UNIVERSITÀ DEGLI STUDI DI PARMA
Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Tesi di Laurea in Analisi e Verifica del Software

Un dominio astratto per l'analisi dei calcoli in virgola mobile

An abstract domain for the analysis of floating point computations

Candidato Fabio Bossi
Relatore Prof. Roberto Bagnara

Anno Accademico 2009/2010

Riassunto

L'aritmetica in virgola mobile basata sullo standard IEEE754 è ampiamente utilizzata nel software grazie alla sua implementazione efficiente su tutti i processori moderni; tuttavia, il suo utilizzo è una frequente causa di errori per conseguenza dell'inesattezza dei risultati e della possibilità di overflow, underflow e operazioni invalide. Questi errori possono talvolta passare inosservati anche da parte di persone che possiedono una solida conoscenza dei principi del calcolo numerico.

Lo scopo di questo lavoro è descrivere l'approccio adottato dall'analizzatore ECLAIR allo scopo di poter analizzare i calcoli in virgola mobile e rilevare una buona parte degli errori che ricadono in questa categoria. ECLAIR è un analizzatore statico per i linguaggi C e C++ basato sui fondamenti teorici dell'interpretazione astratta. I domini astratti utilizzati in questo approccio, così come le operazioni di base che li manipolano, sono per la maggior parte poliedri implementati dalla libreria *Parma Polyhedra Library* (abbreviata in PPL). Fra queste operazioni, alcune sono progettate appositamente per l'analisi dei calcoli in virgola mobile e sono state originariamente concepite da Antoine Miné (Università di Versailles) ed implementate nell'analizzatore ASTRÉE. L'implementazione di tali operazioni all'interno della PPL è concettualmente identica ed è stata realizzata dall'autore e da altri due studenti (Roberto Amadini e Fabio Biselli) nel contesto di un progetto d'esame per il corso di *Analisi e Verifica del Software* tenuto dal prof. Roberto Bagnara.

Al termine dell'opera verranno forniti alcuni esempi di programmi analizzati assieme ai relativi risultati ottenuti da ECLAIR.

Abstract

Floating point arithmetics based on the IEEE754 standard is vastly used by programs due to its efficient implementation on all modern processors; however, its use is also very error prone due to inexactness of results as well as the the possibility of overflows, underflows and invalid operations. These mistakes can occasionally go unnoticed even by people who have a solid knowledge of numerical calculus.

The purpose of this work is to describe the approach used by the ECLAIR analyzer in order to analyze floating point computations and detect many errors that fall into this category. ECLAIR is a fully-featured static analyzer for the C and C++ languages that is based on the theoretical foundations of abstract interpretation. The abstract domains used in this approach, as well as the basic operations that manipulate them, are mostly polyhedral domains implemented by the *Parma Polyhedra Library* (PPL in short). Among these operations, some are specifically designed for floating point computations' analysis and were originally conceieved by Antoine Miné (University of Versailles) and implemented in the ASTRÉE analyzer. The PPL implementation of these operations is conceptually identical and was written by the author and two other students (Roberto Amadini and Fabio Biselli) as an exam project for the *Analisi e Verifica del Software* course held by prof. Roberto Bagnara.

Finally, we will provide a few analysis examples along with their results.

Indice

1 Aritmetica in virgola mobile **1**
- 1.1 Formati in virgola mobile IEEE754 . . . 1
- 1.2 Altri formati in virgola mobile . . . 3
- 1.3 Operazioni in virgola mobile . . . 3
- 1.4 Semantica concreta delle espressioni in virgola mobile . . . 4
- 1.5 Altre operazioni su quantità in virgola mobile . . . 7
- 1.6 Problemi dell'aritmetica in virgola mobile . . . 7

2 Analisi delle computazioni in virgola mobile **9**
- 2.1 Richiami di interpretazione astratta . . . 9
  - 2.1.1 Concetti base . . . 9
  - 2.1.2 Astrarre il flusso di controllo . . . 10
- 2.2 Analisi basata sugli intervalli . . . 11
  - 2.2.1 Semantica astratta basata sugli intervalli . . . 11
  - 2.2.2 Analisi basata sugli intervalli . . . 14
  - 2.2.3 Limiti dell'analisi basata sugli intervalli . . . 15
- 2.3 Linearizzazione delle espressioni in virgola mobile . . . 16
  - 2.3.1 Forme lineari ad intervalli . . . 16
  - 2.3.2 Algoritmo di linearizzazione . . . 17
  - 2.3.3 Strategie di intervallizzazione . . . 19
  - 2.3.4 Propagazione delle forme lineari . . . 20
- 2.4 Analisi con domini relazionali . . . 20
  - 2.4.1 Ottagoni . . . 20
  - 2.4.2 Raffinamento fra i diversi domini astratti . . . 22
  - 2.4.3 Differenze vincolate . . . 23
  - 2.4.4 Poliedri generici . . . 23

3 Dettagli implementativi **25**
- 3.1 Formati in virgola mobile . . . 25
- 3.2 Nuove funzioni della PPL per l'analisi dei calcoli in virgola mobile . . . 25
- 3.3 Interfacciare la PPL ed ECLAIR . . . 27
- 3.4 Implementazione dell'analisi basata sugli intervalli . . . 29
- 3.5 Implementazione dell'analisi basata su poliedri . . . 32

4 Esempi di analisi **34**
4.1 Rate Limiter 34
4.1.1 Rate limiter annotato 34
4.1.2 Correttezza del rate limiter 36
4.1.3 Risultati sperimentali 37
4.2 Limiti delle differenze vincolate 45
4.3 Limiti del widening 48

5 Conclusioni **51**
5.1 Stato attuale 51
5.2 Sviluppi futuri 52

Organizzazione

Capitolo 1. Richiama i concetti fondamentali dell'aritmetica in virgola mobile.

Capitolo 2. Spiega com'è possibile effettuare un'analisi consistente dei più comuni calcoli in virgola mobile applicando l'approccio dell'interpretazione astratta e senza abbandonare i tradizionali domini astratti poliedrici.

Capitolo 3. Fornisce alcuni dettagli riguardanti l'applicazione dei concetti presentati nel capitolo precedente all'interno della PPL e di ECLAIR.

Capitolo 4. Presenta alcuni esempi di analisi assieme ai risultati ottenuti sperimentalmente da ECLAIR.

Capitolo 5. Trae alcune brevi conclusioni e descrive i possibili sviluppi futuri.

1 Aritmetica in virgola mobile

1.1 Formati in virgola mobile IEEE754

Un *formato in virgola mobile* (detto anche *formato floating point*) **f** compatibile con lo standard IEEE754 (si veda [46108]) è definito da un numero intero $\mathbf{p_f}$ detto la base del formato (considereremo solo i formati in base 2 tranne quando esplicitamente dichiarato), dal numero di bit previsti per la mantissa $\mathbf{p_f}$, dal numero di bit per l'esponente $\mathbf{e_f}$ e dal *bias* (o spiazzamento) dell'esponente $\mathbf{bias_f}$. Ricordiamo che la rappresentazione interna di un tale numero è costituita da un bit di segno s seguito da due interi senza segno e ed f che denotano l'esponente (senza spiazzamento) e la mantissa, in questo preciso ordine.

Questa è la lista dei formati in base 2 definiti da IEEE 754-2008, ovvero l'ultima versione di IEEE754 che ha sostituito (e in buona parte inglobato) la precedente revisione IEEE 754-1985:

- Il formato a *mezza precisione* **h**, dove $\mathbf{p_h} = 10$, $\mathbf{e_h} = 5$ e $\mathbf{bias_h} = 15$.
- Il formato a *singola precisione* **s**, dove $\mathbf{p_s} = 23$, $\mathbf{e_s} = 8$ e $\mathbf{bias_s} = 127$.
- Il formato a *doppia precisione* **d**, dove $\mathbf{p_d} = 52$, $\mathbf{e_d} = 11$ e $\mathbf{bias_d} = 1023$.
- Il formato a *quadrupla precisione* **q**, dove $\mathbf{p_q} = 112$, $\mathbf{e_q} = 15$ e $\mathbf{bias_q} = 16383$.

Dato un numero in virgola mobile di formato **f** in base 2 (con opportuni valori per s, e ed f), il suo valore in $\mathbb{R}$ è dato da:

- $(-1)^s \times 2^{e-\mathbf{bias_f}} \times 1.f$, se $1 \leq e \leq 2^{\mathbf{e_f}} - 2$. Questi sono detti *numeri normalizzati.*
- $(-1)^s \times 2^{1-\mathbf{bias_f}} \times 0.f$, se $e = 0$ e $f \neq 0$. Questi sono detti *numeri denormalizzati* (in realtà IEEE 754-2008 ha cambiato il loro nome in *numeri subnormali*, ma qui continueremo ad utilizzare la vecchia denominazione).
- Uno dei due *valori nulli* $+0$ o -0 (a seconda del valore di s), se $e = 0$ e $f = 0$. Poiché questi due valori sono identici per gli scopi delle nostre analisi, indicheremo semplicemente con 0 un generico valore nullo.

Un numero in virgola mobile può anche assumere alcuni valori speciali al di fuori di $\mathbb{R}$; in particolare:

- Uno dei due *infiniti* $+\infty$ o $-\infty$ (a seconda del valore di s), se $e = 2^{\mathbf{e_f}} - 1$ e $f = 0$;
- Un *codice d'errore*, se $e = 2^{\mathbf{e_f}} - 1$ e $f \neq 0$. Un codice d'errore viene solitamente denotato con NaN (Not a Number). I NaN possono essere di due tipi: i *signaling NaN* provocano un errore a tempo d'esecuzione quando vengono generati come risultato di una computazione, mentre al contrario i *quiet NaN* (quelli che sono normalmente generati in C e C++) non interrompono il normale flusso d'esecuzione. Per semplicità, in questo lavoro considereremo solo i quiet NaN (salvo ove esplicitamente indicato).

Indicheremo l'insieme dei numeri in virgola mobile di formato **f** (inclusi gli infiniti ma esclusi i NaN) con $\mathbb{F}_{\mathbf{f}}$.

Dato un formato in virgola mobile **f**, è facile notare che il più piccolo numero positivo in $\mathbb{F}_{\mathbf{f}}$ è pari a $2^{1-\mathbf{bias_f}-\mathbf{p_f}}$; tale numero verrà denotato con $\mathrm{mf}_{\mathbf{f}}$. Allo stesso modo, denoteremo il più grande numero positivo finito $(2 - 2^{-\mathbf{p_f}}) \times 2^{\mathbf{e_f}-\mathbf{bias_f}-2}$ con $\mathrm{Mf}_{\mathbf{f}}$.

1.2 Altri formati in virgola mobile

Lo standard IEEE 754-2008 definisce anche tre formati in base 10, i quali tuttavia sono (perlomeno fino ad oggi) decisamente meno usati rispetto alle loro controparti in base 2.

Esistono inoltre diversi formati floating point che non definiti dallo standard IEEE754. Molti di questi sono specifici per certe piattaforme, per esempio i tre formati in base 16 definiti dalla IBM Floating Point Architecture e utilizzati nei mainframe con architettura System/360, ma ne esistono anche alcuni specifici per determinate applicazioni come i minifloat utilizzati nello standard di codifica audio G.771.

1.3 Operazioni in virgola mobile

Il risultato dell'applicazione di un'operazione aritmetica di base su due numeri in virgola mobile aventi valori in $\mathbb{R}$ è ottenuto applicando la corrispondente operazione esatta sui reali seguita da una *funzione di arrotondamento* $R_{\mathbf{f},\mathbf{r}}$ che mappa il risultato esatto in un risultato possibilmente inesatto in $\mathbb{F}_{\mathbf{f}}$. La funzione di arrotondamento dipende dal formato $\mathbf{f}$ e dalla *modalità di arrotondamento* $\mathbf{r}$ scelti. Se il risultato finale è infinito, si dice che l'operazione va in *overflow* sugli argomenti dati; allo stesso modo, se il risultato dell'operazione esatta nei reali è diverso da 0 ma la funzione d'arrotondamento restituisce 0 quando applicata ad esso, si dice che l'operazione va in *underflow* su tali argomenti.

Lo standard IEEE754 definisce quattro modalità di arrotondamento: *round up* (arrotonda sempre per eccesso), *round down* (arrotonda sempre per difetto), *round towards 0* (arrotonda i numeri negativi per eccesso e quelli positivi per difetto) e *round to nearest* (arrotonda ogni numero in $\mathbb{R}$ al più vicino numero in $\mathbb{F}_{\mathbf{f}}$, con un'eccezione per i numeri molto alti in valori assoluto che vengono arrotondati al più vicino infinito). Tali modalità d'arrotondamento verranno indicate rispettivamente con $+\infty$, $-\infty$, $\mathbf{0}$ e $\mathbf{n}$, cosicché le rispettive funzioni d'arrotondamento per il formato $\mathbf{f}$ saranno indicate con $R_{\mathbf{f},+\infty}$, $R_{\mathbf{f},-\infty}$, $R_{\mathbf{f},\mathbf{0}}$ e $R_{\mathbf{f},\mathbf{n}}$. Si faccia riferimento alla sezione 1.4 per una definizione più precisa e formale.

Un risultato infinito può anche essere generato in casi speciali come l'addizione fra un numero finito e un infinito o la divisione per zero di un numero finito diverso da zero (nel qual caso il segno dell'infinito dipende da quello del dividendo). Tutte le forme indeterminate danno invece luogo a dei NaN, così come qualsiasi operazione alla quale viene dato un argomento NaN (perciò si dice che i NaN "si propagano").

Le quattro operazioni di base sul formato $\mathbf{f}$, comprendenti l'arrotondamento con la modalità $\mathbf{r}$, saranno denotate con $\oplus_{\mathbf{f},\mathbf{r}}$, $\ominus_{\mathbf{f},\mathbf{r}}$, $\otimes_{\mathbf{f},\mathbf{r}}$ e $\oslash_{\mathbf{f},\mathbf{r}}$. Il simbolo $\ominus_{\mathbf{f},\mathbf{r}}$ sarà anche utilizzato per denotare il meno unario (sebbene questo di fatto non richieda un arrotondamento).

1.4 Semantica concreta delle espressioni in virgola mobile

Diamo ora una semantica concreta formale per le espressioni in virgola mobile di formato $\mathbf{f}$ (o quantomeno per un loro ampio sottoinsieme). Nello stesso programma possono ovviamente coesistere espressioni aventi diversi formati in virgola mobile (il che giustifica la presenza dei cast). Anche la modalità di arrotondamento $\mathbf{r}$ può essere cambiata dinamicamente durante l'esecuzione del programma (in C, questo viene normalmente avviene effettuando una chiamata alla funzione `fegetround()` definita nell'header file `fenv.h` della libreria standard).

