti gli argomenti e corrette rispetto alla estensione puntuale della negazione, congiunzione e disgiunzione booleana su $\wp(\text{Bool})$. Per esempio, per L'operazione '$\triangledown$', la correttezza rispetto alla disgiunzione su $\wp(\text{Bool})$, sarà richiesto che,

$$\gamma(t_0^{\sharp\mathsf{b}} \triangledown t_1^{\sharp\mathsf{b}}) \supseteq \left\{ m_0^{\mathsf{b}} \textbf{ or } m_1^{\mathsf{b}} \,.\, m_0^{\mathsf{b}} \in \gamma(t_0^{\sharp\mathsf{b}}), m_1^{\mathsf{b}} \in \gamma(t_1^{\sharp\mathsf{b}}) \right\}$$

per ogni $t_0^{\sharp\mathsf{b}}$ e $t_1^{\sharp\mathsf{b}}$ in $\text{Bool}^\sharp$. Lo stesso varrà per '$\triangle$'. Per l'operazione '$\ominus$' la correttezza rispetto alla negazione su $\wp(\text{Bool})$ richiede che, per ogni $t^{\sharp\mathsf{b}}$ in $\text{Bool}^\sharp$,

$$\gamma(\ominus\, t^{\sharp\mathsf{b}}) \supseteq \left\{ \neg\, t \,.\, t \in \gamma(t^{\sharp\mathsf{b}}) \right\}.$$

In aggiunta, assumiamo che ci siano operazioni astratte '$\triangleq$', '$\not\triangleq$', '$\triangleleft$', '$\trianglelefteq$', '$\trianglerighteq$' e '$\triangleright$' su $\text{Int}_k^{\sharp\mathsf{b}}$ che siano strette sui corrispondenti argomenti e corrette nel rispetto delle estensione puntuali su $\wp(\text{Int})$ dei corrispondenti operatori relazionali '$=$', '$\neq$', '$<$', '$\leq$', '$\geq$' e '$>$' sugli interi, considerati come funzioni con valori in Bool. Per esempio, per l'operazione '$\triangleq$', la correttezza rispetto all'uguaglianza su $\wp(\text{Int})$, richiediamo che

$$\gamma(m_0^{\sharp\mathsf{t}} \triangleq m_1^{\sharp\mathsf{t}}) \supseteq \left\{ m_0^{\mathsf{t}} = m_1^{\mathsf{t}} \,.\, m_0^{\mathsf{t}} \in \gamma(m_0^{\sharp\mathsf{t}}), m_1^{\mathsf{t}} \in \gamma(m_1^{\sharp\mathsf{t}}) \right\}$$

per ogni $m_0^{\sharp\mathsf{t}}$ e $m_1^{\sharp\mathsf{t}}$ entrambi nello stesso dominio di interi astratti.. Lo stesso valga per '$\not\triangleq$', '$\triangleleft$', '$\trianglelefteq$', '$\trianglerighteq$' e '$\triangleright$'.

Approssimazione delle costanti

Il dominio concreto delle costanti $\big(\wp(\text{Con}), \subseteq, \varnothing, \text{Con}, \cap, \cup\big)$ viene astratto dal dominio

$$\text{Con}^\sharp \stackrel{\text{def}}{=} \text{Integers}^\sharp \uplus \{\text{SInt}_\text{k}{}^\sharp \,.\, \mathsf{k} \in \mathsf{K}\} \uplus \{\text{UInt}_\text{k}{}^\sharp \,.\, \mathsf{k} \in \mathsf{K}\} \uplus \text{Bool}^\sharp.$$

Gli elementi di $\text{Con}^\sharp$ saranno indicati con $\text{con}^\sharp$, $\text{con}_0^\sharp$, $\text{con}_1^\sharp$, ... Le ipotesi su $\text{Integers}^\sharp$, $\text{SInt}_\text{k}{}^\sharp$, $\text{UInt}_\text{k}{}^\sharp$ e $\text{Bool}^\sharp$ implicano che l'approssimazione è stretta.

Approssimazione delle eccezioni

Per l'approssimazione delle *run-time exceptions*, assumiamo che ci sia il dominio astratto

$$\big(\text{RTSExcept}^\sharp, \sqsubseteq, \bot, \top, \sqcup\big),$$

legato al dominio concreto

$$\big(\wp(\text{RTSExcept}), \subseteq, \varnothing, \text{RTSExcept}, \cap, \cup\big)$$

da una funzione di concretizzazione. Si richiede inoltre che la funzione parziale di astrazione $\alpha\colon \wp(\text{RTSExcept}) \rightarrowtail \text{RTSExcept}^\sharp$ sia definita su tutti i singoletti. Gli elementi di $\text{RTSExcept}^\sharp$ sono indicati con $\chi^\sharp$, $\chi_0^\sharp$, $\chi_1^\sharp$, ...

Approssimazione delle strutture di memoria

Assumiamo che esista un dominio astratto

$$\big(\text{Mem}^\sharp, \sqsubseteq, \bot, \top, \sqcup\big)$$

che sia collegato tramite una funzione di concretizzazione al dominio concreto

$$\big(\wp(\text{Mem}), \subseteq, \varnothing, \text{Mem}, \cap, \cup\big).$$

Gli elementi di $\text{Mem}^\sharp$ sono indicati con $\sigma^\sharp$, $\sigma_0^\sharp$, $\sigma_1^\sharp$, ... Assumiamo che, per ogni $\sigma \in \text{Mem}$, esista $\sigma^\sharp \in \text{Mem}^\sharp$ tale che $\sigma \propto \sigma^\sharp$.

Approssimazione degli stati con valore e degli stati con eccezione

Il dominio astratto degli stati con valore è

$$\text{ValState}^\sharp \stackrel{\text{def}}{=} \text{Con}^\sharp \otimes \text{Mem}^\sharp.$$

Gli elementi di ValState saranno indicati con $\upsilon^\sharp$, $\upsilon_0^\sharp$, $\upsilon_1^\sharp$, ...

Il dominio astratto degli stati con eccezione è

$$\text{ExceptState}^\sharp \stackrel{\text{def}}{=} \text{Mem}^\sharp \otimes \text{RTSExcept}^\sharp.$$

Gli elementi di $\text{ExceptState}^\sharp$ saranno indicati con $\varepsilon^\sharp$, $\varepsilon_0^\sharp$, $\varepsilon_1^\sharp$, ...

Per migliorare la leggibilità, $\texttt{none}^\sharp$ indicherà l'elemento $\bot \in \text{ExceptState}^\sharp$, per indicare che nessuno stato con eccezione è possibile.

Approssimazione delle operazioni sulla memoria

Gli operatori astratti di lettura e aggiornamento della memoria

$$\begin{aligned} \cdot[\cdot,\cdot]\colon (\text{Mem}^\sharp \times \text{Loc} \times \text{Type}) &\to (\text{ValState}^\sharp \uplus^\sharp \text{ExceptState}^\sharp), \\ \cdot[\cdot :=^\sharp \cdot]\colon \big(\text{Mem}^\sharp \times (\text{Loc} \times \text{Type}) \times \text{Con}^\sharp\big) &\to (\text{Mem}^\sharp \uplus^\sharp \text{ExceptState}^\sharp) \end{aligned}$$

sono assunti essere tali che, per ogni $\sigma^\sharp \in \text{Mem}^\sharp$, $l \in \text{Loc}$, $T \in \text{Type}$ e $\text{con}^\sharp \in \text{Con}^\sharp$:

$$\begin{aligned} \gamma\big(\sigma^\sharp[l,T]\big) &\supseteq \big\{\, \sigma[l,T] \;.\; \sigma \in \gamma(\sigma^\sharp) \,\big\}, \\ \gamma\big(\sigma^\sharp\big[(l,T) :=^\sharp \text{con}^\sharp\big]\big) &\supseteq \big\{\, \sigma\big[(l,T) := \text{con}\big] \;.\; \sigma \in \gamma(\sigma^\sharp), \text{con} \in \gamma(\text{con}^\sharp) \,\big\}. \end{aligned}$$

La funzione astratta di allocazione dati

$$\text{new}^\sharp\colon \text{ValState}^\sharp \to \big((\text{Mem}^\sharp \times \text{Loc}) \uplus^\sharp \text{ExceptState}^\sharp\big)$$

è assunta essere tale che, per ogni $\upsilon^\sharp = (\text{con}^\sharp, \sigma^\sharp) \in \text{ValState}^\sharp$:

$$\gamma\big(\text{new}^\sharp(\upsilon^\sharp)\big) \supseteq \big\{\, \text{new}_\text{d}(\upsilon) \;.\; \upsilon \in \gamma(\upsilon^\sharp) \,\big\},$$

con la precisazione che, per ogni $\sigma \in \text{Mem}$, $\sigma^\sharp \in \text{Mem}^\sharp$, $\{l, l'\} \subseteq \text{Loc}$, valga la seguente proprietà: $(\sigma, l) \in \gamma(\sigma^\sharp, l') \implies l = l'$.

Le funzioni astratte

$$\{\text{mark}^\sharp, \text{unmark}^\sharp\} \subseteq \text{Mem}^\sharp \to \text{Mem}^\sharp$$

sono definite essere tali che, per ogni $\sigma^\sharp \in \text{Mem}^\sharp$:

$$\begin{aligned}\gamma\big(\text{mark}^\sharp(\sigma^\sharp)\big) &\supseteq \big\{\, \text{mark}(\sigma) \;.\; \sigma \in \gamma(\sigma^\sharp) \,\big\},\\ \gamma\big(\text{unmark}^\sharp(\sigma^\sharp)\big) &\supseteq \big\{\, \text{unmark}(\sigma) \;.\; \sigma \in \gamma(\sigma^\sharp) \text{ e } \text{unmark}(\sigma) \text{ è definita} \,\big\}.\end{aligned}$$

Si assume che tutti gli operatori astratti definiti in precedenza siano stretti su tutti i loro argomenti presi da un dominio astratto. Come per quanto definito nel concreto, la funzione astratte $\text{unmark}^\sharp$ è definita anche sugli stati con eccezione. Infatti, per ogni $\varepsilon^\sharp = (\sigma^\sharp, \xi^\sharp) \in \text{ExceptState}^\sharp$,

$$\text{unmark}^\sharp(\sigma^\sharp, \xi^\sharp) \stackrel{\text{def}}{=} \big(\text{unmark}^\sharp(\sigma^\sharp), \xi^\sharp\big).$$

4.3.1 Filtri astratti

Oltre agli operatori astratti definiti in precedenza, saranno definiti altri operatori per la costruzione della semantica astratta in modo da migliorarne la precisione.

Nel momento in cui bisogna gestire le guardie booleane durante l'interpretazione astratta degli statement condizionali e iterativi, può accadere che non siano disponibili sufficienti informazioni. In tali situazioni, l'interpretazione può essere più precisa se la struttura di memoria astratta viene *filtrata* in base alle condizioni che valgono nel ramo di computazione da analizzare.

Definizione 4.3.1 (Filtro) *Un* filtro per la struttura di memoria astratta *è una funzione computabile* $\phi\colon (\text{Env} \times \text{Mem}^\sharp \times \text{Exp}) \to \text{Mem}^\sharp$ *tale che, per ogni* $e \in \text{Exp}$*, ogni* $\beta : I$ *con* $\text{FI}(e) \subseteq I$ *e* $\beta \vdash_I e : \text{boolean}$*, per ogni* $\rho \in \text{Env}$ *con* $\rho : \beta$ *e ogni* $\sigma^\sharp \in \text{Mem}^\sharp$*, se* $\phi(\rho, \sigma^\sharp, e) = \sigma^\sharp_{\text{tt}}$*, allora*

$$\gamma(\sigma^\sharp_{\text{tt}}) \supseteq \big\{\, \sigma_{\text{tt}} \in \text{Mem} \;.\; \sigma \in \gamma(\sigma^\sharp), \rho \vdash_\beta \langle e, \sigma\rangle \to \langle \text{tt}, \sigma_{\text{tt}}\rangle \,\big\}.$$

4.3.2 Cast astratto

La funzione che si occupa di eseguire i cast tra costanti astratte è

$$\text{cast}^\sharp \colon \text{Type} \times \text{Con}^\sharp \to \text{Con}^\sharp \uplus^\sharp \text{RTSExcept}^\sharp$$

Tale funzione deve soddisfare la seguente condizione

$$\gamma\big(\text{cast}^\sharp(T, \text{con}^\sharp)\big) \supseteq \big\{\, \text{cast}(T, \text{con}) \;.\; \text{con} \in \gamma(\text{con}^\sharp) \,\big\},$$

4.3.3 Configurazioni astratte

Definiremo ora le configurazioni terminali e non-terminali del sistema di transizione astratto.

Definizione 4.3.2 (Configurazioni astratte non-terminali) *Gli insiemi delle* configurazioni astratte non-terminali *per espressioni, dichiarazioni e statement, sono date, per ogni $\beta \in \text{TEnv}_I$ rispettivamente da*

$$\begin{aligned}
\Gamma_{\text{e}}^{\beta\sharp} &\stackrel{\text{def}}{=} \big\{\, \langle e, \sigma^\sharp\rangle \in \text{Exp} \times \text{Mem}^\sharp \;.\; \exists T \in \text{Type} \;.\; \beta \vdash_I e : T \,\big\},\\
\Gamma_{\text{d}}^{\beta\sharp} &\stackrel{\text{def}}{=} \big\{\, \langle d, \sigma^\sharp\rangle \in \text{Decl} \times \text{Mem}^\sharp \;.\; \exists \delta \in \text{TEnv} \;.\; \beta \vdash_I d : \delta \,\big\},\\
\Gamma_{\text{s}}^{\beta\sharp} &\stackrel{\text{def}}{=} \big\{\, \langle s, \sigma^\sharp\rangle \in \text{Stmt} \times \text{Mem}^\sharp \;.\; \beta \vdash_I s \,\big\}.
\end{aligned}$$

Scriveremo $N^\sharp$ per indicare una configurazione astratta non terminale.

La relazione di approssimazione tra configurazioni non-terminali di domini concreti e astratti è definita come segue: per ogni $q \in \{\text{e}, \text{d}, \text{s},\}$, $N = \langle q_1, \sigma\rangle \in \Gamma_q^\beta$ e $N^\sharp = \langle q_2, \sigma^\sharp\rangle \in \Gamma_q^{\beta\sharp}$,

$$N \propto N^\sharp \iff (q_1 = q_2 \;\; e \;\; \sigma \propto \sigma^\sharp). \tag{4.2}$$

Definizione 4.3.3 (Configurazioni astratte terminali) *Gli insiemi delle* configurazioni astratte terminali *per espressioni, dichiarazioni e statement, sono date rispettivamente da*

$$\begin{aligned}
T_{\text{e}}^\sharp &\stackrel{\text{def}}{=} \text{ValState}^\sharp \uplus^\sharp \text{ExceptState}^\sharp,\\
T_{\text{d}}^\sharp &\stackrel{\text{def}}{=} (\text{Env} \times \text{Mem}^\sharp) \uplus^\sharp \text{ExceptState}^\sharp,\\
T_{\text{s}}^\sharp &\stackrel{\text{def}}{=} \text{Mem}^\sharp \uplus^\sharp \text{ExceptState}^\sharp.
\end{aligned}$$

Scriveremo $\eta^\sharp$ per indicare una configurazione astratta terminale.