Denoteremo l'insieme delle variabili aventi un tipo in virgola mobile (dette anche semplicemente *variabili in virgola mobile*) di formato $\mathbf{f}$ con $\mathcal{V}_{\mathbf{f}}$. Definiamo uno store (concreto) ρ come una funzione che associa ad ogni variabile in virgola mobile v avente formato $\mathbf{f}$ il suo valore in memoria (appartenente a $\mathbb{F}_{\mathbf{f}}$) a tempo d'esecuzione. Avvisiamo che tali variabili nella nostra accezione non corrispondono alle variabili esplicitamente dichiarate dal programmatore: dovrebbero invece essere interpretate come locazioni di memoria contenenti un valore in virgola mobile (che possono anche essere associate a più di una o a nessuna variabile esplicitamente dichiarata; si pensi all'utilizzo dei riferimenti in C++).

Una generica espressione (concreta) in virgola mobile avente formato $\mathbf{f}$ verrà denotata con $\text{expr}_{\mathbf{f}}$ (tali espressioni potranno anche venire indicate con e_1, e_2 ecc. nel caso in cui avessimo la necessità di indicarne più di una nel medesimo contesto; in tali casi, il formato utilizzato sarà sempre facilmente deducibile dal contesto). I generi d'espressioni in virgola mobile che considereremo sono:

- Le costanti in virgola mobile $c_{\mathbf{f}}$, dove $c \in \mathbb{F}_{\mathbf{f}}$;
- Le variabili in virgola mobile $v_{\mathbf{f}}$, dove $v_{\mathbf{f}} \in \mathcal{V}_{\mathbf{f}}$;
- L'applicazione di un operatore binario $e_1 \odot e_2$, dove $\odot$ è uno fra $\oplus$, $\ominus$, $\otimes$ o $\oslash$, che rappresentano le quattro operazioni aritmetiche di base;
- L'applicazione del meno unario $\ominus\text{expr}_{\mathbf{f}}$;
- Il cast di un'espressione $\text{expr}_{\mathbf{f}'}$, al formato in virgola mobile $\mathbf{f}$, indicato con $\text{cast}_{\mathbf{f}}(\text{expr}_{\mathbf{f}'})$.
- Il cast da un'espressione di tipo intero $\text{expr}_{\mathbb{Z}}(n)$ avente a tempo d'esecuzione valore $n \in \mathbb{Z}$ ad una in virgola mobile di formato $\mathbf{f}$, indicato con $\text{cast}_{\mathbf{f}}(\text{expr}_{\mathbb{Z}}(n))$.

Dal momento che la semantica concreta di un'espressione dipende dalla modalità di arrotondamento utilizzata a tempo d'esecuzione, diamo ora una definizione precisa delle funzioni di arrotondamento.

Definizione 1. (Funzioni di arrotondamento.) Sia $\mathbf{f}$ un formato in virgola mobile e sia $x \in \mathbb{R}$. Definiamo le funzioni di arrotondamento su $\mathbf{f}$ come:

$$R_{\mathbf{f},+\infty}(x) \stackrel{\text{def}}{=} \begin{cases} +\infty, & \text{se } x > \mathrm{Mf}_{\mathbf{f}}; \\ min\{y \in \mathbb{F}_{\mathbf{f}} \mid y \geq x\}, & \text{altrimenti.} \end{cases}$$

$$R_{\mathbf{f},-\infty}(x) \stackrel{\text{def}}{=} \begin{cases} -\infty, & \text{se } x < -\mathrm{Mf}_{\mathbf{f}}; \\ max\{y \in \mathbb{F}_{\mathbf{f}} \mid y \leq x\}, & \text{altrimenti.} \end{cases}$$

$$R_{\mathbf{f},\mathbf{0}}(x) \stackrel{\text{def}}{=} \begin{cases} min\{y \in \mathbb{F}_{\mathbf{f}} \mid y \geq x\}, & \text{se } x \leq 0; \\ max\{y \in \mathbb{F}_{\mathbf{f}} \mid y \leq x\}, & \text{altrimenti.} \end{cases}$$

$$R_{\mathbf{f},\mathbf{n}}(x) \stackrel{\text{def}}{=} \begin{cases} +\infty, & \text{se } x \geq (2 - 2^{-\mathbf{p_f}-1}) \times 2^{2^{\mathbf{e_f}}-\mathbf{bias_f}-2}; \\ -\infty, & \text{se } x \leq -(2 - 2^{-\mathbf{p_f}-1}) \times 2^{2^{\mathbf{e_f}}-\mathbf{bias_f}-2}; \\ \mathrm{Mf}_{\mathbf{f}}, & \text{altrimenti se } x \geq \mathrm{Mf}_{\mathbf{f}}; \\ \mathrm{mf}_{\mathbf{f}}, & \text{altrimenti se } x \leq \mathrm{mf}_{\mathbf{f}}; \\ R_{\mathbf{f},-\infty}(x), & \text{altrimenti se } |R_{\mathbf{f},-\infty}(x) - x| < |R_{\mathbf{f},+\infty}(x) - x|; \\ R_{\mathbf{f},+\infty}(x), & \text{altrimenti se } |R_{\mathbf{f},+\infty}(x) - x| < |R_{\mathbf{f},-\infty}(x) - x|; \\ R_{\mathbf{f},-\infty}(x), & \text{altrimenti se il bit meno significativo di } R_{\mathbf{f},-\infty}(x) \text{ è } 0; \\ R_{\mathbf{f},+\infty}(x), & \text{altrimenti se il bit meno significativo di } R_{\mathbf{f},+\infty}(x) \text{ è } 0. \end{cases}$$

Possiamo ora definire formalmente la semantica concreta di un'espressione in virgola mobile. Per semplicità, nella seguente definizione (e in tutti i futuri casi dove possa essere utile) indicheremo un generico valore infinito o codice d'errore con Ω.

Definizione 2. (Semantica concreta delle espressioni in virgola mobile.) Sia $\mathbf{f}$ un formato in virgola mobile e sia ρ uno store (concreto). La semantica concreta dell'espressione in virgola mobile $\mathrm{expr}_{\mathbf{f}}$ nello store ρ è un numero di $\mathbb{F}_{\mathbf{f}}$ denotato da $[\![\mathrm{expr}_{\mathbf{f}}]\!]\,\rho$ ed è definita induttivamente da:

$$\begin{array}{lll}
[\![c_{\mathbf{f}}]\!]\,\rho & \stackrel{\text{def}}{=} c & \text{(dove } c \in \mathbb{F}_{\mathbf{f}}\text{)}; \\
[\![v_{\mathbf{f}}]\!]\,\rho & \stackrel{\text{def}}{=} \rho(v) & \text{(dove } v_{\mathbf{f}} \in \mathcal{V}_{\mathbf{f}}\text{)}; \\
[\![\mathrm{cast}_{\mathbf{f}}(\mathrm{expr}_{\mathbb{Z}}(n))]\!]\,\rho & \stackrel{\text{def}}{=} R_{\mathbf{f},\mathbf{r}}(n) & \text{(dove } n \in \mathbb{Z}\text{)}; \\
[\![\mathrm{cast}_{\mathbf{f}}(\mathrm{expr}_{\mathbf{f}'})]\!]\,\rho & \stackrel{\text{def}}{=} R_{\mathbf{f},\mathbf{r}}([\![\mathrm{expr}_{\mathbf{f}'}]\!]\,\rho) & \text{se } [\![\mathrm{expr}_{\mathbf{f}'}]\!]\,\rho \neq \Omega; \\
[\![e_1 \odot e_2]\!]\,\rho & \stackrel{\text{def}}{=} R_{\mathbf{f},\mathbf{r}}([\![e_1]\!]\,\rho \cdot [\![e_2]\!]\,\rho) & \text{se } [\![e_1]\!]\,\rho \neq \Omega,\ [\![e_2]\!]\,\rho \neq \Omega,\ \cdot \in \{+,-,\times\}; \\
[\![e_1 \oslash e_2]\!]\,\rho & \stackrel{\text{def}}{=} R_{\mathbf{f},\mathbf{r}}([\![e_1]\!]\,\rho / [\![e_2]\!]\,\rho) & \text{se } [\![e_1]\!]\,\rho \neq \Omega,\ [\![e_2]\!]\,\rho \neq \Omega \text{ e } [\![e_2]\!]\,\rho \neq 0; \\
[\![\ominus \mathrm{expr}_{\mathbf{f}}]\!]\,\rho & \stackrel{\text{def}}{=} -[\![\mathrm{expr}_{\mathbf{f}}]\!]\,\rho & \text{se } [\![\mathrm{expr}_{\mathbf{f}}]\!]\,\rho \neq \Omega; \\
[\![\mathrm{expr}_{\mathbf{f}}]\!]\,\rho & \stackrel{\text{def}}{=} \Omega & \text{negli altri casi.}
\end{array}$$

1.5 Altre operazioni su quantità in virgola mobile

Per quanto riguarda i test su valori in virgola in virgola mobile, essi a differenza delle operazioni aritmetiche sono sempre esatti ed il loro risultato nella maggior parte dei casi è quello ovvio. Vi sono tuttavia alcuni casi speciali che richiedono una menzione esplicita:

- I due valori nulli sono considerati uguali per quanto riguarda l'ordinamento. Da questo segue ad esempio che, date due variabili `a` e `b` aventi due diversi valori nulli, il test `a == b` valuterà a `true`, il test `a != b` valuterà a `false` mentre le espressioni `std::min(a,b)` e `std::max(a,b)` valuteranno ad un valore nullo arbitrario.

- I NaN non sono ordinati, né fra di loro né rispetto ad altri valori, perciò qualunque test della forma `a X b` dove almeno uno fra `a` e `b` ha un valore NaN e dove `X` è un operatore di confronto diverso dall'operatore di disuguaglianza `!=` valuterà a `false` (si noti in particolare che `x == x` valuterà a `false` quando `x` ha un valore NaN). Viceversa, l'operatore `!=` valuterà sempre a `true` (in particolare anche quando un NaN è confrontato con sé stesso).

- Gli operatori logici applicati a valori in virgola mobile considerano entrambi i valori nulli come `false` e tutti i NaN come `true`.

1.6 Problemi dell'aritmetica in virgola mobile

L'inesattezza dell'aritmetica in virgola mobile rende il suo utilizzo ancora più problematico rispetto a quello dell'aritmetica intera. Ecco un elenco (per nulla esaustivo) di possibili cause di errore introdotte da questo tipo di calcoli:

- L'impossibilità di rappresentare esattamente numeri "semplici" come 0.1.

- L'impossibilità di rappresentare esattamente tutte le costanti irrazionali importanti come il pi greco o il numero di Nepero.

- L'assenza delle proprietà associativa e distributività dell'addizione e della moltiplicazione (continua invece a valere la proprietà commutativa).

- Il problema della *cancellazione*, ovvero il fenomeno per il quale sottraendo ad un numero un altro numero quasi uguale si può ottenere un enorme errore relativo sul risultato.

- L'esistenza di *problemi malcondizionati*, ovvero problemi matematici nei quali una piccola variazione nei dati produce una grossa variazione del risultato esatto. Ovviamente tali problemi non si prestano ad essere affrontati per mezzo di un'aritmetica inesatta come quella in virgola mobile.

- L'impossibilità di effettuare test basandosi sui risultati attesi nell'aritmetica esatta.

2 Analisi delle computazioni in virgola mobile

2.1 Richiami di interpretazione astratta

2.1.1 Concetti base

Col termine *analisi statica del software* viene indicato il processo di deduzione di proprietà dei programmi effettuato senza mandare in esecuzione i programmi stessi. Si dice *analizzatore statico* un programma che ha la capacità di effettuare l'analisi statica di programmi.

L'analizzatore statico ECLAIR, oggetto del nostro lavoro, utilizza un approccio di analisi noto come *interpretazione astratta*; tale approccio consiste in linea di massima delle seguenti fasi:

1. Viene definito un insieme di *stati astratti* della macchina avente la struttura di reticolo completo in base a una relazione d'ordine parziale $\sqsubseteq$. L'operazione di estremo superiore su tale reticolo è denotata da $\sqcup$. Questo insieme è detto *dominio astratto* e viene definito in base alle proprietà di interesse che si desidera tentare di dedurre.

2. Viene definita una corrispondenza fra gli insiemi di possibili stati concreti della macchina (i quali comprendono tutti i dettagli rilevanti per l'esecuzione del programma quali lo stato dell'unità di controllo, il contenuto dei registri, il contenuto della memoria eccetera) sulla quale il programma analizzato andrà in esecuzione e gli stati astratti. Si dice *funzione d'astrazione* la funzione α che mappa ogni insieme

di stati concreti nel corrispondente stato astratto. Alla funzione di concretizzazione corrisponde una pseudo-inversa γ detta *funzione di concretizzazione*, la quale mappa ciascuno stato astratto nell'unione di tutti gli insiemi di stati concreti che vengono astratti in esso. Queste due funzioni danno luogo ad una connessione di Galois fra l'insieme delle parti degli stati concreti (visto come un reticolo completo ordinato dalla relazione di inclusione $\subseteq$) e l'insieme degli stati astratti.

3. Per ogni possibile operazione del programma analizzato che modifica lo stato concreto viene definita una corrispondente *operazione astratta* che agisce su uno stato astratto. L'operazione astratta deve essere definita in modo che sia un'*approssimazione consistente* di quella concreta, ovvero sia compatibile con le funzioni di astrazione e concretizzazione. Se si vuole ottenere un analizzatore efficiente, deterministico e a terminazione garantita, bisogna per prima cosa che le operazioni astratte godano di tali proprietà.

4. L'analisi vera e propria, noto uno stato astratto iniziale che approssimi correttamente lo stato concreto iniziale del programma, avviene infine simulando l'esecuzione concreta tramite le corrispondenti operazioni astratte.

Si noti che un analizzatore che opera in tal modo non potrà mai trarre conclusioni errate: nella peggiore delle ipotesi si limiterà a non riuscire a dedurre nulla. In altre parole l'analizzatore è privo di falsi negativi.

Per una trattazione approfondita dei concetti appena riassunti, si vedano [Sch95] e [BHZ09a].

2.1.2 Astrarre il flusso di controllo

Tra gli operatori concreti un'attenzione particolare deve essere prestata a quelli che possono modificare il flusso di controllo in base al verificarsi o meno di determinate condizioni, come avviene ad esempio in C/C++ in corrispondenza di costrutti quali

gli statement `if` e `while`. Per migliorare la precisione dell'analisi in presenza di tali costrutti, tenendo conto ad esempio del fatto che all'interno del corpo di un `if` la guardia booleana deve essere vera, vengono introdotti degli operatori astratti detti *filtri*. Data una condizione d, un filtro trasforma un generico stato astratto a in uno stato a_d tale che $a_d \sqsubseteq a$ e per ogni stato concreto $c \in \gamma(a_d)$ si ha che c soddisfa d.