La relazione di approssimazione $\eta \propto \eta^\sharp$ tra configurazioni terminali di domini concreti e astratti è definita come segue. Per le espressioni,

$$\eta \propto \langle \upsilon^\sharp, \varepsilon^\sharp \rangle \iff \begin{cases} \upsilon \propto \upsilon^\sharp, & \text{se } \eta = \upsilon; \\ \varepsilon \propto \varepsilon^\sharp, & \text{se } \eta = \varepsilon. \end{cases} \tag{4.3}$$

Per le dichiarazioni,

$$\eta \propto \big\langle (\rho_2, \sigma^\sharp), \varepsilon^\sharp \big\rangle \iff \begin{cases} (\rho_1 = \rho_2 \ e \ \sigma \propto \sigma^\sharp), & \text{se } \eta = \langle \rho_1, \sigma \rangle; \\ \varepsilon \propto \varepsilon^\sharp, & \text{se } \eta = \varepsilon. \end{cases} \tag{4.4}$$

Per gli statement,

$$\eta \propto \langle \sigma^\sharp, \varepsilon^\sharp \rangle \iff \begin{cases} \sigma \propto \sigma^\sharp, & \text{se } \eta = \sigma; \\ \varepsilon \propto \varepsilon^\sharp, & \text{se } \eta = \varepsilon. \end{cases} \tag{4.5}$$

La relazione di approssimazione per i conseguenti è facilmente ottenibile dalle relazioni di approssimazione definite per le configurazioni.

Definizione 4.3.4 ('$\propto$' sui conseguenti) *La relazione di approssimazione tra conseguenti concreti e astratti è definita, per ogni $\beta \in \mathrm{TEnv}_I$ e $\rho \in \mathrm{Env}_J$ tali che $\rho : \beta\,|_J$, ogni $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{s}, \}$, $N \in \Gamma_q^\beta$, $\eta \in T_q$, $N^\sharp \in \Gamma_q^{\beta\sharp}$ e $\eta^\sharp \in T_q^\sharp$, da*

$$(\rho \vdash_\beta N \to \eta) \propto (\rho \vdash_\beta N^\sharp \to \eta^\sharp) \iff (N \propto N^\sharp \ e \ \eta \propto \eta^\sharp). \tag{4.6}$$

4.3.4 Espressioni, dichiarazioni e statement supportati

Ogni dominio astratto deve fornire una relazione che dica quale configurazione astratta per espressioni, dichiarazioni e statement esso supporta direttamente; allo stesso modo, deve anche fornire una seconda funzione di valutazione astratta che fornisca una approssimazione corretta di ogni espressione, dichiarazione e statement.

Definizione 4.3.5 (supported**,** eval**)** *Per ogni $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{s}\}$, assumiamo l'esistenza di una relazione e di una funzione parziale entrambe computabili*

$$\mathrm{supported} \subseteq \mathrm{Env} \times \Gamma_q^{\beta\sharp} \qquad e \qquad \mathrm{eval}\colon (\mathrm{Env} \times \Gamma_q^{\beta\sharp}) \rightarrowtail T_q^\sharp,$$

tali che:

1. *ogni qual volta si verifichi che $\rho : \beta$ e $\mathrm{supported}(\rho, N^\sharp)$ vale, $\mathrm{eval}(\rho, N^\sharp)$ è definita ed ha valore $\eta^\sharp \in T_q^\sharp$;*
2. *per ogni $N \in \Gamma_q^\beta$ e ogni $\eta \in T_q$ tali che $N \propto N^\sharp$ e $\rho \vdash_\beta N \to \eta$, abbiamo $\eta \propto \eta^\sharp$;*
3. *se $\rho : I$, $\mathrm{id} \in I$ e $N^\sharp = \langle \mathrm{id}, \sigma^\sharp \rangle \in \Gamma_{\mathrm{e}}^{\beta\sharp}$, allora $\mathrm{supported}(\rho, N^\sharp)$ vale.*

4.3.5 Relazioni di valutazione astratta

Le relazioni di valutazione astratta sono della forma

$$\rho \vdash_\beta N^\sharp \to \eta^\sharp,$$

dove $\beta \in \mathrm{TEnv}$, $\rho : \beta$ e, per qualche $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{s}\}$, $N^\sharp \in \Gamma_q^{\beta\sharp}$ e $\eta^\sharp \in T_q^\sharp$. La definizione è ancora per induzione strutturale, basata su un insieme di schemi di regole.

Come per l'abbreviazione introdotta nella Sezione 4.2.2, quando ci aspetterebbe una costante astratta $\mathrm{con}^\sharp$ ma scriviamo uno degli interi astratti $m^{\sharp\mathsf{i}_\infty}$, $m^{\sharp\mathsf{s}_\mathsf{k}}$ o $m^{\sharp\mathsf{u}_\mathsf{k}}$ o un booleano astratto $m^{\sharp\mathsf{b}}$, allora intendiamo indicare, rispettivamente, la costante astratta $(m^{\sharp\mathsf{i}_\infty}, \bot, \bot, \bot)$, $(\bot, m^{\sharp\mathsf{s}_\mathsf{k}}, \bot, \bot)$, $(\bot, \bot, m^{\sharp\mathsf{u}_\mathsf{k}}, \bot)$ o $(\bot, \bot, \bot, \bot, m^{\sharp\mathsf{b}})$; in modo simile, quando ci si aspetta uno stato con valore $v^\sharp$ o uno stato con eccezione $\varepsilon^\sharp$ e scriviamo solamente il valore $\mathrm{con}^\sharp$ o l'eccezione $\chi^\sharp$, allora si intende esprimere, rispettivamente, l'elemento astratto $(\bot, \mathrm{con}^\sharp)$ o $(\chi^\sharp, \bot)$.

Espressioni non supportate

Le seguenti regole per la valutazione astratta di espressioni, si applicano solamente se $\mathrm{supported}\big(\rho, \langle e, \sigma^\sharp\rangle\big)$ non vale, dove e è l'espressione che si sta valutando. Tale condizione sarà lasciata implicita in modo da non sovraccaricare la presentazione delle regole.

Costante

$$\frac{}{\rho \vdash_\beta \langle \mathrm{con}, \sigma^\sharp\rangle \rightsquigarrow \big\langle \alpha(\{\mathrm{con}\}) \otimes \sigma^\sharp, \mathtt{none}^\sharp\big\rangle} \tag{4.7}$$

Identificatore

$$\frac{}{\rho \vdash_\beta \langle \mathrm{id}, \sigma^\sharp\rangle \rightsquigarrow \sigma^\sharp\big[\rho(\mathrm{id})\big]} \tag{4.8}$$

Cast esplicito

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \to \big\langle (\mathrm{con}^\sharp, \sigma_0^\sharp), \varepsilon^\sharp\big\rangle}{\rho \vdash_\beta \langle (T)\, e, \sigma^\sharp\rangle \rightsquigarrow \big\langle \mathrm{con}_0^\sharp \otimes \sigma_0^\sharp, \varepsilon^\sharp \sqcup (\sigma_0^\sharp \otimes \chi^\sharp))\big\rangle} \tag{4.9}$$

se $\mathrm{cast}_\chi^\sharp(T, \mathrm{con}_0^\sharp) = (\mathrm{con}_0^\sharp, \chi^\sharp)$.

Operazioni aritmetiche

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \to \left\langle (\mathrm{con}^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp\right\rangle}{\rho \vdash_\beta \langle -e, \sigma^\sharp\rangle \rightsquigarrow \left\langle (\mathrm{con}_0^\sharp \otimes \sigma_0^\sharp), \varepsilon_0^\sharp \sqcup (\sigma_0^\sharp \otimes \chi^\sharp)\right\rangle} \quad \text{se } \ominus \mathrm{con}^\sharp = (\mathrm{con}_0^\sharp, \chi^\sharp) \tag{4.10}$$

Consideriamo quindi le rimanenti operazioni aritmetiche. Sia $\boxdot \in \{+, -, *, /, \%\}$ un operatore sintattico e $\circledcirc \in \{\oplus, \ominus, \odot, \oslash, \oplus\}$ la corrispondente operazione astratta. Allora le regole astratte per l'addizione, sottrazione, moltiplicazione, divisione e modulo sono date dallo schema:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma^\sharp\rangle \to \left\langle (\mathrm{con}_0^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp\right\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0^\sharp\rangle \to \left\langle (\mathrm{con}_1^\sharp, \sigma_1^\sharp), \varepsilon_1^\sharp\right\rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma^\sharp\rangle \rightsquigarrow \left\langle (\mathrm{con}_2^\sharp \otimes \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup (\sigma_1^\sharp \otimes \chi^\sharp)\right\rangle} \tag{4.11}$$

se $\mathrm{con}_0^\sharp \circledcirc \mathrm{con}_1^\sharp = (\mathrm{con}_2^\sharp, \chi^\sharp)$.

Operazioni *bitwise*

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \to \left\langle (\mathrm{con}^\sharp, \sigma_0^\sharp), \varepsilon^\sharp\right\rangle}{\rho \vdash_\beta \langle \sim e, \sigma^\sharp\rangle \rightsquigarrow \left\langle (\oslash \mathrm{con}^\sharp, \sigma_0^\sharp), \varepsilon^\sharp\right\rangle} \tag{4.12}$$

Sia $\boxdot \in \{\wedge, \vee, \veebar\}$ un operatore sintattico e $\circledcirc \in \{\owedge, \owedge, \otimes\}$ la corrispondente operazione astratta. Allora le regole astratte per le operazioni di congiunzione bit-a-bit, disgiunzione inclusiva e esclusiva bit-a-bit sono date dal seguente schema:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma^\sharp\rangle \to \left\langle (\mathrm{con}_0^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp\right\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0^\sharp\rangle \to \left\langle (\mathrm{con}_1^\sharp, \sigma_1^\sharp), \varepsilon_1^\sharp\right\rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma^\sharp\rangle \rightsquigarrow \left\langle (\mathrm{con}_0^\sharp \circledcirc \mathrm{con}_1^\sharp, \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\right\rangle} \tag{4.13}$$

Sia $\boxdot \in \{\ll, \gg\}$ un operatore sintattico e $\circledcirc \in \{\circledless, \circledgtr\}$ la corrispondente operazione astratta. Per le operazioni di scorrimento di bit occorre considerare le seguenti regole:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma^\sharp\rangle \to \left\langle (\mathrm{con}_0^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp\right\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0^\sharp\rangle \to \left\langle (\mathrm{con}_1^\sharp, \sigma_1^\sharp), \varepsilon_1^\sharp\right\rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma^\sharp\rangle \rightsquigarrow \left\langle (\mathrm{con}_2^\sharp \otimes \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup (\sigma_1^\sharp \otimes \chi^\sharp)\right\rangle} \tag{4.14}$$

se $\mathrm{con}_0^\sharp \circledcirc \mathrm{con}_1^\sharp = (\mathrm{con}_2^\sharp, \chi^\sharp)$.

TEST ARITMETICI Sia $\circledast \in \{=, \neq, <, \leq, \geq, >\}$ un operatore sintattico astratto e $\bowtie$: $(\mathrm{Int}^\sharp \times \mathrm{Int}^\sharp) \to \mathrm{Bool}^\sharp$ la corrispondente operazione di test astratto in $\{\mathrel{\hat{=}}, \mathrel{\hat{\neq}}, \lhd, \trianglelefteq, \rhd, \trianglerighteq\}$. Allora le regole per i test aritmetici sono date da

$$\frac{\rho \vdash_\beta \langle e_0, \sigma^\sharp\rangle \to \big\langle (\mathrm{con}_0^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp\big\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0^\sharp\rangle \to \big\langle (\mathrm{con}_1^\sharp, \sigma_1^\sharp), \varepsilon_1^\sharp\big\rangle}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\mathrm{con}_0^\sharp \bowtie \mathrm{con}_1^\sharp, \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\big\rangle} \tag{4.15}$$

ESPRESSIONI BOOLEANE

$$\frac{\rho \vdash_\beta \langle b, \sigma^\sharp\rangle \to \big\langle (t^\sharp, \sigma_0^\sharp), \varepsilon^\sharp\big\rangle}{\rho \vdash_\beta \langle \mathbf{not}\ \ b, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\ominus\, t^\sharp, \sigma_0^\sharp), \varepsilon^\sharp\big\rangle} \tag{4.16}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma^\sharp\rangle \to \langle \upsilon_0^\sharp, \varepsilon_0^\sharp\rangle \quad \rho \vdash_\beta \langle b_1, \sigma_{\mathrm{tt}}^\sharp\rangle \to \langle \upsilon_1^\sharp, \varepsilon_1^\sharp\rangle}{\rho \vdash_\beta \langle b_0 \,\mathbf{and}\, b_1, \sigma^\sharp\rangle \rightsquigarrow \big\langle \upsilon_{\mathrm{ff}}^\sharp \sqcup \upsilon_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\big\rangle}, \tag{4.17}$$

se $\sigma_{\mathrm{tt}}^\sharp = \phi(\rho, \sigma^\sharp, b_0)$, $\sigma_{\mathrm{ff}}^\sharp = \phi(\rho, \sigma^\sharp, \mathbf{not}\, b_0)$ e $\upsilon_{\mathrm{ff}}^\sharp = \alpha(\{\mathrm{ff}\}) \otimes \sigma_{\mathrm{ff}}^\sharp$.

$$\frac{\rho \vdash_\beta \langle b_0, \sigma^\sharp\rangle \to \langle \upsilon_0^\sharp, \varepsilon_0^\sharp\rangle \quad \rho \vdash_\beta \langle b_1, \sigma_{\mathrm{ff}}^\sharp\rangle \to \langle \upsilon_1^\sharp, \varepsilon_1^\sharp\rangle}{\rho \vdash_\beta \langle b_0 \,\mathbf{or}\, b_1, \sigma^\sharp\rangle \rightsquigarrow \langle \upsilon_{\mathrm{tt}}^\sharp \sqcup \upsilon_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\rangle} \tag{4.18}$$

se $\sigma_{\mathrm{tt}}^\sharp = \phi(\rho, \sigma^\sharp, b_0)$, $\sigma_{\mathrm{ff}}^\sharp = \phi(\rho, \sigma^\sharp, \mathbf{not}\, b_0)$ e $\upsilon_{\mathrm{tt}}^\sharp = \alpha(\{\mathrm{tt}\}) \otimes \sigma_{\mathrm{tt}}^\sharp$.