Nei punti di programma in cui confluisce un numero limitato di diversi possibili flussi di controllo, come ad esempio al termine degli statement `if` o `switch`, lo stato astratto raggiunto viene normalmente calcolato come l'estremo superiore fra tutti gli stati astratti ottenuti analizzando i diversi flussi di controllo indipendenti.

L'attenzione maggiore va riservata nei punti di programma, come all'interno dei cicli, che possono essere raggiunti da un numero arbitrariamente alto e possibilmente illimitato di possibili flussi di controllo. Una semplice idea per effettuare l'interpretazione astratta di un ciclo potrebbe essere quella di iterare l'analisi del corpo, partendo ad ogni nuova iterazione dallo stato astratto i_n ottenuto calcolando l'estremo superiore $i_{n-1} \sqcup f_{n-1}$ fra il vecchio stato all'inizio del ciclo i_{n-1} e quello risultante dall'ultima iterazione f_{n-1}. Tale approccio, tuttavia, non garantisce la possibilità di raggiungere un punto fisso dopo un numero finito di iterazioni: ecco perché è necessario introdurre gli operatori di *widening*.

Un operatore di widening ∇ è un operatore binario fra stati astratti tale che dati due stati astratti a_1 e a_2 con $a_1 \sqsubseteq a_2$ si abbia $a_2 \sqsubseteq a_1 \nabla a_2$; inoltre, data qualsiasi successione di stati astratti $a_0, a_1, a_2, \ldots$ con $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ il calcolo $a_0 \nabla a_1 \nabla a_2 \nabla \ldots$ deve raggiungere un punto fisso. Avendo a disposizione un operatore di widening ∇, possiamo analizzare un ciclo come nell'approccio precedente ponendo però $i_n = i_{n-1} \nabla (i_{n-1} \sqcup f_{n-1})$.

2.2 Analisi basata sugli intervalli

2.2.1 Semantica astratta basata sugli intervalli

Da questo punto in avanti dobbiamo fare una distinzione fra l'ambiente dove viene eseguito il programma analizzato e quello dove viene eseguito l'analizzatore. Nello

specifico, indicheremo con **f** un generico formato in virgola mobile usato nel programma analizzato (utilizzeremo simboli come **f'** e **f''** nel caso avessimo bisogno di riferirci a più di un formato nel programma analizzato), mentre il formato (unico) utilizzato dall'analizzatore verrà indicato con **fa**.

Nella *analisi basata sugli intervalli*, l'analizzatore approssima ciascun valore concreto in virgola mobile con un intervallo $[lb; ub]$, dove $lb, ub \in \mathbb{F}_{\mathbf{fa}}$ sono rispettivamente il limite inferiore e superiore entro i quali è garantito che il valore concreto debba cadere. Denoteremo con $\mathbb{I}_{\mathbf{fa}}$ l'insieme di tutti gli intervalli con limiti in $\mathbb{F}_{\mathbf{fa}}$. ECLAIR estende ciascun intervallo con un *NaN bit* il cui valore è 0 se siamo certi che il corrispondente valore concreto non sia un NaN (viceversa, il valore 1 indica la *possibilità* che il valore sia un NaN), cosicché l'astrazione diventa di fatto una coppia $\langle [lb; ub], N \rangle$, dove N è il NaN bit. Tali coppie sono dette *astrazioni ad intervallo*.

Questo approccio astrae ciascuno store concreto ρ in uno *store astratto ad intervalli* $\rho^{\#}$. Uno store astratto ad intervalli associa ciascuna variabile in virgola mobile ad una astrazione ad intervallo (dove l'intervallo appartiene a $\mathbb{I}_{\mathbf{fa}}$ indipendentemente dal formato della variabile nel concreto) che è garantita essere un'approssimazione corretta del valore della variabile nello store concreto.

Per essere sicuri di ottenere un risultato corretto l'analizzatore deve affrontare due problemi: il primo è tenere conto della modalità di arrotondamento utilizzata dal programma analizzato, il secondo è scegliere una modalità di arrotondamento consistente per operare sui limiti delle astrazioni ad intervallo. Per quanto riguarda quest'ultimo problema, l'unica scelta possibile è arrotondare sempre i limiti inferiori verso $-\infty$ e i limiti superiori verso $+\infty$. Ciò potrebbe comportare problemi d'efficienza, in quanto cambiare dinamicamente la modalità d'arrotondamento utilizzata dalla CPU può portare all'invalidazione di molte voci all'interno della sua cache; ecco perché manteniamo sempre la CPU impostata per arrotondare verso $+\infty$ e simuliamo l'arrotondamento verso $-\infty$ di una quantità prima negandola, poi arrotondandola verso $+\infty$ ed infine negandola di

nuovo.

Per quanto riguarda invece il primo problema ECLAIR sceglie di utilizzare internamente un formato **fa** arbitrario, ma qualora nel programma concreto venga utilizzato un formato in virgola mobile **f** più preciso di **fa** la precisione dei risultati dell'analisi viene ridotta (utilizzando il metodo `maybe_reduce_precision()`) per essere uguale o inferiore a quella delle operazioni concrete su **f**. Per evitare questa complicazione all'interno delle formule che seguiranno, d'ora in poi supporremo semplicemente che **fa** sia più preciso di qualunque formato **f** che compare nel programma analizzato. Sotto questa ipotesi, siamo ora in grado di definire una semantica astratta consistente per le espressioni concrete in virgola mobile:

Definizione 3. (Semantica astratta per l'analisi basata sugli intervalli.) Sia $\text{expr}_{\mathbf{f}}$ un'espressione concreta e sia **fa** un formato in virgola mobile tale che $\mathbb{F}_{\mathbf{fa}} \subseteq \mathbb{F}_{\mathbf{f}}$; sia inoltre $\rho^{\#}$ uno store astratto ad intervalli. La semantica astratta basata sugli intervalli di $\text{expr}_{\mathbf{f}}$ in $\rho^{\#}$ è un'astrazione ad intervallo denotata con $[\![\text{expr}_{\mathbf{f}}]\!]^{\#}\,\rho^{\#}$ e definita induttivamente da:

Se $[\![e_1]\!]^{\#}\,\rho^{\#} = \langle [l_1; u_1]\,, N_1\rangle$ e $[\![e_2]\!]^{\#}\,\rho^{\#} = \langle [l_2; u_2]\,, N_2\rangle$, allora:

$[\![c_{\mathbf{f}}]\!]^{\#}\,\rho^{\#} \stackrel{\text{def}}{=} \langle [R_{\mathbf{fa},-\infty}(c); R_{\mathbf{fa},+\infty}(c)]\,, 0\rangle$
$[\![v_{\mathbf{f}}]\!]^{\#}\,\rho^{\#} \stackrel{\text{def}}{=} \rho^{\#}\,(v_{\mathbf{f}})$
$[\![\text{cast}_{\mathbf{f}}(\text{expr}_{\mathbb{Z}}(n))]\!]^{\#}\,\rho^{\#} \stackrel{\text{def}}{=} \langle [R_{\mathbf{fa},-\infty}(n); R_{\mathbf{fa},+\infty}(n)]\,, 0\rangle$
$[\![\text{cast}_{\mathbf{f}}(e_1)]\!]^{\#}\,\rho^{\#} \stackrel{\text{def}}{=} [\![e_1]\!]^{\#}\,\rho^{\#}$
$[\![\ominus e_1]\!]^{\#}\,\rho^{\#} \stackrel{\text{def}}{=} \left\langle \ominus^{\#}\,[l_1; u_1]\,, N_1\right\rangle$

$[\![e_1 \odot e_2]\!]^{\#}\,\rho^{\#} \stackrel{\text{def}}{=} \left\langle [l_1; u_1] \odot^{\#} [l_2; u_2]\,, N\right\rangle$
Dove $\odot \in \left\{\oplus, \ominus, \otimes\right\}$, e $N = 1$ se e solo se $N_1 = 1$ oppure $N_2 = 1$
oppure viene generato un NaN durante il calcolo di $[l_1; u_1] \odot^{\#} [l_2; u_2]$

$[\![e_1 \oslash e_2]\!]^{\#}\,\rho^{\#} \stackrel{\text{def}}{=} \left\langle [l_1; u_1] \oslash^{\#} [l_2; u_2]\,, N\right\rangle$
Dove $N = 1$ se e solo se $N_1 = 1$ o $N_2 = 1$ o $l_2 \leq 0 \leq u_2$
oppure viene generato un NaN durante il calcolo di $[-l_1; -u_1] \oslash^{\#} [l_2; u_2]$

Dove $\oplus^{\#}$,$\ominus^{\#}$,$\otimes^{\#}$ e $\oslash^{\#}$ sono operatori consistenti sugli intervalli definiti come:

$$
\begin{array}{rll}
 & \ominus^{\#}\,[l_1;u_1] & \stackrel{\text{def}}{=} \quad [-u_1;-l_1] \\
[l_1;u_1] & \oplus^{\#}\,[l_2;u_2] & \stackrel{\text{def}}{=} \quad [l_1 \oplus_{\mathbf{fa},-\infty} l_2; u_1 \oplus_{\mathbf{fa},+\infty} u_2] \\
[l_1;u_1] & \ominus^{\#}\,[l_2;u_2] & \stackrel{\text{def}}{=} \quad [l_1 \ominus_{\mathbf{fa},-\infty} u_2; u_1 \ominus_{\mathbf{fa},+\infty} l_2] \\
[l_1;u_1] & \otimes^{\#}\,[l_2;u_2] & \stackrel{\text{def}}{=} \quad [\min\{l_1 \otimes_{\mathbf{fa},-\infty} l_2, l_1 \otimes_{\mathbf{fa},-\infty} u_2, u_1 \otimes_{\mathbf{fa},-\infty} l_2, u_1 \otimes_{\mathbf{fa},-\infty} u_2\} \\
 & & \quad ; \max\{l_1 \otimes_{\mathbf{fa},+\infty} l_2, l_1 \otimes_{\mathbf{fa},+\infty} u_2, u_1 \otimes_{\mathbf{fa},+\infty} l_2, u_1 \otimes_{\mathbf{fa},+\infty} u_2\}] \\
[l_1;u_1] & \oslash^{\#}\,[l_2;u_2] & \stackrel{\text{def}}{=} \quad [\min\{l_1 \oslash_{\mathbf{fa},-\infty} l_2, l_1 \oslash_{\mathbf{fa},-\infty} u_2, u_1 \oslash_{\mathbf{fa},-\infty} l_2, u_1 \oslash_{\mathbf{fa},-\infty} u_2\} \\
 & & \quad ; \max\{l_1 \oslash_{\mathbf{fa},+\infty} l_2, l_1 \oslash_{\mathbf{fa},+\infty} u_2, u_1 \oslash_{\mathbf{fa},+\infty} l_2, u_1 \oslash_{\mathbf{fa},+\infty} u_2\}]
\end{array}
$$

Dove min e max ignorano i NaN (si noti che questo potrebbe portare ad intervalli vuoti: ad esempio, $[0;0] \oslash^{\#} [0;0] = \varnothing$. Ciò è consistente col fatto che 0/0 è una forma indeterminata la cui valutazione può unicamente risultare in un NaN).

Teorema 1. (Consistenza della semantica astratta sugli intervalli..) *La semantica astratta presentata nella definizione 3 è consistente.*

Dimostrazione. Si veda [Min05]. □

2.2.2 Analisi basata sugli intervalli

Con questo approccio, la valutazione astratta dell'assegnamento di un'espressione $\text{expr}_{\mathbf{f}}$ ad una variabile $v_{\mathbf{f}}$ consiste semplicemente nella sostituzione all'interno dello store astratto del valore attuale di $\rho^{\#}(v_{\mathbf{f}})$ con la valutazione astratta dell'espressione $[\![\text{expr}_{\mathbf{f}}]\!]^{\#}\,\rho^{\#}$. È inoltre possibile analizzare semplici filtri per punti di biforcazione del flusso di controllo come `if e1 <= e2 true_branch else false_branch` quando almeno una delle due espressioni presenti all'interno della guardia booleana è una singola variabile: ad esempio, il test $X <= Y + Z$ valutato in uno store astratto nel quale $\rho^{\#}(X) = \langle [0;6], 0\rangle, \rho^{\#}(Y) = \langle [-4;4], 0\rangle, \rho^{\#}(Z) = \langle [0;0], 1\rangle$ filtrerà lo store in $\rho^{\#}(X) = \langle [0;4], 0\rangle, \rho^{\#}(Y) = \langle [0;4], 0\rangle, \rho^{\#}(Z) = \langle [0;0], 0\rangle$ per il ramo `true`, mentre lo store per il ramo `false` rimarrà immutato (il motivo è che $Y + Z$ valuterà a NaN se $Z = \text{NaN}$ e, come già detto nella sezione 1.5, lo standard IEEE754 specifica che i NaN non sono ordinati, perciò il test valuterà a `false` per ogni valore di X e Y). Altri tipi di test sono

analizzabili unicamente quando è possibile dimostrare che valuteranno sempre a `true` o a `false` (o che valuteranno sicuramente a `true` o `false` quando nessuna delle variabili coinvolte ha un valore NaN).

L'estremo superiore fra due astrazioni ad intervallo $\langle [l_1; u_1], N_1 \rangle$ e $\langle [l_2; u_2], N_2 \rangle$, utilizzato per l'analisi dei punti di programma dove si possono congiungere diversi flussi di controllo, è banalmente definito da $\langle [min\{l_1, l_2\}; max\{u_1, u_2\}], N_1 \vee N_2 \rangle$, dove $\vee$ è la disgiunzione logica. Gli operatori di widening per i domini astratti basati sugli intervalli, ad esempio quello che fa crescere esponenzialmente i limiti degli intervalli in base a soglie predefinite, sono anch'essi decisamente elementari.

2.2.3 Limiti dell'analisi basata sugli intervalli

Lo svantaggio più grosso dell'analisi basata sugli intervalli è la sua scarsa precisione. Si consideri ad esempio il programma mostrato nel listato 2.1. Tale programma è un cosiddetto *rate limiter*: ad ogni iterazione del ciclo la variabile Y, detta *rate*, riceve una perturbazione che dipende da due valori X e D generati casualmente ad ogni iterazione. Più precisamente Y viene inizializzata a 0, quindi ad ogni iterazione il suo valore viene aggiustato per avvicinarsi il più possibile a quello di X con l'unica limitazione che il nuovo valore di Y non può differire dal vecchio per una quantità maggiore di D (il cui valore è generato casualmente in $[0; 16]$) in valore assoluto. Poiché il valore di X è generato casualmente in $[-128; 128]$, ne segue che anche Y sarà limitata ad assumere valori in quell'intervallo. Sfortunatamente, l'analisi basata sugli intervalli sarà unicamente in grado di dedurre che dopo la n-esima iterazione il valore di Y cade in $[-128 - n; 128 + n]$, perciò indipendentemente dal widening utilizzato Y risulterà possibilmente illimitata dopo un numero arbitrariamente grande di iterazioni.

La motivazione principale per cui si ottiene un risultato così impreciso risiede nell'incapacità di questa analisi di inferire molte relazioni che valgono a tempo d'esecuzione fra i valori delle variabili presenti nel programma analizzato: ecco perché quest'analisi è in fin dei conti insufficiente e dovrà essere estesa con l'utilizzo dei più precisi domini relazionali.

```
1  double Y, S, R, X, D;
2  Y = 0;
3  while (true) {
4    // Genera casualmente X fra -128 e 128.
5    // Genera casualmente D fra 0 e 16.
6
7    S = Y;
8    R = X - S;
9    Y = X;
10
11   if (R <= -D)
12     Y = S - D;
13
14   if (R >= D)
15     Y = S + D;
16 }
```

Listato 2.1: Un rate limiter.

2.3 Linearizzazione delle espressioni in virgola mobile

2.3.1 Forme lineari ad intervalli

Dobbiamo ora trovare un modo per estendere i tradizionali domini astratti relazionali, come quelli poliedrici forniti dalla Parma Polyhedra Library (PPL) descritti in [BHZ08], al fine di poter effettuare un analisi consistente dei calcoli in virgola mobile. Tali domini offrono già delle funzioni per astrarre gli assegnamenti e i test su espressioni complesse, ma tali funzioni fanno affidamento su alcune proprietà algebriche delle operazioni, come l'associatività e la distributibità, che semplicemente non valgono per i calcoli in virgola mobile.

La soluzione proposta da Antoine Miné in [Min04] è basata sull'idea di associare una *forma lineare a coefficienti intervalli* a ciascuna espressione concreta, quindi di definire alcune operazioni sui poliedri che utilizzino tali forme lineari. Una forma lineare è un'espressione del tipo $i + \sum_v i_v v$, dove ciascuna v è una variabile in virgola mobile (qui il formato è irrilevante) e ciascun i, i_v è un intervallo di $\mathbb{I}_{\mathbf{fa}}$. L'analizzatore può quindi effettuare delle operazioni consistenti sulle forme lineari basandosi sugli operatori sugli

intervalli definiti nella sottosezione 2.2.1.

Le operazioni sulle forme lineari non devono produrre alcun infinito o NaN durante la loro computazione: se qualcuna di queste quantità viene generata l'operazione deve terminare immediatamente segnalando un fallimento. Ciò implica fra l'altro che le tecniche basate sull'utilizzo delle forme lineari non possono sostituire del tutto l'analisi basata sugli intervalli: anzi, quest'ultima dev'essere sempre eseguita in parallelo con tutte le tecniche d'analisi che saranno descritte da adesso in poi, in quanto l'algoritmo di linearizzazione descritto nella sottosezione 2.3.2 richiede l'accesso ad uno store astratto ad intervalli.

Definizione 4. (Operazioni sulle forme lineari.) Le operazioni aritmetiche di base sulle forme lineari sono definite come:

$$
\begin{array}{ll}
\boxminus^{\#}\left(i+\sum_v i_v v\right) & \stackrel{\text{def}}{=} \left(\ominus^{\#} i\right) + \sum_v \left(\ominus^{\#} i_v\right) v, \\
\left(i+\sum_v i_v v\right) \boxplus^{\#} \left(i'+\sum_v i'_v v\right) & \stackrel{\text{def}}{=} \left(i \oplus^{\#} i'\right) + \sum_v \left(i_v \oplus^{\#} i'_v\right) v, \\
\left(i+\sum_v i_v v\right) \boxminus^{\#} \left(i'+\sum_v i'_v v\right) & \stackrel{\text{def}}{=} \left(i \ominus^{\#} i'\right) + \sum_v \left(i_v \ominus^{\#} i'_v\right) v, \\
\left(i+\sum_v i_v v\right) \boxtimes^{\#} \left(i'+\sum_v i'_v v\right) & \stackrel{\text{def}}{=} \left(i \otimes^{\#} i'\right) + \sum_v \left(i_v \otimes^{\#} i'_v\right) v, \\
\left(i+\sum_v i_v v\right) \boxslash^{\#} \left(i'+\sum_v i'_v v\right) & \stackrel{\text{def}}{=} \left(i \oslash^{\#} i'\right) + \sum_v \left(i_v \oslash^{\#} i'_v\right) v.
\end{array}
$$

2.3.2 Algoritmo di linearizzazione

Per poter tradurre un'espressione in virgola mobile in una forma lineare che l'approssimi correttamente, dobbiamo necessariamente tener conto degli errori d'arrotondamento. Gli errori associati alle forme lineari sono a loro volta espressi come forme lineari dipendenti dal formato **f** usato nel programma analizzato. Esistono due tipi d'errore: l'*errore assoluto* $\omega_{\mathbf{f}}$ è definito come la forma lineare costituita dal solo intervallo $\left[-max\left\{\mathrm{mf}_{\mathbf{f}}, \mathrm{mf}_{\mathbf{fa}}\right\}; max\left\{\mathrm{mf}_{\mathbf{f}}, \mathrm{mf}_{\mathbf{fa}}\right\}\right]$ e tiene conto dell'errore commesso nelle operazioni che coinvolgono numeri denormalizzati. L'*errore relativo* di una forma lineare l tiene invece conto degli errori commessi sui numeri normalizzati, è denotato con $\epsilon_{\mathbf{f}}(l)$ ed è definito da:

$$\epsilon_{\mathbf{f}}\left([l;u]+\sum_v [l_v;u_v]\,v\right) \stackrel{\text{def}}{=} \begin{array}{l}\left((max\left\{|l|,|u|\right\})\otimes^{\#}[-2^{-\mathbf{p_f}};2^{-\mathbf{p_f}}]\right)\\ +\sum_v\left(max\left\{|l_v|,|u_v|\right\}\otimes^{\#}[-2^{-\mathbf{p_f}};2^{-\mathbf{p_f}}]\right)v.\end{array}$$

Dato uno store astratto $\rho^{\#}$ (ricordiamo che l'algoritmo di linearizzazione richiede che sia effettuata l'analisi basata sugli intervalli) ed una forma lineare l, l'intervallo che approssima meglio l in $\rho^{\#}$ è detto *intervallizzazione* di l in $\rho^{\#}$ e viene indicato con $\iota\,(l)\,\rho^{\#}$. Può essere calcolato come:

$$\iota\left(i+\sum_v i_v v\right)\rho^{\#} = i\oplus^{\#}\left(\bigoplus_v^{\#} i_v\otimes^{\#}\rho^{\#}(v)\right).$$

Possiamo ora definire la linearizzazione di un'espressione in virgola mobile:

Definizione 5. (Linearizazazione di un'espressione in virgola mobile.) Data un'espressione (concreta) in virgola mobile $\text{expr}_{\mathbf{f}}$ ed uno store astratto ad intervalli $\rho^{\#}$, la linearizzazione di $\text{expr}_{\mathbf{f}}$ in $\rho^{\#}$, se esiste, è una forma lineare con intervalli di $\mathbb{I}_{\mathbf{fa}}$ come cofficienti che viene indicata con $(\!|\text{expr}_{\mathbf{f}}|\!)\rho^{\#}$ ed è definita induttivamente da:

$$\begin{array}{ll}
(\!|c_{\mathbf{f}}|\!)\rho^{\#} & \stackrel{\text{def}}{=} [R_{\mathbf{fa},-\infty}(c);R_{\mathbf{fa},+\infty}(c)]\,;\\
(\!|v_{\mathbf{f}}|\!)\rho^{\#} & \stackrel{\text{def}}{=} [1;1]\,v_{\mathbf{f}};\\
(\!|\text{cast}_{\mathbf{f}}(\text{expr}_{\mathbb{Z}}(n))|\!)\rho^{\#} & \stackrel{\text{def}}{=} [R_{\mathbf{fa},-\infty}(n);R_{\mathbf{fa},+\infty}(n)]\boxplus^{\#}\epsilon_{\mathbf{f}}\left([R_{\mathbf{fa},-\infty}(n);R_{\mathbf{fa},+\infty}(n)]\right)\boxplus^{\#}\omega_{\mathbf{f}};\\
(\!|\text{cast}_{\mathbf{f'}}(\text{expr}_{\mathbf{f}})|\!)\rho^{\#} & \stackrel{\text{def}}{=} \begin{cases}(\!|\text{expr}_{\mathbf{f}}|\!)\rho^{\#}, & \text{se } \mathbb{F}_{\mathbf{f}}\subseteq\mathbb{F}_{\mathbf{f'}},\\ (\!|\text{expr}_{\mathbf{f}}|\!)\rho^{\#}\boxplus^{\#}\epsilon_{\mathbf{f'}}\left((\!|\text{expr}_{\mathbf{f}}|\!)\rho^{\#}\right)\boxplus^{\#}\omega_{\mathbf{f'}}, & \text{altrimenti;}\end{cases}\\
(\!|e_1\oplus e_2|\!)\rho^{\#} & \stackrel{\text{def}}{=} (\!|e_1|\!)\rho^{\#}\boxplus^{\#}(\!|e_2|\!)\rho^{\#}\boxplus^{\#}\epsilon_{\mathbf{f}}\left((\!|e_1|\!)\rho^{\#}\right)\boxplus^{\#}\epsilon_{\mathbf{f}}\left((\!|e_2|\!)\rho^{\#}\right)\boxplus^{\#}\omega_{\mathbf{f}};\\
(\!|e_1\ominus e_2|\!)\rho^{\#} & \stackrel{\text{def}}{=} (\!|e_1|\!)\rho^{\#}\boxminus^{\#}(\!|e_2|\!)\rho^{\#}\boxplus^{\#}\epsilon_{\mathbf{f}}\left((\!|e_1|\!)\rho^{\#}\right)\boxplus^{\#}\epsilon_{\mathbf{f}}\left((\!|e_2|\!)\rho^{\#}\right)\boxplus^{\#}\omega_{\mathbf{f}};\\
(\!|[l;u]\otimes e_2|\!)\rho^{\#} & \stackrel{\text{def}}{=} \left([l;u]\boxtimes^{\#}(\!|e_2|\!)\rho^{\#}\right)\boxplus^{\#}\left([l;u]\boxtimes^{\#}\epsilon_{\mathbf{f}}\left((\!|e_2|\!)\rho^{\#}\right)\right)\boxplus^{\#}\omega_{\mathbf{f}};\\
(\!|e_1\otimes[l;u]|\!)\rho^{\#} & \stackrel{\text{def}}{=} (\!|[l;u]\otimes e_1|\!)\rho^{\#};\\
(\!|e_1\otimes e_2|\!)\rho^{\#} & \stackrel{\text{def}}{=} (\!|\iota\left((\!|e_1|\!)\rho^{\#}\right)\rho^{\#}\otimes e_2|\!)\rho^{\#};\\
(\!|e_1\oslash[l;u]|\!)\rho^{\#} & \stackrel{\text{def}}{=} \left((\!|e_1|\!)\rho^{\#}\boxslash^{\#}[l;u]\right)\boxplus^{\#}\left(\epsilon_{\mathbf{f}}\left((\!|e_1|\!)\rho^{\#}\right)\boxslash^{\#}[l;u]\right)\boxplus^{\#}\omega_{\mathbf{f}};\\
(\!|e_1\oslash e_2|\!)\rho^{\#} & \stackrel{\text{def}}{=} (\!|e_1\oslash\iota\left((\!|e_2|\!)\rho^{\#}\right)\rho^{\#}|\!)\rho^{\#}.
\end{array}$$

La linearizzazione $(\!|\text{expr}_{\mathbf{f}}|\!)\rho^{\#}$ non esiste quando qualunque operazione sulle forme lineari fallisce durante il suo calcolo (normalmente in seguito alla generazione di un infinito o di un NaN). In tal caso, si dice che la linearizzazione di $\text{expr}_{\mathbf{f}}$ in $\rho^{\#}$ fallisce.

Teorema 2. (Consistenza della linearizzazione..) *La linearizzazione di un'espressione in virgola mobile calcolata in base alla definizione 5 è un'approssimazione consistente per l'espressione.*

Dimostrazione. Si veda [Min05]. □

Quando la linearizzazione di un'espressione del programma fallisce, ciò tipicamente significa che non siamo in grado di effettuare alcun ulteriore tipo di analisi e siamo limitati alla semplice analisi basata sugli intervalli.

2.3.3 Strategie di intervallizzazione

Nella definizione 5, calcoliamo $(\!|e_1 \otimes e_2|\!)\rho^\#$ come $(\!|\iota\left((\!|e_1|\!)\rho^\#\right)\rho^\# \otimes e_2|\!)\rho^\#$. In realtà, possiamo scegliere se intervallizzare $(\!|e_1|\!)\rho^\#$ o $(\!|e_2|\!)\rho^\#$; il criterio con cui viene effettuata tale scelta è detto *strategia di intervallizzazione*. Miné presenta diverse strategie di intervallizzazione in [Min05], la cui complessità computazionale è direttamente proporzionale alla loro precisione; noi attualmente adottiamo quella più semplice, chiamata *Interval-Size Local Strategy*, che consiste nel tentare di intervallizzare entrambe le forme lineari e scegliere quella la cui intervallizzazione è più stretta.

Altre strategie che prenderemo in considerazione per il futuro sono:

- La *Relative-Size Local Strategy*, che invece di scegliere l'intervallizzazione con ampiezza minore sceglie quella con ampiezza *relativa* minore. L'ampiezza relativa di un intervallo è definita come la sua ampiezza assoluta divisa per la somma dei valori assoluti dei due limiti.

- La *Simplification-Driven Global Strategy*, che cerca di ridurre il numero di variabili che saranno presenti nell'espressione linearizzata finale. Questa strategia può essere utile in quanto alcuni domini relazionali come gli ottagoni (si veda la sottosezione 2.4.1) e le forme alle differenze vincolate (si veda la sottosezione 2.4.3) riescono a rappresentare esattamente soltanto i vincoli che sussistono fra al più due variabili.