Dichiarazioni non supportate

Le seguenti regole valgono solo se la condizione $\mathrm{supported}\big(\rho, \langle d, \sigma^\sharp\rangle\big)$ non vale, dove $d \in \mathrm{Decl}$ è la dichiarazione che si sta valutando. Anche in questo caso tale condizione è lasciata implicita.

NIL

$$\frac{}{\rho \vdash_\beta \langle \mathbf{nil}, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\varnothing, \sigma^\sharp), \mathtt{none}^\sharp\big\rangle} \tag{4.19}$$

DICHIARAZIONE DI VARIABILE

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \to \langle \upsilon^\sharp, \varepsilon_0^\sharp\rangle}{\rho \vdash_\beta \langle \mathbf{lvar}\,\mathrm{id} : T = e, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\rho_1, \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\big\rangle} \tag{4.20}$$

se $\mathrm{new}_{\mathrm{s}}{}^\sharp(\upsilon^\sharp) = \big((\sigma_1^\sharp, i), \varepsilon_1^\sharp\big)$ e $\rho_1 = \big\{\mathrm{id} \mapsto (i, T)\big\}$.

Concatenazione di dichiarazioni

$$\frac{\rho \vdash_\beta \langle d_0, \sigma^\sharp\rangle \to \big\langle(\rho_0, \sigma_0^\sharp), \varepsilon_0^\sharp\big\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle d_1, \sigma_0^\sharp\rangle \to \big\langle(\rho_1, \sigma_1^\sharp), \varepsilon_1^\sharp\big\rangle}{\rho \vdash_\beta \langle d_0; d_1, \sigma^\sharp\rangle \rightsquigarrow \big\langle(\rho_0[\rho_1], \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup_{\mathrm{s}} \varepsilon_1^\sharp\big\rangle} \tag{4.21}$$

se $\beta \vdash_I d_0 : \beta_0$ e $\mathrm{FI}(d_0) \subseteq I$.

Statement non supportati

Le seguenti regole valgono solo se l'implicita condizione $\mathrm{supported}\big(\rho, \langle s, \sigma^\sharp\rangle\big)$ non vale, quando s è lo statement che si sta valutando.

Nop

$$\frac{}{\rho \vdash_\beta \langle \mathbf{nop}, \sigma^\sharp\rangle \rightsquigarrow \sigma^\sharp} \tag{4.22}$$

Assegnamento

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \to \big\langle(\mathrm{con}^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp\big\rangle}{\rho \vdash_\beta \langle \mathrm{id} := e, \sigma^\sharp\rangle \rightsquigarrow \langle \sigma_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\rangle} \quad \text{if } \sigma_0^\sharp\big[\rho(\mathrm{id}) :=^\sharp \mathrm{con}^\sharp\big] = (\sigma_1^\sharp, \varepsilon_1^\sharp) \tag{4.23}$$

Blocco

$$\frac{\rho \vdash_\beta \big\langle d, \mathrm{mark}^\sharp(\sigma^\sharp)\big\rangle \to \big\langle(\rho_0, \sigma_0^\sharp), \varepsilon_0^\sharp\big\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle s, \sigma_0^\sharp\rangle \to \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle}{\rho \vdash_\beta \langle d; s, \sigma^\sharp\rangle \rightsquigarrow \big\langle \mathrm{unmark}^\sharp(\sigma_1^\sharp), \mathrm{unmark}^\sharp(\varepsilon_0^\sharp) \sqcup \mathrm{unmark}^\sharp(\varepsilon_1^\sharp)\big\rangle} \tag{4.24}$$

se $\beta \vdash_{\mathrm{FI}(d)} d : \beta_0$.

Concatenazione di statement

$$\frac{\rho \vdash_\beta \langle s_0, \sigma^\sharp\rangle \to \langle \sigma_0^\sharp, \varepsilon_0^\sharp\rangle \quad \rho \vdash_\beta \langle s_1, \sigma_0^\sharp\rangle \to \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle}{\rho \vdash_\beta \langle s_0; s_1, \sigma^\sharp\rangle \rightsquigarrow \big\langle \sigma_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\big\rangle} \tag{4.25}$$

Test

$$\frac{\begin{array}{r}\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \to \big\langle(t^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp\big\rangle \quad \rho \vdash_\beta \langle s_0, \sigma_{\mathrm{tt}}^\sharp\rangle \to \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle \\ \rho \vdash_\beta \langle s_1, \sigma_{\mathrm{ff}}^\sharp\rangle \to \langle \sigma_2^\sharp, \varepsilon_2^\sharp\rangle\end{array}}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma^\sharp\rangle \rightsquigarrow \langle \sigma_1^\sharp \sqcup \sigma_2^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp\rangle} \tag{4.26}$$

se $\sigma_{\mathrm{tt}}^\sharp = \phi(\rho, \sigma^\sharp, e)$ e $\sigma_{\mathrm{ff}}^\sharp = \phi(\rho, \sigma^\sharp, \mathbf{not}\, e)$.

LOOP

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp \rangle \to \left\langle (t^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp \right\rangle \quad \rho \vdash_\beta \langle s, \sigma_{\mathrm{tt}}^\sharp \rangle \to \langle \sigma_1^\sharp, \varepsilon_1^\sharp \rangle \quad \rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma_1^\sharp \rangle \to \langle \sigma_2^\sharp, \varepsilon_2^\sharp \rangle}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma^\sharp \rangle \rightsquigarrow \langle \sigma_{\mathrm{ff}}^\sharp \sqcup \sigma_2^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp \rangle} \tag{4.27}$$

se $\sigma_{\mathrm{tt}}^\sharp = \phi(\rho, \sigma^\sharp, e)$ e $\sigma_{\mathrm{ff}}^\sharp = \phi(\rho, \sigma^\sharp, \mathbf{not}\, e)$.

Espressioni, dichiarazioni e statement supportati

Siano $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{s}\}$ e $N^\sharp \in \Gamma_q^{\beta\sharp}$. Allora ogni volta che supported$(\rho, N^\sharp)$ vale, per ogni regola prima specificata

$$\frac{P_0 \quad \cdots \quad P_{\ell-1}}{\rho \vdash_\beta N^\sharp \rightsquigarrow \eta^\sharp} \quad \text{se (condizione) e supported}(\rho, N^\sharp) \text{ non vale}$$

abbiamo anche la regola

$$\frac{P_0 \quad \cdots \quad P_{\ell-1}}{\rho \vdash_\beta N^\sharp \rightsquigarrow \mathrm{eval}(\rho, N^\sharp)} \quad \text{se (condizione) e supported}(\rho, N^\sharp) \text{ vale}$$

Capitolo 5

Proprietà aritmetiche degli operatori *bitwise*

5.1 Numeri interi e loro rappresentazione

Denotiamo con $B \stackrel{\text{def}}{=} \{0,1\}^\star$ l'insieme delle rappresentazioni binarie dei numeri interi. Con il simbolo b_n indichiamo una sequenza di bit in B composta da n bit mentre con $b(i)$ indichiamo il bit in posizione i dell'elemento $b \in B$ ottenuto contando le posizioni da destra verso sinistra, iniziando con il valore 0. Si specifica inoltre che $b(i)$ è sempre definito mentre $b_n(i)$ ha significato solo se $i < n$. Con $\bar{b}_n$ indichiamo invece la sequenza infinita di bit ottenuta imponendo $b_n(i) = b_n(n-1)$ per ogni $i \geq n$.

- Sia $\bar{b}_n \in B$. Se tale sequenza di bit è utilizzata per rappresentare un numero $m \in$ Integers in complemento a due, allora il valore di m è ottenuto nel seguente modo:

$$m = \begin{cases} \displaystyle\sum_{i=0}^{n-2} \bar{b}_n(i) \cdot 2^i, & \text{se } \bar{b}_n(n-1) = 0; \\ \displaystyle -\left(\sum_{i=0}^{n-2} (\neg\, \bar{b}_n(i)) \cdot 2^i\right) - 1, & \text{altrimenti.} \end{cases} \tag{5.1}$$

  Il bit di segno dell'intero m corrisponde al bit $\bar{b}_n(n-1)$.

- Sia $b_k \in B$ la rappresentazione binaria in complemento a due di un

intero $m \in \mathrm{SInt_k}$. Allora

$$m = \begin{cases} \sum_{i=0}^{k-2} b_k(i) \cdot 2^i, & \text{se } b_k(k-1) = 0 \\ -\left(\sum_{i=0}^{k-2} (\neg\, b_k(i)) \cdot 2^i\right) - 1, & \text{altrimenti.} \end{cases} \tag{5.2}$$

Il bit di segno di m corrisponde al bit $b_k(k-1)$.

- Sia $b_{k+1} \in B$ la rappresentazione binaria di un intero $m \in \mathrm{UInt_k}$. Allora

$$m = \sum_{i=0}^{k-1} b_{k+1}(i) \cdot 2^i \tag{5.3}$$

Dato che si tratta di interi senza segno, e quindi sempre positivi, richiediamo che $b_{k+1}(k) = 0$ per ogni rappresentazione b_{k+1} dei numeri in $\mathrm{UInt_k}$. Il bit di segno di m corrisponde al bit $b_{k+1}(k)$.

5.2 Complemento a uno

Sia con $\in$ Con e $b_k \in B$ la sua rappresentazione binaria. Se type(con) $\in$ Signed allora per la definizione del complemento a uno,

$$\sim \mathrm{con} = \begin{cases} -\left(\sum_{i=0}^{k-2} b_k(i) \cdot 2^i\right) - 1, & \text{se con} \geq 0; \\ \sum_{i=0}^{k-2} (\neg\, b_k(i)) \cdot 2^i, & \text{altrimenti.} \end{cases} \tag{5.4}$$

Se invece type(con) = $\mathrm{uint_k}$, allora

$$\sim \mathrm{con} = \sum_{i=0}^{k} (\neg\, b_{k+1}(i)) \cdot 2^i \tag{5.5}$$

Teorema 5.2.1 *Sia* con $\in$ Con *tale che* type(con) $\in$ Signed*. Se* con ≥ 0 *allora* $\sim$ con < 0*; altrimenti* $\sim$ con ≥ 0*.*

Dimostrazione Sia $b_k \in B$ la rappresentazione binaria di con. Consideriamo il caso in cui con ≥ 0 e dimostriamo che $\sim$ con ≤ -1. Per per la formula (5.4),

$$\mathrm{con}_0 = -\left(\sum_{i=0}^{k-2} b_k(i) \cdot 2^i\right) - 1. \tag{5.6}$$

Per ogni $i = 0 \dots k-2$, $b_k(i) \cdot 2^i$ è nullo o positivo, quindi $\sim \text{con} \leq -1$.

Consideriamo ora il caso in cui $\text{con} < 0$ e dimostriamo che $\text{con}_0 \geq 0$. Per per la formula (5.4),

$$\text{con}_0 = \sum_{i=0}^{k-2} (\neg\, b_k(i)) \cdot 2^i. \tag{5.7}$$

Come osservato in precedenza e per la definizione di $\neg$, per ogni $i = 0 \dots k-2$, $\neg b_k(i) \cdot 2^i$ è nullo o positivo, quindi $\sim \text{con} \geq 0$.

□

5.3 Congiunzione bit-a-bit

Siano $\text{con}_0, \text{con}_1 \in \text{Con}$ due costanti e $b_k^0, b_k^1 \in B$ le loro corrispondenti rappresentazioni binarie. Se tali costanti sono entrambe in Signed,

$$\text{con}_0 \wedge \text{con}_1 = \begin{cases} -\left(\sum_{i=0}^{k-2} (\neg\, b_k(i)) \cdot 2^i\right) - 1, & \text{se } \text{con}_0 \leq -1 \text{ e } \text{con}_1 \leq -1, \\ \sum_{i=0}^{k-2} b_k(i) \cdot 2^i, & \text{altrimenti.} \end{cases} \tag{5.8}$$

Se invece $\text{type}(\text{con}_0) = \text{type}(\text{con}_0) = \text{uint}_\text{k}$

$$\text{con}_0 \wedge \text{con}_1 = \sum_{i=0}^{k-1} b_k(i) \cdot 2^i \tag{5.9}$$

In entrambe le definizioni (5.8) e (5.9), b_k è la rappresentazione ottenuta a partire da b_k^0 e b_k^1 nel seguente modo:

$$b_k(i) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{se } b_k^0(i) = 0; \\ b_k^1(i), & \text{se } b_k^0(i) = 1. \end{cases} \tag{5.10}$$

Teorema 5.3.1 *Siano* $\text{con}_0, \text{con}_1 \in \text{Con}$ *tali che* $\text{type}(con_0) = \text{type}(con_1) \in \text{aType}$. *Se* $\text{con}_0 \geq 0$ *e* $\text{con}_1 \geq 0$ *allora* $\text{con}_0 \wedge \text{con}_1 \geq 0$ *e* $\text{con}_0 \wedge \text{con}_1 \leq \min(\text{con}_0, \text{con}_1)$.

Dimostrazione Supponiamo che $\text{type}(\text{con}_0) = \text{type}(\text{con}_1) \in \text{Signed}$. Se, come da ipotesi, entrambe le costanti assumono solo valori positivi o nulli, allora per la formula (5.8),

$$\text{con}_0 \wedge \text{con}_1 = \sum_{i=0}^{k-2} b_k(i) \cdot 2^i.$$

La limitazione inferiore $\mathrm{con}_0 \wedge \mathrm{con}_1 \geq 0$ risulta essere banalmente vera in quanto $\mathrm{con}_0 \wedge \mathrm{con}_1$ è ottenuto come somma di un numero finito di termini nulli o positivi.

Dimostriamo ora che vale la limitazione superiore facendo vedere che il valore $\mathrm{con}_0 \wedge \mathrm{con}_1$ è inferiore di entrambi con_0 e con_1.

1. $\mathrm{con}_0 \wedge \mathrm{con}_1 \leq \mathrm{con}_0$
   Per le formule (5.8), (5.1) e (5.2), dobbiamo dimostrare che

   $$\sum_{i=0}^{k-2} b_k(i) \cdot 2^i \leq \sum_{i=0}^{k-2} b_k^0(i) \cdot 2^i.$$

   Tale disequazione è riconducibile alla seguente:

   $$\sum_{i=0}^{k-2} \big(b_k(i) - b_k^0(i)\big) \cdot 2^i \leq 0. \tag{5.11}$$

   Per ogni $i = 0 \ldots k-2$, per la formula (5.10) possiamo dire che

   - se $b_k^0(i) = 0$, allora $b_k(i) = 0$. Quindi $b_k(i) - b_k^0(i) = 0$;
   - se $b_k^0(i) = 1$, allora $b_k(i) = b_k^1(i)$. Quindi $b_k(i) - b_k^0(i) \leq 0$.