2.3.4 Propagazione delle forme lineari

Come già accennato nella precedente sottosezione 2.3.3, spesso è desiderabile che l'algoritmo di linearizzazione restituisca forme lineari contenenti poche variabili. Ciò suggerisce di adottare alcune tecniche di semplificazione per le forme lineari come quella proposta da Miné che consiste nel memorizzare in un apposito store astratto $\rho_l^\#$ l'ultima forma lineare che è stata assegnata a ciascuna variabile, quindi di utilizzare questa informazione nell'algoritmo di linearizzazione per cercare di ridurre il numero di variabili. Per esempio, in presenza della sequenza di assegnamenti `X = 2*A; Y = X + A`, sarebbe probabilmente preferibile linearizzare l'espressione `X + A` come se fosse `2*A + A`.

Sfortunatamente non è affatto garantito che l'approccio banale, che consiste nel linearizzare sempre l'espressione $v_\mathbf{f}$ in $\rho_l^\#(v_\mathbf{f})$ qualora tale forma lineare esista, porti sempre ad ottenere forme lineari più semplici (si pensi ad esempio alla sequenza di assegnamenti `X = A + B; Y = X + C + D`): l'unico modo per esserne sicuri è provare sempre entrambe le linearizzazioni $[1;1]\,v_\mathbf{f}$ e $\rho_l^\#(v_\mathbf{f})$ e scegliere quella che porta ad un risultato finale più semplice, ottenendo però una complessità computazionale esponenziale nel numero delle variabili. Se si sceglie di utilizzare lo store aggiuntivo $\rho_l^\#$ è inoltre necessario aver cura di scartare dopo ogni assegnamento tutte le forme lineari in $\rho_l^\#$ che fanno riferimento alla variabile assegnata. Per tutte queste ragioni abbiamo per il momento accantonato l'utilizzo di questa tecnica di propagazione all'interno di ECLAIR, nonostante l'algoritmo di linearizzazione della PPL (approfondito brevemente nella sezione 3.3) fornisca comunque un supporto parziale per essa.

2.4 Analisi con domini relazionali

2.4.1 Ottagoni

Un ottagono in uno spazio vettoriale di dimensione arbitraria è un sottoinsieme dello spazio definito da un insieme di vincoli aventi la forma $\pm x \pm y \leq c$, dove x e y sono dimensioni dello spazio (eventualmente coincidenti) e c è uno scalare. L'analizzatore

stabilisce una corrispondenza biunivoca fra le variabili analizzate e le dimensioni dello spazio, cosicché i termini "variabile" e "dimensione dello spazio" saranno spesso utilizzati come se fossero equivalenti.

Gli ottagoni utilizzati da ECLAIR sono istanze della classe `Octagonal_Shape` della Parma Polyhedra Library. La rappresentazione interna per un ottagono adottata dalla PPL consiste in una matrice triangolare dove ciascuna cella contiene quattro valori scalari. I valori contenuti nella cella alla riga i, colonna j (con $i \leq j$) sono i quattro scalari c_1, c_2, c_3, c_4 che compaiono nei vincoli $v_i + v_j \leq c_1$, $v_i - v_j \leq c_2$, $-v_i + v_j \leq c_3$ e $-v_i - v_j \leq c_4$ che definiscono l'ottagono (se $i = j$, i valori di c_2 e c_3 sono inutili perciò tali locazioni di memoria contengono in realtà valori invalidi o informazioni di debug). Denoteremo con $u_{\mathbf{o}}(Q)$ il limite superiore per la quantità Q che è memorizzato all'interno dell'ottagono. Gli scalari utilizzati per la nostra analisi saranno ovviamente scelti in $\mathbb{F}_{\mathbf{fa}}$.

La valutazione astratta di un assegnamento di un'espressione che è stata linearizzata con successo in una forma lineare l ad una variabile v_k può essere effettuata rimpiazzando tutti i vincoli dell'ottagono che coinvolgono v_k con i seguenti vincoli:

$$
\begin{array}{lll}
v_k + v_i & \leq U_{\mathbf{o}}\left(l \boxplus^{\#} v_i\right), & \forall i \neq k; \\
v_k - v_i & \leq U_{\mathbf{o}}\left(l \boxminus^{\#} v_i\right), & \forall i \neq k; \\
-v_k + v_i & \leq U_{\mathbf{o}}\left(v_i \boxminus^{\#} l\right), & \forall i \neq k; \\
-v_k - v_i & \leq U_{\mathbf{o}}\left(\boxminus^{\#}\left(l \boxplus^{\#} v_i\right)\right), & \forall i \neq k; \\
v_k & \leq U_{\mathbf{o}}(l); & \\
-v_k & \leq U_{\mathbf{o}}\left(\boxminus^{\#} l\right). &
\end{array}
$$

dove $U_{\mathbf{o}}(l)$ è un limite superiore per la forma lineare l calcolabile come:

$$
\begin{array}{l}
U_{\mathbf{o}}\left([l;u] + \sum_v [l_v;u_v]\, v\right) = u \oplus_{\mathbf{fa},+\infty} \\
\left(\bigoplus^{v}_{\mathbf{fa},+\infty} max\left\{u_{\mathbf{o}}(v) \otimes_{\mathbf{fa},+\infty} u_v, \ominus_{\mathbf{fa}}\left(u_{\mathbf{o}}(-v) \otimes_{\mathbf{fa},-\infty} u_v\right),\right.\right. \\
\left.\left. u_{\mathbf{o}}(v) \otimes_{\mathbf{fa},+\infty} l_v, \ominus_{\mathbf{fa}}\left(u_{\mathbf{o}}(-v) \otimes_{\mathbf{fa},-\infty} l_v\right)\right\}\right)
\end{array}
$$

(Si noti che tale algoritmo richiede che i vincoli unari su v_k vengano agggiornati alla fine, in quanto i corrispondenti valori vengono utilizzati durante il calcolo).

Un filtro astratto su una guardia booleana $e_1 \leq e_2$ dove $(\!|e_1|\!)\rho^{\#} = l_1$ e $(\!|e_2|\!)\rho^{\#} = l_2$ può invece essere realizzato aggiungendo i seguenti vincoli per ogni coppia di variabili $v_i, v_j, i \neq j$ che compaiono in l_1 o l_2 (se qualcuno di questi vincoli non è più stretto di uno preesistente, viene mantenuto il vecchio vincolo):

$$\begin{aligned}
v_j + v_i &\leq U_{\mathbf{o}}\left(l_2 \boxminus^{\#} l_1 \boxplus^{\#} v_i \boxplus^{\#} v_j\right) \\
v_j - v_i &\leq U_{\mathbf{o}}\left(l_2 \boxminus^{\#} l_1 \boxminus^{\#} v_i \boxplus^{\#} v_j\right) \\
-v_j + v_i &\leq U_{\mathbf{o}}\left(l_2 \boxminus^{\#} l_1 \boxplus^{\#} v_i \boxminus^{\#} v_j\right) \\
-v_j - v_i &\leq U_{\mathbf{o}}\left(l_2 \boxminus^{\#} l_1 \boxminus^{\#} v_i \boxminus^{\#} v_j\right) \\
v_i &\leq U_{\mathbf{o}}\left(l_2 \boxminus^{\#} l_1 \boxplus^{\#} v_i\right)
\end{aligned}$$

(Anche qui vale il solito accorgimento per cui i vincoli unari vanno aggiornati per ultimi).

Teorema 3. (Correttezza degli algoritmi sugli ottagoni..) *Gli algoritmi sugli ottagoni qui presentati sono corretti.*

Dimostrazione. Si veda [Min05]. □

Il calcolo degli estremi superiori o dei widening con questo approccio non richiede alcuna particolare precauzione, perciò a questi scopi possiamo utilizzare gli operatori della PPL preesistenti descritti in [BHZ08], [BHRZ05], [BHZ06] e [BHZ09b].

2.4.2 Raffinamento fra i diversi domini astratti

I risultati dell'analisi basata sugli intervalli e di quella basata su poliedri possono essere raffinati l'un l'altro a ciascun passo dell'analisi migliorando in questo modo la precisione di entrambi (infatti l'analisi basata sui poliedri può in molti casi non essere strettamente più precisa di quella sugli intervalli, come viene ad esempio mostrato nella sottosezione 4.1.3): se $\rho^{\#}(v) = \langle [l_i; u_i], N \rangle$, $u_{\mathbf{o}}(-v) = l_p$ e $u_{\mathbf{o}}(v) = u_p$, sostituiamo sia l_i che l_p con $max\{l_i, -l_p\}$ e allo stesso modo sostituiamo sia u_i che u_p con $min\{u_i, u_p\}$.

2.4.3 Differenze vincolate

Una *forma alle differenze vincolate* è un poliedro definito da un insieme di vincoli della forma $x - y \leq c$ o $x \leq c$. Si noti che le forme alle differenze vincolate sono un sottoinsieme stretto degli ottagoni.

Le forme alle differenze vincolate utilizzate da ECLAIR sono istanze della classe `BD_Shape` della PPL. La rappresentazione in memoria consiste in una matrice quadrata di scalari, dove la cella situata alla riga i, colonna j con $i \neq j$ contiene il valore c per il vincolo $v_i - v_j \leq c$. Le celle sulla diagonale principale (dove $i = j$) contengono invece i valori per i vincoli della forma $v_i \leq c$.

Gli algoritmi utilizzati per effettuare l'analisi dei calcoli in virgola mobile utilizzando questo dominio astratto sono concettualmente identici a quelli usati per gli ottagoni, con l'ovvia eccezione che ci sono meno vincoli da aggiornare. Le operazioni con le differenze vincolate sono perciò più veloci rispetto a quelle sugli ottagoni ma questo dominio, pur rimanendo in grado di catturare diverse relazioni interessanti (può ad esempio provare che la variabile Y del rate limiter presentato nel listato 2.1 è limitata), è chiaramente meno preciso di quello ottagonale (si veda a tal proposito la sezione 4.2).

2.4.4 Poliedri generici

Invece di limitarci ad un sottoinsieme proprio dei poliedri, possiamo decidere di utilizzare l'intero insieme. Le operazioni sui poliedri generici sono particolarmente costose, ma la complessità computazionale viene bilanciata dalla notevole precisione di questo dominio astratto.

Un poliedro è definito da un insieme di vincoli della forma $\sum_v c_v v \leq c$. La PPL fornisce una classe astratta `C_Polyhedron` per rappresentare i poliedri; a differenza degli ottagoni e delle forme alle differenze vincolate, gli scalari non possono essere presi in $\mathbb{F}_{\mathbf{fa}}$ ma sono invece numeri interi in $\mathbb{Z}$ a precisione arbitraria forniti dalla GNU Multi-Precision Library (in particolare dalla classe `mpz_class`, si veda `http://gmplib.org/manual/`

per la relativa documentazione): ciò è dovuto al fatto che implementare degli operatori consistenti sui poliedri utilizzando coefficienti in virgola mobile è estremamente difficoltoso. Per questa ragione, dobbiamo trovare un modo per mantenere la consistenza continuando comunque ad affidarci alle tradizionali operazioni sui poliedri, la cui implementazione deve restare immutata: la soluzione proposta da Miné consiste nel sovrapprossimare ciascuna espressione linearizzata $l = [l; u] + \sum_v [l_v; u_v]\, v$ con una forma lineare l' in cui tutti gli intervalli, ad eccezione eventualmente del termine noto, sono singoletti. l' è calcolabile come:

$$\begin{aligned} l' = \; & \Big([l; u] \oplus^{\#} \bigoplus_v^{\#} (u_v \ominus_{\mathbf{fa},+\infty} l_v) \otimes^{\#} [0.5; 0.5] \otimes^{\#} \rho^{\#}(v) \Big) + \\ & \textstyle\sum_v \big((l_v \oplus_{\mathbf{fa},+\infty} u_v) \otimes^{\#} [0.5; 0.5] \big)\, v. \end{aligned}$$

Questa forma lineare può quindi essere convertita in due espressioni lineari a coefficienti razionali (una che prende il limite inferiore del termine noto, l'altra che prende invece il limite superiore) che sono dei limiti corretti per l'espressione originale e possono quindi essere passate come parametri agli algoritmi standard sui poliedri per ottenere un'analisi consistente.

3 Dettagli implementativi

3.1 Formati in virgola mobile

Un numero in virgola mobile di formato $\mathbf{f}$ è definito dalla Parma Polyhedra Library (PPL) tramite una struct come quella presentata nel listato 3.1, dove i campi statici contengono le informazioni specifiche sul formato (ovvero comuni a tutti i numeri di quel formato) come i valori di $\mathbf{e_f}$, $\mathbf{p_f}$ e $\mathbf{bias_f}$. In realtà gli algoritmi presentati in quest'opera fanno uso esclusivamente di questi campi statici, in quanto qualsiasi altra informazione sulle espressioni in virgola mobile del programma analizzato viene richiesta direttamente ad ECLAIR per mezzo di opportune interfacce (si veda la sezione 3.3).

3.2 Nuove funzioni della PPL per l'analisi dei calcoli in virgola mobile

Una forma lineare nella PPL viene rappresentata come un oggetto della classe `Linear_Form`, che è una classe generica il cui unico argomento di template è il tipo dei coefficienti. Ricordiamo che le forme lineari utilizzate per la nostra analisi hanno un tipo di coefficienti corrispondente ad un intervallo con limiti in virgola mobile in $\mathbb{F}_{\mathbf{fa}}$ (sottosezione 2.3.1).

```
1  struct float_ieee754_single {
2    uint32_t word;
3    static const uint32_t SGN_MASK = 0x80000000;
4    static const uint32_t EXP_MASK = 0x7f800000;
5    static const uint32_t POS_INF = 0x7f800000;
6    static const uint32_t NEG_INF = 0xff800000;
7    static const uint32_t POS_ZERO = 0x00000000;
8    static const uint32_t NEG_ZERO = 0x80000000;
9    static const unsigned int BASE = 2;
10   static const unsigned int EXPONENT_BITS = 8;
11   static const unsigned int MANTISSA_BITS = 23;
12   static const int EXPONENT_MAX = (1 << (EXPONENT_BITS - 1))
       - 1;
13   static const int EXPONENT_BIAS = EXPONENT_MAX;
14   static const int EXPONENT_MIN = -EXPONENT_MAX + 1;
15   static const int EXPONENT_MIN_DENORM
16      = EXPONENT_MIN - static_cast<int>(MANTISSA_BITS);
17   static const Floating_Point_Format floating_point_format =
       IEEE754_SINGLE;
18   int is_inf() const;
19   int is_nan() const;
20   int is_zero() const;
21   int sign_bit() const;
22   void negate();
23   void dec();
24   void inc();
25   void set_max(bool negative);
26   void build(bool negative, mpz_t mantissa, int exponent);
27 };
```

Listato 3.1: Una struct che definisce un formato floating point.

Tutti gli algoritmi descritti nella sezione 2.4 sono stati implementati nella PPL per ognuno dei domini astratti interessati (`BD_Shape`, `Octagonal_Shape` e `Polyhedron`). I corrispondenti metodi sono:

- `affine_form_image`, utilizzato per astrarre un assegnamento di un espressione in virgola mobile ad una variabile. I suoi argomenti sono la variabile che sta venendo assegnata e la forma lineare corrispondente all'espressione linearizzata.

- `refine_with_linear_form_inequality`, utilizzato per astrarre un filtro sull'espressione booleana $e_1 \leq e_2$. I suoi argomenti sono le linearizzazioni delle due sottoespressioni.

3.3 Interfacciare la PPL ed ECLAIR

L'algoritmo di linearizzazione implementato dalla PPL nel file `linearize.hh` è concettualmente quasi identico a quello presentato nella sottosezione 2.3.2, pertanto richiede l'accesso ad alcune informazioni sulle espressioni in virgola mobile del programma analizzato. Dal momento che la PPL dovrebbe rimanere completamente indipendente dalla rappresentazione delle espressioni utilizzata da qualsiasi particolare analizzatore, essa si limita a definire l'interfaccia `Concrete_Expression_Common` per una classe generica `Concrete_Expression` (comprendente anche le espressioni di tipo intero) e delega all'analizzatore il compito di implementarla concretamente. La stessa cosa viene effettuata per ogni sottoclasse di `Concrete_Expression` che definisce un più specifico genere di espressione quale `Floating_Point_Constant` (costante in virgola mobile) o `Approximable_Reference` (riferimento a una qualche quantità in memoria).

Le dichiarazioni per `Concrete_Expression` e `Concrete_Expression_Common` sono visibili nel listato 3.2; si presume che un analizzatore implementi una specializzazione templatica di `Concrete_Expression` dove l'argomento di template dato per `Target` viene utilizzato unicamente come un segnaposto che distingue le diverse implementazioni.

```
1  // La classe base di tutte le espressioni concrete.
2  template <typename Target>
3  class Concrete_Expression;
4
5  template <typename Target>
6  class Concrete_Expression_Common {
7  public:
8    // Restituisce il tipo di *this.
9    Concrete_Expression_Type type() const;
10
11   // Restituisce il genere di *this.
12   Concrete_Expression_Kind kind() const;
13
14   // Testa se *this ha lo stesso genere di Derived<Target>.
15   template <template <typename T> class Derived>
16   bool is() const;
17
18   // Restituisce this convertito al tipo Derived<Target>*.
19   template <template <typename T> class Derived>
20   Derived<Target>* as();
21
22   // Restituisce this convertito al tipo const Derived<Target>*.
23   template <template <typename T> class Derived>
24   const Derived<Target>* as() const;
25 };
```

Listato 3.2: Dichiarazioni di Concrete_Expression e Concrete_Expression_Common.

I metodi `type()` e `kind()` restituiscono rispettivamente un identificatore per il tipo (come intero o virgola mobile a precisione singola) e il genere (come costante, operatore o riferimento) dell'espressione, mentre i metodi `is()` e `as()` sono rispettivamente utilizzati per testare il genere dell'espressione o per effettuarne il cast ad una del genere specificato.

La rappresentazione concreta delle espressioni in ECLAIR è un nodo dell'albero della sintassi astratta (AST) ottenuto parsando il programma con Clang (un compiler front end sviluppato da Apple; si veda `http://clang.llvm.org/` per maggiori informazioni); la versione di `Concrete_Expression` implementata da ECLAIR, chiamata `PPL_Concrete_Expression`, è semplicemente un'interfaccia alternativa per un nodo dell'AST di Clang (il che significa che un puntatore ad una `PPL_Concrete_Expression` può

essere interpretato come un puntatore ad una `clang::Expr` e viceversa) i cui metodi sono implementati interrogando l'AST tramite le funzionalità offerte dallo stesso Clang.

Ci sono altre informazioni che dobbiamo ottenere dall'analizzatore per poter essere in grado di implementare l'algoritmo di linearizzazione, come ad esempio i valori degli intervalli contenuti in $\rho^{\#}$ o la dimensione dello spazio associata ad una particolare variabile o locazione di memoria. Ancora una volta, la soluzione consiste nello specificare un'interfaccia la cui implementazione concreta è lasciata all'analizzatore: tale interfaccia è chiamata `FP_Oracle` e la sua dichiarazione è visibile nel listato 3.3.

Da questo listato è possibile notare un'importante differenza fra il nostro algoritmo di linearizzazione e quello di Miné, in quanto le nostre espressioni possono essere riferimenti associati a diverse possibili dimensioni dello spazio. Il motivo è che ECLAIR implementa l'analisi points-to (si veda [Sof08]), che gli permette di approssimare la locazione di memoria associata ad un generico lvalue in un insieme di possibili dimensioni dello spazio associate (questo è utile in molti casi, come ad esempio quando l'espressione è la dereferenziazione di un puntatore che è stato manipolato tramite aritmetica dei puntatori). Un `Approximable_Reference` che può riferirsi a più dimensioni dello spazio $v_1, v_2, \ldots, v_n$ è semplicemente linearizzato nell'estremo superiore di $\rho^{\#}(v_1), \rho^{\#}(v_2), \ldots, \rho^{\#}(v_n)$ (ovvero il minimo intervallo che li contiene tutti).

3.4 Implementazione dell'analisi basata sugli intervalli

In ECLAIR, la struttura concreta della memoria viene astratta in un oggetto della classe `Abstract_Memory_Structure_Product`. Tale classe combina un dominio astratto utilizzato per l'analisi points-to con un altro utilizzato invece per l'analisi dei calcoli numerici, compresi quelli in virgola mobile. Quest'ultimo dominio, implementato dalla classe `AI_Numerical_Abstraction`, è stato ovviamente il punto focale del nostro lavoro.

```
// Classe usata per interrogare un analizzatore esterno.
template <typename Target, typename FP_Interval_Type>
class FP_Oracle {
public:
  /*
    Richiede un intervallo che approssimi l'entita' in
    virgola mobile referenziata da dim e memorizza il
    risultato in result. Restituisce true se l'analizzatore
    riesce a trovare un'approssimazione corretta, false
    altrimenti.
  */
  virtual bool get_interval(dimension_type dim,
                FP_Interval_Type& result) const = 0;

  /*
    Richiede un intervallo che approssimi la costante in
    virgola mobile expr e memorizza il risultato in result.
    Restituisce true se l'analizzatore riesce a trovare
    un'approssimazione corretta, false altrimenti.
  */
  virtual bool get_fp_constant_value(
                const Floating_Point_Constant<Target>& expr,
                      FP_Interval_Type& result) const = 0;

  /*
    Richiede un intervallo che approssimi il valore
    dell'espressione expr di tipo intero e memorizza il
    risultato in result.  Restituisce true se l'analizzatore
    riesce a trovare un'approssimazione corretta, false
    altrimenti.
  */
  virtual bool get_integer_expr_value(
                const Concrete_Expression<Target>& expr,
                FP_Interval_Type& result) const = 0;

  /*
    Richiede le possibili dimensioni dello spazio associate
    al riferimento expr e memorizza il risultato in result.
    Restituisce true se l'analizzatore e' riuscito a
    restituire l'insieme, falso altrimenti.
  */
  virtual bool get_associated_dimensions(
          const Approximable_Reference<Target>& expr,
          std::set<dimension_type>& result) const = 0;
};
```

Listato 3.3: Dichiarazione di FP_Oracle.

```
template <int max_exp, int min_exp, unsigned int precision>
class Floating_Point_Abstraction : public
    Individual_Abstraction {
public:
  // Funzioni d'utilità generale.
  ...

  // L'intervallo che approssima i possibili valori numerici.
  floating_point_interval_type interval;

  // Bitmask che approssima i possibili valori speciali.
  FP_Special special;
};

class IEEE754_Single_Abstraction
    : public Floating_Point_Abstraction<127, -126, 24>;
```

Listato 3.4: Floating_Point_Abstraction e IEEE_Single_Abstraction.

Internamente, una `AI_Numerical_Abstraction` contiene una `Cartesian_Abstraction` che ha lo scopo di memorizzare, per ogni variabile numerica (ricordiamo ancora una volta che le nostre variabili non corrispondono a quelle dichiarate dal programmatore), un'astrazione individuale che l'approssimi correttamente. Un'astrazione individuale per una variabile in virgola mobile è un'istanza della classe `Floating_Point_Abstraction`, la cui definizione è mostrata nel listato 3.4. Come è possibile notare, si tratta in realtà di un'astrazione ad intervallo come quelle definite nella sottosezione 2.2.1, dove `interval` è un intervallo della PPL con limiti in virgola mobile e `special` svolge il ruolo del NaN bit. In questo modo una `Cartesian_Abstraction` svolge il ruolo dello store astratto $\rho^{\#}$.

I tre argomenti di template definiscono il formato floating point: il primo argomento è il massimo valore E tale che 2^E sia esattamente rappresentabile nel formato, il secondo è il minimo valore e tale che 2^e sia un numero normalizzato nel formato, mentre il terzo è il numero di bit che il formato prescrive per il significando (mantissa + bit di segno). Il listato 3.4 mostra anche, come esempio di specializzazione di `Floating_Point_Abstraction`, la dichiarazione per la classe `IEEE754_Single_Abstraction` che è un'astrazione per una variabile avente il formato IEEE754 a precisione singola.

L'analisi basata sugli intervalli condotta dall'analizzatore ECLAIR è concettualmente identica a quella presentata nella sottosezione 2.2.2, con l'unica eccezione che quando un riferimento può corrispondere a più di una variabile lo valutiamo all'estremo superiore di tutti gli intervalli nella `Cartesian_Abstraction` associati ad ogni possibile variabile. Analogamente, se il riferimento compare sul lato sinistro di un assegnamento, effettuiamo l'assegnamento in modo indipendente su ciascuna possibile variabile e prendiamo come risultato finale l'estremo superiore fra tutte le `Cartesian_Abstraction` così ottenute.

3.5 Implementazione dell'analisi basata su poliedri

Un nuovo campo `fp_domain`, che è un puntatore ad un dominio astratto poliedrico, è stato aggiunto alla classe `AI_Numerical_Abstraction` per migliorare la precisione dell'analisi oltre quella fornita dalla semplice analisi basata sugli intervalli (si veda la sottosezione 2.2.3).

Attualmente, il poliedro può essere sia un `Octagon` che una `BD_Shape` a seconda del bilanciamento fra precisione e complessità computazionale che desideriamo ottenere. Il supporto per i poliedri generici è tuttora incompleto in quanto alcuni metodi necessari della classe `Polyhedron` devono ancora essere implementati.

Questa analisi estesa utilizza l'algoritmo di linearizzazione della PPL (si veda la sezione 3.3) ed è basata sugli algoritmi presentati nel capitolo precedente con la solita estensione per il caso in cui il lato sinistro di un assegnamento può riferirsi a più di una variabile, nel quale procediamo effettuando l'assegnamento astratto del lato destro linearizzato indipendentemente su ciascuna possibile dimensione dello spazio e tenendo come risultato finale l'estremo superiore (ottenuto utilizzando il metodo `join_assign` dei poliedri) di tutti i poliedri così ottenuti.

L'analisi basata sui poliedri viene eseguita insieme a quella basata sugli intervalli: dopo ciascuna operazione d'analisi, i due domini astratti vengono raffinati l'un l'altro come descritto nella sottosezione 2.4.2 chiamando il metodo `refine_internal_domains` della

classe `AI_Numerical_Abstraction`. Poiché i risultati della linearizzazione dipendono da quelli dell'analisi basata sugli intervalli, è necessario adottare l'accorgimento di effettuare sempre la linearizzazione del lato destro di un assegnamento prima di aggiornare la `Cartesian_Abstraction`, in quanto altrimenti il risultato della linearizzazione potrebbe essere scorretto nel caso in cui il valore dell'espressione da linearizzare dipenda da quello della variabile assegnata.

È possibile arrestare in qualunque momento l'analisi basata sui poliedri chiamando il metodo `abort_extended_fp_analysis` di `AI_Numerical_Abstraction`; questo metodo utilizza il poliedro per raffinare la `Cartesian_Abstraction` in modo da ridurre la perdita di informazione, quindi dealloca il poliedro ed imposta il puntatore `fp_domain` al valore speciale `NULL`.

4 Esempi di analisi

4.1 Rate Limiter

4.1.1 Rate limiter annotato

Abbiamo già discusso come il rate limiter presentato nel listato 2.1 non possa essere analizzato in modo soddisfacente utilizzando esclusivamente l'analisi basata sugli intervalli. Mostreremo ora come ECLAIR sia (quasi) in grado di dimostrare che la variabile Y del programma è limitata utilizzando l'analisi basata sui poliedri, e in particolare sugli ottagoni; sfortunatamente ECLAIR non è ancora in grado di produrre un output leggibile da umano, ma stampa semplicemente delle informazioni di debug difficilmente leggibili per chiunque non conosca i dettagli interni dell'analizzatore, per cui non mostreremo l'output vero e proprio ma solamente una sua interpretazione.

Per agevolare la presentazione dei risultati considereremo la versione annotata del rate limiter mostrata nel listato 4.1, nella quale abbiamo individuato alcuni punti del programma di particolare interesse.

```
1  double Y, S, R, X, D;
2  Y = 0;
3  while (true) {
4    /* PUNTO 1 */
5
6    // Genera casualmente un valore per X fra -128 e 128.
7    // Genera casualmente un valore per D fra 0 e 16.
8
9    /* PUNTO 2 */
10
11   S = Y;
12   /* PUNTO 3 */
13   R = X - S;
14   /* PUNTO 4 */
15   Y = X;
16   /* PUNTO 5 */
17
18   if (R <= -D) {
19     /* PUNTO 6 */
20     Y = S - D;
21     /* PUNTO 7 */
22   }
23
24   /* PUNTO 8 */
25
26   if (R >= D) {
27     /* PUNTO 9 */
28     Y = S + D;
29     /* PUNTO 10 */
30   }
31
32   /* PUNTO 11 */
33 }
```

Listato 4.1: Rate limiter annotato.

4.1.2 Correttezza del rate limiter

È facile dimostrare per induzione sul numero di iterazioni che al punto 11 il valore di Y è limitato da $[-128; 128]$. Si noti che nella prima iterazione al punto 4 si ha $Y = S = 0$, il che implica $S - D = 0 - D = -D \in [-16; 0] \subseteq [-128; 128]$ e $S + D = 0 + D = D \in [0; 16] \subseteq [-128; 128]$, perciò alla fine del ciclo $Y \in [-128; 128]$ indipendentemente da quali assegnamenti sono o non sono stati effettuati (si noti che in ogni formato floating point ragionevole ± 16 e ± 128 sono esattamente rappresentabili, per cui non c'è alcun rischio di superarli come conseguenza di un errore di arrotondamento).