   La disequazione (5.11) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

2. $\mathrm{con}_0 \wedge \mathrm{con}_1 \leq \mathrm{con}_1$
   Per le formule (5.8), (5.1) e (5.2) dobbiamo dimostrare che

   $$\sum_{i=0}^{k-2} b_k(i) \cdot 2^i \leq \sum_{i=0}^{k-2} b_k^1(i) \cdot 2^i.$$

   Tale disequazione è riconducibile alla seguente:

   $$\sum_{i=0}^{k-2} \big(b_k(i) - b_k^1(i)\big) \cdot 2^i \leq 0. \tag{5.12}$$

   Per ogni $i = 0 \ldots k-2$, per la formula (5.10) possiamo dire che

   - se $b_k^0(i) = 0$, allora $b_k(i) = 0$. Quindi, $b_k(i) - b_k^1(i) \leq 0$;
   - se $b_k^0(i) = 1$ allora $b_k(i) = b_k^1(i)$. Quindi $b_k(i) - b_k^1(i) = 0$.

   La disequazione (5.12) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

Supponiamo ora che $\text{type}(\text{con}_0) = \text{type}(\text{con}_1) = \text{uint}_\text{k}$. Per la formula (5.9) abbiamo quindi che

$$\text{con}_0 \wedge \text{con}_1 = \sum_{i=0}^{k-1} b_k(i) \cdot 2^i.$$

La limitazione inferiore $\text{con}_0 \wedge \text{con}_1 \geq 0$ è vera in quanto i termini della sommatoria sono nulli o positivi. Dimostriamo infine che $\text{con}_0 \wedge \text{con}_1 \leq \min(\text{con}_0, \text{con}_1)$ facendo vedere che valgono se seguenti disequazioni:

3. $\text{con}_0 \wedge \text{con}_1 \leq \text{con}_0$
   La dimostrazione coincide con quella introdotta per il precedente caso 1.

4. $\text{con}_0 \wedge \text{con}_1 \leq \text{con}_1$
   La dimostrazione coincide con quella introdotta per il precedente caso 2.

□

Teorema 5.3.2 *Siano* $\text{con}_0, \text{con}_1 \in \text{Con}$ *tali che* $\text{type}(con_0) = \text{type}(con_1) \in \text{aType}$. *Se* $\text{con}_0 \geq 0$ *e* $\text{con}_1 \leq -1$ *allora* $\text{con}_0 \wedge \text{con}_1 \geq 0$ *e* $\text{con}_0 \wedge \text{con}_1 \leq \text{con}_0$.

Dimostrazione Se per ipotesi abbiamo che $\text{con}_0 \geq 0$ e $\text{con}_1 \leq -1$ allora per la formula (5.8),

$$\text{con}_0 \wedge \text{con}_1 = \sum_{i=0}^{k-2} b_k(i) \cdot 2^i.$$

La disequazione $\text{con}_0 \wedge \text{con}_1 \geq 0$ risulta essere banalmente vera in quanto i termini della sommatoria sono nulli o positivi. Dimostriamo ora che vale $\text{con}_0 \wedge \text{con}_1 \leq \text{con}_0$. In questo caso la dimostrazione coincide con quella introdotta nel punto 1 della dimostrazione del teorema 5.3.1.

□

Teorema 5.3.3 *Siano* $\text{con}_0, \text{con}_1 \in \text{Con}$ *tali che* $\text{type}(con_0) = \text{type}(con_1) \in \text{aType}$. *Se* $\text{con}_0 \leq -1$ *e* $\text{con}_1 \geq 0$ *allora* $\text{con}_0 \wedge \text{con}_1 \geq 0$ *e* $\text{con}_0 \wedge \text{con}_1 \leq \text{con}_1$.

Dimostrazione Se per ipotesi abbiamo che $\text{con}_0 \leq -1$ e $\text{con}_1 \geq 0$ allora per la formula (5.8),

$$\text{con}_0 \wedge \text{con}_1 = \sum_{i=0}^{k-2} b_k(i) \cdot 2^i.$$

La disequazione $\text{con}_0 \wedge \text{con}_1 \geq 0$ risulta essere banalmente vera in quanto i termini della sommatoria sono nulli o positivi. Dimostriamo ora che vale $\text{con}_0 \wedge \text{con}_1 \leq \text{con}_0$. In questo caso la dimostrazione coincide con quella introdotta nel punto 2 della dimostrazione del teorema 5.3.1.

□

Teorema 5.3.4 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \leq -1$ *e* $\mathrm{con}_1 \leq -1$ *allora* $\mathrm{con}_0 \wedge \mathrm{con}_1 \geq 0$ *e* $\mathrm{con}_0 \wedge \mathrm{con}_1 \leq \min(\mathrm{con}_0, \mathrm{con}_1)$.

Dimostrazione Se, come da ipotesi, entrambe le costanti assumono solo valori negativi, allora per la formula (5.8),

$$\mathrm{con}_0 \wedge \mathrm{con}_1 = -\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1.$$

Dimostriamo che vale la limitazione inferiore $\mathrm{con}_0 + \mathrm{con}_1$ nel caso di operandi con segno. Per le formule (5.8), (5.1) e (5.2), dobbiamo dimostrare che

$$-\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i \geq -\sum_{i=0}^{k-2}(\neg\, b_k^0(i)) \cdot 2^i - \left(\sum_{i=0}^{k-2}(\neg\, b_k^1(i)) \cdot 2^i\right) - 1.$$

Tale disequazione può essere ricondotta a

$$\sum_{i=0}^{k-2}\big((\neg\, b_k(i)) - (\neg\, b_k^0(i)) - (\neg\, b_k^1(i))\big) \cdot 2^i \leq 1. \tag{5.13}$$

Per ogni $i = 0 \ldots k-2$, per la formula (5.10) possiamo dire che

1. se $b_k^0(i) = 0$ allora $b_k(i) = 0$. Quindi,

$$(\neg\, b_k(i)) - (\neg\, b_k^0(i)) - (\neg\, b_k^1(i)) \leq 0;$$

2. se $b_k^0(i) = 1$ allora $b_k(i) = b_k^1(i)$. Quindi,

$$(\neg\, b_k(i)) - (\neg\, b_k^0(i)) - (\neg\, b_k^1(i)) = 0.$$

La disequazione (5.13) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

Dimostriamo ora che vale la limitazione superiore $\min(\mathrm{con}_0, \mathrm{con}_1)$ nel caso di operandi con segno. Occorre dimostrare che $\mathrm{con}_0 \wedge \mathrm{con}_1$ è minore o uguale di entrambi i valori con_0 e con_1.

3. $\mathrm{con}_0 \wedge \mathrm{con}_1 \leq \mathrm{con}_0$
   Per la formule (5.8), (5.1) e (5.2) dobbiamo dimostrare che

$$-\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1 \leq -\left(\sum_{i=0}^{k-2}(\neg\, b_k^0(i)) \cdot 2^i\right) - 1.$$

   Tale disequazione può essere ricondotta alla seguente:

$$\sum_{i=0}^{k-2}\left((\neg\, b_k(i)) - (\neg\, b_k^0(i))\right) \cdot 2^i \geq 0. \tag{5.14}$$

   Per ogni $i = 0 \ldots k-2$, per la formula (5.10) possiamo dire che

   (a) se $b_k^0(i) = 0$ allora $b_k(i) = 0$. Quindi $(\neg\, b_k(i)) - (\neg\, b_k^0(i)) = 0$;

   (b) se $b_k^0(i) = 1$ allora $b_k(i) = b_k^1(i)$. Quindi $(\neg\, b_k(i)) - (\neg\, b_k^0(i)) \geq 0$.

   La disequazione (5.14) risulta vera in quanto tutti i termini della sommatoria sono nulli o positivi.

4. $\mathrm{con}_0 \wedge \mathrm{con}_1 \leq \mathrm{con}_1$
   Per le formule (5.8), (5.1) e (5.2) dobbiamo dimostrare che

$$-\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1 \leq -\left(\sum_{i=0}^{k-2}(\neg\, b_k^1(i)) \cdot 2^i\right) - 1.$$

   Tale disequazione può essere ricondotta alla seguente:

$$\sum_{i=0}^{k-2}\left((\neg\, b_k(i)) - (\neg\, b_k^1(i))\right) \cdot 2^i \geq 0. \tag{5.15}$$

   Per ogni $i = 0 \ldots k-2$, per la formula (5.10) possiamo dire che

   (a) se $b_k^0(i) = 0$ allora $b_k(i) = 0$. Quindi, $(\neg\, b_k(i)) - (\neg\, b_k^1(i)) \geq 0$;

   (b) se $b_k^0(i) = 1$ allora $b_k(i) = b_k^1(i)$. Quindi $(\neg\, b_k(i)) - (\neg\, b_k^1(i)) = 0$.

   La disequazione (5.15) risulta vera in quanto tutti i termini della sommatoria sono nulli o positivi.

□

5.4 Disgiunzione inclusiva bit-a-bit

Siano $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ due costanti e $b_k^0, b_k^1 \in B$ le loro corrispondenti rappresentazioni binarie. Se tali costanti sono entrambe in Signed,

$$\mathrm{con}_0 \vee \mathrm{con}_1 = \begin{cases} \displaystyle\sum_{i=0}^{k-2} b_k(i) \cdot 2^i, & \text{se } \mathrm{con}_0 \geq 0 \text{ e } \mathrm{con}_1 \geq 0; \\ \displaystyle -\left(\sum_{i=0}^{k-2} (\neg\, b_k(i)) \cdot 2^i \right) - 1, & \text{altrimenti.} \end{cases} \tag{5.16}$$

Se invece $\mathrm{type}(\mathrm{con}_0) = \mathrm{type}(\mathrm{con}_0) = \mathrm{uint_k}$, allora

$$\mathrm{con}_0 \vee \mathrm{con}_1 = \sum_{i=0}^{k-1} b_k(i) \cdot 2^i \tag{5.17}$$

In entrambe le definizioni (5.16) e (5.17), il termine $b_k(i)$ è definito come segue:

$$b_k(i) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{se } b_k^0(i) = 1; \\ b_k^1(i), & \text{se } b_k^0(i) = 0. \end{cases} \tag{5.18}$$

Teorema 5.4.1 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \geq 0$ *e* $\mathrm{con}_1 \geq 0$ *allora* $\mathrm{con}_0 \vee \mathrm{con}_1 \geq \max(\mathrm{con}_0, \mathrm{con}_1)$ *e* $\mathrm{con}_0 \vee \mathrm{con}_1 \leq \mathrm{con}_0 + \mathrm{con}_1$.

Dimostrazione Supponiamo che $\mathrm{type}(\mathrm{con}_0) = \mathrm{type}(\mathrm{con}_1) \in \mathrm{Signed}$. Se, come da ipotesi, entrambe le costanti assumono solo valori positivi o nulli, allora per la formula (5.16),

$$\mathrm{con}_0 \vee \mathrm{con}_1 = \sum_{i=0}^{k-2} b_k(i) \cdot 2^i.$$

Dimostriamo che vale la limitazione inferiore $\max(\mathrm{con}_0, \mathrm{con}_1)$ nel caso in cui gli operandi sono di tipo con segno. Occorre dimostrare che il risultato dell'operazione binaria $\vee$ è maggiore o uguale di entrambi i valori di con_0 e con_1.

1. $\mathrm{con}_0 \vee \mathrm{con}_1 \geq \mathrm{con}_0$
   Per le formule (5.16), (5.1) e (5.2), dobbiamo dimostrare che

$$\sum_{i=0}^{k-2} b_k(i) \cdot 2^i \geq \sum_{i=0}^{k-2} b_k^0(i) \cdot 2^i.$$

Tale disequazione può essere ricondotta a

$$\sum_{i=0}^{k-2} \big(b_k(i) - b_k^0(i)\big) \cdot 2^i \geq 0. \tag{5.19}$$

Per ogni $i = 0 \dots k-2$, per la formula (5.18) possiamo dire che

(a) Se $b_k^0(i) = 1$ allora $b_k(i) = 1$. Quindi, $b_k(i) - b_k^0(i) = 0$;
(b) se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi, $b_k(i) - b_k^0(i) =\geq 0$;

La disequazione (5.19) risulta vera in quanto tutti i termini della sommatoria sono nulli o positivi.

2. $\mathrm{con}_0 \vee \mathrm{con}_1 \geq \mathrm{con}_1$
Per le formule (5.16), (5.1) e (5.2), dobbiamo dimostrare che

$$\sum_{i=0}^{k-2} b_k(i) \cdot 2^i \geq \sum_{i=0}^{k-2} b_k^1(i) \cdot 2^i.$$

Tale disequazione può essere ricondotta a

$$\sum_{i=0}^{k-2} \big(b_k(i) - b_k^1(i)\big) \cdot 2^i \geq 0. \tag{5.20}$$

Per ogni $i = 0 \dots k-2$, per la formula (5.18) possiamo dire che

(a) Se $b_k^0(i) = 1$ allora $b_k(i) = 1$. Quindi, $b_k(i) - b_k^1(i) \geq 0$;
(b) se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi, $b_k(i) - b_k^1(i) = 0$;

La disequazione (5.20) risulta vera in quanto tutti i termini della sommatoria sono nulli o positivi.

Dimostriamo ora che vale la limitazione superiore $\mathrm{con}_0 + \mathrm{con}_1$. Per le formule (5.16), (5.1) e (5.2), dobbiamo dimostrare che

$$\sum_{i=0}^{k-2} b_k(i) \cdot 2^i \leq \sum_{i=0}^{k-2} b_k^0(i) \cdot 2^i + \sum_{i=0}^{k-2} b_k^1(i) \cdot 2^i.$$

Tale disequazione è riconducibile alla seguente:

$$\sum_{i=0}^{k-2} \big(b_k(i) - b_k^0(i) - b_k^1(i)\big) \cdot 2^i \leq 0. \tag{5.21}$$

Per ogni $i = 0 \dots k-2$, per la formula (5.18) possiamo dire che

1. se $b_k^0(i) = 1$ allora $b_k(i) = 1$. Quindi, $b_k(i) - b_k^0(i) - b_k^1(i) \leq 0$;
2. se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi, $b_k(i) - b_k^0(i) - b_k^1(i) = 0$.

La disequazione (5.21) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

Supponiamo ora che $\text{type}(\text{con}_0) = \text{type}(\text{con}_1) = \text{uint}_\text{k}$. Dimostriamo che vale la limitazione inferiore $\max(\text{con}_0, \text{con}_1)$ nel caso di operandi senza segno. Occorre quindi dimostrare che il risultato dell'operazione binaria $\vee$ è maggiore o uguale di entrambi i valori di con_0 e con_1.

3. $\text{con}_0 \vee \text{con}_1 \geq \text{con}_0$
   Per le formule (5.17) e (5.3) dobbiamo dimostrare che

$$\sum_{i=0}^{k-1} b_k(i) \cdot 2^i \geq \sum_{i=0}^{k-1} b_{k+1}^0(i) \cdot 2^i$$

   Tale dimostrazione coincide con quella introdotta nel precedente caso 1.