Si supponga ora che $Y \in [-128; 128]$ al termine dell'iterazione numero $n - 1$. Nell'iterazione n al punto 4 avremo ovviamente $Y \in [-128; 128]$ (dal momento che non è stato modificato dal termine dell'iterazione precedente) e ci sono tre possibilità:

- Se $-D < R < D$, entrambe le guardie booleane $R <= -D$ e $R >= D$ valuteranno a falso, quindi alla fine del ciclo $Y = X \in [-128; 128]$.

- Se $R \leq -D$, alla fine del ciclo abbiamo $Y = S - D$. Ora poiché $R = X - S$ abbiamo $X - S \leq -D$ dunque $S - D \geq X$ e $X \leq Y$. Ovviamente abbiamo anche $Y \leq S$ (dal momento che Y è ottenuto sottraendo da S una quantità non negativa). Infine, $X \leq Y \leq S$ dove $X, S \in [-128; 128]$ (dal momento che X viene generato in quell'intervallo e S è il vecchio valore di Y che soddisfava l'ipotesi induttiva), perciò abbiamo anche $Y \in [-128; 128]$.

- Se $R \geq D$, alla fine del ciclo abbiamo $Y = S + D$. Ora poiché $R = X - S$ abbiamo $X - S \geq D$ dunque $S + D \leq X$ e $Y \leq X$. Ovviamente abbiamo anche $S \leq Y$ (dal momento che Y è ottenuto sommando ad S una quantità non negativa). Infine, $S \leq Y \leq X$ dove $X, S \in [-128; 128]$ (dal momento che X viene generato in quell'intervallo e S è il vecchio valore di Y che soddisfava l'ipotesi induttiva), perciò abbiamo anche $Y \in [-128; 128]$.

4.1.3 Risultati sperimentali

Qui riportiamo alcuni risultati ottenuti da ECLAIR durante l'analisi del rate limiter effettuata utilizzando il dominio degli ottagoni.

Prima iterazione, punto 5:

```
Y >= -128, Y <= 128, X >= -128, X <= 128, D >= 0, D <= 16, S = 0,
R >= -128.000030517578125, R <= 128.000030517578125,
Y - X = 0, Y + X >= -256,
Y + X <= 256, Y - D <= 128, D - Y <= 144, Y + D >= -128, Y + D <= 144,
X - D <= 128, D - X <= 144, X + D >= -128, X + D <= 144, Y - S <= 128,
S - Y <= 128, Y + S >= -128, Y + S <= 128, X - S <= 128, S - X <= 128,
X + S >= -128, X + S <= 128, D - S <= 16, S - D <= 0,
D + S >= 0, D + S <= 16,
Y - R <= 1.525879088148940354585647583007812 5e-05,
R - Y <= 1.525879088148940354585647583007812 5e-05,
Y + R >= -256.000030517578125, Y + R <= 256.000030517578125,
X - R <= 1.525879088148940354585647583007812 5e-05,
R - X <= 1.525879088148940354585647583007812 5e-05,
X + R >= -256.000030517578125, X + R <= 256.000030517578125,
D - R <= 144.000030517578125, R - D <= 128.000030517578125,
D + R >= -128.000030517578125, D + R <= 144.000030517578125,
S - R <= 128.000030517578125, R - S <= 128.000030517578125,
S + R >= -128.000030517578125, S + R <= 128.000030517578125
```

Qui si può immediatamente notare che nonostante il dominio degli ottagoni non sia in grado di rappresentare vincoli ternari come $R = X - S$, è comunque in grado di dedurre relazioni non banali come $S + R \in [-128; 128]$; qui in realtà nel risultato è stato aggiunto un errore di arrotondamento inesistente, ma esso sarà poi rimosso raffinando l'ottagono con la `Cartesian_Abstraction`. Questo è un esempio che dimostra come l'analisi basata sui poliedri non sia necessariamente strettamente più precisa di quella basata sugli intervalli.

Prima iterazione, punto 6:

```
Y >= -128, Y <= 1.525879088148940354585647583007812 5e-05, X >= -128,
X <= 1.525879088148940354585647583007812 5e-05,
D >= 0, D <= 16, S = 0,
R >= -128, R <= 0, Y - X = 0, Y + X >= -256,
Y + X <= 3.051758176297880709171295166015625e-05,
Y - D <= 1.525879088148940354585647583007812 5e-05,
D - Y <= 144, Y + D >= -128,
Y + D <= 1.525879088148940354585647583007812 5e-05,
X - D <= 1.525879088148940354585647583007812 5e-05,
D - X <= 144, X + D >= -128,
X + D <= 1.525879088148940354585647583007812 5e-05,
Y - S <= 1.525879088148940354585647583007812 5e-05,
S - Y <= 128, Y + S >= -128,
Y + S <= 1.525879088148940354585647583007812 5e-05,
X - S <= 1.525879088148940354585647583007812 5e-05,
S - X <= 128, X + S >= -128,
X + S <= 1.525879088148940354585647583007812 5e-05,
D - S <= 16, S - D <= 0,
D + S >= 0, D + S <= 16,
```

```
Y - R <= 1.525879088148940354585647583007812 5e-05,
R - Y <= 1.525879088148940354585647583007812 5e-05, Y + R >= -256,
Y + R <= 1.525879088148940354585647583007812 5e-05,
X - R <= 1.525879088148940354585647583007812 5e-05,
R - X <= 1.525879088148940354585647583007812 5e-05, X + R >= -256,
X + R <= 1.525879088148940354585647583007812 5e-05,
D - R <= 144, R - D <= 0,
D + R >= -128, D + R <= 0, S - R <= 128, R - S <= 0,
S + R >= -128, S + R <= 0
```

Qui viene mostrata l'applicazione di un filtro. Si notino le deduzioni non banali come $S + R \in [-128; 0]$ o $Y <= 1.525879088148940354585647583007812 5e - 05$.

Prima iterazione, punto 7:

```
Y >= -16.000003814697265625,
Y <= 1.401298464324817070923729583289916131280261941876515771 7e-45,
X >= -128, X <= 1.525879088148940354585647583007812 5e-05,
D >= 0, D <= 16,
S = 0, R >= -128, R <= 0, Y - X <= 128.0000152587890625,
X - Y <= 1.716614133329130709171295166015625e-05,
Y + X >= -144.0000152587890625,
Y + X <= 1.525879270047880709171295166015625e-05,
Y - D <= 1.401298464324817070923729583289916131280261941876515772e-45,
D - Y <= 32.000003814697265625,
Y + D >= -1.907348860186175443232059478759765625e-06,
Y + D <= 9.536744300930877216160297393798828125e-07,
X - D <= 1.525879088148940354585647583007812 5e-05,
D - X <= 144, X + D >= -128, X + D <= 1.525879088148940354585647 6e-05,
```

```
Y - S <= 1.401298464324817070923729583289916131280261941876515772e-45,
S - Y <= 16.000003814697265625, Y + S >= -16.000003814697265625,
Y + S <= 1.401298464324817070923729583289916131280261941876515772e-45,
X - S <= 1.525879088148940354585647583007812 5e-05,
S - X <= 128, X + S >= -128,
X + S <= 1.525879088148940354585647583007812 5e-05,
D - S <= 16, S - D <= 0, D + S >= 0, D + S <= 16,
Y - R <= 128.0000152587890625,
R - Y <= 1.907348860186175443232059478759765625e-06,
Y + R >= -144.0000152587890625,
Y + R <= 1.401298464324817070923729583289916131280261941876515772e-45,
X - R <= 1.525879088148940354585647583007812 5e-05,
R - X <= 1.525879088148940354585647583007812 5e-05, X + R >= -256,
X + R <= 1.525879088148940354585647583007812 5e-05,
D - R <= 144, R - D <= 0, D + R >= -128, D + R <= 0, S - R <= 128,
R - S <= 0, S + R >= -128, S + R <= 0
```

(Nulla in particolare da notare qui)

Prima iterazione, punto 8:

```
Y >= -16.000003814697265625,
Y <= 1.401298464324817070923729583289916131280261941876515771 7571e-45,
X >= -128, X <= 1.525879088148940354585647583007812 5e-05,
D >= 0, D <= 16, S = 0,
R >= -128, R <= 0, Y - X <= 128.0000152587890625,
X - Y <= 1.716614133329130709171295166015625e-05,
Y + X >= -144.0000152587890625,
Y + X <= 1.525879270047880709171295166015625e-05,
```

```
Y - D <= 1.401298464324817070923729583289916131280261941876515772e-45,
D - Y <= 32.000003814697265625,
Y + D >= -1.907348860186175443232059478759765625e-06,
Y + D <= 9.536744300930877216160297393798828125e-07,
X - D <= 1.525879088148940354585647583007812 5e-05,
D - X <= 144, X + D >= -128,
X + D <= 1.525879088148940354585647583007812 5e-05,
Y - S <= 1.401298464324817070923729583289916131280261941876515772e-45,
S - Y <= 16.000003814697265625, Y + S >= -16.000003814697265625,
Y + S <= 1.401298464324817070923729583289916131280261941876515772e-45,
X - S <= 1.525879088148940354585647583007812 5e-05,
S - X <= 128, X + S >= -128,
X + S <= 1.525879088148940354585647583007812 5e-05,
D - S <= 16, S - D <= 0,
D + S >= 0, D + S <= 16, Y - R <= 128.0000152587890625,
R - Y <= 1.907348860186175443232059478759765625e-06,
Y + R >= -144.0000152587890625,
Y + R <= 1.401298464324817070923729583289916131280261941876515772e-45,
X - R <= 1.525879088148940354585647583007812 5e-05,
R - X <= 1.525879088148940354585647583007812 5e-05,
X + R >= -256,
X + R <= 1.525879088148940354585647583007812 5e-05,
D - R <= 144, R - D <= 0,
D + R >= -128, D + R <= 0, S - R <= 128, R - S <= 0,
S + R >= -128, S + R <= 0
```

Qui vengono mostrati i risultati per un punto di programma in cui confluiscono diversi rami del flusso di controllo.

Prima iterazione, punto 11:

```
Y >= -16.0000171661376953125, Y <= 128,
X >= -128, X <= 128, D >= 0, D <= 16,
S = 0, R >= -128, R <= 128,
Y - X <= 128, X - Y <= 128,
Y + X >= -144, Y + X <= 144.000030517578125,
Y - D <= 128, D - Y <= 32.000019073486328125,
Y + D >= -1.525879088148940354585647583007812e-05,
Y + D <= 144, X - D <= 128,
D - X <= 144, X + D >= -128, X + D <= 144, Y - S <= 128,
S - Y <= 16.0000171661376953125, Y + S >= -16.0000171661376953125,
Y + S <= 128, X - S <= 128, S - X <= 128,
X + S >= -128, X + S <= 128,
D - S <= 16, S - D <= 0, D + S >= 0, D + S <= 16,
Y - R <= 128, R - Y <= 128,
Y + R >= -144, Y + R <= 144,
X - R <= 1.525879088148940354585647583007812e-05,
R - X <= 1.525879088148940354585647583007812e-05,
X + R >= -256, X + R <= 256,
D - R <= 144, R - D <= 128,
D + R >= -128, D + R <= 144, S - R <= 128, R - S <= 128,
S + R >= -128, S + R <= 128
```

Mostra come Y sia correttamente limitata dopo la prima iterazione e come venga dedotto il vincolo $S + R \in [-128; 128]$.

Seconda iterazione, punto 11:

```
Y >= -32.00003814697265625, Y <= 136.00006103515625,
X >= -128, X <= 128,
D >= 0, D <= 16, S >= -16.0000171661376953125,
S <= 128, R >= -256,
R <= 144.000030517578125, Y - X <= 256,
X - Y <= 144.0000762939453125,
Y + X >= -160.000030517578125, Y + X <= 264.00006103515625,
Y - D <= 128.000030517578125, D - Y <= 48.00003814697265625,
Y + D >= -16.00003814697265625, Y + D <= 144.0000762939453125,
X - D <= 128,
D - X <= 144, X + D >= -128, X + D <= 144,
Y - S <= 144.000030517578125,
S - Y <= 160.0000457763671875, Y + S >= -48.000057220458984375,
Y + S <= 256.000091552734375,
X - S <= 144.000030517578125, S - X <= 256,
X + S >= -144.000030517578125, X + S <= 256,
D - S <= 32.000019073486328125,
S - D <= 128, D + S >= -16.0000171661376953125,
D + S <= 144, Y - R <= 384,
R - Y <= 160.00006103515625, Y + R >= -144.00006103515625,
Y + R <= 144.0000762939453125, X - R <= 128.0000457763671875,
R - X <= 16.00003814697265625,
X + R >= -384, X + R <= 272.000030517578125,
D - R <= 272, R - D <= 144.000030517578125,
D + R >= -256,
D + R <= 160.000030517578125,
```

```
S - R <= 384, R - S <= 160.00006103515625,
S + R >= -128.0000457763671875, S + R <= 128.0000457763671875
```

Mostra come, dopo un'ulteriore iterazione, il limite superiore dedotto per Y raggiunga un valore (circa 136) che eccede il vero limite superiore (il quale vale 128). In effetti il dominio degli ottagoni non riesce a trovare i limiti più precisi, non essendo in grado di rappresentare il vincolo $R = X - S$. Tuttavia, il vincolo $S + R \in [-128; 128]$ continua a valere e viene correttamente dedotto a meno di un errore di arrotondamento.

Per quanto riguarda le iterazioni successive, sfortunatamente al momento ECLAIR non è in grado di analizzarle in modo preciso a causa dell'inadeguatezza dell'operatore di widening che viene usato attualmente (si veda anche la sezione 5.2): dopo la prima iterazione dove $Y \in [-16; 128]$ e la seconda dove $Y \in [-32; 136]$, tale widening procede immediatamente a scartare tutti i vincoli su Y. Tuttavia sappiamo, in base ad alcuni esperimenti condotti (si veda il file `digitalfilters1.cc` all'interno della test suite della Parma Polyhedra Library), che cambiare la modalità di widening sarà sufficiente a dimostrare che Y è limitata. In particolare, qualunque widening che vada a restringere Y ad un intervallo $[-B; B]$ dove $B >= 144$ renderà i vincoli su Y stabili per tutte le iterazioni successive. Per convincersi, si supponga che all'inizio di un'iterazione si abbia $Y \in [-B; B]$. Allora al punto 3 $S \in [-B; B]$ e l'analisi procederà in questo modo:

- Al punto 6, la guardia booleana implica $R + D \leq 0$ e poiché $D \geq 0$ abbiamo anche $-R \geq 0$. Come abbiamo già visto, il dominio degli ottagoni è in grado di inferire che $S + R \in [-128; 128]$ perciò $S = (S + R) - R \geq -128$ e $S \in [-128; B]$. Al punto 7 abbiamo $Y - S = -D \in [-16; 0]$ dunque $Y = (Y - S) + S \in [-144; B]$.