4. $\text{con}_0 \vee \text{con}_1 \geq \text{con}_1$
   Per le formule (5.17) e (5.3) dobbiamo dimostrare che

$$\sum_{i=0}^{k-1} b_k(i) \cdot 2^i \geq \sum_{i=0}^{k-1} b_{k+1}^1(i) \cdot 2^i$$

   Tale dimostrazione coincide con quella introdotta nel precedente caso 2.

Dimostriamo che vale la limitazione superiore $\text{con}_0 + \text{con}_1$ nel caso di operandi senza segno. Dobbiamo quindi dimostrare che

$$\sum_{i=0}^{k-1} b_k(i) \cdot 2^i \geq \sum_{i=0}^{k-1} b_{k+1}^0(i) \cdot 2^i + \sum_{i=0}^{k-1} b_{k+1}^1(i) \cdot 2^i.$$

Tale disequazione è riconducibile alla seguente:

$$\sum_{i=0}^{k-1} \left(b_k(i) - b_k^0(i) - b_k^1(i) \right) \cdot 2^i \geq 0.$$

La dimostrazione di tale limitazione superiore coincide con la dimostrazione della limitazione superiore nel caso di operandi con segno.

□

Teorema 5.4.2 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \geq 0$ *e* $\mathrm{con}_1 \leq -1$ *allora* $\mathrm{con}_0 \vee \mathrm{con}_1 \geq \mathrm{con}_1$ *e* $\mathrm{con}_0 \vee \mathrm{con}_1 \leq -1$.

Dimostrazione Se per ipotesi abbiamo che $\mathrm{con}_0 \geq 0$ e $\mathrm{con}_1 \leq -1$ allora per la formula (5.16),

$$\mathrm{con}_0 \vee \mathrm{con}_1 = -\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1.$$

Dimostriamo che vale la limitazione inferiore $\mathrm{con}_0 \vee \mathrm{con}_1 \geq \mathrm{con}_1$. Per le formule (5.16) e (5.1) dobbiamo quindi dimostrare che

$$-\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1 \geq -\left(\sum_{i=0}^{k-2}(\neg\, b_k^1(i)) \cdot 2^i\right) - 1.$$

Tale limitazione superiore si può ricondurre alla disequazione

$$\sum_{i=0}^{k-2}\left((\neg\, b_k(i)) - (\neg\, b_k^1(i))\right) \cdot 2^i \leq 0 \tag{5.22}$$

Per ogni $i = 0 \ldots k-2$, per la formula (5.18) possiamo dire che

1. Se $b_k^0(i) = 1$ allora $b_k(i) = 1$. Quindi $(\neg\, b_k(i)) - (\neg\, b_k^1(i)) \leq 0$;
2. se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi $(\neg\, b_k(i)) - (\neg\, b_k^1(i)) = 0$.

La disequazione (5.22) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

Dimostriamo ora che vale la limitazione superiore $\mathrm{con}_0 \vee \mathrm{con}_1 \leq -1$. Dobbiamo quindi dimostrare che

$$-\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1 \leq -1,$$

ovvero che

$$\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i \geq 0.$$

Tale disequazione è vera in quanto tutti i termini della sommatoria sono nulli o positivi.

□

Teorema 5.4.3 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \leq -1$ *e* $\mathrm{con}_1 \geq 0$ *allora* $\mathrm{con}_0 \wedge \mathrm{con}_1 \geq \mathrm{con}_0$ *e* $\mathrm{con}_0 \wedge \mathrm{con}_1 \leq -1$.

Dimostrazione Se per ipotesi abbiamo che $\mathrm{con}_0 \leq -1$ e $\mathrm{con}_1 \geq 0$ allora per la formula (5.16),

$$\mathrm{con}_0 \vee \mathrm{con}_1 = -\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1.$$

Dimostriamo che vale la limitazione inferiore $\mathrm{con}_0 \vee \mathrm{con}_1 \geq \mathrm{con}_0$. Per le formule (5.16) e (5.1) dobbiamo quindi dimostrare che

$$-\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1 \geq -\left(\sum_{i=0}^{k-2}(\neg\, b_k^0(i)) \cdot 2^i\right) - 1$$

Tale limitazione superiore si può ricondurre alla disequazione

$$\sum_{i=0}^{k-2}\left((\neg\, b_k(i)) - (\neg\, b_k^0(i))\right) \cdot 2^i \leq 0 \tag{5.23}$$

Per ogni $i = 0 \ldots k-2$, per la formula (5.18) possiamo dire che

1. se $b_k^0(i) = 1$ allora $b_k(i) = 1$. Quindi, $(\neg\, b_k(i)) - (\neg\, b_k^0(i)) = 0$;
2. se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi, $(\neg\, b_k(i)) - (\neg\, b_k^0(i)) \leq 0$.

La disequazione (5.23) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

La dimostrazione della limitazione superiore $\mathrm{con}_0 \vee \mathrm{con}_1 \leq -1$ coincide con la dimostrazione della limitazione superiore introdotta nel precedente teorema 5.4.2.

□

Teorema 5.4.4 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \leq -1$ *e* $\mathrm{con}_1 \leq -1$ *allora* $\mathrm{con}_0 \wedge \mathrm{con}_1 \geq \max(\mathrm{con}_0, \mathrm{con}_1)$ *e* $\mathrm{con}_0 \wedge \mathrm{con}_1 \leq -1$.

Dimostrazione Dimostriamo ora che vale la limitazione inferiore $\mathrm{con}_0 \vee \mathrm{con}_1 \geq \max(\mathrm{con}_0, \mathrm{con}_1)$. In questo caso la dimostrazione coincide con quella introdotta nel nella dimostrazione del precedente teorema 5.4.1.

Dimostriamo che vale la limitazione superiore $\mathrm{con}_0 \vee \mathrm{con}_1 \leq -1$. Tale dimostrazione coincide con quella del precedente teorema 5.4.2.

□

5.5 Disgiunzione esclusiva bit-a-bit

Siano $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ due costanti e $b_k^0, b_k^1 \in B$ le loro corrispondenti rappresentazioni binarie. Se tali costanti sono entrambe in Signed,

$$\mathrm{con}_0 \veebar \mathrm{con}_1 = \begin{cases} \displaystyle\sum_{i=0}^{k-2} b_k(i) \cdot 2^i, & \text{se } \mathrm{con}_0 \geq 0 \text{ e } \mathrm{con}_1 \geq 0 \\ & \text{o } \mathrm{con}_0 \leq -1 \text{ e } \mathrm{con}_1 \leq -1; \\ \displaystyle -\left(\sum_{i=0}^{k-2} (\neg\, b_k(i)) \cdot 2^i\right) - 1, & \text{altrimenti.} \end{cases} \tag{5.24}$$

Se invece $\mathrm{type}(\mathrm{con}_0) = \mathrm{type}(\mathrm{con}_0) = \mathrm{uint_k}$, allora

$$\mathrm{con}_0 \veebar \mathrm{con}_1 = \sum_{i=0}^{k-1} b_k(i) \cdot 2^i \tag{5.25}$$

In entrambe le definizioni (5.24) e (5.25), il termine $b_k(i)$ è definito come segue:

$$b_k(i) \stackrel{\text{def}}{=} \begin{cases} b_k^1(i), & \text{se } b_k^0(i) = 0; \\ \neg\, b_k^1(i), & \text{altrimenti.} \end{cases} \tag{5.26}$$

Teorema 5.5.1 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \geq 0$ *e* $\mathrm{con}_1 \geq 0$ *allora* $\mathrm{con}_0 \veebar \mathrm{con}_1 \geq 0$ *e* $\mathrm{con}_0 \veebar \mathrm{con}_1 \leq \mathrm{con}_0 + \mathrm{con}_1$.

Dimostrazione Supponiamo che $\mathrm{type}(\mathrm{con}_0) = \mathrm{type}(\mathrm{con}_1) \in \mathrm{Signed}$. Se, come da ipotesi, entrambe le costanti assumono solo valori positivi o nulli, allora per la formula (5.24),

$$\mathrm{con}_0 \veebar \mathrm{con}_1 = \sum_{i=0}^{k-2} b_k(i) \cdot 2^i.$$

Dimostriamo che vale la limitazione inferiore nel caso in cui gli operandi sono di tipo con segno. Per la formula (5.24) dobbiamo quindi dimostrare che

$$\sum_{i=0}^{k-2} b_k(i) \cdot 2^i \geq 0$$

Tale disequazione è vera in quanto i termini della sommatoria sono nulli o positivi.

Dimostriamo ora che vale la limitazione superiore $\mathrm{con}_0 + \mathrm{con}_1$ nel caso in cui gli operandi sono di tipo con segno. Per le formule (5.24), (5.1) e (5.2), dobbiamo dimostrare che

$$\sum_{i=0}^{k-2} b_k(i) \cdot 2^i \leq \sum_{i=0}^{k-2} b_k^0(i) \cdot 2^i + \sum_{i=0}^{k-2} b_k^1(i) \cdot 2^i.$$

Tale disequazione è riconducibile alla seguente:

$$\sum_{i=0}^{k-2} \big(b_k(i) - b_k^0(i) - b_k^1(i)\big) \cdot 2^i \leq 0. \tag{5.27}$$

Per ogni $i = 0 \ldots k-2$, per la formula (5.26) possiamo dire che

1. Se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi $b_k(i) - b_k^0(i) - b_k^1(i) = 0$;
2. se $b_k^0(i) = 1$ allora $b_k(i) = \neg\, b_k^1(i)$. Quindi $b_k(i) - b_k^0(i) - b_k^1(i) \leq 0$.

La disequazione (5.27) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

Supponiamo ora che $\mathrm{type}(\mathrm{con}_0) = \mathrm{type}(\mathrm{con}_1) = \mathrm{uint_k}$. Per la formula (5.25) abbiamo quindi che

$$\mathrm{con}_0 \veebar \mathrm{con}_1 = \sum_{i=0}^{k-1} b_k(i) \cdot 2^i.$$

La limitazione inferiore $\mathrm{con}_0 \veebar \mathrm{con}_1 \geq 0$ è sempre vera nel caso di operandi senza segno, in quanto una somma finita di valori non negativi è non negativa.

Verifichiamo ora che valga la limitazione superiore $\mathrm{con}_0 + \mathrm{con}_1$ nel caso di operandi senza segno. Per le formule (5.25) e (5.3) dobbiamo dimostrare che

$$\sum_{i=0}^{k-1} b_k(i) \cdot 2^i \leq \sum_{i=0}^{k-1} b_{k+1}^0(i) \cdot 2^i + \sum_{i=0}^{k-1} b_{k+1}^1(i) \cdot 2^i.$$

Tale disequazione è riconducibile alla seguente

$$\sum_{i=0}^{k-1} \big(b_k(i) - b_{k+1}^0(i) - b_{k+1}^1(i)\big) \cdot 2^i \leq 0.$$

La dimostrazione coincide quindi con quella introdotta per la limitazione superiore del risultato ottenuto quando gli operandi sono con segno.

□

Teorema 5.5.2 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \geq 0$ *e* $\mathrm{con}_1 \leq -1$ *allora* $\mathrm{con}_0 \veebar \mathrm{con}_1 \geq \mathrm{con}_1 - \mathrm{con}_0$ *e* $\mathrm{con}_0 \veebar \mathrm{con}_1 \leq -1$.

Dimostrazione Se per ipotesi abbiamo che $\mathrm{con}_0 \geq 0$ e $\mathrm{con}_1 \leq -1$ allora per la formula (5.24),

$$\mathrm{con}_0 \veebar \mathrm{con}_1 = -\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1.$$

Iniziamo con il verificare la limitazione inferiore $\mathrm{con}_1 - \mathrm{con}_0$. Per le formule (5.24), (5.1) e (5.2), dobbiamo dimostrare che

$$-\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1 \geq -\left(\sum_{i=0}^{k-2}(\neg\, b_k^1(i)) \cdot 2^i\right) - 1 - \sum_{i=0}^{k-2} b_k^0(i) \cdot 2^i.$$

Tale disequazione è riconducibile a

$$-\sum_{i=0}^{k-2}\left((\neg\, b_k(i)) - b_k^0(i) - (\neg\, b_k^1(i))\right) \cdot 2^i \geq 0 \tag{5.28}$$

Per ogni $i = 0 \ldots k-2$, per la formula (5.18) possiamo dire che

1. se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi, $(\neg\, b_k(i)) - b_k^0(i) - (\neg\, b_k^1(i)) = 0$;
2. se $b_k^0(i) = 1$ allora $b_k(i) = \neg\, b_k^1(i)$. Quindi, $(\neg\, b_k(i)) - b_k^0(i) - (\neg\, b_k^1(i)) \geq 0$.

La disequazione (5.28) risulta vera in quanto tutti i termini della sommatoria sono nulli o positivi.

Verifichiamo ora la limitazione superiore, ovvero che $\mathrm{con}_0 \veebar \mathrm{con}_1 \leq -1$. Dobbiamo quindi verificare che

$$-\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1 \leq -1$$

Tale disequazione è riconducibile a

$$\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i \geq 0.$$

Tale disequazione è vera in quanto la somma di finiti termini nulli o positivi è un valore non negativo.

□

Teorema 5.5.3 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \leq -1$ *e* $\mathrm{con}_1 \geq 0$ *allora* $\mathrm{con}_0 \veebar \mathrm{con}_1 \geq \mathrm{con}_0 - \mathrm{con}_1 0$ *e* $\mathrm{con}_0 \veebar \mathrm{con}_1 \leq -1$.

Dimostrazione Se per ipotesi abbiamo che $\mathrm{con}_0 \leq -1$ e $\mathrm{con}_1 \geq 0$ allora per la formula (5.24),

$$\mathrm{con}_0 \veebar \mathrm{con}_1 = -\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1.$$

Iniziamo con il verificare la limitazione inferiore $\mathrm{con}_0 - \mathrm{con}_1$. Per le formule (5.24), (5.1) e (5.2), dobbiamo dimostrare che

$$-\left(\sum_{i=0}^{k-2}(\neg\, b_k(i)) \cdot 2^i\right) - 1 \geq -\left(\sum_{i=0}^{k-2}(\neg\, b_k^0(i)) \cdot 2^i\right) - 1 - \sum_{i=0}^{k-2} b_k^1(i) \cdot 2^i.$$

Tale disequazione è riconducibile a

$$\sum_{i=0}^{k-2}\left((\neg\, b_k(i)) - (\neg\, b_k^0(i)) - b_k^1(i)\right) \cdot 2^i \leq 0. \tag{5.29}$$

Per ogni $i = 0 \ldots k-2$, per la formula (5.26) possiamo dire che

1. Se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi, $(\neg\, b_k(i)) - (\neg\, b_k^0(i)) - b_k^1(i) = 0$
2. Se $b_k^0(i) = 1$ allora $b_k(i) = \neg\, b_k^1(i)$. Quindi, $(\neg\, b_k(i)) - (\neg\, b_k^0(i)) - b_k^1(i) \leq 0$

La disequazione (5.29) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

La dimostrazione della limitazione superiore coincide con la dimostrazione della limitazione superiore introdotta nel precedente teorema 5.5.2.

□

Teorema 5.5.4 *Siano* $\mathrm{con}_0, \mathrm{con}_1 \in \mathrm{Con}$ *tali che* $\mathrm{type}(con_0) = \mathrm{type}(con_1) \in \mathrm{aType}$. *Se* $\mathrm{con}_0 \leq -1$ *e* $\mathrm{con}_1 \leq -1$ *allora* $\mathrm{con}_0 \veebar \mathrm{con}_1 \geq 0$ *e* $\mathrm{con}_0 \veebar \mathrm{con}_1 \leq -\mathrm{con}_0 - \mathrm{con}_1$.

Dimostrazione Supponiamo che $\text{type}(\text{con}_0) = \text{type}(\text{con}_1) \in \text{Signed}$. Se, come da ipotesi, entrambe le costanti assumono solo valori negativi, allora per la formula (5.24),

$$\text{con}_0 \veebar \text{con}_1 = \sum_{i=0}^{k-2} b_k(i) \cdot 2^i.$$

La dimostrazione della limitazione inferiore coincide con la dimostrazione della limitazione inferiore introdotta nel teorema 5.5.1 nel caso di operandi con segno.

Dimostriamo ora che vale la limitazione superiore $-\text{con}_0 - \text{con}_1$. Per le formule (5.24), (5.1) e (5.2), dobbiamo dimostrare che

$$\sum_{i=0}^{k-2} b_k(i) \cdot 2^i \leq \left(\sum_{i=0}^{k-2} (\neg\, b_k^0(i)) \cdot 2^i \right) + 1 + \left(\sum_{i=0}^{k-2} (\neg\, b_k^1(i)) \cdot 2^i \right) + 1.$$

Tale disequazione è riconducibile alla seguente:

$$\sum_{i=0}^{k-2} \left(b_k(i) - (\neg\, b_k^0(i)) - (\neg\, b_k^1(i)) \right) \cdot 2^i \leq 2. \tag{5.30}$$

Per ogni $i = 0 \ldots k-2$, per la formula (5.26) possiamo dire che

1. Se $b_k^0(i) = 0$ allora $b_k(i) = b_k^1(i)$. Quindi, $b_k(i) - (\neg\, b_k^0(i)) - (\neg\, b_k^1(i)) \leq 0$;
2. Se $b_k^0(i) = 1$ allora $b_k(i) = \neg\, b_k^1(i)$. Quindi, $b_k(i) - (\neg\, b_k^0(i)) - (\neg\, b_k^1(i)) = 0$.

La disequazione (5.30) risulta vera in quanto tutti i termini della sommatoria sono nulli o negativi.

□

Capitolo 6

Implementazione degli operatori *bitwise* sul dominio degli intervalli

In questo capitolo introdurremo il dominio astratto degli intervalli di interi seguendo lo schema proposto in [3]. Useremo gli intervalli di interi per dare un esempio di interpretazione astratta degli operatori di manipolazione dei bit. Gli algoritmi che verranno proposti per calcolare l'intervallo risultante dalle operazioni di complemento a uno, congiunzione bit-a-bit, disgiunzione inclusiva e esclusiva bit-a-bit sono presi da [2].

6.1 Il dominio degli intervalli

Il dominio dei valori concreti preso in considerazione è:

$$\big(\wp(\mathbb{Z}), \subseteq, \varnothing, \mathbb{Z}, \cap, \cup\big).$$

Ogni insieme di numeri interi, ovvero ogni elemento di $\wp(\mathbb{Z})$ è associabile ad un intervallo, ovvero un elemento di un dominio astratto; definiamo quindi l'insieme

$$I \stackrel{\text{def}}{=} \big\{ \big[\, \bar{l},\, \bar{u} \,\big] \;.\; \bar{l} \in \bar{\mathbb{Z}}, \bar{u} \in \bar{\mathbb{Z}} \text{ e } \bar{l} \leq \bar{u} \big\} \uplus \big\{ \perp \big\}.$$

Occorre sottolineare che la relazione di ordinamento $\leq$ utilizzata nella definizione precedente corrisponde all'estensione dell'ordinamento sull'insieme $\bar{\mathbb{Z}}$; in particolare,

$$\forall\, \bar{z} \in \bar{\mathbb{Z}}, \qquad -\infty \leq \bar{z} \text{ e } \bar{z} \leq +\infty$$

La stessa estensione è applicabile all'ordinamento $\geq$. Sia $\bar{z}$ un generico elemento in $\bar{\mathbb{Z}}$ e $i = \big[\, \bar{l},\, \bar{u} \,\big]$ un intervallo. Allora diremo che $\bar{z}$ appartiene

all'intervallo i, e lo indicheremo con $\bar{z} \in i$, se e soltanto se Diremo inoltre che $i \geq 0$ (risp. $i \leq 0$) se $\gamma(i_0) \subseteq \mathbb{N}$ (risp. $\gamma(i_1) \nsubseteq \mathbb{N}$).

Il dominio degli intervalli sugli interi, come precedentemente indicato nella sezione 4.1, è definito dal reticolo limitato

$$\big(I, \sqsubseteq, \bot, \sqcup\big)$$

I generici elementi di I sono indicati con i, i_0, i_1,...

Siano $i_0 = \big[\, \bar{l}_0,\, \bar{u}_0 \,\big]$ e $i_1 = \big[\, \bar{l}_1,\, \bar{u}_1 \,\big]$ intervalli in I. L'operatore di ordinamento parziale astratto $\sqsubseteq$ è definito in modo che

$$i_0 \sqsubseteq i_1 \;\Leftrightarrow\; (\bar{l}_1 \leq \bar{l}_0 \text{ e } \bar{u}_0 \leq \bar{u}_1),$$

mentre l'operatore di least upper bound $\sqcup$ è definito come

$$i_0 \sqcup i_1 \stackrel{\text{def}}{=} \big[\, \inf(\bar{l}_0, \bar{l}_1),\; \sup(\bar{u}_0, \bar{u}_1) \,\big].$$

Anche in questo caso, le funzioni inf e sup sono da considerarsi estese all'insieme $\bar{\mathbb{Z}}$.

La corrispondenza tra un insieme di valori concreti ed un valore astratto è stabilita dalla funzione di astrazione $\alpha\colon \wp(\mathbb{Z}) \to I$, che opera nel seguente modo:

$$\forall\, S \subseteq \mathbb{Z}, \quad \alpha(S) = \big[\, \bar{l},\, \bar{u} \,\big], \qquad \text{dove } \bar{l} = \inf_{s \in S}(s) \text{ e } \bar{u} = \sup_{s \in S}(s).$$

La funzione di concretizzazione $\gamma\colon I \to \wp(\mathbb{Z})$, associa ad ogni intervallo il corrispondente insieme di valori concreti. Tale funzione opera come segue:

$$\forall\, i = \big[\, \bar{l},\, \bar{u} \,\big] \in I, \quad \gamma(i) = \big\{\, z \in \mathbb{Z} \,.\, \bar{l} \leq z \leq \bar{u} \,\big\}.$$

L'elemento *bottom* $\bot$ è associato, tramite la funzione γ, ad un intervallo privo di elementi (da identificare con l'insieme vuoto), ovvero $\gamma(\bot) = \varnothing$, mentre l'elemento *top* (che corrisponde all'intervallo $\big[\, -\infty,\, +\infty \,\big]$)$\top$ coincide, sempre tramite γ, con l'intero insieme $\mathbb{Z}$, ovvero $\gamma(\top) = \mathbb{Z}$.

Le funzioni lowerBound, upperBound$\colon I \to \bar{\mathbb{Z}}$ associano ad ogni intervallo il corrispondente estremo inferiore $\bar{l}$ o superiore $\bar{u}$.

6.2 Complemento a uno

Sia '$\oslash$' l'operazione astratta di complemento a uno $\oslash\colon I \to I$ definita sul dominio degli intervalli e $i = \big[\, \bar{l},\, \bar{u} \,\big]$ un intervallo in I; allora

$$\oslash i = \big[\, \nsim\, \bar{u},\, \nsim\, \bar{l} \,\big].$$

dove l'operatore $\nsim\colon \bar{\mathbb{Z}} \to \bar{\mathbb{Z}}$ opera come segue:

$$\forall\, \bar{z} \in \bar{\mathbb{Z}}, \qquad \nsim \bar{z} \stackrel{\text{def}}{=} \begin{cases} +\infty & \text{se } \bar{z} = -\infty; \\ -\infty & \text{se } \bar{z} = +\infty; \\ \sim \bar{z} & \text{se } \bar{z} \in \mathbb{Z}. \end{cases}$$

Lo pseudocodice che implementa tale operazione è il seguente:

```
Interval
interval_not(Interval i) {
  if (i.isBottom() || i.isTop())
    return i;
  Interval k;
  k.setLowerBound(~i.getUpperBound());
  k.setUpperBound(~i.getLowerBound());
  return k;
}
```

Tale funzione inizialmente si occupa di controllare che l'inervallo non coincida con $\perp$: in tal caso l'intervallo risultante è anch'esso $\perp$. In seguito si chiamano le funzioni che si occupano di impostare i valori della limitazione inferiore e superiore del risultato. L'operatore ~ utilizzato corriposnde all'implementazione di $\nsim$ definito sugli interi estesi.

6.3 Congiunzione bit-a-bit

Sia '$\owedge$' l'operazione astratta di and bit-a-bit $\owedge\colon I \times I \to I$, definita sul dominio degli intervalli e siano i_0 e i_1 due intervalli in I. Allora

$$i_0 \owedge i_1 = i$$

dove i è l'intervallo ottenuto dalla funzione `interval_and` la cui definizione è riportata in seguito. Tale intervallo deve soddisfare le seguenti proprietà: se $i = \left[\bar{l},\, \bar{u}\right]$, allora

$$\forall\, \bar{z}_0 \in i_0,\ \forall\, \bar{z}_1 \in i_1, \quad \bar{l} \leq (\bar{z}_0 \wedge \bar{z}_1) \text{ e } (\bar{z}_0 \wedge \bar{z}_1) \leq \bar{u}$$

L'algoritmo che riporteremo sarà in grado di calcolare solamente la limitazione superiore dei valori dell'intervallo i; per la limitazione inferiore sfrutteremo la proprietà di De Morgan:

$$\forall\, z_0, z_1 \in \mathbb{Z}, \qquad z_0 \wedge z_1 \stackrel{\text{def}}{=} \sim (\sim z_0 \vee \sim z_1)\,.$$

Lo pseudocodice che implementa l'operazione astratta di and bit-a-bit è il seguente:

```
Interval
interval_and(Interval i0, Interval i1) {
  Interval k.
  if (i0.isBottom() || i1.isBottom()) {
    k.setToBottom();
    return k;
  }
  Interval not_i0(interval_not(i0));
  Interval not_i1(interval_not(i1));
  k.setUpperBound(signed_and_right_limitation(i0, i1));
  k.setLowerBound(signed_or_right_limitation(not_i0, not_i1));
  return k;
}
```

Tale funzione inizialmente si occupa di controllare che uno dei due intervalli non coincida con l'elemento $\perp$: in tal caso l'intervallo risultante è anch'esso $\perp$. In seguito richiama le funzioni che si occupano di determinare la limitazione superiore del risultato nel caso in cui gli operandi siano intervalli di numeri con segno. Questo è corretto in quanto tali funzioni discriminano il segno dei rispettivi intervalli i_0 e i_1 e tale operazione coincide con la valutazione del segno del tipo su cui si sta operando. Non ci resta che definire la funzione `signed_and_right_limitation` che, dati due intervalli i_0 e i_1, restituisce l'elemento $\bar{z} \in \bar{\mathbb{Z}}$ tale che

$$\forall\, \bar{z}_0 \in i_0,\ \forall\, \bar{z}_1 \in i_1, \quad (\bar{z}_0 \wedge \bar{z}_1) \leq \bar{z}$$

Tale funzione si comporta nel seguente modo:

1. Se $i_0 \geq 0$ e
   (a) se l'intervallo $i_1 \geq 0$, allora si deve utilizzare l'algoritmo che gestisce l'and-bit-a-bit degli interi senza segno usando gli stessi intervalli i_0 e i_1. Ricondursi all'algoritmo che gestisce solo interi senza segno (e quindi valori sempre positivi o nulli) è un passaggio corretto in quanto, se gli interi con segno sono positivi o nulli, i loro bit si interpretano nello stesso modo dei bit degli interi senza segno, come descritto nelle formule (5.1), (5.2) e (5.3).
   (b) Se $i_1 \leq -1$, per quanto dimostrato nel teorema 5.3.2, la limitazione superiore del risultato dell'and bit-a-bit è pari all'operando sinistro e quindi $\bar{z}$ sarà sempre inferiore dell'estremo destro di i_0. Quindi $\bar{z} \leq \text{upperBound}(i_0)$.
   (c) Altrimenti $\bar{z} = \text{upperBound}(i_0)$.
2. Se $i_0 \leq -1$ e

   (a) se $i_1 \geq 0$, per quanto dimostrato nel teorema 5.3.3, la limitazione superiore del risultato dell'and bit-a-bit è pari all'operando destro e quindi $\bar{z}$ sarà sempre inferiore dell'estremo destro di i_1. Quindi $\bar{z} \leq \text{upperBound}(i_1)$.

   (b) se $i_1 \leq -1$, allora si deve utilizzare l'algoritmo che gestisce l'and bit-a-bit degli interi senza segno usando gli stessi intervalli i_0 e i_1;

   (c) altrimenti occorre sostituire $\text{lowerBound}(i_1)$ con il valore '0' ed usare l'algoritmo che gestisce l'and bit-a-bit degli interi senza segno con l'intervallo i_0 ed il nuovo intervallo i_1.

3. altrimenti

   (a) se l'intervallo $i_1 \geq 0$, allora $\bar{z} = \text{upperBound}(i_1)$;

   (b) se $i_1 \leq -1$, allora occorre sostituire $\text{lowerBound}(i_0)$ con il valore '0' ed usare l'algoritmo che gestisce l'and bit-a-bit degli interi senza segno con il nuovo intervallo i_0 e l'intervallo i_1;

   (c) altrimenti $\bar{z} = \max(\text{upperBound}(i_0), \text{upperBound}(i_1))$.