- Al punto 9, la guardia booleana implica $R - D \geq 0$ e poiché $D \geq 0$ abbiamo anche $-R \leq 0$. Come abbiamo già visto, il dominio degli ottagoni è in grado di inferire che $S + R \in [-128; 128]$ perciò $S = (S + R) - R \leq 128$ e $S \in [-B; 128]$. Al punto 10 abbiamo $Y - S = D \in [0; 16]$ dunque $Y = (Y - S) + S \in [-B; 144]$.

```
1   double X, Y;
2
3   // Genera casualmente un valore per X fra -100 e 100.
4
5   Y = X;
6
7   // PUNTO 1
8   if (Y <= 0) {
9     // PUNTO 2
10    Y = -Y;
11    // PUNTO 3
12  }
13  else {
14    // PUNTO 4
15  }
16
17  // PUNTO 5
18
19  if (Y <= 69) {
20    // PUNTO 6
21  }
```

Listato 4.2: Semplice programma che mostra un limite delle differenze vincolate.

- Al punto 11, i risultati dei punti 7 e 10 vengono uniti ed otteniamo $Y \in [-max\{128, M\}; max\{128, M\}]$.

Lo stesso limite per Y è stato anche trovato sperimentalmente utilizzando le differenze vincolate al posto degli ottagoni.

4.2 Limiti delle differenze vincolate

L'analisi del programma presentato nel listato 4.2 costituisce un esempio di come gli ottagoni possano fornire un risultato migliore rispetto alle differenze vincolate.

Ecco i risultati ottenuti utilizzando differenze vincolate ed ottagoni:

Differenze vincolate, punto 1:

```
X >= -100, X <= 100, Y >= -100, Y <= 100, X - Y = 0
```

Ottagono, punto 1:

```
X >= -100, X <= 100, Y >= -100, Y <= 100, X - Y = 0,
X + Y <= 200, X + Y >= -200
```

Differenze vincolate, punto 2:

```
X >= -100, X <= 0, Y >= -100, Y <= 0, X - Y = 0
```

Ottagono, punto 2:

```
X >= -100, X <= 0, Y >= -100, Y <= 0, X - Y = 0,
X + Y <= 0, X + Y >= -200
```

Differenze vincolate, punto 3:

```
X >= -100, X <= 0, Y >= 0, Y <= 100, X - Y <= 0, Y - X <= 200
```

Ottagono, punto 3:

```
X >= -100, X <= 0, Y >= 0, Y <= 100, X - Y >= -200, X - Y <= 0, X + Y = 0
```

Differenze vincolate, punto 4:

```
X >= 0, X <= 100, Y >= 0, Y <= 100, X - Y = 0
```

Ottagono, punto 4:

```
X >= 0, X <= 100, Y >= 0, Y <= 100, X - Y = 0, X + Y >= 0, X + Y <= 200
```

Differenze vincolate, punto 5:

```
X >= -100, X <= 100, Y >= 0, Y <= 100, X - Y <= 0, Y - X <= 200
```

Ottagono, punto 5:

```
X >= -100, X <= 100, Y >= 0, Y <= 100, X - Y >= -200, X - Y <= 0,
X + Y >= 0, X + Y <= 200
```

Differenze vincolate, punto 6:

```
X >= -100, X <= 69, Y >= 0, Y <= 69, X - Y <= 0, Y - X <= 169
```

Ottagono, punto 6:

```
X >= -69, X <= 69, Y >= 0, Y <= 69, X - Y >= -138, X - Y <= 0,
X + Y >= 0, X + Y <= 138
```

Da questi risultati si nota come il limite inferiore trovato per X sia considerevolmente più preciso con l'analisi basata sugli ottagoni; la ragione sta nel fatto che le differenze vincolate non possono rappresentare il vincolo $X + Y = 0$.

```
1 double A, B;
2 A = 16;
3 B = 1;
4 while (A > 0) {
5   B = B + 1;
6   A = A - 1;
7 }
```

Listato 4.3: Un semplice ciclo su una quantità decrescente.

4.3 Limiti del widening

Nella sezione 4.1 abbiamo visto come l'utilizzo di operatori di widening troppo imprecisi possa portare a risultati dell'analisi insoddisfacenti. Il listato 4.3 mostra ora un esempio in cui qualunque operatore di widening, per quanto preciso, sarebbe insufficiente.

Ecco i risultati ottenuti da ECLAIR (utilizzando ancora una volta il dominio degli ottagoni) sul programma in questione al termine di varie iterazioni del ciclo:

Iterazione 0 (prima entrata nel ciclo):

```
A = 16, B = 1, A - B = 15, A + B = 17
```

Iterazione 1:

```
A >= 14.9999980926513671875, A <= 15.00000286102294921875,
B >= 1.999999761581420898437 5, B <= 2.000000476837158203125,
A - B <= 13.000003814697265625, A - B >= 12.999971389770507812 5,
A + B >= 16.999996185302734375, A + B <= 17.000003814697265625
```

Per il momento la precisione è soddisfacente, anche perché il widening deve ancora intervenire. All'inizio dell'iterazione 2, il widening fra i risultati per le iterazioni 0 e 1 restituirà

```
A >= 15, A <= 16, B >= 1, B <= 2, A - B <= 15,
A - B >= 13, A + B = 17
```

Iterazione 2:

```
A >= 13.9999980926513671875, A <= 15.00000286102294921875,
B >= 1.999999761581420898437 5, B <= 3.000000476837158203125,
A - B <= 13.000003814697265625, A - B >= 10.99999713897705078125,
A + B >= 15.99999713897705078125, A + B <= 18.000003814697265625
```

Benché i valori esatti per A e B dopo la seconda iterazione nel concreto siano rispettivamente 14 e 3, la nostra analisi per come è progettata deve restituire un'approssimazione che valga anche per TUTTE le iterazioni precedenti che hanno raggiunto il termine del ciclo. Si può già cominciare ad intuire come questo ci porterà ad ottenere un risultato finale molto impreciso all'uscita del ciclo.

Iterazione 3:

```
A >= 12.9999980926513671875, A <= 15.00000286102294921875,
B >= 1.999999761581420898437 5, B <= 4.00000095367431640625,
A - B <= 13.000003814697265625, A - B >= 8.99999713897705078125,
A + B >= 14.99999713897705078125, A + B <= 19.000003814697265625
```

Ora dovrebbe essere chiaro come procederà l'analisi per tutte le iterazioni successive: mentre i vincoli `A <= 15.00000286102294921875` e `B >= 1.999999761581420898437 5` rimarranno stabili, gli altri due limiti per A e B resteranno instabili e saranno prima o poi scartati dal widening, ottenendo un risultato finale decisamente insoddisfacente.

Risultato finale:

```
A >= -1, A <= 0, B >= 2, B - A >= 2, A + B >= 1
```

Per quanto detto finora, tale risultato è ben lontano da quello più preciso in cui $A = 0$ (nonostante la guardia booleana del ciclo ci permetta di ottenere comunque "gratuitamente" una discreta approssimazione per A) e soprattutto $B = 16$. Come si è

visto, il problema è qui dovuto ad una limitazione intrinseca dell'approccio basato sui soli widening; questo problema può essere risolto affiancando all'operatore di widening un operatore cosiddetto di *narrowing*. Nella teoria dell'interpretazione astratta un operatore di narrowing è in un certo senso l'inverso di un operatore di widening in quanto invece di partire dallo stato astratto all'inizio del ciclo e iterare in avanti, parte dallo stato finale e cerca di renderlo più preciso ripercorrendo le iterazioni all'indietro; per maggiori informazioni su tali operatori si veda [Cou05].

Attualmente ECLAIR non supporta (ancora) il narrowing né alcun'altra tecnica aggiuntiva per migliorare la precisione in presenza di cicli, perciò per il momento siamo costretti ad accontentarci di questi risultati.

5 Conclusioni

5.1 Stato attuale

Abbiamo visto come ECLAIR sia ora in grado di approssimare il valore dei calcoli numerici (sia interi che in virgola mobile) utilizzando un dominio astratto strutturato come un prodotto cartesiano di astrazioni individuali; questo dominio mappa ogni locazione di memoria contenente un valore numerico in un intervallo che sovrapprossima correttamente il valore concreto. Le operazioni astratte sugli intervalli con limiti in virgola mobile approssimano correttamente le corrispondenti operazioni concrete in quanto utilizzano un formato floating point non più preciso di quello che viene usato nel concreto e fanno un uso consistente dell'arrotondamento. L'analisi points-to permette inoltre di tener traccia delle possibili locazioni di memoria associate agli lvalue nel programma, permettendo così di analizzare i programmi che fanno uso di funzionalità "complicate" quali ad esempio l'aritmetica dei puntatori.

Abbiamo inoltre visto come la precisione di tale analisi sia spesso insufficiente per le applicazioni reali, non essendo essa in grado di dedurre vincoli relazionali. I domini astratti forniti dalla Parma Polyhedra Library (PPL) erano già in grado di affrontare con successo questo problema per quanto riguarda l'aritmetica esatta, ma non fornivano ancora alcun metodo per astrarre assegnamenti o filtri che coinvolgessero quantità in virgola mobile. Abbiamo perciò presentato un modo per estendere questi domini aggiungendo degli operatori consistenti che fanno uso di forme lineari; tali forme lineari, prodotte dall'algoritmo di linearizzazione, sono approssimazioni consistenti delle espressioni in virgola mobile del programma concreto e tengono conto di tutti i possibili errori di

arrotondamento.

Tutte queste nuove operazioni sono ora implementate all'interno della PPL e possono essere utilizzate con successo da ECLAIR per catturare molti vincoli relazionali che sussistono fra le quantità in virgola mobile all'interno dei programmi.

Atttraverso la definizione di opportune interfacce astratte (`FP_Oracle` e `Concrete_Expression`), la nostra implementazione garantisce l'indipendenza della PPL sia rispetto all'analizzatore che rispetto alla rappresentazione delle espresssioni concrete.

5.2 Sviluppi futuri

La struttura delle astrazioni ad intervallo sta attualmente venendo rivista; la nuova futura rappresentazione dovrebbe fra l'altro essere in grado supportare i signaling NaN. Il codice responsabile dell'analisi dei calcoli in virgola mobile all'interno di ECLAIR dovrà essere pesantemente ottimizzato, in particolare per quanto riguarda l'uso della memoria (al momento molta memoria viene sprecata dai domini relazionali per dimensioni dello spazio che non vengono utilizzate in quanto non corrispondono ad alcuna quantità floating point).

Verrà presto aggiunto anche un dominio astratto per l'analisi dei calcoli sugli interi, la cui interazione con i due domini già esistenti dovrà essere attentamente pianificata.

L'utilizzo del widening all'interno di ECLAIR sta venendo totalmente rivisto in modo da migliorare la precisione dell'analisi in presenza di cicli. In futuro l'utente finale potrà specificare degli insiemi di vincoli che si aspetta di veder verificati in diversi punti del programma e tale sistema di vincoli potrà essere sfruttato allo scopo di decidere quali widening utilizzare.

Dopo aver migliorato l'uso del widening sarà necessario migliorare ulteriormente la precisione dell'analisi, in particolare per risolvere le problematiche evidenziate nella sezione 4.3. Bisognerà infine ampliare il numero di formati in virgola mobile supportati

(al momento gli unici ad essere stati adeguatamente testati sono quelli di IEEE754 in base 2).

Bibliografia

[46108] *Ieee standard for floating-point arithmetic.* IEEE Std 754-2008, pagine 1–58, 2008.

[BHRZ05] Bagnara, R., P. M. Hill, E. Ricci e E. Zaffanella: *Precise widening operators for convex polyhedra.* Science of Computer Programming, 58(1–2):28–56, 2005. `http://www.cs.unipr.it/ppl/Documentation/BagnaraHRZ05SCP.pdf`.

[BHZ06] Bagnara, R., P. M. Hill e E. Zaffanella: *Widening operators for powerset domains.* Software Tools for Technology Transfer, 8(4/5):449–466, 2006. `http://www.cs.unipr.it/ppl/Documentation/BagnaraHZ06STTT.pdf`, Nella versione stampata di questo articolo, tutte le figure sono state stampate in modo improprio (rendendole inutili). Si veda [BHZ07].

[BHZ07] Bagnara, R., P. M. Hill e E. Zaffanella: *Widening operators for powerset domains.* Software Tools for Technology Transfer, 9(3/4):413–414, 2007. Correzione di [BHZ06] contenente tutte le figure correttamente stampate.

[BHZ08] Bagnara, R., P. M. Hill e E. Zaffanella: *The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems.* Science of Computer Programming, 72(1–2):3–21, 2008. `http://www.cs.unipr.it/ppl/Documentation/BagnaraHZ08SCP.pdf`.

[BHZ09a] Bagnara, R., P. M. Hill e E. Zaffanella: *Applications of polyhedral computations to the analysis and verification of hardware and software systems.* Theoretical

Computer Science, 410(46):4672–4691, 2009. http://www.cs.unipr.it/ppl/Documentation/BagnaraHZ09TCS.pdf.

[BHZ09b] Bagnara, R., P. M. Hill e E. Zaffanella: *Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness*. Formal Methods in System Design, 35(3):279–323, 2009. http://www.cs.unipr.it/ppl/Documentation/BagnaraHZ09FMSD.pdf.

[Cou05] Cousot, P.: *Abstract interpretation*. 2005.

[Min04] Miné, A.: *Relational abstract domains for the detection of floating-point run-time errors*. Nel Schmidt, D. (curatore): *Programming Languages and Systems: Proceedings of the 13th European Symposium on Programming*, volume 2986 della serie *Lecture Notes in Computer Science*, pagine 3–17, Barcelona, Spain, 2004. Springer-Verlag, Berlin, ISBN 3-540-213213-9.

[Min05] Miné, A.: *Weakly Relational Numerical Abstract Domains*. tesi di dottorato, École Polytechnique, Paris, France, marzo 2005.

[Sch95] Schmidt, D. A.: *Natural-semantics-based abstract interpretation (preliminary version)*. Nel Mycroft, A. (curatore): *Static Analysis: Proceedings of the 2nd International Symposium*, volume 983 della serie *Lecture Notes in Computer Science*, pagine 1–18, Glasgow, UK, 1995. Springer-Verlag, Berlin, ISBN 3-540-60360-3.

[Sof08] Soffia, Stefano: *Definition and implementation of a points-to analysis for c-like languages*. 2008.