Lo pseudocodice che implementa tale funzione è il seguente:

```
Extended_integer
signed_and_right_limitation(Interval i0, Interval i1) {
  if (i0.onlyPositiveValues()) {
    if (i1.onlyPositiveValues())
      if (i0.getUpperBound() != PLUS_INFTY)
        if (i1.getUpperBound() != PLUS_INFTY)
          return unsigned_and_right_limitation(i0, i1);
        else
          return i1.getUpperBound();
      else
        return PLUS_INFTY;
    else
      return i0.getUpperBound();
  }
  else if (i0.onlyNegativeValues()) {
    if (i1.onlyPositiveValues())
      return i1.getUpperBound();
    else if (i1.onlyNegativeValues())
      return unsigned_and_right_limitation(i0, i1);
    else {
      if (i1.getUpperBound() != PLUS_INFTY) {
        Interval h(0, i1.getUpperBound());
        return unsigned_and_right_limitation(i1, h);
```

```
      }
      else
        return PLUS_INFTY;
    }
  }
  else if (i1.onlyPositiveValues())
    return i1.getUpperBound();
  else if (i1.onlyNegativeValues())
    if (i0.getUpperBound() != PLUS_INFTY) {
      Interval h(0, i0.getUpperBound());
      return unsigned_and_right_limitation(h, i1);
    }
    else
      return PLUS_INFTY;
  else
    if (i0.getUpperBound() != PLUS_INFTY
        && i1.getUpperBound() != PLUS_INFTY)
      return max(i0.getUpperBound(), i1.getUpperBound());
    else
      return PLUS_INFTY;
}
```

Si osservi che le guardie che controllano se gli estremi destri degli intervalli coincidono col valore `PLUS_INFTY`, sono state introdotte per garantire che la funzione `unsigned_and_right_limitation` sia sempre eseguita con intervalli il cui estremo destro è un numero intero.

Concludiamo la trattazione dell'operatore 'and *bitwise*' descrivendo l'implementazione di questa ultima funzione, che si occupa di calcolare la limitazione superiore nel caso di operandi senza segno. Tale funzione si comporta nel seguente modo:

1. calcola il valore della posizione del bit più significativo di entrambi i valori upperBound(i_0) e upperBound(i_1): questo è sempre possibile in quanto tali valori non possono essere $+\infty$;

2. si memorizza il massimo dei due valori trovati nel punto 1 in una nuova variabile (`start_position`);

3. si preparano due variabili per memorizzare dei valori di calcolo temporanei, ognuna associata al corrispondente intervallo (`tmp0` e `tmp1`). Tali variabili vanno inizializzate con una sequenza di `start_position` bit settati a 1 se il corrispondente intervallo contiene solo valori negativi, altrimenti le si inizializzano a '0' (ovvero una sequenza di tutti zeri). Questa inizializzazione è necessaria, nonostante sia stato definito che la funzione lavori solo

su valori senza segno. Questo è dovuto al fatto che la funzione è utilizzata anche con valori con segno, come richiesto nell'implementazione della funzione precedente (si osservino i tre casi 2b, 2c e 3b della descrizione di `signed_and_right_limitation`).

4. si esegue un algoritmo ricorsivo che scorre e valuta i bit nelle corrispondenti posizioni di upperBound(i_0) e upperBound(i_1), partendo con i valori temporanei `tmp0` e `tmp1` trovati al punto 3 e con il valore di posizione `start_position` trovato al punto 2. Tale algoritmo funziona nel seguente modo:

   (a) se si sono esaminati tutti i bit senza aver trovato la limitazione superiore, allora tale limitazione è data da upperBound(i_0) $\wedge$ upperBound(i_0);

   (b) altrimenti si estraggono i valori dei bit nella posizione che si sta valutando e si fanno le seguenti considerazioni:

      i. se entrambi i bit sono pari a '0' allora si riesegue ricorsivamente la funzione partendo dal punto 4a, usando un valore di `start_position` decrementato di uno e senza variare i valori delle due variabili temporanee;

      ii. se entrambi i bit sono pari a '1' allora si setta a '1' il bit di entrambe le variabili temporanee nella posizione corrente e si riesegue ricorsivamente la funzione partendo dal punto 4a, usando un valore di `start_position` decrementato di uno e con i nuovi valori delle variabili temporanee;

      iii. altrimenti si consideri la variabile temporanea associata all'intervallo il cui estremo ha '1' nella posizione corrente; si consideri il valore di tale variabile temporanea se si settasse il bit nella posizione corrente con il valore '0' e si mettessero degli '1' nelle rimanenti posizioni sulla destra:

         - se tale valore è contenuto nell'intervallo associato alla variabile temporanea in questione, allora l'estremo destro è dato dall'and bit-a-bit tra la variabile temporanea e l'estremo destro dell'altro intervallo;
         - altrimenti si inserisce un '1' nella posizione corrente della variabile temporanea presa in considerazione al punto 4(b)iii e si riesegue ricorsivamente la funzione partendo dal punto 4a, usando un valore di `start_position` decrementato di uno e con il nuovo valore della variabile temporanea.

Lo pseudocodice che implementa tali funzioni è il seguente:

```
Extended_integer
unsigned_and_right_limitation(Interval i0, Interval i1) {
```

```
  int pos0 = most_significant_bit_position(i0.getUpperBound());
  int pos1 = most_significant_bit_position(i1.getUpperBound());
  int start_position = max(pos0, pos1);
  signed int start_value0 = 0;
  if (i0.getUpperBound() < 0)
    start_value = -pow(2,start_position+1);
  signed int start_value1 = 0;
  if (i1.getUpperBound() < 0)
    start_value = -pow(2,start_position+1);
  return and_limitation(start_value0, start_value1,
                        i0, i1, start_position);
}

Extended_integer
and_limitation(signed int tmp0, signed int tmp1,
               Interval i0, Interval i1, int position) {
  if (position == -1)
    return (i0.getUpperBound() & i1.getUpperBound());
  else {
    int next_position = position-1;
    bool bit0 = (i0.getUpperBound() & pow(2,position)) >> position;
    bool bit1 = (i1.getUpperBound() & pow(2,position)) >> position;
    if (!bit0)
      if (!bit1)
        // bit0 = 0 && bit1 = 0
        return and_limitation(tmp0, tmp1,
                              i0, i1, next_position);
      else {
        // bit0 = 0 && bit1 = 1
        signed int temp = tmp1 | (pow(2,position)-1);
        if (i1.getLowerBound() <= temp
            && temp <= i1.getUpperBound())
          return (temp & i0.getUpperBound());
        else {
          signed_int new_value1 = tmp1 | pow(2,position);
          return and_limitation(tmp0, new_value1,
                                i0, i1, next_position);
        }
    }
    else
      if (bit1) {
        // bit0 = 1 && bit1 = 1
        signed_in new_value0 = tmp0 | pow(2,position);
```

```
        signed_in new_value1 = tmp1 | pow(2,position);
        return and_limitation(new_value0, new_value1,
                              i0, i1, next_position);
      }
      else {
        // bit0 = 1 && bit1 = 0
        signed int temp = tmp0 | (pow(2,position)-1);
        if (i0.getLowerBound() <= temp
            && temp <= i0.getUpperBound())
          return (temp & i1.getUpperBound());
        else {
          signed_int new_value0 = tmp0 | pow(2,position);
          return and_limitation(new_value0, tmp1
                                i0, i1, next_position);
        }
      }
  }
}
```

6.4 Disgiunzione inclusiva bit-a-bit

Sia '$\barwedge$' l'operazione astratta di or bit-a-bit $\barwedge : I \times I \to I$, definita sul dominio degli intervalli e siano i_0 e i_1 due intervalli in I. Allora

$$i_0 \barwedge i_1 = i$$

dove i è l'intervallo ottenuto dalla funzione `interval_or` la cui definizione è riportata in seguito. Tale intervallo deve soddisfare le seguenti proprietà: se $i = \left[\bar{l},\, \bar{u}\right]$, allora

$$\forall\, \bar{z}_0 \in i_0,\ \forall\, \bar{z}_1 \in i_1, \quad \bar{l} \leq (\bar{z}_0 \vee \bar{z}_1) \text{ e } (\bar{z}_0 \vee \bar{z}_1) \leq \bar{u}$$

Come nella sezione 6.3, l'algoritmo che riporteremo sarà in grado di calcolare solamente la limitazione superiore dei valori dell'intervallo i_2; per la limitazione inferiore sfrutteremo la proprietà di De Morgan:

$$\forall\, z_0, z_1 \in S \subseteq \mathbb{Z}, \qquad z_0 \vee z_1 \stackrel{\text{def}}{=} \sim (\sim z_0 \wedge \sim z_1)\,.$$

Lo pseudocodice che implementa l'operazione astratta di and bit-a-bit è il seguente:

```
Interval
interval_or(Interval i0, Interval i1) {
  Interval k;
```

```
  if (i0.isBottom() || i1.isBottom()) {
    k.setToBottom();
    return k;
  }
  Interval not_i0(interval_not(i0));
  Interval not_i1(interval_not(i1));
  k.setUpperBound(signed_or_right_limitation(i0, i1));
  k.setLowerBound(signed_and_right_limitation(not_i0, not_i1));
  return k;
}
```

Anche in questo caso l'or bit-a-bit di due intervalli risulta essere $\perp$ nel caso in cui uno dei due intervalli sia $\perp$. Si tratta ora di definire la funzione `signed_or_right_limitation` che, dati due intervalli i_0 e i_1, restituisce l'elemento $\bar{z} \in \bar{\mathbb{Z}}$ tale che

$$\forall\ \bar{z}_0 \in i_0,\ \forall\ \bar{z}_1 \in i_1, \quad (\bar{z}_0 \vee \bar{z}_1) \leq \bar{z}$$

Iniziamo con la funzione che gestisce gli interi con segno. Tale funzione si comporta nel seguente modo:

1. Se $i_0 \geq 0$ e
   (a) se $i_1 \geq 0$, allora si deve utilizzare l'algoritmo che gestisce l'or bit-a-bit degli interi senza segno usando gli stessi intervalli i_0 e i_1. Ricondursi all'algoritmo che gestisce solo interi senza segno (e quindi valori sempre positivi o nulli) è un passaggio corretto in quanto, se gli interi con segno sono positivi o nulli, i loro bit si interpretano nello stesso modo dei bit degli interi senza segno, come descritto nelle formule (5.1), (5.2) e (5.3).
   (b) se $i_1 \leq -1$, allora $\bar{z} \leq -1$;
   (c) altrimenti occorre sostituire lowerBound(i_1) con il valore '0' ed usare l'algoritmo che gestisce l'or bit-a-bit degli interi senza segno con l'intervallo i_0 ed il nuovo intervallo i_1.
2. Se $i_0 \leq -1$ e
   (a) se $i_1 \geq 0$, per quanto dimostrato nel teorema 5.4.3, la limitazione superiore del risultato dell'and bit-a-bit è pari al valore -1 quindi $\bar{z}$ sarà sempre inferiore a tale valore, ovvero $\bar{z} \leq -1$.
   (b) se $i_1 \leq -1$, allora si deve utilizzare l'algoritmo che gestisce l'or bit-a-bit degli interi senza segno usando gli stessi intervalli i_0 e i_1;
   (c) altrimenti $\bar{z} = -1$.
3. altrimenti

(a) se $i_1 \geq 0$, allora occorre sostituire lowerBound(i_0) con il valore '0' ed usare l'algoritmo che gestisce l'or bit-a-bit degli interi senza segno con il nuovo intervallo i_0 e l'intervallo i_1;

(b) se $i_1 \leq -1$, allora $\bar{z} = -1$;

(c) altrimenti occorre sostituire i valori lowerBound(i_0) e lowerBound(i_1) con il valore '0' ed usare l'algoritmo che gestisce l'or bit-a-bit degli interi senza segno con i nuovi intervalli i_0 e i_1.

Lo pseudocodice che implementa tale funzione è il seguente:

```
Extended_integer
signed_or_right_limitation(Interval i0, Interval i1) {
  if (i0.onlyPositiveValues()) {
    if (i1.onlyPositiveValues())
      if (i0.getUpperBound() != PLUS_INFTY
          && i1.getUpperBound() != PLUS_INFTY)
        return unsigned_or_right_limitation(i0, i1);
      else
        return PLUS_INFTY;
    else if (i1.onlyNegativeValues())
      return (Extended_integer)-1;
    else
      if (i0.getUpperBound() != PLUS_INFTY
          && i1.getUpperBound() != PLUS_INFTY) {
        Interval k(0, i1.getUpperBound());
        return unsigned_or_right_limitation(i0, k);
      }
      else
        return PLUS_INFTY;
  }
  else if (i0.onlyNegativeValues()) {
    if (i1.onlyNegativeValues)
      return unsigned_or_right_limitation(i0, i1);
    else
      return (Extended_integer)-1;
  }
  else {
    Interval k(0, i0.getUpperBound());
    Interval h(0, i1.getUpperBound());
    if (i1.onlyPositiveValues())
      if (i0.getUpperBound() != PLUS_INFTY
          && i1.getUpperBound() != PLUS_INFTY)
        return unsigned_or_right_limitation(k, i1);
```

```
      else
        return PLUS_INFTY;
    else {
      if (i1.onlyNegativeValues())
        return (Extended_integer)-1;
      else
        if (i0.getUpperBound() != PLUS_INFTY
            && i1.getUpperBound() != PLUS_INFTY)
          return unsigned_or_right_limitation(k, h);
        else
          return PLUS_INFTY;
    }
  }
}
```

Anche in questo caso, le guardie che controllano se gli estremi destri degli intervalli coincidono col valore `PLUS_INFTY` servono per garantire che la funzione `unsigned_or_right_limitation` sia sempre eseguita con intervalli il cui estremo destro è un numero intero. Tale funzione si occupa di trovare la limitazione superiore nel caso di operando senza segno e si comporta esattamente come la `unsigned_and_right_limitation`, tranne nel chiamare la funzione ricorsiva `or_limitation`, che si comporta nel seguente modo:

1. se si sono esaminati tutti i bit senza aver trovato la limitazione superiore, allora tale limitazione è data da upperBound(i_0) $\vee$ upperBound(i_0);
2. altrimenti si estraggono i valori dei bit nella posizione che si sta valutando e si fanno le seguenti considerazioni:
   (a) se entrambi i bit sono pari a '0' allora si riesegue ricorsivamente la funzione partendo dal punto 1, usando un valore di `start_position` decrementato di uno e senza variare i valori delle due variabili temporanee;
   (b) se entrambi i bit sono pari a '1' allora si considerino, a turno, entrambe le variabili temporanee; si inseriscano degli '1' in tutte le posizioni a destra della posizione corrente: se tale valore è contenuto nell'intervallo associato alla variabile temporanea in questione, allora l'estremo destro è dato dall'or bit-a-bit tra la variabile temporanea e l'estremo destro dell'altro intervallo. Se questo non si verifica per entrambe le variabili, allora si setta a '1' il bit nella posizione corrente di entrambe le variabili temporanee e si si riesegue ricorsivamente la funzione partendo dal punto 1, usando un valore di `start_position` decrementato di uno e con i nuovi valori delle due variabili temporanee;
   (c) se i bit sono di valore diverso occorre settare un '1' nella posizione corrente della variabile temporanea corrispondente all'intervallo il cui

estremo destro ha l''1' e rieseguire ricorsivamente la funzione partendo dal punto 1, usando un valore di `start_position` decrementato di uno e con una delle due variabili temporanee modificata.

Lo pseudocodice che implementa tale funzione è il seguente:

```
Extended_integer
or_limitation(signed int tmp0, signed int tmp1,
              Interval i0, Interval i1, int position) {
  if (position == -1)
    return (i0.getUpperBound() | i1.getUpperBound());
  else {
    int next_position = position-1;
    bool bit0 = (i0.getUpperBound() & pow(2^position)) >> position;
    bool bit1 = (i1.getUpperBound() & pow(2^position)) >> position;
    if (!bit0) {
      if (!bit1)
        // bit0 = 0 && bit1 = 0
        return or_limitation(tmp0, tmp1,
                             i0, i1, next_position);
      else {
        // bit0 = 0 && bit1 = 1
        signed int new_value1 = tmp1 | pow(2,position);
        return or_limitation(tmp0, new_value1,
                             i0, i1, next_position);
      }
    }
    else
      if (bit1) {
        // bit0 = 1 && bit1 = 1
        signed int temp0 = tmp0 | (pow(2,Position)-1);
        if (i0.getLowerBound() <= temp0
            && temp0 <= i0.getUpperBound())
          return (temp1 | i1.getUpperBound());
        else {
          signed int temp1 = tmp1 | (pow(2,position)-1);
          if (i1.getLowerBound() <= temp1
              && temp1 <= i1.getUpperBound())
            return (temp1 | io.getUpperBound());
          else {
            signed int new_value0 = tmp0 | pow(2,position);
            signed int new_value1 = tmp1 | pow(2,position);
            return or_limitation(new_value0, new_value1,
                                 i0, i1, next_position);
```

```
        }
      }
    }
    else {
      // bit0 = 1 && bit1 = 0
      signed int new_value_0 = tmp0 | pow(2,position);
      return or_limitation(new_value_0, tmp1,
                           i0, i1, next_position);
    }
}
```

6.5 Disgiunzione esclusiva bit-a-bit

Sia '$\otimes$' l'operazione astratta di or esclusivo bit-a-bit $\otimes\colon I \times I \to I$, definita sul dominio degli intervalli e siano i_0 e i_1 due intervalli in I. Allora

$$i_0 \otimes i_1 = i_2$$

dove i_2 è l'intervallo ottenuto dalla funzione `interval_xor` nella cui implementazione si utilizza una definizione alternativa dell'operatore 'or esclusivo' per sfruttare le già implementate funzioni in grado di gestire il calcolo delle limitazioni dell'and e dell'or bit-a-bit. Consideriamo quindi la seguente definizione di or esclusivo:

$$\forall\, z_0, z_1 \in \mathbb{Z}, \qquad z_0 \veebar z_1 \stackrel{\text{def}}{=} (z_0 \wedge \sim z_1) \vee (\sim z_0 \wedge z_1).$$

Lo pseudocodice è quindi

```
Interval
interval_xor(Interval i0, Interval i1) {
  if (i0.isBottom() || i1.isBottom()) {
    Interval k.
    k.setToBottom();
    return k;
  }
  Interval not_i0(interval_not(i0));
  Interval not_i1(interval_not(i1));
  Interval left(interval_and(i0, not_i1));
  Interval right(interval_and(not_i0, i1));
  return interval_or(left, right);
}
```

6.6 Scorrimento a sinistra

Sia '$\circledcirc$' l'operazione astratta di or bit-a-bit $\circledcirc\colon I \times I \to I$, definita sul dominio degli intervalli e siano i_0 e i_1 due intervalli in I. Allora

$$i_0 \circledcirc i_1 = i$$

dove i è l'intervallo ottenuto dalla funzione `interval_lshift` la cui definizione è riportata in seguito. Tale intervallo deve soddisfare le seguenti proprietà: se $i = \left[\bar{l},\, \bar{u}\right]$, allora

$$\forall\, \bar{z}_0 \in i_0,\ \forall\, \bar{z}_1 \in i_1, \quad \bar{l} \leq (\bar{z}_0 \ll \bar{z}_1) \text{ e } (\bar{z}_0 \ll \bar{z}_1) \leq \bar{u}$$

Descriviamo ora il comportamento della funzione che gestisce lo *shift* sinistro.

1. se uno dei due intervalli coincide con l'elemento *bottom*, allora lo scorrimento a sinistra restituisce un intervallo che coincide con il medesimo valore $\perp$;

2. se l'intervallo corrispondente all'operando destro contiene valori negativi, allora il risultato corrisponde con l'elemento *top* del dominio degli intervalli. Da questo punto in poi possiamo assumere che l'intervallo i_1 contiene solo interi positivi;

3. se il tipo dell'operando sinistro è senza segno oppure è con segno, il valore è non negativo ed il risultato è rappresentabile nel tipo del risultato, allora possiamo approssimare il risultato dello scorrimento a sinistra con i seguenti valori:

$$\bar{l} = \text{lowerBound}(i_0) \ll_z \text{lowerBound}(i_1)$$
$$\bar{u} = \text{upperBound}(i_0) \ll_z \text{upperBound}(i_1).$$

   Altrimenti il risultato dello scorrimento a sinistra coincide con l'elemento *top* del dominio degli intervalli.

Con l'operatore $\ll_z$ si intende l'ovvia estensione dell'operazione di scorrimento su $\bar{\mathbb{Z}}$. Segue lo pseudocodice che implementa l'algoritmo appena descritto.

```
Interval
interval_lshift(Interval i0, type i0type, Interval i1) {
  Interval k;
  if (i0.isBottom() || i1.isBottom()) {
    k.setToBottom();
    return k;
  }
  unsigned int i0size = syzeof(i0type) * 8;
  unsigned int overflow_limit = i0size - msb(i0.getUpperBound);
```

```
  if (!i1.onlyPositiveValues()
      || (bounded_type(i0type)
          && i1.getUpperBound() >= i0size))
      return k.setToTop();
  if (unsigned_type(i0type)
      || (i0.onlyPositiveValues()
          && (unbounded_type(i0type)
              || i1.getUpperBound() >= i0size - overflow_limit))) {
    k.setLowerBound(i0.getLowerBound() << i1.getLowerBound());
    if (i0.getUpperBound() == PLUS_INFTY
        || << i1.getUpperBound() == PLUS_INFTY)
      k.setUpperBound(PLUS_INFTY);
    else
      k.setUpperBound(i0.getUpperBound() << i1.getUpperBound());
  }
  else
     k.setToTop();
  return k;
}
```

Nel calcolo dell'estremo sinistro dell'intervallo i (l'intervallo restituito dall'intera funzione), si utilizzano le limitazioni sinistre di entrambi gli intervalli i_0 e i_1: tali valori sono sempre numeri interi, in quanto i test eseguiti in precedenza garantiscono che in quel punto del codice i due intervalli sono chiusi a sinistra. Questo non vale per il calcolo dell'estremo destro di i ed occorre introdurre un test che controlli che i valori utilizzati per il calcolo siano effettivamente dei numeri interi.

6.7 Scorrimento a destra

Sia '$\circledcirc$' l'operazione astratta di or bit-a-bit $\circledcirc\colon I \times I \to I$, definita sul dominio degli intervalli e siano i_0 e i_1 due intervalli in I. Allora

$$i_0 \circledcirc i_1 = i$$

dove i è l'intervallo ottenuto dalla funzione `interval_rshift` la cui definizione è riportata in seguito. Tale intervallo deve soddisfare le seguenti proprietà: se $i = \left[\, \bar{l},\, \bar{u} \,\right]$, allora

$$\forall\, \bar{z}_0 \in i_0,\ \forall\, \bar{z}_1 \in i_1, \quad \bar{l} \leq (\bar{z}_0 \gg \bar{z}_1) \text{ e } (\bar{z}_0 \gg \bar{z}_1) \leq \bar{u}$$

Descriviamo ora il comportamento della funzione che gestisce lo *shift* destro.

1. se uno dei due intervalli coincide con l'elemento *bottom*, allora lo scorrimento a destra restituisce un intervallo che coincide con il medesimo valore $\perp$;

2. se l'intervallo corrispondente all'operando destro contiene valori negativi, allora il risultato corrisponde con l'elemento *top* del dominio degli intervalli. Da questo punto in poi possiamo assumere che l'intervallo i_1 contiene solo interi positivi;

3. se il tipo dell'operando sinistro è limitato, allora occorre verificare che i valori dell'operando destro siano tutti inferiori alla lunghezza della rappresentazione binaria dell'operando sinistro; se questo non si verifica allora il risultato dello *shift* corrisponde con l'elemento *top* del dominio degli intervalli;

4. se il tipo dell'intervallo i_0 è senza segno oppure ha tipo con segno e valore non negativo allora possiamo approssimare il risultato dello scorrimento a destra con i seguenti valori:

$$\bar{l} = \text{lowerBound}(i_0) \gg_z \text{upperBound}(i_1)$$
$$\bar{u} = \text{upperBound}(i_0) \gg_z \text{lowerBound}(i_1).$$

   Altrimenti il risultato dello scorrimento a destra coincide con l'elemento *top* del dominio degli intervalli.

Con l'operatore $\gg_z$ si intende l'ovvia estensione dell'operazione di scorrimento su $\bar{\mathbb{Z}}$. Segue lo pseudocodice che implementa l'algoritmo appena descritto.

```
Interval
interval_rshift(Interval i0, type i0type, Interval i1) {
  Interval k;
  if (i0.isBottom() || i1.isBottom()) {
    k.setToBottom();
    return k;
  }
  unsigned int i0size = syzeof(i0type) * 8;
  if (!i1.onlyPositiveValues()
      || (bounded_type(i0type)
          && i1.getUpperBound() >= i0size))
      return k.setToTop();
  if (unsigned_type(i0type)
      || i0.onlyPositiveValues()) {
    if (i1.getUpperBound() == PLUS_INFTY)
      k.setLowerBound((Extended_integer) 0);
    else
      k.setLowerBound(i0.getLowerBound() >> i1.getUpperBound());
    if (i0.getUpperBound() == PLUS_INFTY)
```

```
      k.setLowerBound(PLUS_INFTY);
    else
      k.setUpperBound(i0.getUpperBound() >> i1.getLowerBound());
  }
  else
    k.setToTop();
  return k;
}
```

Anche in questo caso, nel momento in cui si eseguono dei calcoli per trovare il valore degli estremi dell'intervallo i, abbiamo la garanzia che i due intervalli i_0 e i_1 sono chiusi a sinistra. Per questo motivo, per essere sicuri che le operazioni di scorrimento siano sempre eseguite su numeri interi, sono stati introdotti controlli sul valore dell'estremo destro di entrambi gli intervalli.

Capitolo 7

Conclusioni

Realizzare l'interpretazione astratta di un elemento o un componente di un generico linguaggio imperativo (come un tipo di dato strutturato o la gestione delle definizioni e chiamate di funzioni) significa dover definire una serie di elementi in stretta correlazione tra loro. In questa tesi gli elementi al centro della trattazione sono gli operatori aritmetici per la manipolazione di bit.

Innanzitutto sono state definite tutte le caratteristiche degli elementi da analizzare: il primo passo è stato quello di valutare quali specifiche considerare per la semantica degli operatori. Si è dovuto definire quali comportamenti si vogliono implementare ed anche in quali circostanze questi si verificano. Il considerare diversi tipi di numeri interi è stata, per esempio, un'esigenza dovuta alla scelta di implementare i comportamenti defini dallo standard C in [4]: nel caso degli operatori di scorrimento, infatti, la semantica è molto legata al tipo di dato utilizzato per le operazioni.

Gli operatori per la manipolazione di bit sono stati poi inseriti in un linguaggio, chiamto CMM. La capacità espressiva di tale linguaggio è stata limitata al minimo indispensabile per concentrare l'attenzione sui meccanismi più legati all'approssimazione degli operatori. Infatti, i costrutti di programmazione più significativi che sono stati considerati (controllo e iterazione) eseguono una valutazione di un'espressione, che può per definizione contenere operatori *bitwise*. Dato che secondo lo standard C ci sono circostanze per cui con certi valori numerici le operazioni, o il loro risultato, sono prive di signifacato, si è dovuti introdurre un, seppur semplice, meccanismo di generazione e propagazione degli errori che avvengono a tempo di esecuzione.

Dopo aver definito tutti gli elementi del linguaggio, è stato descritto il modello scelti per approssimare i comportamenti del concreto definiti nelle fasi precedenti. In questa circostanza si è scelto di far corrispondere ad ogni dominio di elementi concreti (valori numerici, eccezioni, operazioni,...) un dominio composto da elementi definiti come astratti (ovvero rappresentanti

un insieme di valori o una proprietà). La corrispondenza tra le coppie di domini è stata realizzata definendo una coppia di funzioni secondo il modello proposto in [6]: una funzione di 'astrazione' ed una di 'concretizzzione'. Inoltre, si è anche specificato quando questi elementi astratti sono considerabili come un'approssimazione corretta degli elementi concreti.

Sono state studiate le proprietà aritmetiche degli operatori $\sim$, $\wedge$, $\vee$ e $\veebar$. Tuttavia è possibile estendere (e dimostrare) l'insieme di tali proprietà aggiungendo, ad esempio, che per ogni $m \in \mathbb{Z}$, $\sim m = -m - 1$ oppure, che per ogni $x, y \in \mathbb{Z}$, $x \veebar y = 0 \Leftrightarrow x = y$. Inoltre non è stata definita alcuna proprietà aritmetica per gli operatori di scorrimento.

Dopo aver definito quali sono le esigenze teoriche per reliazzare un'approssimazione corretta, stati riportati algoritmi che utilizzano gli intervalli di interi come dominio di analisi. Tali algoritmi implementano un comportamento corretto ma manca una dimostrazione formale della loro precisione.

Bibliografia

[1] R. Bagnara, P. M. Hill, A. Pescetti, and E. Zaffanella. On the design of generic static analyzers for modern imperative languages. Quaderno, Dipartimento di Matematica, Università di Parma, Italy, 2007. In corso di pubblicazione.

[2] International Business Machines Corporation. Range analysis with bitwise operations. Distributed by `http://www.ip.com`, 2003.

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In B. Robinet, editor, *Proceedings of the Second International Symposium on Programming*, pages 106–130, Paris, France, 1976. Dunod, Paris, France.

[4] Information Technology Industry Council (ITI). Programming languages — c, 1999.

[5] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Compiler Construction: Proceedings of the 11th International Conference (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, Grenoble, France, 2002. Springer-Verlag, Berlin.

[6] D. A. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In A. Mycroft, editor, *Static Analysis: Proceedings of the 2nd International Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 1–18, Glasgow, UK, 1995. Springer-Verlag, Berlin.